PSPP Developers Guide

GNU PSPP Statistical Analysis Software
Release 2.0.1-g5a65e7

Ben Pfaff
John Darrington

This manual is for GNU PSPP version 2.0.1-g5a65e7, software for statistical analysis.

Table of Contents

This manual describes the file formats that PSPP supports.

1 System File Format

An SPSS system file holds a set of cases and dictionary information that describes how they may be interpreted. The system file format dates back 40+ years and has evolved greatly over that time to support new features, but in a way to facilitate interchange between even the oldest and newest versions of software. This chapter describes the system file format.

System files use four data types: 8-bit characters, 32-bit integers, 64-bit integers, and 64-bit floating points, called here `char`, `int32`, `int64`, and `flt64`, respectively. Data is not necessarily aligned on a word or double-word boundary: the long variable name record (see Section 1.12 [Long Variable Names Record], page 17) and very long string records (see Section 1.13 [Very Long String Record], page 17) have arbitrary byte length and can therefore cause all data coming after them in the file to be misaligned.

Integer data in system files may be big-endian or little-endian. A reader may detect the endianness of a system file by examining `layout_code` in the file header record (see [`layout_code`], page 4).

Floating-point data in system files may nominally be in IEEE 754, IBM, or VAX formats. A reader may detect the floating-point format in use by examining `bias` in the file header record (see [`bias`], page 5).

PSPP detects big-endian and little-endian integer formats in system files and translates as necessary. PSPP also detects the floating-point format in use, as well as the endianness of IEEE 754 floating-point numbers, and translates as needed. However, only IEEE 754 numbers with the same endianness as integer data in the same file have actually been observed in system files, and it is likely that other formats are obsolete or were never used.

System files use a few floating point values for special purposes:

SYSMIS
: The system-missing value is represented by the largest possible negative number in the floating point format (`-DBL_MAX`).

HIGHEST
: HIGHEST is used as the high end of a missing value range with an unbounded maximum. It is represented by the largest possible positive number (`DBL_MAX`).

LOWEST
: LOWEST is used as the low end of a missing value range with an unbounded minimum. It was originally represented by the second-largest negative number (in IEEE 754 format, `0xffeffffffffffffe`). System files written by SPSS 21 and later instead use the largest negative number (`-DBL_MAX`), the same value as SYSMIS. This does not lead to ambiguity because LOWEST appears in system files only in missing value ranges, which never contain SYSMIS.

System files may use most character encodings based on an 8-bit unit. UTF-16 and UTF-32, based on wider units, appear to be unacceptable. `rec_type` in the file header record is sufficient to distinguish between ASCII and EBCDIC based encodings. The best way to determine the specific encoding in use is to consult the character encoding record (see Section 1.14 [Character Encoding Record], page 19), if present, and failing that the `character_code` in the machine integer info record (see Section 1.6 [Machine Integer Info Record], page 10). The same encoding should be used for the dictionary and the data in the file, although it is possible to artificially synthesize files that use different encodings (see Section 1.14 [Character Encoding Record], page 19).

1.1 System File Record Structure

System files are divided into records with the following format:

```
int32               type;
char                data[];
```

This header does not identify the length of the `data` or any information about what it contains, so the system file reader must understand the format of `data` based on `type`. However, records with type 7, called *extension records*, have a stricter format:

```
int32               type;
int32               subtype;
int32               size;
int32               count;
char                data[size * count];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. This value identifies a particular kind of extension record.

`int32 size;`
: The size of each piece of data that follows the header, in bytes. Known extension records use 1, 4, or 8, for `char`, `int32`, and `flt64` format data, respectively.

`int32 count;`
: The number of pieces of data that follow the header.

`char data[size * count];`
: Data, whose format and interpretation depend on the subtype.

An extension record contains exactly `size * count` bytes of data, which allows a reader that does not understand an extension record to skip it. Extension records provide only nonessential information, so this allows for files written by newer software to preserve backward compatibility with older or less capable readers.

Records in a system file must appear in the following order:

- File header record.
- Variable records.
- All pairs of value labels records and value label variables records, if present.
- Document record, if present.
- Extension (type 7) records, in ascending numerical order of their subtypes.

  System files written by SPSS include at most one of each kind of extension record. This is generally true of system files written by other software as well, with known exceptions noted below in the individual sections about each type of record.
- Dictionary termination record.
- Data record.

We advise authors of programs that read system files to tolerate format variations. Various kinds of misformatting and corruption have been observed in system files written by SPSS and other software alike. In particular, because extension records provide nonessential information, it is generally better to ignore an extension record entirely than to refuse to read a system file.

The following sections describe the known kinds of records.

1.2 File Header Record

A system file begins with the file header, with the following format:

```
char                rec_type[4];
char                prod_name[60];
int32               layout_code;
int32               nominal_case_size;
int32               compression;
int32               weight_index;
int32               ncases;
flt64               bias;
char                creation_date[9];
char                creation_time[8];
char                file_label[64];
char                padding[3];
```

`char rec_type[4];`

Record type code, either '`$FL2`' for system files with uncompressed data or data compressed with simple bytecode compression, or '`$FL3`' for system files with ZLIB compressed data.

This is truly a character field that uses the character encoding as other strings. Thus, in a file with an ASCII-based character encoding this field contains `24 46 4c 32` or `24 46 4c 33`, and in a file with an EBCDIC-based encoding this field contains `5b c6 d3 f2`. (No EBCDIC-based ZLIB-compressed files have been observed.)

`char prod_name[60];`

Product identification string. This always begins with the characters '`@(#) SPSS DATA FILE`'. PSPP uses the remaining characters to give its version and the operating system name; for example, '`GNU pspp 0.1.4 - sparc-sun-solaris2.5.2`'. The string is truncated if it would be longer than 60 characters; otherwise it is padded on the right with spaces.

The product name field allow readers to behave differently based on quirks in the way that particular software writes system files. See Section 1.4 [Value Labels Records], page 9, for the detail of the quirk that the PSPP system file reader tolerates in files written by ReadStat, which has `https://github.com/WizardMac/ReadStat` in `prod_name`.

`int32 layout_code;`

Normally set to 2, although a few system files have been spotted in the wild with a value of 3 here. PSPP use this value to determine the file's integer endianness (see Chapter 1 [System File Format], page 2).

`int32 nominal_case_size;`
: Number of data elements per case. This is the number of variables, except that long string variables add extra data elements (one for every 8 characters after the first 8). However, string variables do not contribute to this value beyond the first 255 bytes. Further, some software always writes -1 or 0 in this field. In general, it is unsafe for systems reading system files to rely upon this value.

`int32 compression;`
: Set to 0 if the data in the file is not compressed, 1 if the data is compressed with simple bytecode compression, 2 if the data is ZLIB compressed. This field has value 2 if and only if `rec_type` is '`$FL3`'.

`int32 weight_index;`
: If one of the variables in the data set is used as a weighting variable, set to the dictionary index of that variable, plus 1 (see [Dictionary Index], page 6). Otherwise, set to 0.

`int32 ncases;`
: Set to the number of cases in the file if it is known, or -1 otherwise.

  In the general case it is not possible to determine the number of cases that will be output to a system file at the time that the header is written. The way that this is dealt with is by writing the entire system file, including the header, then seeking back to the beginning of the file and writing just the `ncases` field. For files in which this is not valid, the seek operation fails. In this case, `ncases` remains -1.

`flt64 bias;`
: Compression bias, ordinarily set to 100. Only integers between `1 - bias` and `251 - bias` can be compressed.

  By assuming that its value is 100, PSPP uses `bias` to determine the file's floating-point format and endianness (see Chapter 1 [System File Format], page 2). If the compression bias is not 100, PSPP cannot auto-detect the floating-point format and assumes that it is IEEE 754 format with the same endianness as the system file's integers, which is correct for all known system files.

`char creation_date[9];`
: Date of creation of the system file, in '`dd mmm yy`' format, with the month as standard English abbreviations, using an initial capital letter and following with lowercase. If the date is not available then this field is arbitrarily set to '`01 Jan 70`'.

`char creation_time[8];`
: Time of creation of the system file, in '`hh:mm:ss`' format and using 24-hour time. If the time is not available then this field is arbitrarily set to '`00:00:00`'.

`char file_label[64];`
: File label declared by the user, if any (see Section "FILE LABEL" in *PSPP Users Guide*). Padded on the right with spaces.

  A product that identifies itself as `VOXCO INTERVIEWER 4.3` uses CR-only line ends in this field, rather than the more usual LF-only or CR LF line ends.

`char padding[3];`
: Ignored padding bytes to make the structure a multiple of 32 bits in length. Set to zeros.

1.3 Variable Record

There must be one variable record for each numeric variable and each string variable with width 8 bytes or less. String variables wider than 8 bytes have one variable record for each 8 bytes, rounding up. The first variable record for a long string specifies the variable's correct dictionary information. Subsequent variable records for a long string are filled with dummy information: a type of -1, no variable label or missing values, print and write formats that are ignored, and an empty string as name. A few system files have been encountered that include a variable label on dummy variable records, so readers should take care to parse dummy variable records in the same way as other variable records.

The *dictionary index* of a variable is a 1-based offset in the set of variable records, including dummy variable records for long string variables. The first variable record has a dictionary index of 1, the second has a dictionary index of 2, and so on.

The system file format does not directly support string variables wider than 255 bytes. Such very long string variables are represented by a number of narrower string variables. See Section 1.13 [Very Long String Record], page 17, for details.

A system file should contain at least one variable and thus at least one variable record, but system files have been observed in the wild without any variables (thus, no data either).

```
int32               rec_type;
int32               type;
int32               has_var_label;
int32               n_missing_values;
int32               print;
int32               write;
char                name[8];

/* Present only if has_var_label is 1. */
int32               label_len;
char                label[];

/* Present only if n_missing_values is nonzero. */
flt64               missing_values[];
```

`int32 rec_type;`
: Record type code. Always set to 2.

`int32 type;`
: Variable type code. Set to 0 for a numeric variable. For a short string variable or the first part of a long string variable, this is set to the width of the string. For the second and subsequent parts of a long string variable, set to -1, and the remaining fields in the structure are ignored.

`int32 has_var_label;`
: If this variable has a variable label, set to 1; otherwise, set to 0.

`int32 n_missing_values;`

If the variable has no missing values, set to 0. If the variable has one, two, or three discrete missing values, set to 1, 2, or 3, respectively. If the variable has a range for missing variables, set to -2; if the variable has a range for missing variables plus a single discrete value, set to -3.

A long string variable always has the value 0 here. A separate record indicates missing values for long string variables (see Section 1.16 [Long String Missing Values Record], page 21).

`int32 print;`

Print format for this variable. See below.

`int32 write;`

Write format for this variable. See below.

`char name[8];`

Variable name. The variable name must begin with a capital letter or the at-sign ('`@`'). Subsequent characters may also be digits, octothorpes ('`#`'), dollar signs ('`$`'), underscores ('`_`'), or full stops ('`.`'). The variable name is padded on the right with spaces.

The '`name`' fields should be unique within a system file. System files written by SPSS that contain very long string variables with similar names sometimes contain duplicate names that are later eliminated by resolving the very long string names (see Section 1.13 [Very Long String Record], page 17). PSPP handles duplicates by assigning them new, unique names.

`int32 label_len;`

This field is present only if `has_var_label` is set to 1. It is set to the length, in characters, of the variable label. The documented maximum length varies from 120 to 255 based on SPSS version, but some files have been seen with longer labels. PSPP accepts labels of any length.

`char label[];`

This field is present only if `has_var_label` is set to 1. It has length `label_len`, rounded up to the nearest multiple of 32 bits. The first `label_len` characters are the variable's variable label.

`flt64 missing_values[];`

This field is present only if `n_missing_values` is nonzero. It has the same number of 8-byte elements as the absolute value of `n_missing_values`. Each element is interpreted as a number for numeric variables (with HIGHEST and LOWEST indicated as described in the chapter introduction). For string variables of width less than 8 bytes, elements are right-padded with spaces; for string variables wider than 8 bytes, only the first 8 bytes of each missing value are specified, with the remainder implicitly all spaces.

For discrete missing values, each element represents one missing value. When a range is present, the first element denotes the minimum value in the range, and the second element denotes the maximum value in the range. When a range plus a value are present, the third element denotes the additional discrete missing value.

The `print` and `write` members of sysfile_variable are output formats coded into `int32` types. The least-significant byte of the `int32` represents the number of decimal places, and the next two bytes in order of increasing significance represent field width and format type, respectively. The most-significant byte is not used and should be set to zero.

Format types are defined as follows:

Value	**Meaning**
0	Not used.
1	`A`
2	`AHEX`
3	`COMMA`
4	`DOLLAR`
5	`F`
6	`IB`
7	`PIBHEX`
8	`P`
9	`PIB`
10	`PK`
11	`RB`
12	`RBHEX`
13	Not used.
14	Not used.
15	`Z`
16	`N`
17	`E`
18	Not used.
19	Not used.
20	`DATE`
21	`TIME`
22	`DATETIME`
23	`ADATE`
24	`JDATE`
25	`DTIME`
26	`WKDAY`
27	`MONTH`
28	`MOYR`
29	`QYR`
30	`WKYR`
31	`PCT`
32	`DOT`
33	`CCA`
34	`CCB`
35	`CCC`
36	`CCD`
37	`CCE`
38	`EDATE`
39	`SDATE`

40 `MTIME`
41 `YMDHMS`

A few system files have been observed in the wild with invalid `write` fields, in particular with value 0. Readers should probably treat invalid `print` or `write` fields as some default format.

1.4 Value Labels Records

The value label records documented in this section are used for numeric and short string variables only. Long string variables may have value labels, but their value labels are recorded using a different record type (see Section 1.15 [Long String Value Labels Record], page 19).

ReadStat (see Section 1.2 [File Header Record], page 4) writes value labels that label a single value more than once. In more detail, it emits value labels whose values are longer than string variables' widths, that are identical in the actual width of the variable, e.g. labels for values `ABC123` and `ABC456` for a string variable with width 3. For files written by this software, PSPP ignores such labels.

The value label record has the following format:

```
int32               rec_type;
int32               label_count;

/* Repeated n_label times. */
char                value[8];
char                label_len;
char                label[];
```

`int32 rec_type;`
: Record type. Always set to 3.

`int32 label_count;`
: Number of value labels present in this record.

The remaining fields are repeated `count` times. Each repetition specifies one value label.

`char value[8];`
: A numeric value or a short string value padded as necessary to 8 bytes in length. Its type and width cannot be determined until the following value label variables record (see below) is read.

`char label_len;`
: The label's length, in bytes. The documented maximum length varies from 60 to 120 based on SPSS version. PSPP supports value labels up to 255 bytes long.

`char label[];`
: `label_len` bytes of the actual label, followed by up to 7 bytes of padding to bring `label` and `label_len` together to a multiple of 8 bytes in length.

The value label record is always immediately followed by a value label variables record with the following format:

```
int32               rec_type;
```

```
int32               var_count;
int32               vars[];
```

`int32 rec_type;`
: Record type. Always set to 4.

`int32 var_count;`
: Number of variables that the associated value labels from the value label record are to be applied.

`int32 vars[];`
: A list of 1-based dictionary indexes of variables to which to apply the value labels (see [Dictionary Index], page 6). There are `var_count` elements.

  String variables wider than 8 bytes may not be specified in this list.

1.5 Document Record

The document record, if present, has the following format:

```
int32               rec_type;
int32               n_lines;
char                lines[][80];
```

`int32 rec_type;`
: Record type. Always set to 6.

`int32 n_lines;`
: Number of lines of documents present. This should be greater than zero, but ReadStats writes system files with zero `n_lines`.

`char lines[][80];`
: Document lines. The number of elements is defined by `n_lines`. Lines shorter than 80 characters are padded on the right with spaces.

1.6 Machine Integer Info Record

The integer info record, if present, has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Data. */
int32               version_major;
int32               version_minor;
int32               version_revision;
int32               machine_code;
int32               floating_point_rep;
int32               compression_code;
int32               endianness;
```

```
    int32               character_code;
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 3.

`int32 size;`
: Size of each piece of data in the data part, in bytes. Always set to 4.

`int32 count;`
: Number of pieces of data in the data part. Always set to 8.

`int32 version_major;`
: PSPP major version number. In version *x.y.z*, this is *x*.

`int32 version_minor;`
: PSPP minor version number. In version *x.y.z*, this is *y*.

`int32 version_revision;`
: PSPP version revision number. In version *x.y.z*, this is *z*.

`int32 machine_code;`
: Machine code. PSPP always set this field to value to -1, but other values may appear.

`int32 floating_point_rep;`
: Floating point representation code. For IEEE 754 systems this is 1. IBM 370 sets this to 2, and DEC VAX E to 3.

`int32 compression_code;`
: Compression code. Always set to 1, regardless of whether or how the file is compressed.

`int32 endianness;`
: Machine endianness. 1 indicates big-endian, 2 indicates little-endian.

`int32 character_code;`
: Character code. The following values have been actually observed in system files:

  1 EBCDIC.

  2 7-bit ASCII.

  1250 The `windows-1250` code page for Central European and Eastern European languages.

  1252 The `windows-1252` code page for Western European languages.

  28591 ISO 8859-1.

  65001 UTF-8.

  The following additional values are known to be defined:

  3 8-bit “ASCII”.

4 DEC Kanji.

Other Windows code page numbers are known to be generally valid.

Old versions of SPSS for Unix and Windows always wrote value 2 in this field, regardless of the encoding in use. Newer versions also write the character encoding as a string (see Section 1.14 [Character Encoding Record], page 19).

1.7 Machine Floating-Point Info Record

The floating-point info record, if present, has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Data. */
flt64               sysmis;
flt64               highest;
flt64               lowest;
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 4.

`int32 size;`
: Size of each piece of data in the data part, in bytes. Always set to 8.

`int32 count;`
: Number of pieces of data in the data part. Always set to 3.

`flt64 sysmis;`
`flt64 highest;`
`flt64 lowest;`
: The system missing value, the value used for HIGHEST in missing values, and the value used for LOWEST in missing values, respectively. See Chapter 1 [System File Format], page 2, for more information.

The SPSSWriter library in PHP, which identifies itself as `FOM SPSS 1.0.0` in the file header record `prod_name` field, writes unexpected values to these fields, but it uses the same values consistently throughout the rest of the file.

1.8 Multiple Response Sets Records

The system file format has two different types of records that represent multiple response sets (see Section "MRSETS" in *PSPP Users Guide*). The first type of record describes multiple response sets that can be understood by SPSS before version 14. The second type of record, with a closely related format, is used for multiple dichotomy sets that use the CATEGORYLABELS=COUNTEDVALUES feature added in version 14.

```
/* Header. */
```

```
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
char                mrsets[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Set to 7 for records that describe multiple response sets understood by SPSS before version 14, or to 19 for records that describe dichotomy sets that use the CATEGORYLABELS=COUNTEDVALUES feature added in version 14.

`int32 size;`
: The size of each element in the `mrsets` member. Always set to 1.

`int32 count;`
: The total number of bytes in `mrsets`.

`char mrsets[];`
: Zero or more line feeds (byte 0x0a), followed by a series of multiple response sets, each of which consists of the following:

- The set's name (an identifier that begins with '`$`'), in mixed upper and lower case.
- An equals sign ('`=`').
- '`C`' for a multiple category set, '`D`' for a multiple dichotomy set with CATEGORYLABELS=VARLABELS, or '`E`' for a multiple dichotomy set with CATEGORYLABELS=COUNTEDVALUES.
- For a multiple dichotomy set with CATEGORYLABELS=COUNTEDVALUES, a space, followed by a number expressed as decimal digits, followed by a space. If LABELSOURCE=VARLABEL was specified on MRSETS, then the number is 11; otherwise it is 1.[1]
- For either kind of multiple dichotomy set, the counted value, as a positive integer count specified as decimal digits, followed by a space, followed by as many string bytes as specified in the count. If the set contains numeric variables, the string consists of the counted integer value expressed as decimal digits. If the set contains string variables, the string contains the counted string value. Either way, the string may be padded on the right with spaces (older versions of SPSS seem to always pad to a width of 8 bytes; newer versions don't).
- A space.

[1] This part of the format may not be fully understood, because only a single example of each possibility has been examined.

- The multiple response set's label, using the same format as for the counted value for multiple dichotomy sets. A string of length 0 means that the set does not have a label. A string of length 0 is also written if LABELSOURCE=VARLABEL was specified.
- A space.
- The short names of the variables in the set, converted to lowercase, each separated from the previous by a single space.

  Even though a multiple response set must have at least two variables, some system files contain multiple response sets with no variables or one variable. The source and meaning of these multiple response sets is unknown. (Perhaps they arise from creating a multiple response set then deleting all the variables that it contains?)
- One line feed (byte 0x0a). Sometimes multiple, even hundreds, of line feeds are present.

Example: Given appropriate variable definitions, consider the following MRSETS command:

```
MRSETS /MCGROUP NAME=$a LABEL='my mcgroup' VARIABLES=a b c
       /MDGROUP NAME=$b VARIABLES=g e f d VALUE=55
       /MDGROUP NAME=$c LABEL='mdgroup #2' VARIABLES=h i j VALUE='Yes'
       /MDGROUP NAME=$d LABEL='third mdgroup' CATEGORYLABELS=COUNTEDVALUES
        VARIABLES=k l m VALUE=34
       /MDGROUP NAME=$e CATEGORYLABELS=COUNTEDVALUES LABELSOURCE=VARLABEL
        VARIABLES=n o p VALUE='choice'.
```

The above would generate the following multiple response set record of subtype 7:

```
$a=C 10 my mcgroup a b c
$b=D2 55 0  g e f d
$c=D3 Yes 10 mdgroup #2 h i j
```

It would also generate the following multiple response set record with subtype 19:

```
$d=E 1 2 34 13 third mdgroup k l m
$e=E 11 6 choice 0  n o p
```

1.9 Extra Product Info Record

This optional record appears to contain a text string that describes the program that wrote the file and the source of the data. (This is redundant with the file label and product info found in the file header record.)

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
```

```
char                info[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 10.

`int32 size;`
: The size of each element in the `info` member. Always set to 1.

`int32 count;`
: The total number of bytes in `info`.

`char info[];`
: A text string. A product that identifies itself as `VOXCO INTERVIEWER 4.3` uses CR-only line ends in this field, rather than the more usual LF-only or CR LF line ends.

1.10 Variable Display Parameter Record

The variable display parameter record, if present, has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Repeated count times. */
int32               measure;
int32               width;           /* Not always present. */
int32               alignment;
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 11.

`int32 size;`
: The size of `int32`. Always set to 4.

`int32 count;`
: The number of sets of variable display parameters (ordinarily the number of variables in the dictionary), times 2 or 3.

The remaining members are repeated `count` times, in the same order as the variable records. No element corresponds to variable records that continue long string variables. The meanings of these members are as follows:

`int32 measure;`
: The measurement level of the variable:

0 Unknown

1 Nominal

2 Ordinal

3 Scale

An "unknown" `measure` of 0 means that the variable was created in some way that doesn't make the measurement level clear, e.g. with a `COMPUTE` transformation. PSPP sets the measurement level the first time it reads the data using the rules documented in Section "Measurement Level" in *PSPP Users Guide*, so this should rarely appear.

`int32 width;`
: The width of the display column for the variable in characters.

  This field is present if *count* is 3 times the number of variables in the dictionary. It is omitted if *count* is 2 times the number of variables.

`int32 alignment;`
: The alignment of the variable for display purposes:

  0 Left aligned

  1 Right aligned

  2 Centre aligned

1.11 Variable Sets Record

The SPSS GUI offers users the ability to arrange variables in sets. Users may enable and disable sets individually, and the data editor and analysis dialog boxes only show enabled sets. Syntax does not use variable sets.

The variable sets record, if present, has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of text. */
char                text[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 5.

`int32 size;`
: Always set to 1.

`int32 count;`
: The total number of bytes in `text`.

`char text[];`

The variable sets, in a text-based format.

Each variable set occupies one line of text, each of which ends with a line feed (byte 0x0a), optionally preceded by a carriage return (byte 0x0d).

Each line begins with the name of the variable set, followed by an equals sign (‘`=`’) and a space (byte 0x20), followed by the long variable names of the members of the set, separated by spaces. A variable set may be empty, in which case the equals sign and the space following it are still present.

1.12 Long Variable Names Record

If present, the long variable names record has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
char                var_name_pairs[];
```

`int32 rec_type;`

Record type. Always set to 7.

`int32 subtype;`

Record subtype. Always set to 13.

`int32 size;`

The size of each element in the `var_name_pairs` member. Always set to 1.

`int32 count;`

The total number of bytes in `var_name_pairs`.

`char var_name_pairs[];`

A list of *key*–*value* tuples, where *key* is the name of a variable, and *value* is its long variable name. The *key* field is at most 8 bytes long and must match the name of a variable which appears in the variable record (see Section 1.3 [Variable Record], page 6). The *value* field is at most 64 bytes long. The *key* and *value* fields are separated by a ‘`=`’ byte. Each tuple is separated by a byte whose value is 09. There is no trailing separator following the last tuple. The total length is `count` bytes.

1.13 Very Long String Record

Old versions of SPSS limited string variables to a width of 255 bytes. For backward compatibility with these older versions, the system file format represents a string longer than 255 bytes, called a *very long string*, as a collection of strings no longer than 255 bytes each. The strings concatenated to make a very long string are called its *segments*; for consistency, variables other than very long strings are considered to have a single segment.

A very long string with a width of w has $n = (w + 251) / 252$ segments, that is, one segment for every 252 bytes of width, rounding up. It would be logical, then, for each of the segments except the last to have a width of 252 and the last segment to have the remainder, but this is not the case. In fact, each segment except the last has a width of 255 bytes. The last segment has width $w - (n - 1) * 252$; some versions of SPSS make it slightly wider, but not wide enough to make the last segment require another 8 bytes of data.

Data is packed tightly into segments of a very long string, 255 bytes per segment. Because 255 bytes of segment data are allocated for every 252 bytes of the very long string's width (approximately), some unused space is left over at the end of the allocated segments. Data in unused space is ignored.

Example: Consider a very long string of width 20,000. Such a very long string has 20,000 / 252 = 80 (rounding up) segments. The first 79 segments have width 255; the last segment has width 20,000 - 79 * 252 = 92 or slightly wider (up to 96 bytes, the next multiple of 8). The very long string's data is actually stored in the 19,890 bytes in the first 78 segments, plus the first 110 bytes of the 79th segment (19,890 + 110 = 20,000). The remaining 145 bytes of the 79th segment and all 92 bytes of the 80th segment are unused.

The very long string record explains how to stitch together segments to obtain very long string data. For each of the very long string variables in the dictionary, it specifies the name of its first segment's variable and the very long string variable's actual width. The remaining segments immediately follow the named variable in the system file's dictionary.

The very long string record, which is present only if the system file contains very long string variables, has the following format:

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
char                string_lengths[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 14.

`int32 size;`
: The size of each element in the `string_lengths` member. Always set to 1.

`int32 count;`
: The total number of bytes in `string_lengths`.

`char string_lengths[];`
: A list of *key–value* tuples, where *key* is the name of a variable, and *value* is its length. The *key* field is at most 8 bytes long and must match the name of a variable which appears in the variable record (see Section 1.3 [Variable Record], page 6). The *value* field is exactly 5 bytes long. It is a zero-padded, ASCII-encoded string that is the length of the variable. The *key* and *value*

fields are separated by a '=' byte. Tuples are delimited by a two-byte sequence {00, 09}. After the last tuple, there may be a single byte 00, or {00, 09}. The total length is `count` bytes.

1.14 Character Encoding Record

This record, if present, indicates the character encoding for string data, long variable names, variable labels, value labels and other strings in the file.

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
char                encoding[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 20.

`int32 size;`
: The size of each element in the `encoding` member. Always set to 1.

`int32 count;`
: The total number of bytes in `encoding`.

`char encoding[];`
: The name of the character encoding. Normally this will be an official IANA character set name or alias. See `http://www.iana.org/assignments/character-sets`. Character set names are not case-sensitive, but SPSS appears to write them in all-uppercase.

This record is not present in files generated by older software. See also the `character_code` field in the machine integer info record (see [character-code], page 11).

When the character encoding record and the machine integer info record are both present, all system files observed in practice indicate the same character encoding, e.g. 1252 as `character_code` and `windows-1252` as `encoding`, 65001 and `UTF-8`, etc.

If, for testing purposes, a file is crafted with different `character_code` and `encoding`, it seems that `character_code` controls the encoding for all strings in the system file before the dictionary termination record, including strings in data (e.g. string missing values), and `encoding` controls the encoding for strings following the dictionary termination record.

1.15 Long String Value Labels Record

This record, if present, specifies value labels for long string variables.

```
/* Header. */
int32               rec_type;
```

```
int32               subtype;
int32               size;
int32               count;

/* Repeated up to exactly count bytes. */
int32               var_name_len;
char                var_name[];
int32               var_width;
int32               n_labels;
long_string_label   labels[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 21.

`int32 size;`
: Always set to 1.

`int32 count;`
: The number of bytes following the header until the next header.

`int32 var_name_len;`
`char var_name[];`
: The number of bytes in the name of the variable that has long string value labels, plus the variable name itself, which consists of exactly `var_name_len` bytes. The variable name is not padded to any particular boundary, nor is it null-terminated.

`int32 var_width;`
: The width of the variable, in bytes, which will be between 9 and 32767.

`int32 n_labels;`
`long_string_label labels[];`
: The long string labels themselves. The `labels` array contains exactly `n_labels` elements, each of which has the following substructure:

```
int32               value_len;
char                value[];
int32               label_len;
char                label[];
```

`int32 value_len;`
`char value[];`
: The string value being labeled. `value_len` is the number of bytes in `value`; it is equal to `var_width`. The `value` array is not padded or null-terminated.

`int32 label_len;`
`char label[];`
: The label for the string value. `label_len`, which must be between 0 and 120, is the number of bytes in `label`. The `label` array is not padded or null-terminated.

1.16 Long String Missing Values Record

This record, if present, specifies missing values for long string variables.

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Repeated up to exactly count bytes. */
int32               var_name_len;
char                var_name[];
char                n_missing_values;
int32               value_len;
char                values[values_len * n_missing_values];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 22.

`int32 size;`
: Always set to 1.

`int32 count;`
: The number of bytes following the header until the next header.

`int32 var_name_len;`
`char var_name[];`
: The number of bytes in the name of the long string variable that has missing values, plus the variable name itself, which consists of exactly `var_name_len` bytes. The variable name is not padded to any particular boundary, nor is it null-terminated.

`char n_missing_values;`
: The number of missing values, either 1, 2, or 3. (This is, unusually, a single byte instead of a 32-bit number.)

`int32 value_len;`
: The length of each missing value string, in bytes. This value should be 8, because long string variables are at least 8 bytes wide (by definition), only the first 8 bytes of a long string variable's missing values are allowed to be non-spaces, and any spaces within the first 8 bytes are included in the missing value here.

`char values[values_len * n_missing_values]`
: The missing values themselves, without any padding or null terminators.

An earlier version of this document stated that `value_len` was repeated before each of the missing values, so that there was an extra `int32` value of 8 before each missing value after the first. Old versions of PSPP wrote data files in this format. Readers can tolerate this mistake, if they wish, by noticing and skipping the extra `int32` values, which wouldn't ordinarily occur in strings.

1.17 Data File and Variable Attributes Records

The data file and variable attributes records represent custom attributes for the system file or for individual variables in the system file, as defined on the DATAFILE ATTRIBUTE (see Section "DATAFILE ATTRIBUTE" in *PSPP Users Guide*) and VARIABLE ATTRIBUTE commands (see Section "VARIABLE ATTRIBUTE" in *PSPP Users Guide*), respectively.

```
/* Header. */
int32               rec_type;
int32               subtype;
int32               size;
int32               count;

/* Exactly count bytes of data. */
char                attributes[];
```

`int32 rec_type;`
: Record type. Always set to 7.

`int32 subtype;`
: Record subtype. Always set to 17 for a data file attribute record or to 18 for a variable attributes record.

`int32 size;`
: The size of each element in the `attributes` member. Always set to 1.

`int32 count;`
: The total number of bytes in `attributes`.

`char attributes[];`
: The attributes, in a text-based format.

In record subtype 17, this field contains a single attribute set. An attribute set is a sequence of one or more attributes concatenated together. Each attribute consists of a name, which has the same syntax as a variable name, followed by, inside parentheses, a sequence of one or more values. Each value consists of a string enclosed in single quotes (`'`) followed by a line feed (byte 0x0a). A value may contain single quote characters, which are not themselves escaped or quoted or required to be present in pairs. There is no apparent way to embed a line feed in a value. There is no distinction between an attribute with a single value and an attribute array with one element.

In record subtype 18, this field contains a sequence of one or more variable attribute sets. If more than one variable attribute set is present, each one after the first is delimited from the previous by `/`. Each variable attribute set consists of a long variable name, followed by `:`, followed by an attribute set with the same syntax as on record subtype 17.

System files written by `Stata 14.1/-savespss- 1.77 by S.Radyakin` may include multiple records with subtype 18, one per variable that has variable attributes.

The total length is `count` bytes.

Example

A system file produced with the following VARIABLE ATTRIBUTE commands in effect:

```
VARIABLE ATTRIBUTE VARIABLES=dummy ATTRIBUTE=fred[1]('23') fred[2]('34').
VARIABLE ATTRIBUTE VARIABLES=dummy ATTRIBUTE=bert('123').
```

will contain a variable attribute record with the following contents:

```
0000  07 00 00 00 12 00 00 00  01 00 00 00 22 00 00 00  |............"...|
0010  64 75 6d 6d 79 3a 66 72  65 64 28 27 32 33 27 0a  |dummy:fred('23'.|
0020  27 33 34 27 0a 29 62 65  72 74 28 27 31 32 33 27  |'34'.)bert('123'|
0030  0a 29                                             |.)              |
```

1.17.1 Variable Roles

A variable's role is represented as an attribute named `$@Role`. This attribute has a single element whose values and their meanings are:

`0` Input. This, the default, is the most common role.

`1` Output.

`2` Both.

`3` None.

`4` Partition.

`5` Split.

1.18 Extended Number of Cases Record

The file header record expresses the number of cases in the system file as an int32 (see Section 1.2 [File Header Record], page 4). This record allows the number of cases in the system file to be expressed as a 64-bit number.

```
int32               rec_type;
int32               subtype;
int32               size;
int32               count;
int64               unknown;
int64               ncases64;
```

`int32 rec_type;`
Record type. Always set to 7.

`int32 subtype;`
Record subtype. Always set to 16.

`int32 size;`
Size of each element. Always set to 8.

`int32 count;`
Number of pieces of data in the data part. Alway set to 2.

`int64 unknown;`
Meaning unknown. Always set to 1.

`int64 ncases64;`
Number of cases in the file as a 64-bit integer. Presumably this could be -1 to indicate that the number of cases is unknown, for the same reason as `ncases` in the file header record, but this has not been observed in the wild.

1.19 Other Informational Records

This chapter documents many specific types of extension records are documented here, but others are known to exist. PSPP ignores unknown extension records when reading system files.

The following extension record subtypes have also been observed, with the following believed meanings:

6 Date info, probably related to USE (according to Aapi Hämäläinen).

12 A UUID in the format described in RFC 4122. Only two examples observed, both written by SPSS 13, and in each case the UUID contained both upper and lower case.

24 XML that describes how data in the file should be displayed on-screen.

1.20 Dictionary Termination Record

The dictionary termination record separates all other records from the data records.

```
int32               rec_type;
int32               filler;
```

`int32 rec_type;`
Record type. Always set to 999.

`int32 filler;`
Ignored padding. Should be set to 0.

1.21 Data Record

The data record must follow all other records in the system file. Every system file must have a data record that specifies data for at least one case. The format of the data record varies depending on the value of `compression` in the file header record:

0: no compression
Data is arranged as a series of 8-byte elements. Each element corresponds to the variable declared in the respective variable record (see Section 1.3 [Variable Record], page 6). Numeric values are given in `flt64` format; string values are literal characters string, padded on the right when necessary to fill out 8-byte units.

1: bytecode compression
The first 8 bytes of the data record is divided into a series of 1-byte command codes. These codes have meanings as described below:

0
: Ignored. If the program writing the system file accumulates compressed data in blocks of fixed length, 0 bytes can be used to pad out extra bytes remaining at the end of a fixed-size block.

1 through 251
: A number with value *code* - *bias*, where *code* is the value of the compression code and *bias* is the variable `bias` from the file header. For example, code 105 with bias 100.0 (the normal value) indicates a numeric variable of value 5.

  A code of 0 (after subtracting the bias) in a string field encodes null bytes. This is unusual, since a string field normally encodes text data, but it exists in real system files.

252
: End of file. This code may or may not appear at the end of the data stream. PSPP always outputs this code but its use is not required.

253
: A numeric or string value that is not compressible. The value is stored in the 8 bytes following the current block of command bytes. If this value appears twice in a block of command bytes, then it indicates the second group of 8 bytes following the command bytes, and so on.

254
: An 8-byte string value that is all spaces.

255
: The system-missing value.

The end of the 8-byte group of bytecodes is followed by any 8-byte blocks of non-compressible values indicated by code 253. After that follows another 8-byte group of bytecodes, then those bytecodes' non-compressible values. The pattern repeats to the end of the file or a code with value 252.

2: ZLIB compression
: The data record consists of the following, in order:

  - ZLIB data header, 24 bytes long.
  - One or more variable-length blocks of ZLIB compressed data.
  - ZLIB data trailer, with a 24-byte fixed header plus an additional 24 bytes for each preceding ZLIB compressed data block.

  The ZLIB data header has the following format:

```
int64               zheader_ofs;
int64               ztrailer_ofs;
int64               ztrailer_len;
```

`int64 zheader_ofs;`
: The offset, in bytes, of the beginning of this structure within the system file.

`int64 ztrailer_ofs;`
: The offset, in bytes, of the first byte of the ZLIB data trailer.

`int64 ztrailer_len;`
: The number of bytes in the ZLIB data trailer. This and the previous field sum to the size of the system file in bytes.

The data header is followed by `(ztrailer_len - 24) / 24` ZLIB compressed data blocks. Each ZLIB compressed data block begins with a ZLIB header as specified in RFC 1950, e.g. hex bytes `78 01` (the only header yet observed in practice). Each block decompresses to a fixed number of bytes (in practice only `0x3ff000`-byte blocks have been observed), except that the last block of data may be shorter. The last ZLIB compressed data block gends just before offset `ztrailer_ofs`.

The result of ZLIB decompression is bytecode compressed data as described above for compression format 1.

The ZLIB data trailer begins with the following 24-byte fixed header:

```
int64               bias;
int64               zero;
int32               block_size;
int32               n_blocks;
```

`int64 int_bias;`
: The compression bias as a negative integer, e.g. if `bias` in the file header record is 100.0, then `int_bias` is −100 (this is the only value yet observed in practice).

`int64 zero;`
: Always observed to be zero.

`int32 block_size;`
: The number of bytes in each ZLIB compressed data block, except possibly the last, following decompression. Only `0x3ff000` has been observed so far.

`int32 n_blocks;`
: The number of ZLIB compressed data blocks, always exactly `(ztrailer_len - 24) / 24`.

The fixed header is followed by `n_blocks` 24-byte ZLIB data block descriptors, each of which describes the compressed data block corresponding to its offset. Each block descriptor has the following format:

```
int64               uncompressed_ofs;
int64               compressed_ofs;
int32               uncompressed_size;
int32               compressed_size;
```

`int64 uncompressed_ofs;`
: The offset, in bytes, that this block of data would have in a similar system file that uses compression format 1. This is `zheader_ofs` in the first block descriptor, and in each succeeding block descriptor it is the sum of the previous desciptor's `uncompressed_ofs` and `uncompressed_size`.

`int64 compressed_ofs;`
: The offset, in bytes, of the actual beginning of this compressed data block. This is `zheader_ofs + 24` in the first block descriptor,

and in each succeeding block descriptor it is the sum of the previous descriptor's `compressed_ofs` and `compressed_size`. The final block descriptor's `compressed_ofs` and `compressed_size` sum to `ztrailer_ofs`.

`int32 uncompressed_size;`
: The number of bytes in this data block, after decompression. This is `block_size` in every data block except the last, which may be smaller.

`int32 compressed_size;`
: The number of bytes in this data block, as stored compressed in this system file.

2 SPSS Viewer File Format

SPSS Viewer or `.spv` files, here called SPV files, are written by SPSS 16 and later to represent the contents of its output editor. This chapter documents the format, based on examination of a corpus of about 8,000 files from a variety of sources. This description is detailed enough to both read and write SPV files.

SPSS 15 and earlier versions instead use `.spo` files, which have a completely different output format based on the Microsoft Compound Document Format. This format is not documented here.

An SPV file is a Zip archive that can be read with `zipinfo` and `unzip` and similar programs. The final member in the Zip archive is the *manifest*, a file named `META-INF/MANIFEST.MF`. This structure makes SPV files resemble Java "JAR" files (and ODF files), but whereas a JAR manifest contains a sequence of colon-delimited key/value pairs, an SPV manifest contains the string '`allowPivoting=true`', without a new-line. PSPP uses this string to identify an SPV file; it is invariant across the corpus.[1][2]

The rest of the members in an SPV file's Zip archive fall into two categories: *structure* and *detail* members. Structure member names take the form with `outputViewer`*`number`*`.xml` or `outputViewer`*`number`*`_heading.xml`, where *number* is an 10-digit decimal number. Each of these members represents some kind of output item (a table, a heading, a block of text, etc.) or a group of them. The member whose output goes at the beginning of the document is numbered 0, the next member in the output is numbered 1, and so on.

Structure members contain XML. This XML is sometimes self-contained, but it often references detail members in the Zip archive, which are named as follows:

`prefix``_table.xml` and *`prefix`*`_tableData.bin`
`prefix``_lightTableData.bin`
: The structure of a table plus its data. Older SPV files pair a *`prefix`*`_table.xml` file that describes the table's structure with a binary *`prefix`*`_tableData.bin` file that gives its data. Newer SPV files (the majority of those in the corpus) instead include a single *`prefix`*`_lightTableData.bin` file that incorporates both into a single binary format.

`prefix``_warning.xml` and *`prefix`*`_warningData.bin`
`prefix``_lightWarningData.bin`
: Same format used for tables, with a different name.

`prefix``_notes.xml` and *`prefix`*`_notesData.bin`
`prefix``_lightNotesData.bin`
: Same format used for tables, with a different name.

`prefix``_chartData.bin` and *`prefix`*`_chart.xml`
: The structure of a chart plus its data. Charts do not have a "light" format.

---

[1] SPV files always begin with the 7-byte sequence 50 4b 03 04 14 00 08, but this is not a useful magic number because most Zip archives start the same way.

[2] SPSS writes `META-INF/MANIFEST.MF` to every SPV file, but it does not read it or even require it to exist, so using different contents, e.g. as '`allowingPivot=false`' has no effect.

*prefix*_Imagegeneric.png
*prefix*_PastedObjectgeneric.png
*prefix*_imageData.bin
: A PNG image referenced by an `object` element (in the first two cases) or an `image` element (in the final case). See Section 2.1.9 [SPV Structure object and image Elements], page 37.

*prefix*_pmml.scf
*prefix*_stats.scf
*prefix*_model.xml
: Not yet investigated. The corpus contains few examples.

The *prefix* in the names of the detail members is typically an 11-digit decimal number that increases for each item, tending to skip values. Older SPV files use different naming conventions for detail members. Structure member refer to detail members by name, and so their exact names do not matter to readers as long as they are unique.

SPSS tolerates corrupted Zip archives that Zip reader libraries tend to reject. These can be fixed up with `zip -FF`.

2.1 Structure Member Format

A structure member lays out the high-level structure for a group of output items such as heading, tables, and charts. Structure members do not include the details of tables and charts but instead refer to them by their member names.

Structure members' XML files claim conformance with a collection of XML Schemas. These schemas are distributed, under a nonfree license, with SPSS binaries. Fortunately, the schemas are not necessary to understand the structure members. The schemas can even be deceptive because they document elements and attributes that are not in the corpus and do not document elements and attributes that are commonly found in the corpus.

Structure members use a different XML namespace for each schema, but these namespaces are not entirely consistent. In some SPV files, for example, the `viewer-tree` schema is associated with namespace '`http://xml.spss.com/spss/viewer-tree`' and in others with '`http://xml.spss.com/spss/viewer/viewer-tree`' (note the additional `viewer/`). Under either name, the schema URIs are not resolvable to obtain the schemas themselves.

One may ignore all of the above in interpreting a structure member. The actual XML has a simple and straightforward form that does not require a reader to take schemas or namespaces into account. A structure member's root is `heading` element, which contains `heading` or `container` elements (or a mix), forming a tree. In turn, `container` holds a `label` and one more child, usually `text` or `table`.

The following sections document the elements found in structure members in a context-free grammar-like fashion. Consider the following example, which specifies the attributes and content for the `container` element:

```
container
   :visibility=(visible | hidden)
   :page-break-before=(always)?
   :text-align=(left | center)?
   :width=dimension
```

```
=> label (table | container_text | graph | model | object | image | tree)
```

Each attribute specification begins with ':' followed by the attribute's name. If the attribute's value has an easily specified form, then '=' and its description follows the name. Finally, if the attribute is optional, the specification ends with '?'. The following value specifications are defined:

`(a | b | ...)`
: One of the listed literal strings. If only one string is listed, it is the only acceptable value. If `OTHER` is listed, then any string not explicitly listed is also accepted.

`bool`
: Either `true` or `false`.

`dimension`
: A floating-point number followed by a unit, e.g. `10pt`. Units in the corpus include `in` (inch), `pt` (points, 72/inch), `px` ("device-independent pixels", 96/inch), and `cm`. If the unit is omitted then points should be assumed. The number and unit may be separated by white space.

  The corpus also includes localized names for units. A reader must understand these to properly interpret the dimension:

  inch: , `pol.`, `cala`, `cali`

  point

  centimeter

`real`
: A floating-point number.

`int`
: An integer.

`color`
: A color in one of the forms `#`*`rrggbb`* or *`rrggbb`*, or the string `transparent`, or one of the standard Web color names.

`ref`

`ref` *`element`*
`ref(`*`elem1`* `|` *`elem2`* `| ...)`
: The name from the `id` attribute in some element. If one or more elements are named, the name must refer to one of those elements, otherwise any element is acceptable.

All elements have an optional `id` attribute. If present, its value must be unique. In practice many elements are assigned `id` attributes that are never referenced.

The content specification for an element supports the following syntax:

`element`
: An element.

`a` *`b`*
: *a* followed by *b*.

`a |` *`b`* `|` *`c`*
: One of *a* or *b* or *c*.

`a?`
: Zero or one instances of *a*.

`a*`
: Zero or more instances of *a*.

b+ One or more instances of *a*.

(*subexpression*)
Grouping for a subexpression.

EMPTY No content.

TEXT Text and CDATA.

Element and attribute names are sometimes suffixed by another name in square brackets to distinguish different uses of the same name. For example, structure XML has two `text` elements, one inside `container`, the other inside `pageParagraph`. The former is defined as `text[container_text]` and referenced as `container_text`, the latter defined as `text[pageParagraph_text]` and referenced as `pageParagraph_text`.

This language is used in the PSPP source code for parsing structure and detail XML members. Refer to `src/output/spv/structure-xml.grammar` and `src/output/spv/detail-xml.grammar` for the full grammars.

The following example shows the contents of a typical structure member for a DESCRIPTIVES procedure. A real structure member is not indented. This example also omits most attributes, all XML namespace information, and the CSS from the embedded HTML:

```
<?xml version="1.0" encoding="utf-8"?>
<heading>
  <label>Output</label>
  <heading commandName="Descriptives">
    <label>Descriptives</label>
    <container>
      <label>Title</label>
      <text commandName="Descriptives" type="title">
        <html lang="en">
<![CDATA[<head><style type="text/css">...</style></head><BR>Descriptives]]>
        </html>
      </text>
    </container>
    <container visibility="hidden">
      <label>Notes</label>
      <table commandName="Descriptives" subType="Notes" type="note">
        <tableStructure>
          <dataPath>00000000001_lightNotesData.bin</dataPath>
        </tableStructure>
      </table>
    </container>
    <container>
      <label>Descriptive Statistics</label>
      <table commandName="Descriptives" subType="Descriptive Statistics"
             type="table">
        <tableStructure>
          <dataPath>00000000002_lightTableData.bin</dataPath>
        </tableStructure>
```

```
      </table>
    </container>
  </heading>
</heading>
```

2.1.1 The heading Element

```
heading[root_heading]
   :creator-version?
   :creator?
   :creation-date-time?
   :lockReader=bool?
   :schemaLocation?
=> label pageSetup? (container | heading)*

heading
   :creator-version?
   :commandName?
   :visibility[heading_visibility]=(collapsed)?
   :locale?
   :olang?
=> label (container | heading)*
```

A `heading` represents a tree of content that appears in an output viewer window. It contains a `label` text string that is shown in the outline view ordinarily followed by content containers or further nested (sub)-sections of output. Unlike heading elements in HTML and other common document formats, which precede the content that they head, `heading` contains the elements that appear below the heading.

The root of a structure member is a special `heading`. The direct children of the root `heading` elements in all structure members in an SPV file are siblings. That is, the root `heading` in all of the structure members conceptually represent the same node. The root heading's `label` is ignored (see see Section 2.1.2 [SPV Structure label Element], page 33). The root heading in the first structure member in the Zip file may contain a `pageSetup` element.

The schema implies that any `heading` may contain a sequence of any number of `heading` and `container` elements. This does not work for the root `heading` in practice, which must actually contain exactly one `container` or `heading` child element. Furthermore, if the root heading's child is a `heading`, then the structure member's name must end in `_heading.xml`; if it is a `container` child, then it must not.

The following attributes have been observed on both document root and nested `heading` elements.

`creator-version` [Attribute]

The version of the software that created this SPV file. A string of the form `xxyyzzww` represents software version xx.yy.zz.ww, e.g. `21000001` is version 21.0.0.1. Trailing pairs of zeros are sometimes omitted, so that `21`, `210000`, and `21000000` are all version 21.0.0.0 (and the corpus contains all three of those forms).

The following attributes have been observed on document root `heading` elements only:

`creator` [Attribute]
: The directory in the file system of the software that created this SPV file.

`creation-date-time` [Attribute]
: The date and time at which the SPV file was written, in a locale-specific format, e.g. `Friday, May 16, 2014 6:47:37 PM PDT` or `lunedì 17 marzo 2014 3.15.48 CET` or even `Friday, December 5, 2014 5:00:19 o'clock PM EST`.

`lockReader` [Attribute]
: Whether a reader should be allowed to edit the output. The possible values are `true` and `false`. The value `false` is by far the most common.

`schemaLocation` [Attribute]
: This is actually an XML Namespace attribute. A reader may ignore it.

The following attributes have been observed only on nested `heading` elements:

`commandName` [Attribute]
: A locale-invariant identifier for the command that produced the output, e.g. `Frequencies`, `T-Test`, `Non Par Corr`.

`visibility` [Attribute]
: If this attribute is absent, the heading's content is expanded in the outline view. If it is set to `collapsed`, it is collapsed. (This attribute is never present in a root `heading` because the root node is always expanded when a file is loaded, even though the UI can be used to collapse it interactively.)

`locale` [Attribute]
: The locale used for output, in Windows format, which is similar to the format used in Unix with the underscore replaced by a hyphen, e.g. `en-US`, `en-GB`, `el-GR`, `sr-Cryl-RS`.

`olang` [Attribute]
: The output language, e.g. `en`, `it`, `es`, `de`, `pt-BR`.

2.1.2 The `label` Element

```
label => TEXT
```

Every `heading` and `container` holds a `label` as its first child. The label text is what appears in the outline pane of the GUI's viewer window. PSPP also puts it into the outline of PDF output. The label text doesn't appear in the output itself.

The text in `label` describes what it labels, often by naming the statistical procedure that was executed, e.g. "Frequencies" or "T-Test". Labels are often very generic, especially within a `container`, e.g. "Title" or "Warnings" or "Notes". Label text is localized according to the output language, e.g. in Italian a frequency table procedure is labeled "Frequenze".

The user can edit labels to be anything they want. The corpus contains a few examples of empty labels, ones that contain no text, probably as a result of user editing.

The root `heading` in an SPV file has a `label`, like every `heading`. It normally contains "Output" but its content is disregarded anyway. The user cannot edit it.

2.1.3 The container Element

```
container
   :visibility=(visible | hidden)
   :page-break-before=(always)?
   :text-align=(left | center)?
   :width=dimension
=> label (table | container_text | graph | model | object | image | tree)
```

A `container` serves to contain and label a `table`, `text`, or other kind of item.

This element has the following attributes.

`visibility` [Attribute]
: Whether the container's content is displayed. "Notes" tables are often hidden; other data is usually visible.

`text-align` [Attribute]
: Alignment of text within the container. Observed with nested `table` and `text` elements.

`width` [Attribute]
: The width of the container, e.g. `1097px`.

All of the elements that nest inside `container` (except the `label`) have the following optional attribute.

`commandName` [Attribute]
: As on the `heading` element. The corpus contains one example of where `commandName` is present but set to the empty string.

2.1.4 The text Element (Inside container)

```
text[container_text]
  :type[text_type]=(title | log | text | page-title)
  :commandName?
  :creator-version?
=> html
```

This `text` element is nested inside a `container`. There is a different `text` element that is nested inside a `pageParagraph`.

This element has the following attributes.

`commandName` [Attribute]
: See Section 2.1.3 [SPV Structure container Element], page 34. For output not specific to a command, this is simply `log`.

`type` [Attribute]
: The semantics of the text.

`creator-version` [Attribute]
: As on the `heading` element.

2.1.5 The `html` Element

```
html :lang=(en) => TEXT
```

The element contains an HTML document as text (or, in practice, as CDATA). In some cases, the document starts with `<html>` and ends with `</html>`; in others the `html` element is implied. Generally the HTML includes a `head` element with a CSS stylesheet. The HTML body often begins with `<BR>`.

The HTML document uses only the following elements:

`html` Sometimes, the document is enclosed with `<html>`...`</html>`.

`br` The HTML body often begins with `<BR>` and may contain it as well.

`b`
`i`
`u` Styling.

`font` The attributes `face`, `color`, and `size` are observed. The value of `color` takes one of the forms `#rrggbb` or `rgb (r, g, b)`. The value of `size` is a number between 1 and 7, inclusive.

The CSS in the corpus is simple. To understand it, a parser only needs to be able to skip white space, `<!--`, and `-->`, and parse style only for `p` elements. Only the following properties matter:

`color` In the form *rrggbb*, e.g. `000000`, with no leading '`#`'.

`font-weight`
Either `bold` or `normal`.

`font-style`
Either `italic` or `normal`.

`text-decoration`
Either `underline` or `normal`.

`font-family`
A font name, commonly `Monospaced` or `SansSerif`.

`font-size`
Values claim to be in points, e.g. `14pt`, but the values are actually in "device-independent pixels" (px), at 96/inch.

This element has the following attributes.

`lang` [Attribute]
This always contains `en` in the corpus.

2.1.6 The `table` Element

```
table
   :VDPId?
   :ViZmlSource?
   :activePageId=int?
   :commandName
```

```
    :creator-version?
    :displayFiltering=bool?
    :maxNumCells=int?
    :orphanTolerance=int?
    :rowBreakNumber=int?
    :subType
    :tableId
    :tableLookId?
    :type[table_type]=(table | note | warning)
=> tableProperties? tableStructure

tableStructure => path? dataPath csvPath?
```

This element has the following attributes.

`commandName` [Attribute]
: See Section 2.1.3 [SPV Structure container Element], page 34.

`type` [Attribute]
: One of `table`, `note`, or `warning`.

`subType` [Attribute]
: The locale-invariant command ID for the particular kind of output that this table represents in the procedure. This can be the same as `commandName` e.g. `Frequencies`, or different, e.g. `Case Processing Summary`. Generic subtypes `Notes` and `Warnings` are often used.

`tableId` [Attribute]
: A number that uniquely identifies the table within the SPV file, typically a large negative number such as `-4147135649387905023`.

`creator-version` [Attribute]
: As on the `heading` element. In the corpus, this is only present for version 21 and up and always includes all 8 digits.

See Section 2.4.14 [SPV Detail Legacy Properties], page 81, for details on the `tableProperties` element.

2.1.7 The graph Element

```
graph
    :VDPId?
    :ViZmlSource?
    :commandName?
    :creator-version?
    :dataMapId?
    :dataMapURI?
    :editor?
    :refMapId?
    :refMapURI?
```

```
    :csvFileIds?
    :csvFileNames?
=> dataPath? path csvPath?
```

This element represents a graph. The `dataPath` and `path` elements name the Zip members that give the details of the graph. Normally, both elements are present; there is only one counterexample in the corpus.

`csvPath` only appears in one SPV file in the corpus, for two graphs. In these two cases, `dataPath`, `path`, and `csvPath` all appear. These `csvPath` name Zip members with names of the form `number_csv.bin`, where *number* is a many-digit number and the same as the `csvFileIds`. The named Zip members are CSV text files (despite the `.bin` extension). The CSV files are encoded in UTF-8 and begin with a U+FEFF byte-order marker.

2.1.8 The `model` Element

```
model
    :PMMLContainerId?
    :PMMLId
    :StatXMLContainerId
    :VDPId
    :auxiliaryViewName
    :commandName
    :creator-version
    :mainViewName
=> ViZml? dataPath? path | pmmlContainerPath statsContainerPath

pmmlContainerPath => TEXT

statsContainerPath => TEXT

ViZml :viewName? => TEXT
```

This element represents a model. The `dataPath` and `path` elements name the Zip members that give the details of the model. Normally, both elements are present; there is only one counterexample in the corpus.

The details are unexplored. The `ViZml` element contains base-64 encoded text, that decodes to a binary format with some embedded text strings, and `path` names an Zip member that contains XML. Alternatively, `pmmlContainerPath` and `statsContainerPath` name Zip members with `.scf` extension.

2.1.9 The `object` and `image` Elements

```
object
    :commandName?
    :type[object_type]=(unknown)?
    :uri
=> EMPTY

image
    :commandName?
```

```
        :VDPId
    => dataPath
```

These two elements represent an image in PNG format. They are equivalent and the corpus contains examples of both. The only difference is the syntax: for `object`, the `uri` attribute names the Zip member that contains a PNG file; for `image`, the text of the inner `dataPath` element names the Zip member.

PSPP writes `object` in output but there is no strong reason to choose this form.

The corpus only contains PNG image files.

2.1.10 The `tree` Element

```
tree
   :commandName
   :creator-version
   :name
   :type
=> dataPath path
```

This element represents a tree. The `dataPath` and `path` elements name the Zip members that give the details of the tree. The details are unexplored.

2.1.11 Path Elements

```
dataPath => TEXT

path => TEXT

csvPath => TEXT
```

These element contain the name of the Zip members that hold details for a container. For tables:

- When a “light” format is used, only `dataPath` is present, and it names a `.bin` member of the Zip file that has `light` in its name, e.g. `0000000001437_lightTableData.bin` (see Section 2.2 [SPV Light Detail Member Format], page 40).
- When the legacy format is used, both are present. In this case, `dataPath` names a Zip member with a legacy binary format that contains relevant data (see Section 2.3 [SPV Legacy Detail Member Binary Format], page 56), and `path` names a Zip member that uses an XML format (see Section 2.4 [SPV Legacy Detail Member XML Format], page 58).

Graphs normally follow the legacy approach described above. The corpus contains one example of a graph with `path` but not `dataPath`. The reason is unexplored.

Models use `path` but not `dataPath`. See Section 2.1.7 [SPV Structure graph Element], page 36, for more information.

These elements have no attributes.

2.1.12 The `pageSetup` Element

```
pageSetup
   :initial-page-number=int?
```

```
    :chart-size=(as-is | full-height | half-height | quarter-height | OTHER)?
    :margin-left=dimension?
    :margin-right=dimension?
    :margin-top=dimension?
    :margin-bottom=dimension?
    :paper-height=dimension?
    :paper-width=dimension?
    :reference-orientation?
    :space-after=dimension?
=> pageHeader pageFooter

pageHeader => pageParagraph?

pageFooter => pageParagraph?

pageParagraph => pageParagraph_text
```

The `pageSetup` element has the following attributes.

`initial-page-number` [Attribute]
: The page number to put on the first page of printed output. Usually `1`.

`chart-size` [Attribute]
: One of the listed, self-explanatory chart sizes, `quarter-height`, or a localization (!) of one of these (e.g. `dimensione attuale`, `Wie vorgegeben`).

`margin-left` [Attribute]
`margin-right` [Attribute]
`margin-top` [Attribute]
`margin-bottom` [Attribute]
: Margin sizes, e.g. `0.25in`.

`paper-height` [Attribute]
`paper-width` [Attribute]
: Paper sizes.

`reference-orientation` [Attribute]
: Indicates the orientation of the output page. Either `0deg` (portrait) or `90deg` (landscape),

`space-after` [Attribute]
: The amount of space between printed objects, typically `12pt`.

2.1.13 The `text` Element (Inside `pageParagraph`)

```
text[pageParagraph_text] :type=(title | text) => TEXT
```

This `text` element is nested inside a `pageParagraph`. There is a different `text` element that is nested inside a `container`.

The element is either empty, or contains CDATA that holds almost-XHTML text: in the corpus, either an `html` or `p` element. It is *almost*-XHTML because the `html` element designates the default namespace as '`http://xml.spss.com/spss/viewer/viewer-tree`' instead of an XHTML namespace, and because the CDATA can contain substitution variables. The following variables are supported:

`&[Date]`
`&[Time]` The current date or time in the preferred format for the locale.

`&[Head1]`
`&[Head2]`
`&[Head3]`
`&[Head4]` First-, second-, third-, or fourth-level heading.

`&[PageTitle]`
The page title.

`&[Filename]`
Name of the output file.

`&[Page]` The page number.

`&[Page]` for the page number and `&[PageTitle]` for the page title.

Typical contents (indented for clarity):

```
<html xmlns="http://xml.spss.com/spss/viewer/viewer-tree">
    <head></head>
    <body>
        <p style="text-align:right; margin-top: 0">Page &[Page]</p>
    </body>
</html>
```

This element has the following attributes.

`type` [Attribute]
Always `text`.

2.2 Light Detail Member Format

This section describes the format of "light" detail `.bin` members. These members have a binary format which we describe here in terms of a context-free grammar using the following conventions:

NonTerminal ⇒ . . .
Nonterminals have CamelCaps names, and ⇒ indicates a production. The right-hand side of a production is often broken across multiple lines. Break points are chosen for aesthetics only and have no semantic significance.

00, 01, . . . , ff.
A bytes with a fixed value, written as a pair of hexadecimal digits.

i0, i1, . . . , i9, i10, i11, . . .
ib0, ib1, . . . , ib9, ib10, ib11, . . .
A 32-bit integer in little-endian or big-endian byte order, respectively, with a fixed value, written in decimal. Prefixed by '`i`' for little-endian or '`ib`' for big-endian.

byte — A byte.

bool — A byte with value 0 or 1.

int16
be16 — A 16-bit unsigned integer in little-endian or big-endian byte order, respectively.

int32
be32 — A 32-bit unsigned integer in little-endian or big-endian byte order, respectively.

int64
be64 — A 64-bit unsigned integer in little-endian or big-endian byte order, respectively.

double — A 64-bit IEEE floating-point number.

float — A 32-bit IEEE floating-point number.

string
bestring — A 32-bit unsigned integer, in little-endian or big-endian byte order, respectively, followed by the specified number of bytes of character data. (The encoding is indicated by the Formats nonterminal.)

x? — *x* is optional, e.g. 00? is an optional zero byte.

*x***n* — *x* is repeated *n* times, e.g. byte*10 for ten arbitrary bytes.

x[*name*] — Gives *x* the specified *name*. Names are used in textual explanations. They are also used, also bracketed, to indicate counts, e.g. `int32[n] byte*[n]` for a 32-bit integer followed by the specified number of arbitrary bytes.

a | *b* — Either *a* or *b*.

(*x*) — Parentheses are used for grouping to make precedence clear, especially in the presence of |, e.g. in 00 (01 | 02 | 03) 00.

count(*x*)
becount(*x*) — A 32-bit unsigned integer, in little-endian or big-endian byte order, respectively, that indicates the number of bytes in *x*, followed by *x* itself.

v1(*x*) — In a version 1 `.bin` member, *x*; in version 3, nothing. (The `.bin` header indicates the version.)

v3(*x*) — In a version 3 `.bin` member, *x*; in version 1, nothing.

PSPP uses this grammar to parse light detail members. See `src/output/spv/light-binary.grammar` in the PSPP source tree for the full grammar.

Little-endian byte order is far more common in this format, but a few pieces of the format use big-endian byte order.

Light detail members express linear units in two ways: points (pt), at 72/inch, and "device-independent pixels" (px), at 96/inch. To convert from pt to px, multiply by 1.33 and round up. To convert from px to pt, divide by 1.33 and round down.

A "light" detail member `.bin` consists of a number of sections concatenated together, terminated by an optional byte 01:

```
Table =>
```

```
Header Titles Footnotes
Areas Borders PrintSettings TableSettings Formats
Dimensions Axes Cells
01?
```

The following sections go into more detail.

2.2.1 Header

An SPV light member begins with a 39-byte header:

```
Header =>
    01 00
    (i1 | i3)[version]
    bool[x0]
    bool[x1]
    bool[rotate-inner-column-labels]
    bool[rotate-outer-row-labels]
    bool[x2]
    int32[x3]
    int32[min-col-heading-width] int32[max-col-heading-width]
    int32[min-row-heading-width] int32[max-row-heading-width]
    int64[table-id]
```

`version` is a version number that affects the interpretation of some of the other data in the member. We will refer to "version 1" and "version 3" later on and use v1(...) and v3(...) for version-specific formatting (as described previously).

If `rotate-inner-column-labels` is 1, then column labels closest to the data are rotated 90° counterclockwise; otherwise, they are shown in the normal way.

If `rotate-outer-row-labels` is 1, then row labels farthest from the data are rotated 90° counterclockwise; otherwise, they are shown in the normal way.

`min-col-heading-width`, `max-col-heading-width`, `min-row-heading-width`, and `max-row-heading-width` are measurements in 1/96 inch units (called "device independent pixel" units in Windows) whose values influence column widths. For the purpose of interpreting these values, a table is divided into the three regions shown below:

```
+------------------+-------------------------------------------------+
|                  |                 column headings                 |
|                  +-------------------------------------------------+
|      corner      |                                                 |
|       and        |                                                 |
|   row headings   |                      data                       |
|                  |                                                 |
|                  |                                                 |
+------------------+-------------------------------------------------+
```

`min-col-heading-width` and `max-col-heading-width` apply to the columns in the column headings region. `min-col-heading-width` is the minimum width that any of these columns will be given automatically. In addition, `max-col-heading-width` is the maximum width that a column will be assigned to accommodate a long label in the column headings cells. These columns will still be made wider to accommodate wide data values in the data region.

`min-row-heading-width` is the minimum width that a column in the corner and row headings region will be given automatically. `max-row-heading-width` is the maximum width that a column in this region will be assigned to accomodate a long label. This region doesn't include data, so data values don't affect column widths.

`table-id` is a binary version of the `tableId` attribute in the structure member that refers to the detail member. For example, if `tableId` is `-4122591256483201023`, then `table-id` would be 0xc6c99d183b300001.

The meaning of the other variable parts of the header is not known. A writer may safely use version 3, true for `x0`, false for `x1`, true for `x2`, and 0x15 for `x3`.

2.2.2 Titles

```
Titles =>
    Value[title] 01?
    Value[subtype] 01? 31
    Value[user-title] 01?
    (31 Value[corner-text] | 58)
    (31 Value[caption] | 58)
```

The Titles follow the Header and specify the table's title, caption, and corner text.

The `user-title` reflects any user editing of the title text or style. The `title` is the title originally generated by the procedure. Both of these are appropriate for presentation and localized to the user's language. For example, for a frequency table, `title` and `user-title` normally name the variable and `c` is simply "Frequencies".

`subtype` is the same as the `subType` attribute in the `table` structure XML element that referred to this member. See Section 2.1.6 [SPV Structure table Element], page 35, for details.

The `corner-text`, if present, is shown in the upper-left corner of the table, above the row headings and to the left of the column headings. It is usually absent. When row dimension labels are displayed in the corner (see `show-row-labels-in-corner`), corner text is hidden.

The `caption`, if present, is shown below the table. `caption` reflects user editing of the caption.

2.2.3 Footnotes

```
Footnotes => int32[n-footnotes] Footnote*[n-footnotes]
Footnote => Value[text] (58 | 31 Value[marker]) int32[show]
```

Each footnote has `text` and an optional custom `marker` (such as '`*`').

The syntax for Value would allow footnotes (and their markers) to reference other footnotes, but in practice this doesn't work.

`show` is a 32-bit signed integer. It is positive to show the footnote or negative to hide it. Its magnitude is often 1, and in other cases tends to be the number of references to the footnote. It is safe to write 1 to show a footnote and -1 to hide it.

2.2.4 Areas

```
Areas => 00? Area*8
Area =>
    byte[index] 31
    string[typeface] float[size] int32[style] bool[underline]
    int32[halign] int32[valign]
    string[fg-color] string[bg-color]
    bool[alternate] string[alt-fg-color] string[alt-bg-color]
    v3(int32[left-margin] int32[right-margin] int32[top-margin] int32[bottom-margin])
```

Each Area represents the style for a different area of the table, in the following order: title, caption, footer, corner, column labels, row labels, data, and layers.

`index` is the 1-based index of the Area, i.e. 1 for the first Area, through 8 for the final Area.

`typeface` is the string name of the font used in the area. In the corpus, this is `SansSerif` in over 99% of instances and `Times New Roman` in the rest.

`size` is the size of the font, in px (see Section 2.2 [SPV Light Detail Member Format], page 40). The most common size in the corpus is 12 px. Even though `size` has a floating-point type, in the corpus its values are always integers.

`style` is a bit mask. Bit 0 (with value 1) is set for bold, bit 1 (with value 2) is set for italic.

`underline` is 1 if the font is underlined, 0 otherwise.

`halign` specifies horizontal alignment: 0 for center, 2 for left, 4 for right, 61453 for decimal, 64173 for mixed. Mixed alignment varies according to type: string data is left-justified, numbers and most other formats are right-justified.

`valign` specifies vertical alignment: 0 for center, 1 for top, 3 for bottom.

`fg-color` and `bg-color` are the foreground color and background color, respectively. In the corpus, these are always `#000000` and `#ffffff`, respectively.

`alternate` is 1 if rows should alternate colors, 0 if all rows should be the same color. When `alternate` is 1, `alt-fg-color` and `alt-bg-color` specify the colors for the alternate rows; otherwise they are empty strings.

`left-margin`, `right-margin`, `top-margin`, and `bottom-margin` are measured in px.

2.2.5 Borders

```
Borders =>
    count(
        ib1[endian]
        be32[n-borders] Border*[n-borders]
        bool[show-grid-lines]
        00 00 00)

Border =>
    be32[border-type]
    be32[stroke-type]
    be32[color]
```

The Borders reflect how borders between regions are drawn.

The fixed value of `endian` can be used to validate the endianness.

`show-grid-lines` is 1 to draw grid lines, otherwise 0.

Each Border describes one kind of border. `n-borders` seems to always be 19. Each `border-type` appears once (although in an unpredictable order) and correspond to the following borders:

0 Title.

1...4 Left, top, right, and bottom outer frame.

5...8 Left, top, right, and bottom inner frame.

9, 10 Left and top of data area.

11, 12 Horizontal and vertical dimension rows.

13, 14 Horizontal and vertical dimension columns.

15, 16 Horizontal and vertical category rows.

17, 18 Horizontal and vertical category columns.

`stroke-type` describes how a border is drawn, as one of:

0 No line.

1 Solid line.

2 Dashed line.

3 Thick line.

4 Thin line.

5 Double line.

`color` is an RGB color. Bits 24–31 are alpha, bits 16–23 are red, 8–15 are green, 0–7 are blue. An alpha of 255 indicates an opaque color, therefore opaque black is 0xff000000.

2.2.6 Print Settings

```
PrintSettings =>
    count(
        ib1[endian]
        bool[all-layers]
        bool[paginate-layers]
        bool[fit-width]
        bool[fit-length]
        bool[top-continuation]
        bool[bottom-continuation]
        be32[n-orphan-lines]
        bestring[continuation-string])
```

The PrintSettings reflect settings for printing. The fixed value of `endian` can be used to validate the endianness.

`all-layers` is 1 to print all layers, 0 to print only the layer designated by `current-layer` in TableSettings (see Section 2.2.7 [SPV Light Member Table Settings], page 46).

`paginate-layers` is 1 to print each layer at the start of a new page, 0 otherwise. (This setting is honored only `all-layers` is 1, since otherwise only one layer is printed.)

`fit-width` and `fit-length` control whether the table is shrunk to fit within a page's width or length, respectively.

`n-orphan-lines` is the minimum number of rows or columns to put in one part of a table that is broken across pages.

If `top-continuation` is 1, then `continuation-string` is printed at the top of a page when a table is broken across pages for printing; similarly for `bottom-continuation` and the bottom of a page. Usually, `continuation-string` is empty.

2.2.7 Table Settings

```
TableSettings =>
    count(
      v3(
        ib1[endian]
        be32[x5]
        be32[current-layer]
        bool[omit-empty]
        bool[show-row-labels-in-corner]
        bool[show-alphabetic-markers]
        bool[footnote-marker-superscripts]
        byte[x6]
        becount(
          Breakpoints[row-breaks] Breakpoints[column-breaks]
          Keeps[row-keeps] Keeps[column-keeps]
          PointKeeps[row-point-keeps] PointKeeps[column-point-keeps]
        )
        bestring[notes]
        bestring[table-look]
        )...)

Breakpoints => be32[n-breaks] be32*[n-breaks]

Keeps => be32[n-keeps] Keep*[n-keeps]
Keep => be32[offset] be32[n]

PointKeeps => be32[n-point-keeps] PointKeep*[n-point-keeps]
PointKeep => be32[offset] be32 be32
```

The TableSettings reflect display settings. The fixed value of `endian` can be used to validate the endianness.

`current-layer` is the displayed layer. Suppose there are d layers, numbered 1 through d in the order given in the Dimensions (see Section 2.2.9 [SPV Light Member Dimensions], page 50), and that the displayed value of dimension i is d_i, $0 \leq x_i < n_i$, where n_i is the number of categories in dimension i. Then `current-layer` is calculated by the following algorithm:

> let `current-layer` = 0
> for each i from d downto 1:
>   `current-layer` = (n_i × `current-layer`) + x_i

If `omit-empty` is 1, empty rows or columns (ones with nothing in any cell) are hidden; otherwise, they are shown.

If `show-row-labels-in-corner` is 1, then row labels are shown in the upper left corner; otherwise, they are shown nested.

If `show-alphabetic-markers` is 1, markers are shown as letters (e.g. '`a`', '`b`', '`c`', ...); otherwise, they are shown as numbers starting from 1.

When `footnote-marker-superscripts` is 1, footnote markers are shown as superscripts, otherwise as subscripts.

The Breakpoints are rows or columns after which there is a page break; for example, a row break of 1 requests a page break after the second row. Usually no breakpoints are specified, indicating that page breaks should be selected automatically.

The Keeps are ranges of rows or columns to be kept together without a page break; for example, a row Keep with `offset` 1 and `n` 10 requests that the 10 rows starting with the second row be kept together. Usually no Keeps are specified.

The PointKeeps seem to be generated automatically based on user-specified Keeps. They seems to indicate a conversion from rows or columns to pixel or point offsets.

`notes` is a text string that contains user-specified notes. It is displayed when the user hovers the cursor over the table, like text in the `title` attribute in HTML. It is not printed. It is usually empty.

`table-look` is the name of a SPSS "TableLook" table style, such as "Default" or "Academic"; it is often empty.

TableSettings ends with an arbitrary number of null bytes. A writer may safely write 82 null bytes.

A writer may safely use 4 for `x5` and 0 for `x6`.

2.2.8 Formats

```
Formats =>
    int32[n-widths] int32*[n-widths]
    string[locale]
    int32[current-layer]
    bool[x7] bool[x8] bool[x9]
    Y0
    CustomCurrency
    count(
      v1(X0?)
      v3(count(X1 count(X2)) count(X3)))
Y0 => int32[epoch] byte[decimal] byte[grouping]
CustomCurrency => int32[n-ccs] string*[n-ccs]
```

If `n-widths` is nonzero, then the accompanying integers are column widths as manually adjusted by the user.

`locale` is a locale including an encoding, such as `en_US.windows-1252` or `it_IT.windows-1252`. (`locale` is often duplicated in Y1, described below).

`epoch` is the year that starts the epoch. A 2-digit year is interpreted as belonging to the 100 years beginning at the epoch. The default epoch year is 69 years prior to the current year; thus, in 2017 this field by default contains 1948. In the corpus, `epoch` ranges from 1943 to 1948, plus some contain -1.

`decimal` is the decimal point character. The observed values are '`.`' and '`,`'.

`grouping` is the grouping character. Usually, it is '`,`' if `decimal` is '`.`', and vice versa. Other observed values are '`'`' (apostrophe), '` `' (space), and zero (presumably indicating that digits should not be grouped).

`n-ccs` is observed as either 0 or 5. When it is 5, the following strings are CCA through CCE format strings. See Section "Custom Currency Formats" in *PSPP*. Most commonly these are all `-,,,` but other strings occur.

A writer may safely use false for `x7`, `x8`, and `x9`.

X0

X0 only appears, optionally, in version 1 members.

```
X0 => byte*14 Y1 Y2
Y1 =>
    string[command] string[command-local]
    string[language] string[charset] string[locale]
    bool[x10] bool[include-leading-zero] bool[x12] bool[x13]
    Y0
Y2 => CustomCurrency byte[missing] bool[x17]
```

`command` describes the statistical procedure that generated the output, in English. It is not necessarily the literal syntax name of the procedure: for example, NPAR TESTS becomes "Nonparametric Tests." `command-local` is the procedure's name, translated into the output language; it is often empty and, when it is not, sometimes the same as `command`.

`include-leading-zero` is the `LEADZERO` setting for the table, where false is `OFF` (the default) and true is `ON`. See Section "SET LEADZERO" in *PSPP*.

`missing` is the character used to indicate that a cell contains a missing value. It is always observed as '`.`'.

A writer may safely use false for `x10` and `x17` and true for `x12` and `x13`.

X1

X1 only appears in version 3 members.

```
X1 =>
    bool[x14]
    byte[show-title]
    bool[x16]
    byte[lang]
    byte[show-variables]
```

```
        byte[show-values]
        int32[x18] int32[x19]
        00*17
        bool[x20]
        bool[show-caption]
```

lang may indicate the language in use. Some values seem to be 0: en, 1: de, 2: es, 3: it, 5: ko, 6: pl, 8: zh-tw, 10: pt_BR, 11: fr.

show-variables determines how variables are displayed by default. A value of 1 means to display variable names, 2 to display variable labels when available, 3 to display both (name followed by label, separated by a space). The most common value is 0, which probably means to use a global default.

show-values is a similar setting for values. A value of 1 means to display the value, 2 to display the value label when available, 3 to display both. Again, the most common value is 0, which probably means to use a global default.

show-title is 1 to show the caption, 10 to hide it.

show-caption is true to show the caption, false to hide it.

A writer may safely use false for x14, false for x16, 0 for lang, -1 for x18 and x19, and false for x20.

X2

X2 only appears in version 3 members.

```
X2 =>
    int32[n-row-heights] int32*[n-row-heights]
    int32[n-style-map] StyleMap*[n-style-map]
    int32[n-styles] StylePair*[n-styles]
    count((i0 i0)?)
StyleMap => int64[cell-index] int16[style-index]
```

If present, n-row-heights and the accompanying integers are row heights as manually adjusted by the user.

The rest of X2 specifies styles for data cells. At first glance this is odd, because each data cell can have its own style embedded as part of the data, but in practice X2 specifies a style for a cell only if that cell is empty (and thus does not appear in the data at all). Each StyleMap specifies the index of a blank cell, calculated the same was as in the Cells (see Section 2.2.12 [SPV Light Member Cells], page 52), along with a 0-based index into the accompanying StylePair array.

A writer may safely omit the optional i0 i0 inside the count(...).

X3

X3 only appears in version 3 members.

```
X3 =>
    01 00 byte[x21] 00 00 00
    Y1
    double[small] 01
    (string[dataset] string[datafile] i0 int32[date] i0)?
```

```
Y2
(int32[x22] i0 01?)?
```

`small` is a small real number. In the corpus, it overwhelmingly takes the value 0.0001, with zero occasionally seen. Nonzero numbers with format 40 (see Section 2.2.13 [SPV Light Member Value], page 53) whose magnitudes are smaller than displayed in scientific notation. (Thus, a `small` of zero prevents scientific notation from being chosen.)

`dataset` is the name of the dataset analyzed to produce the output, e.g. `DataSet1`, and `datafile` the name of the file it was read from, e.g. `C:\Users\foo\bar.sav`. The latter is sometimes the empty string.

`date` is a date, as seconds since the epoch, i.e. since January 1, 1970. Pivot tables within an SPV file often have dates a few minutes apart, so this is probably a creation date for the table rather than for the file.

Sometimes `dataset`, `datafile`, and `date` are present and other times they are absent. The reader can distinguish by assuming that they are present and then checking whether the presumptive `dataset` contains a null byte (a valid string never will).

`x22` is usually 0 or 2000000.

A writer may safely use 4 for `x21` and omit `x22` and the other optional bytes at the end.

Encoding

Formats contains several indications of character encoding:

- `locale` in Formats itself.
- `locale` in Y1 (in version 1, Y1 is optionally nested inside X0; in version 3, Y1 is nested inside X3).
- `charset` in version 3, in Y1.
- `lang` in X1, in version 3.

`charset`, if present, is a good indication of character encoding, and in its absence the encoding suffix on `locale` in Formats will work.

`locale` in Y1 can be disregarded: it is normally the same as `locale` in Formats, and it is only present if `charset` is also.

`lang` is not helpful and should be ignored for character encoding purposes.

However, the corpus contains many examples of light members whose strings are encoded in UTF-8 despite declaring some other character set. Furthermore, the corpus contains several examples of light members in which some strings are encoded in UTF-8 (and contain multibyte characters) and other strings are encoded in another character set (and contain non-ASCII characters). PSPP treats any valid UTF-8 string as UTF-8 and only falls back to the declared encoding for strings that are not valid UTF-8.

The `pspp-output` program's `strings` command can help analyze the encoding in an SPV light member. Use `pspp-output --help-dev` to see its usage.

2.2.9 Dimensions

A pivot table presents multidimensional data. A Dimension identifies the categories associated with each dimension.

```
Dimensions => int32[n-dims] Dimension*[n-dims]
```

```
Dimension =>
    Value[name] DimProperties
    int32[n-categories] Category*[n-categories]
DimProperties =>
    byte[x1]
    byte[x2]
    int32[x3]
    bool[hide-dim-label]
    bool[hide-all-labels]
    01 int32[dim-index]
```

`name` is the name of the dimension, e.g. `Variables`, `Statistics`, or a variable name.

The meanings of `x1` and `x3` are unknown. `x1` is usually 0 but many other values have been observed. A writer may safely use 0 for `x1` and 2 for `x3`.

`x2` is 0, 1, or 2. For a pivot table with L layer dimensions, R row dimensions, and C column dimensions, `x2` is 2 for the first L dimensions, 0 for the next R dimensions, and 1 for the remaining C dimensions. This does not mean that the layer dimensions must be presented first, followed by the row dimensions, followed by the column dimensions—on the contrary, they are frequently in a different order—but `x2` must follow this pattern to prevent the pivot table from being misinterpreted.

If `hide-dim-label` is 00, the pivot table displays a label for the dimension itself. Because usually the group and category labels are enough explanation, it is usually 01.

If `hide-all-labels` is 01, the pivot table omits all labels for the dimension, including group and category labels. It is usually 00. When `hide-all-labels` is 01, `show-dim-label` is ignored.

`dim-index` is usually the 0-based index of the dimension, e.g. 0 for the first dimension, 1 for the second, and so on. Sometimes it is -1. There is no visible difference. A writer may safely use the 0-based index.

2.2.10 Categories

Categories are arranged in a tree. Only the leaf nodes in the tree are really categories; the others just serve as grouping constructs.

```
Category => Value[name] (Leaf | Group)
Leaf => 00 00 00 i2 int32[leaf-index] i0
Group =>
    bool[merge] 00 01 int32[x23]
    i-1 int32[n-subcategories] Category*[n-subcategories]
```

`name` is the name of the category (or group).

A Leaf represents a leaf category. The Leaf's `leaf-index` is a nonnegative integer unique within the Dimension and less than `n-categories` in the Dimension. If the user does not sort or rearrange the categories, then `leaf-index` starts at 0 for the first Leaf in the dimension and increments by 1 with each successive Leaf. If the user does sorts or rearrange the categories, then the order of categories in the file reflects that change and `leaf-index` reflects the original order.

A dimension can have no leaf categories at all. A table that contains such a dimension necessarily has no data at all.

A Group is a group of nested categories. Usually a Group contains at least one Category, so that `n-subcategories` is positive, but Groups with zero subcategories have been observed.

If a Group's `merge` is 00, the most common value, then the group is really a distinct group that should be represented as such in the visual representation and user interface. If `merge` is 01, the categories in this group should be shown and treated as if they were direct children of the group's containing group (or if it has no parent group, then direct children of the dimension), and this group's name is irrelevant and should not be displayed. (Merged groups can be nested!)

Writers need not use merged groups.

A Group's `x23` appears to be i2 when all of the categories within a group are leaf categories that directly represent data values for a variable (e.g. in a frequency table or crosstabulation, a group of values in a variable being tabulated) and i0 otherwise. A writer may safely write a constant 0 in this field.

2.2.11 Axes

After the dimensions come assignment of each dimension to one of the axes: layers, rows, and columns.

```
Axes =>
    int32[n-layers] int32[n-rows] int32[n-columns]
    int32*[n-layers] int32*[n-rows] int32*[n-columns]
```

The values of `n-layers`, `n-rows`, and `n-columns` each specifies the number of dimensions displayed in layers, rows, and columns, respectively. Any of them may be zero. Their values sum to `n-dimensions` from Dimensions (see Section 2.2.9 [SPV Light Member Dimensions], page 50).

The following `n-dimensions` integers, in three groups, are a permutation of the 0-based dimension numbers. The first `n-layers` integers specify each of the dimensions represented by layers, the next `n-rows` integers specify the dimensions represented by rows, and the final `n-columns` integers specify the dimensions represented by columns. When there is more than one dimension of a given kind, the inner dimensions are given first. (For the layer axis, this means that the first dimension is at the bottom of the list and the last dimension is at the top when the current layer is displayed.)

2.2.12 Cells

The final part of an SPV light member contains the actual data.

```
Cells => int32[n-cells] Cell*[n-cells]
Cell => int64[index] v1(00?) Value
```

A Cell consists of an `index` and a Value. Suppose there are d dimensions, numbered 1 through d in the order given in the Dimensions previously, and that dimension i has n_i categories. Consider the cell at coordinates x_i, $1 \le i \le d$, and note that $0 \le x_i < n_i$. Then the index is calculated by the following algorithm:

let $index = 0$
for each i from 1 to d:
  $index = (n_i \times index) + x_i$

For example, suppose there are 3 dimensions with 3, 4, and 5 categories, respectively. The cell at coordinates (1, 2, 3) has index $5 \times (4 \times (3 \times 0 + 1) + 2) + 3 = 33$. Within a given dimension, the index is the `leaf-index` in a Leaf.

2.2.13 Value

Value is used throughout the SPV light member format. It boils down to a number or a string.

```
Value => 00? 00? 00? 00? RawValue
RawValue =>
    01 ValueMod int32[format] double[x]
  | 02 ValueMod int32[format] double[x]
    string[var-name] string[value-label] byte[show]
  | 03 string[local] ValueMod string[id] string[c] bool[fixed]
  | 04 ValueMod int32[format] string[value-label] string[var-name]
    byte[show] string[s]
  | 05 ValueMod string[var-name] string[var-label] byte[show]
  | 06 string[local] ValueMod string[id] string[c]
  | ValueMod string[template] int32[n-args] Argument*[n-args]
Argument =>
    i0 Value
  | int32[x] i0 Value*[x]      /* x > 0 */
```

There are several possible encodings, which one can distinguish by the first nonzero byte in the encoding.

01
: The numeric value `x`, intended to be presented to the user formatted according to `format`, which is about the same as the format described for system files (see [System File Output Formats], page 8). The exception is that format 40 is not MTIME but instead approximately a synonym for F format with a different rule for whether a value is shown in scientific notation: a value in format 40 is shown in scientific notation if and only if it is nonzero and its magnitude is less than `small` (see Section 2.2.8 [SPV Light Member Formats], page 47).

  Most commonly, `format` has width 40 (the maximum).

  An `x` with the maximum negative double value `-DBL_MAX` represents the system-missing value SYSMIS. (HIGHEST and LOWEST have not been observed.) See Chapter 1 [System File Format], page 2, for more about these special values.

02
: Similar to `01`, with the additional information that `x` is a value of variable `var-name` and has value label `value-label`. Both `var-name` and `value-label` can be the empty string, the latter very commonly.

  `show` determines whether to show the numeric value or the value label. A value of 1 means to show the value, 2 to show the label, 3 to show both, and 0 means to use the default specified in `show-values` (see Section 2.2.8 [SPV Light Member Formats], page 47).

03
: A text string, in two forms: `c` is in English, and sometimes abbreviated or obscure, and `local` is localized to the user's locale. In an English-language locale, the two strings are often the same, and in the cases where they differ,

`local` is more appropriate for a user interface, e.g. `c` of "Not a PxP table for MCN..." versus `local` of "Computed only for a PxP table, where P must be greater than 1."

`c` and `local` are always either both empty or both nonempty.

`id` is a brief identifying string whose form seems to resemble a programming language identifier, e.g. `cumulative_percent` or `factor_14`. It is not unique.

`fixed` is 00 for text taken from user input, such as syntax fragment, expressions, file names, data set names, and 01 for fixed text strings such as names of procedures or statistics. In the former case, `id` is always the empty string; in the latter case, `id` is still sometimes empty.

04 The string value `s`, intended to be presented to the user formatted according to `format`. The format for a string is not too interesting, and the corpus contains many clearly invalid formats like A16.39 or A255.127 or A134.1, so readers should probably entirely disregard the format. PSPP only checks `format` to distinguish AHEX format.

`s` is a value of variable `var-name` and has value label `value-label`. `var-name` is never empty but `value-label` is commonly empty.

`show` has the same meaning as in the encoding for 02.

05 Variable `var-name` with variable label `var-label`. In the corpus, `var-name` is rarely empty and `var-label` is often empty.

`show` determines whether to show the variable name or the variable label. A value of 1 means to show the name, 2 to show the label, 3 to show both, and 0 means to use the default specified in `show-variables` (see Section 2.2.8 [SPV Light Member Formats], page 47).

06 Similar to type 03, with `fixed` assumed to be true.

otherwise When the first byte of a RawValue is not one of the above, the RawValue starts with a ValueMod, whose syntax is described in the next section. (A ValueMod always begins with byte 31 or 58.)

This case is a template string, analogous to `printf`, followed by one or more Arguments, each of which has one or more values. The template string is copied directly into the output except for the following special syntax,

`\%`
`\:`
`\[`
`\]` Each of these expands to the character following '\\', to escape characters that have special meaning in template strings. These are effective inside and outside the `[...]` syntax forms described below.

`\n` Expands to a new-line, inside or outside the `[...]` forms described below.

`^i` Expands to a formatted version of argument *i*, which must have only a single value. For example, `^1` expands to the first argument's `value`.

`[:a:]i` Expands *a* for each of the values in *i*. *a* should contain one or more `^j` conversions, which are drawn from the values for argument *i* in order. Some examples from the corpus:

`[:^1:]1` All of the values for the first argument, concatenated.

`[:^1\n:]1`
Expands to the values for the first argument, each followed by a new-line.

`[:^1 = ^2:]2`
Expands to `x = y` where *x* is the second argument's first value and *y* is its second value. (This would be used only if the argument has two values. If there were more values, the second and third values would be directly concatenated, which would look funny.)

`[a:b:]i` This extends the previous form so that the first values are expanded using *a* and later values are expanded using *b*. For an unknown reason, within *a* the `^j` conversions are instead written as `%j`. Some examples from the corpus:

`[%1:*^1:]1`
Expands to all of the values for the first argument, separated by '`*`'.

`[%1 = %2:, ^1 = ^2:]1`
Given appropriate values for the first argument, expands to `X = 1, Y = 2, Z = 3`.

`[%1:, ^1:]1`
Given appropriate values, expands to `1, 2, 3`.

The template string is localized to the user's locale.

A writer may safely omit all of the optional 00 bytes at the beginning of a Value, except that it should write a single 00 byte before a templated Value.

2.2.14 ValueMod

A ValueMod can specify special modifications to a Value.

```
ValueMod =>
    58
  | 31
    int32[n-refs] int16*[n-refs]
    int32[n-subscripts] string*[n-subscripts]
    v1(00 (i1 | i2) 00? 00? int32 00? 00?)
    v3(count(TemplateString StylePair))

TemplateString => count((count((i0 (58 | 31 55))?) (58 | 31 string[id]))?)

StylePair =>
    (31 FontStyle | 58)
```

```
        (31 CellStyle | 58)

    FontStyle =>
        bool[bold] bool[italic] bool[underline] bool[show]
        string[fg-color] string[bg-color]
        string[typeface] byte[size]

    CellStyle =>
        int32[halign] int32[valign] double[decimal-offset]
        int16[left-margin] int16[right-margin]
        int16[top-margin] int16[bottom-margin]
```

A ValueMod that begins with "31" specifies special modifications to a Value.

Each of the `n-refs` integers is a reference to a Footnote (see Section 2.2.3 [SPV Light Member Footnotes], page 43) by 0-based index. Footnote markers are shown appended to the main text of the Value, as superscripts or subscripts.

The `subscripts`, if present, are strings to append to the main text of the Value, as subscripts. Each subscript text is a brief indicator, e.g. '`a`' or '`b`', with its meaning indicated by the table caption. When multiple subscripts are present, they are displayed separated by commas.

The `id` inside the TemplateString, if present, is a template string for substitutions using the syntax explained previously. It appears to be an English-language version of the localized template string in the Value in which the Template is nested. A writer may safely omit the optional fixed data in TemplateString.

FontStyle and CellStyle, if present, change the style for this individual Value. In FontStyle, `bold`, `italic`, and `underline` control the particular style. `show` is ordinarily 1; if it is 0, then the cell data is not shown. `fg-color` and `bg-color` are strings in the format `#rrggbb`, e.g. `#ff0000` for red or `#ffffff` for white. The empty string is occasionally observed also. The `size` is a font size in units of 1/128 inch.

In CellStyle, `halign` is 0 for center, 2 for left, 4 for right, 6 for decimal, 0xffffffad for mixed. For decimal alignment, `decimal-offset` is the decimal point's offset from the right side of the cell, in pt (see Section 2.2 [SPV Light Detail Member Format], page 40). `valign` specifies vertical alignment: 0 for center, 1 for top, 3 for bottom. `left-margin`, `right-margin`, `top-margin`, and `bottom-margin` are in pt.

2.3 Legacy Detail Member Binary Format

Whereas the light binary format represents everything about a given pivot table, the legacy binary format conceptually consists of a number of named sources, each of which consists of a number of named variables, each of which is a 1-dimensional array of numbers or strings or a mix. Thus, the legacy binary member format is quite simple.

This section uses the same context-free grammar notation as in the previous section, with the following additions:

vAF(*x*) In a version 0xaf legacy member, *x*; in other versions, nothing. (The legacy member header indicates the version; see below.)

vB0(*x*) In a version 0xb0 legacy member, *x*; in other versions, nothing.

A legacy detail member `.bin` has the following overall format:

```
LegacyBinary =>
    00 byte[version] int16[n-sources] int32[member-size]
    Metadata*[n-sources]
    #Data*[n-sources]
    #Strings?
```

`version` is a version number that affects the interpretation of some of the other data in the member. Versions 0xaf and 0xb0 are known. We will refer to "version 0xaf" and "version 0xb0" members later on.

A legacy member consists of `n-sources` data sources, each of which has Metadata and Data.

`member-size` is the size of the legacy binary member, in bytes.

The Data and Strings above are commented out because the Metadata has some oddities that mean that the Data sometimes seems to start at an unexpected place. The following section goes into detail.

2.3.1 Metadata

```
Metadata =>
    int32[n-values] int32[n-variables] int32[data-offset]
    vAF(byte*28[source-name])
    vB0(byte*64[source-name] int32[x])
```

A data source has `n-variables` variables, each with `n-values` data values.

`source-name` is a 28- or 64-byte string padded on the right with 0-bytes. The names that appear in the corpus are very generic: usually `tableData` for pivot table data or `source0` for chart data.

A given Metadata's `data-offset` is the offset, in bytes, from the beginning of the member to the start of the corresponding Data. This allows programs to skip to the beginning of the data for a particular source. In every case in the corpus, the Data follow the Metadata in the same order, but it is important to use `data-offset` instead of reading sequentially through the file because of the exception described below.

One SPV file in the corpus has legacy binary members with version 0xb0 but a 28-byte `source-name` field (and only a single source). In practice, this means that the 64-byte `source-name` used in version 0xb0 has a lot of 0-bytes in the middle followed by the `variable-name` of the following Data. As long as a reader treats the first 0-byte in the `source-name` as terminating the string, it can properly interpret these members.

The meaning of `x` in version 0xb0 is unknown.

2.3.2 Numeric Data

```
Data => Variable*[n-variables]
Variable => byte*288[variable-name] double*[n-values]
```

Data follow the Metadata in the legacy binary format, with sources in the same order (but readers should use the `data-offset` in Metadata records, rather than reading sequentially). Each Variable begins with a `variable-name` that generally indicates its role in the pivot table, e.g. "cell", "cellFormat", "dimension0categories", "dimension0group0", followed by the numeric data, one double per datum. A double with the maximum negative double `-DBL_MAX` represents the system-missing value SYSMIS.

2.3.3 String Data

```
Strings => SourceMaps[maps] Labels

SourceMaps => int32[n-maps] SourceMap*[n-maps]

SourceMap => string[source-name] int32[n-variables] VariableMap*[n-variables]
VariableMap => string[variable-name] int32[n-data] DatumMap*[n-data]
DatumMap => int32[value-idx] int32[label-idx]

Labels => int32[n-labels] Label*[n-labels]
Label => int32[frequency] string[label]
```

Each variable may include a mix of numeric and string data values. If a legacy binary member contains any string data, Strings is present; otherwise, it ends just after the last Data element.

The string data overlays the numeric data. When a variable includes any string data, its Variable represents the string values with a SYSMIS or NaN placeholder. (Not all such values need be placeholders.)

Each SourceMap provides a mapping between SYSMIS or NaN values in source `source-name` and the string data that they represent. `n-variables` is the number of variables in the source that include string data. More precisely, it is the 1-based index of the last variable in the source that includes any string data; thus, it would be 4 if there are 5 variables and only the fourth one includes string data.

A VariableMap repeats its variable's name, but variables are always present in the same order as the source, starting from the first variable, without skipping any even if they have no string values. Each VariableMap contains DatumMap nonterminals, each of which maps from a 0-based index within its variable's data to a 0-based label index, e.g. pair `value-idx` = 2, `label-idx` = 3, means that the third data value (which must be SYSMIS or NaN) is to be replaced by the string of the fourth Label.

The labels themselves follow the pairs. The valuable part of each label is the string `label`. Each label also includes a `frequency` that reports the number of DatumMaps that reference it (although this is not useful).

2.4 Legacy Detail Member XML Format

The design of the detail XML format is not what one would end up with for describing pivot tables. This is because it is a special case of a much more general format ("visualization XML" or "VizML") that can describe a wide range of visualizations. Most of this generality is overkill for tables, and so we end up with a funny subset of a general-purpose format.

An XML Schema for VizML is available, distributed with SPSS binaries, under a nonfree license. It contains documentation that is occasionally helpful.

This section describes the detail XML format using the same notation already used for the structure XML format (see Section 2.1 [SPV Structure Member Format], page 29). See src/output/spv/detail-xml.grammar in the PSPP source tree for the full grammar that it uses for parsing.

The important elements of the detail XML format are:

- Variables. See Section 2.4.2 [SPV Detail Variable Elements], page 61.
- Assignment of variables to axes. A variable can appear as columns, or rows, or layers. The faceting element and its sub-elements describe this assignment.
- Styles and other annotations.

This description is not detailed enough to write legacy tables. Instead, write tables in the light binary format.

2.4.1 The visualization Element

```
visualization
   :creator
   :date
   :lang
   :name
   :style[style_ref]=ref style
   :type
   :version
   :schemaLocation?
=> visualization_extension?
   userSource
   (sourceVariable | derivedVariable)+
   categoricalDomain?
   graph
   labelFrame[lf1]*
   container?
   labelFrame[lf2]*
   style+
   layerController?

extension[visualization_extension]
   :numRows=int?
   :showGridline=bool?
   :minWidthSet=(true)?
   :maxWidthSet=(true)?
=> EMPTY

userSource :missing=(listwise | pairwise)? => EMPTY

categoricalDomain => variableReference simpleSort

simpleSort :method[sort_method]=(custom) => categoryOrder

container :style=ref style => container_extension? location+ labelFrame*
```

```
extension[container_extension] :combinedFootnotes=(true) => EMPTY

layerController
   :source=(tableData)
   :target=ref label?
=> EMPTY
```

The `visualization` element is the root of detail XML member. It has the following attributes:

`creator` [Attribute]
: The version of the software that created this SPV file, as a string of the form `xxyyzz`, which represents software version xx.yy.zz, e.g. `160001` is version 16.0.1. The corpus includes major versions 16 through 19.

`date` [Attribute]
: The date on the which the file was created, as a string of the form `YYYY-MM-DD`.

`lang` [Attribute]
: The locale used for output, in Windows format, which is similar to the format used in Unix with the underscore replaced by a hyphen, e.g. `en-US`, `en-GB`, `el-GR`, `sr-Cryl-RS`.

`name` [Attribute]
: The title of the pivot table, localized to the output language.

`style` [Attribute]
: The base style for the pivot table. In every example in the corpus, the `style` element has no attributes other than `id`.

`type` [Attribute]
: A floating-point number. The meaning is unknown.

`version` [Attribute]
: The visualization schema version number. In the corpus, the value is one of 2.4, 2.5, 2.7, and 2.8.

The `userSource` element has no visible effect.

The `extension` element as a child of `visualization` has the following attributes.

`numRows` [Attribute]
: An integer that presumably defines the number of rows in the displayed pivot table.

`showGridline` [Attribute]
: Always set to `false` in the corpus.

`minWidthSet` [Attribute]
`maxWidthSet` [Attribute]
: Always set to `true` in the corpus.

The `extension` element as a child of `container` has the following attribute

`combinedFootnotes` [Attribute]
Meaning unknown.

The `categoricalDomain` and `simpleSort` elements have no visible effect.

The `layerController` element has no visible effect.

2.4.2 Variable Elements

A "variable" in detail XML is a 1-dimensional array of data. Each element of the array may, independently, have string or numeric content. All of the variables in a given detail XML member either have the same number of elements or have zero elements.

Two different elements define variables and their content:

`sourceVariable`
These variables' data comes from the associated `tableData.bin` member.

`derivedVariable`
These variables are defined in terms of a mapping function from a source variable, or they are empty.

A variable named `cell` always exists. This variable holds the data displayed in the table.

Variables in detail XML roughly correspond to the dimensions in a light detail member. Each dimension has the following variables with stylized names, where *n* is a number for the dimension starting from 0:

`dimension`*n*`categories`
The dimension's leaf categories (see Section 2.2.10 [SPV Light Member Categories], page 51).

`dimension`*n*`group0`
Present only if the dimension's categories are grouped, this variable holds the group labels for the categories. Grouping is inferred through adjacent identical labels. Categories that are not part of a group have empty-string data in this variable.

`dimension`*n*`group1`
Present only if the first-level groups are further grouped, this variable holds the labels for the second-level groups. There can be additional variables with further levels of grouping.

`dimension`*n*
An empty variable.

Determining the data for a (non-empty) variable is a multi-step process:

1. Draw initial data from its source, for a `sourceVariable`, or from another named variable, for a `derivedVariable`.
2. Apply mappings from `valueMapEntry` elements within the `derivedVariable` element, if any.
3. Apply mappings from `relabel` elements within a `format` or `stringFormat` element in the `sourceVariable` or `derivedVariable` element, if any.

4. If the variable is a `sourceVariable` with a `labelVariable` attribute, and there were no mappings to apply in previous steps, then replace each element of the variable by the corresponding value in the label variable.

A single variable's data can be modified in two of the steps, if both `valueMapEntry` and `relabel` are used. The following example from the corpus maps several integers to 2, then maps 2 in turn to the string "Input":

```
<derivedVariable categorical="true" dependsOn="dimension0categories"
                 id="dimension0group0map" value="map(dimension0group0)">
  <stringFormat>
    <relabel from="2" to="Input"/>
    <relabel from="10" to="Missing Value Handling"/>
    <relabel from="14" to="Resources"/>
    <relabel from="0" to=""/>
    <relabel from="1" to=""/>
    <relabel from="13" to=""/>
  </stringFormat>
  <valueMapEntry from="2;3;5;6;7;8;9" to="2"/>
  <valueMapEntry from="10;11" to="10"/>
  <valueMapEntry from="14;15" to="14"/>
  <valueMapEntry from="0" to="0"/>
  <valueMapEntry from="1" to="1"/>
  <valueMapEntry from="13" to="13"/>
</derivedVariable>
```

2.4.2.1 The `sourceVariable` Element

```
sourceVariable
   :id
   :categorical=(true)
   :source
   :domain=ref categoricalDomain?
   :sourceName
   :dependsOn=ref sourceVariable?
   :label?
   :labelVariable=ref sourceVariable?
=> variable_extension* (format | stringFormat)?
```

This element defines a variable whose data comes from the `tableData.bin` member that corresponds to this `.xml`.

This element has the following attributes.

`id` [Attribute]

An `id` is always present because this element exists to be referenced from other elements.

`categorical` [Attribute]

Always set to `true`.

`source` [Attribute]
: Always set to `tableData`, the `source-name` in the corresponding `tableData.bin` member (see Section 2.3.1 [SPV Legacy Member Metadata], page 57).

`sourceName` [Attribute]
: The name of a variable within the source, corresponding to the `variable-name` in the `tableData.bin` member (see Section 2.3.2 [SPV Legacy Member Numeric Data], page 57).

`label` [Attribute]
: The variable label, if any.

`labelVariable` [Attribute]
: The `variable-name` of a variable whose string values correspond one-to-one with the values of this variable and are suitable for use as value labels.

`dependsOn` [Attribute]
: This attribute doesn't affect the display of a table.

2.4.2.2 The `derivedVariable` Element

```
derivedVariable
   :id
   :categorical=(true)
   :value
   :dependsOn=ref sourceVariable?
=> variable_extension* (format | stringFormat)? valueMapEntry*
```

Like `sourceVariable`, this element defines a variable whose values can be used elsewhere in the visualization. Instead of being read from a data source, the variable's data are defined by a mathematical expression.

This element has the following attributes.

`id` [Attribute]
: An `id` is always present because this element exists to be referenced from other elements.

`categorical` [Attribute]
: Always set to `true`.

`value` [Attribute]
: An expression that defines the variable's value. In theory this could be an arbitrary expression in terms of constants, functions, and other variables, e.g. $(var1 + var2)/2$. In practice, the corpus contains only the following forms of expressions:

  `constant(0)`
  `constant(`*`variable`*`)`
  : All zeros. The reason why a variable is sometimes named is unknown. Sometimes the "variable name" has spaces in it.

  `map(`*`variable`*`)`
  : Transforms the values in the named *variable* using the `valueMapEntry`s contained within the element.

`dependsOn` [Attribute]

This attribute doesn't affect the display of a table.

2.4.2.3 The valueMapEntry Element

```
valueMapEntry :from :to => EMPTY
```

A `valueMapEntry` element defines a mapping from one or more values of a source expression to a target value. (In the corpus, the source expression is always just the name of a variable.) Each target value requires a separate `valueMapEntry`. If multiple source values map to the same target value, they can be combined or separate.

In the corpus, all of the source and target values are integers.

`valueMapEntry` has the following attributes.

`from` [Attribute]

A source value, or multiple source values separated by semicolons, e.g. `0` or `13;14;15;16`.

`to` [Attribute]

The target value, e.g. `0`.

2.4.3 The extension Element

This is a general-purpose "extension" element. Readers that don't understand a given extension should be able to safely ignore it. The attributes on this element, and their meanings, vary based on the context. Each known usage is described separately below. The current extensions use attributes exclusively, without any nested elements.

container Parent Element

```
extension[container_extension] :combinedFootnotes=(true) => EMPTY
```

With `container` as its parent element, `extension` has the following attributes.

`combinedFootnotes` [Attribute]

Always set to `true` in the corpus.

sourceVariable and derivedVariable Parent Element

```
extension[variable_extension] :from :helpId => EMPTY
```

With `sourceVariable` or `derivedVariable` as its parent element, `extension` has the following attributes. A given parent element often contains several `extension` elements that specify the meaning of the source data's variables or sources, e.g.

```
<extension from="0" helpId="corrected_model"/>
<extension from="3" helpId="error"/>
<extension from="4" helpId="total_9"/>
<extension from="5" helpId="corrected_total"/>
```

More commonly they are less helpful, e.g.

```
<extension from="0" helpId="notes"/>
<extension from="1" helpId="notes"/>
<extension from="2" helpId="notes"/>
```

```
<extension from="5" helpId="notes"/>
<extension from="6" helpId="notes"/>
<extension from="7" helpId="notes"/>
<extension from="8" helpId="notes"/>
<extension from="12" helpId="notes"/>
<extension from="13" helpId="no_help"/>
<extension from="14" helpId="notes"/>
```

`from` [Attribute]

An integer or a name like "dimension0".

`helpId` [Attribute]

An identifier.

2.4.4 The graph Element

```
graph
   :cellStyle=ref style
   :style=ref style
=> location+ coordinates faceting facetLayout interval

coordinates => EMPTY
```

`graph` has the following attributes.

`cellStyle` [Attribute]
`style` [Attribute]

Each of these is the `id` of a `style` element (see Section 2.4.12 [SPV Detail style Element], page 80). The former is the default style for individual cells, the latter for the entire table.

2.4.5 The location Element

```
location
   :part=(height | width | top | bottom | left | right)
   :method=(sizeToContent | attach | fixed | same)
   :min=dimension?
   :max=dimension?
   :target=ref (labelFrame | graph | container)?
   :value?
=> EMPTY
```

Each instance of this element specifies where some part of the table frame is located. All the examples in the corpus have four instances of this element, one for each of the parts `height`, `width`, `left`, and `top`. Some examples in the corpus add a fifth for part `bottom`, even though it is not clear how all of `top`, `bottom`, and `height` can be honored at the same time. In any case, `location` seems to have little importance in representing tables; a reader can safely ignore it.

`part` [Attribute]

The part of the table being located.

`method` [Attribute]

How the location is determined:

`sizeToContent`
: Based on the natural size of the table. Observed only for parts `height` and `width`.

`attach`
: Based on the location specified in `target`. Observed only for parts `top` and `bottom`.

`fixed`
: Using the value in `value`. Observed only for parts `top`, `bottom`, and `left`.

`same`
: Same as the specified `target`. Observed only for part `left`.

`min` [Attribute]

Minimum size. Only observed with value `100pt`. Only observed for part `width`.

`target` [Dependent]

Required when `method` is `attach` or `same`, not observed otherwise. This identifies an element to attach to. Observed with the ID of `title`, `footnote`, `graph`, and other elements.

`value` [Dependent]

Required when `method` is `fixed`, not observed otherwise. Observed values are `0%`, `0px`, `1px`, and `3px` on parts `top` and `left`, and `100%` on part `bottom`.

2.4.6 The `faceting` Element

```
faceting => layer[layers1]* cross layer[layers2]*

cross => (unity | nest) (unity | nest)

unity => EMPTY

nest => variableReference[vars]+

variableReference :ref=ref (sourceVariable | derivedVariable) => EMPTY

layer
   :variable=ref (sourceVariable | derivedVariable)
   :value
   :visible=bool?
   :method[layer_method]=(nest)?
   :titleVisible=bool?
=> EMPTY
```

The `faceting` element describes the row, column, and layer structure of the table. Its `cross` child determines the row and column structure, and each `layer` child (if any) represents a layer. Layers may appear before or after `cross`.

The `cross` element describes the row and column structure of the table. It has exactly two children, the first of which describes the table's columns and the second the table's rows.

Each child is a `nest` element if the table has any dimensions along the axis in question, otherwise a `unity` element.

A `nest` element contains of one or more dimensions listed from innermost to outermost, each represented by `variableReference` child elements. Each variable in a dimension is listed in order. See Section 2.4.2 [SPV Detail Variable Elements], page 61, for information on the variables that comprise a dimension.

A `nest` can contain a single dimension, e.g.:

```
<nest>
  <variableReference ref="dimension0categories"/>
  <variableReference ref="dimension0group0"/>
  <variableReference ref="dimension0"/>
</nest>
```

A `nest` can contain multiple dimensions, e.g.:

```
<nest>
  <variableReference ref="dimension1categories"/>
  <variableReference ref="dimension1group0"/>
  <variableReference ref="dimension1"/>
  <variableReference ref="dimension0categories"/>
  <variableReference ref="dimension0"/>
</nest>
```

A `nest` may have no dimensions, in which case it still has one `variableReference` child, which references a `derivedVariable` whose `value` attribute is `constant(0)`. In the corpus, such a `derivedVariable` has `row` or `column`, respectively, as its `id`. This is equivalent to using a `unity` element in place of `nest`.

A `variableReference` element refers to a variable through its `ref` attribute.

Each `layer` element represents a dimension, e.g.:

```
<layer value="0" variable="dimension0categories" visible="true"/>
<layer value="dimension0" variable="dimension0" visible="false"/>
```

`layer` has the following attributes.

`variable` [Attribute]

Refers to a `sourceVariable` or `derivedVariable` element.

`value` [Attribute]

The value to select. For a category variable, this is always `0`; for a data variable, it is the same as the `variable` attribute.

`visible` [Attribute]

Whether the layer is visible. Generally, category layers are visible and data layers are not, but sometimes this attribute is omitted.

`method` [Attribute]

When present, this is always `nest`.

2.4.7 The `facetLayout` Element

```
facetLayout => tableLayout setCellProperties[scp1]*
               facetLevel+ setCellProperties[scp2]*

tableLayout
   :verticalTitlesInCorner=bool
   :style=ref style?
   :fitCells=(ticks both)?
=> EMPTY
```

The `facetLayout` element and its descendants control styling for the table.

Its `tableLayout` child has the following attributes

`verticalTitlesInCorner` [Attribute]
: If true, in the absence of corner text, row headings will be displayed in the corner.

`style` [Attribute]
: Refers to a `style` element.

`fitCells` [Attribute]
: Meaning unknown.

The `facetLevel` Element

```
facetLevel :level=int :gap=dimension? => axis

axis :style=ref style => label? majorTicks

majorTicks
   :labelAngle=int
   :length=dimension
   :style=ref style
   :tickFrameStyle=ref style
   :labelFrequency=int?
   :stagger=bool?
=> gridline?

gridline
   :style=ref style
   :zOrder=int
=> EMPTY
```

Each `facetLevel` describes a `variableReference` or `layer`, and a table has one `facetLevel` element for each such element. For example, an SPV detail member that contains four `variableReference` elements and two `layer` elements will contain six `facetLevel` elements.

In the corpus, `facetLevel` elements and the elements that they describe are always in the same order. The correspondence may also be observed in two other ways. First, one may use the `level` attribute, described below. Second, in the corpus, a `facetLevel` always has an `id` that is the same as the `id` of the element it describes with `_facetLevel` appended. One should not formally rely on this, of course, but it is usefully indicative.

`level` [Attribute]
: A 1-based index into the `variableReference` and `layer` elements, e.g. a `facetLayout` with a `level` of 1 describes the first `variableReference` in the SPV detail member, and in a member with four `variableReference` elements, a `facetLayout` with a `level` of 5 describes the first `layer` in the member.

`gap` [Attribute]
: Always observed as `0pt`.

Each `facetLevel` contains an `axis`, which in turn may contain a `label` for the `facetLevel` (see Section 2.4.8 [SPV Detail label Element], page 69) and does contain a `majorTicks` element.

`labelAngle` [Attribute]
: Normally 0. The value -90 causes inner column or outer row labels to be rotated vertically.

`style` [Attribute]
`tickFrameStyle` [Attribute]
: Each refers to a `style` element. `style` is the style of the tick labels, `tickFrameStyle` the style for the frames around the labels.

2.4.8 The `label` Element

```
label
   :style=ref style
   :textFrameStyle=ref style?
   :purpose=(title | subTitle | subSubTitle | layer | footnote)?
=> text+ | descriptionGroup

descriptionGroup
   :target=ref faceting
   :separator?
=> (description | text)+

description :name=(variable | value) => EMPTY

text
   :usesReference=int?
   :definesReference=int?
   :position=(subscript | superscript)?
   :style=ref style
=> TEXT
```

This element represents a label on some aspect of the table.

`style` [Attribute]
`textFrameStyle` [Attribute]
Each of these refers to a `style` element. `style` is the style of the label text, `textFrameStyle` the style for the frame around the label.

`purpose` [Attribute]
The kind of entity being labeled.

A `descriptionGroup` concatenates one or more elements to form a label. Each element can be a `text` element, which contains literal text, or a `description` element that substitutes a value or a variable name.

`target` [Attribute]
The `id` of an element being described. In the corpus, this is always `faceting`.

`separator` [Attribute]
A string to separate the description of multiple groups, if the `target` has more than one. In the corpus, this is always a new-line.

Typical contents for a `descriptionGroup` are a value by itself:

```
<description name="value"/>
```

or a variable and its value, separated by a colon:

```
<description name="variable"/><text>:</text><description name="value"/>
```

A `description` is like a macro that expands to some property of the target of its parent `descriptionGroup`. The `name` attribute specifies the property.

2.4.9 The setCellProperties Element

```
setCellProperties
   :applyToConverse=bool?
=> (setStyle | setFrameStyle | setFormat | setMetaData)* union[union_]?
```

The `setCellProperties` element sets style properties of cells or row or column labels.

Interpreting `setCellProperties` requires answering two questions: which cells or labels to style, and what styles to use.

Which Cells?

```
union => intersect+

intersect => where+ | intersectWhere | alternating | EMPTY

where
   :variable=ref (sourceVariable | derivedVariable)
   :include
=> EMPTY

intersectWhere
   :variable=ref (sourceVariable | derivedVariable)
   :variable2=ref (sourceVariable | derivedVariable)
```

```
=> EMPTY

alternating => EMPTY
```

When `union` is present with `intersect` children, each of those children specifies a group of cells that should be styled, and the total group is all those cells taken together. When `union` is absent, every cell is styled. One attribute on `setCellProperties` affects the choice of cells:

`applyToConverse` [Attribute]
: If true, this inverts the meaning of the cell selection: the selected cells are the ones *not* designated. This is confusing, given the additional restrictions of `union`, but in the corpus `applyToConverse` is never present along with `union`.

An `intersect` specifies restrictions on the cells to be matched. Each `where` child specifies which values of a given variable to include. The attributes of `intersect` are:

`variable` [Attribute]
: Refers to a variable, e.g. `dimension0categories`. Only "categories" variables make sense here, but other variables, e.g. `dimension0group0map`, are sometimes seen. The reader may ignore these.

`include` [Attribute]
: A value, or multiple values separated by semicolons, e.g. `0` or `13;14;15;16`.

PSPP ignores `setCellProperties` when `intersectWhere` is present.

What Styles?

```
setStyle
   :target=ref (labeling | graph | interval | majorTicks)
   :style=ref style
=> EMPTY

setMetaData :target=ref graph :key :value => EMPTY

setFormat
   :target=ref (majorTicks | labeling)
   :reset=bool?
=> format | numberFormat | stringFormat+ | dateTimeFormat | elapsedTimeFormat

setFrameStyle
   :style=ref style
   :target=ref majorTicks
=> EMPTY
```

The `set*` children of `setCellProperties` determine the styles to set.

When `setCellProperties` contains a `setFormat` whose `target` references a `labeling` element, or if it contains a `setStyle` that references a `labeling` or `interval` element, the `setCellProperties` sets the style for table cells. The format from the `setFormat`, if present, replaces the cells' format. The style from the `setStyle` that references `labeling`, if present, replaces the label's font and cell styles, except that the background color is taken instead from the `interval`'s style, if present.

When `setCellProperties` contains a `setFormat` whose `target` references a `majorTicks` element, or if it contains a `setStyle` whose `target` references a `majorTicks`, or if it contains a `setFrameStyle` element, the `setCellProperties` sets the style for row or column labels. In this case, the `setCellProperties` always contains a single `where` element whose `variable` designates the variable whose labels are to be styled. The format from the `setFormat`, if present, replaces the labels' format. The style from the `setStyle` that references `majorTicks`, if present, replaces the labels' font and cell styles, except that the background color is taken instead from the `setFrameStyle`'s style, if present.

When `setCellProperties` contains a `setStyle` whose `target` references a `graph` element, and one that references a `labeling` element, and the `union` element contains `alternating`, the `setCellProperties` sets the alternate foreground and background colors for the data area. The foreground color is taken from the style referenced by the `setStyle` that targets the `graph`, the background color from the `setStyle` for `labeling`.

A reader may ignore a `setCellProperties` that only contains `setMetaData`, as well as `setMetaData` within other `setCellProperties`.

A reader may ignore a `setCellProperties` whose only `set*` child is a `setStyle` that targets the `graph` element.

The `setStyle` Element

```
setStyle
   :target=ref (labeling | graph | interval | majorTicks)
   :style=ref style
=> EMPTY
```

This element associates a style with the target.

`target` [Attribute]
: The `id` of an element whose style is to be set.

`style` [Attribute]
: The `id` of a `style` element that identifies the style to set on the target.

2.4.10 The `setFormat` Element

```
setFormat
   :target=ref (majorTicks | labeling)
   :reset=bool?
=> format | numberFormat | stringFormat+ | dateTimeFormat | elapsedTimeFormat
```

This element sets the format of the target, "format" in this case meaning the SPSS print format for a variable.

The details of this element vary depending on the schema version, as declared in the root `visualization` element's `version` attribute (see Section 2.4.1 [SPV Detail visualization Element], page 59). A reader can interpret the content without knowing the schema version.

The `setFormat` element itself has the following attributes.

`target` [Attribute]
Refers to an element whose style is to be set.

`reset` [Attribute]
If this is `true`, this format replaces the target's previous format. If it is `false`, the modifies the previous format.

2.4.10.1 The `numberFormat` Element

```
numberFormat
   :minimumIntegerDigits=int?
   :maximumFractionDigits=int?
   :minimumFractionDigits=int?
   :useGrouping=bool?
   :scientific=(onlyForSmall | whenNeeded | true | false)?
   :small=real?
   :prefix?
   :suffix?
=> affix*
```

Specifies a format for displaying a number. The available options are a superset of those available from PSPP print formats. PSPP chooses a print format type for a `numberFormat` as follows:

1. If `scientific` is `true`, uses `E` format.
2. If `prefix` is `$`, uses `DOLLAR` format.
3. If `suffix` is `%`, uses `PCT` format.
4. If `useGrouping` is `true`, uses `COMMA` format.
5. Otherwise, uses `F` format.

For translating to a print format, PSPP uses `maximumFractionDigits` as the number of decimals, unless that attribute is missing or out of the range [0,15], in which case it uses 2 decimals.

`minimumIntegerDigits` [Attribute]
Minimum number of digits to display before the decimal point. Always observed as `0`.

`maximumFractionDigits` [Attribute]
`minimumFractionDigits` [Attribute]
Maximum or minimum, respectively, number of digits to display after the decimal point. The observed values of each attribute range from 0 to 9.

`useGrouping` [Attribute]
Whether to use the grouping character to group digits in large numbers.

`scientific` [Attribute]
This attribute controls when and whether the number is formatted in scientific notation. It takes the following values:

`onlyForSmall`
Use scientific notation only when the number's magnitude is smaller than the value of the `small` attribute.

`whenNeeded`
: Use scientific notation when the number will not otherwise fit in the available space.

`true`
: Always use scientific notation. Not observed in the corpus.

`false`
: Never use scientific notation. A number that won't otherwise fit will be replaced by an error indication (see the `errorCharacter` attribute). Not observed in the corpus.

`small` [Attribute]

Only present when the `scientific` attribute is `onlyForSmall`, this is a numeric magnitude below which the number will be formatted in scientific notation. The values `0` and `0.0001` have been observed. The value `0` seems like a pathological choice, since no real number has a magnitude less than 0; perhaps in practice such a choice is equivalent to setting `scientific` to `false`.

`prefix` [Attribute]
`suffix` [Attribute]

Specifies a prefix or a suffix to apply to the formatted number. Only `suffix` has been observed, with value '`%`'.

2.4.10.2 The `stringFormat` Element

```
stringFormat => relabel* affix*

relabel :from=real :to => EMPTY
```

The `stringFormat` element specifies how to display a string. By default, a string is displayed verbatim, but `relabel` can change it.

The `relabel` element appears as a child of `stringFormat` (and of `format`, when it is used to format strings). It specifies how to display a given value. It is used to implement value labels and to display the system-missing value in a human-readable way. It has the following attributes:

`from` [Attribute]

The value to map. In the corpus this is an integer or the system-missing value `-1.797693134862316E300`.

`to` [Attribute]

The string to display in place of the value of `from`. In the corpus this is a wide variety of value labels; the system-missing value is mapped to '`.`'.

2.4.10.3 The `dateTimeFormat` Element

```
dateTimeFormat
   :baseFormat[dt_base_format]=(date | time | dateTime)
   :separatorChars?
   :mdyOrder=(dayMonthYear | monthDayYear | yearMonthDay)?
   :showYear=bool?
   :yearAbbreviation=bool?
   :showQuarter=bool?
```

```
      :quarterPrefix?
      :quarterSuffix?
      :showMonth=bool?
      :monthFormat=(long | short | number | paddedNumber)?
      :showWeek=bool?
      :weekPadding=bool?
      :weekSuffix?
      :showDayOfWeek=bool?
      :dayOfWeekAbbreviation=bool?
      :dayPadding=bool?
      :dayOfMonthPadding=bool?
      :hourPadding=bool?
      :minutePadding=bool?
      :secondPadding=bool?
      :showDay=bool?
      :showHour=bool?
      :showMinute=bool?
      :showSecond=bool?
      :showMillis=bool?
      :dayType=(month | year)?
      :hourFormat=(AMPM | AS_24 | AS_12)?
   => affix*
```

This element appears only in schema version 2.5 and earlier (see Section 2.4.1 [SPV Detail visualization Element], page 59).

Data to be formatted in date formats is stored as strings in legacy data, in the format `yyyy-mm-ddTHH:MM:SS.SSS` and must be parsed and reformatted by the reader.

The following attribute is required.

`baseFormat` [Attribute]
: Specifies whether a date and time are both to be displayed, or just one of them.

Many of the attributes' meanings are obvious. The following seem to be worth documenting.

`separatorChars` [Attribute]
: Exactly four characters. In order, these are used for: decimal point, grouping, date separator, time separator. Always '`.,-:`'.

`mdyOrder` [Attribute]
: Within a date, the order of the days, months, and years. `dayMonthYear` is the only observed value, but one would expect that `monthDayYear` and `yearMonthDay` to be reasonable as well.

`showYear` [Attribute]
`yearAbbreviation` [Attribute]
: Whether to include the year and, if so, whether the year should be shown abbreviated, that is, with only 2 digits. Each is `true` or `false`; only values of `true` and `false`, respectively, have been observed.

`showMonth` [Attribute]
`monthFormat` [Attribute]

Whether to include the month (`true` or `false`) and, if so, how to format it. `monthFormat` is one of the following:

`long` The full name of the month, e.g. in an English locale, `September`.

`short` The abbreviated name of the month, e.g. in an English locale, `Sep`.

`number` The number representing the month, e.g. 9 for September.

`paddedNumber`
A two-digit number representing the month, e.g. 09 for September.

Only values of `true` and `short`, respectively, have been observed.

`dayType` [Attribute]

This attribute is always `month` in the corpus, specifying that the day of the month is to be displayed; a value of `year` is supposed to indicate that the day of the year, where 1 is January 1, is to be displayed instead.

`hourFormat` [Attribute]

`hourFormat`, if present, is one of:

`AMPM` The time is displayed with an `am` or `pm` suffix, e.g. `10:15pm`.

`AS_24` The time is displayed in a 24-hour format, e.g. `22:15`.
This is the only value observed in the corpus.

`AS_12` The time is displayed in a 12-hour format, without distinguishing morning or evening, e.g. `10;15`.

`hourFormat` is sometimes present for `elapsedTime` formats, which is confusing since a time duration does not have a concept of AM or PM. This might indicate a bug in the code that generated the XML in the corpus, or it might indicate that `elapsedTime` is sometimes used to format a time of day.

For a `baseFormat` of `date`, PSPP chooses a print format type based on the following rules:

1. If `showQuarter` is true: `QYR`.
2. Otherwise, if `showWeek` is true: `WKYR`.
3. Otherwise, if `mdyOrder` is `dayMonthYear`:
   a. If `monthFormat` is `number` or `paddedNumber`: `EDATE`.
   b. Otherwise: `DATE`.
4. Otherwise, if `mdyOrder` is `yearMonthDay`: `SDATE`.
5. Otherwise, `ADATE`.

For a `baseFormat` of `dateTime`, PSPP uses `YMDHMS` if `mdyOrder` is `yearMonthDay` and `DATETIME` otherwise. For a `baseFormat` of `time`, PSPP uses `DTIME` if `showDay` is true, otherwise `TIME` if `showHour` is true, otherwise `MTIME`.

For a `baseFormat` of `date`, the chosen width is the minimum for the format type, adding 2 if `yearAbbreviation` is false or omitted. For other base formats, the chosen width is the minimum for its type, plus 3 if `showSecond` is true, plus 4 more if `showMillis` is also true. Decimals are 0 by default, or 3 if `showMillis` is true.

2.4.10.4 The `elapsedTimeFormat` Element

```
elapsedTimeFormat
   :baseFormat[dt_base_format]=(date | time | dateTime)
   :dayPadding=bool?
   :hourPadding=bool?
   :minutePadding=bool?
   :secondPadding=bool?
   :showYear=bool?
   :showDay=bool?
   :showHour=bool?
   :showMinute=bool?
   :showSecond=bool?
   :showMillis=bool?
=> affix*
```

This element specifies the way to display a time duration.

Data to be formatted in elapsed time formats is stored as strings in legacy data, in the format `H:MM:SS.SSS`, with additional hour digits as needed for long durations, and must be parsed and reformatted by the reader.

The following attribute is required.

`baseFormat` [Attribute]

Specifies whether a day and a time are both to be displayed, or just one of them.

The remaining attributes specify exactly how to display the elapsed time.

For `baseFormat` of `time`, PSPP converts this element to print format type `DTIME`; otherwise, if `showHour` is true, to `TIME`; otherwise, to `MTIME`. The chosen width is the minimum for the chosen type, adding 3 if `showSecond` is true, adding 4 more if `showMillis` is also true. Decimals are 0 by default, or 3 if `showMillis` is true.

2.4.10.5 The `format` Element

```
format
   :baseFormat[f_base_format]=(date | time | dateTime | elapsedTime)?
   :errorCharacter?
   :separatorChars?
   :mdyOrder=(dayMonthYear | monthDayYear | yearMonthDay)?
   :showYear=bool?
   :showQuarter=bool?
   :quarterPrefix?
   :quarterSuffix?
   :yearAbbreviation=bool?
   :showMonth=bool?
   :monthFormat=(long | short | number | paddedNumber)?
   :dayPadding=bool?
   :dayOfMonthPadding=bool?
   :showWeek=bool?
   :weekPadding=bool?
```

```
    :weekSuffix?
    :showDayOfWeek=bool?
    :dayOfWeekAbbreviation=bool?
    :hourPadding=bool?
    :minutePadding=bool?
    :secondPadding=bool?
    :showDay=bool?
    :showHour=bool?
    :showMinute=bool?
    :showSecond=bool?
    :showMillis=bool?
    :dayType=(month | year)?
    :hourFormat=(AMPM | AS_24 | AS_12)?
    :minimumIntegerDigits=int?
    :maximumFractionDigits=int?
    :minimumFractionDigits=int?
    :useGrouping=bool?
    :scientific=(onlyForSmall | whenNeeded | true | false)?
    :small=real?
    :prefix?
    :suffix?
    :tryStringsAsNumbers=bool?
    :negativesOutside=bool?
=> relabel* affix*
```

This element is the union of all of the more-specific format elements. It is interpreted in the same way as one of those format elements, using `baseFormat` to determine which kind of format to use.

There are a few attributes not present in the more specific formats:

`tryStringsAsNumbers` [Attribute]
: When this is `true`, it is supposed to indicate that string values should be parsed as numbers and then displayed according to numeric formatting rules. However, in the corpus it is always `false`.

`negativesOutside` [Attribute]
: If true, the negative sign should be shown before the prefix; if false, it should be shown after.

2.4.10.6 The `affix` Element

```
affix
    :definesReference=int
    :position=(subscript | superscript)
    :suffix=bool
    :value
=> EMPTY
```

This defines a suffix (or, theoretically, a prefix) for a formatted value. It is used to insert a reference to a footnote. It has the following attributes:

`definesReference` [Attribute]

This specifies the footnote number as a natural number: 1 for the first footnote, 2 for the second, and so on.

`position` [Attribute]

Position for the footnote label. Always `superscript`.

`suffix` [Attribute]

Whether the affix is a suffix (`true`) or a prefix (`false`). Always `true`.

`value` [Attribute]

The text of the suffix or prefix. Typically a letter, e.g. `a` for footnote 1, `b` for footnote 2, ... The corpus contains other values: `*`, `**`, and a few that begin with at least one comma: `,b`, `,c`, `,,b`, and `,,c`.

2.4.11 The `interval` Element

```
interval :style=ref style => labeling footnotes?

labeling
   :style=ref style?
   :variable=ref (sourceVariable | derivedVariable)
=> (formatting | format | footnotes)*

formatting :variable=ref (sourceVariable | derivedVariable) => formatMapping*

formatMapping :from=int => format?

footnotes
   :superscript=bool?
   :variable=ref (sourceVariable | derivedVariable)
=> footnoteMapping*

footnoteMapping :definesReference=int :from=int :to => EMPTY
```

The `interval` element and its descendants determine the basic formatting and labeling for the table's cells. These basic styles are overridden by more specific styles set using `setCellProperties` (see Section 2.4.9 [SPV Detail setCellProperties Element], page 70).

The `style` attribute of `interval` itself may be ignored.

The `labeling` element may have a single `formatting` child. If present, its `variable` attribute refers to a variable whose values are format specifiers as numbers, e.g. value 0x050802 for F8.2. However, the numbers are not actually interpreted that way. Instead, each number actually present in the variable's data is mapped by a `formatMapping` child of `formatting` to a `format` that specifies how to display it.

The `labeling` element may also have a `footnotes` child element. The `variable` attribute of this element refers to a variable whose values are comma-delimited strings that list the 1-based indexes of footnote references. (Cells without any footnote references are numeric 0 instead of strings.)

Each `footnoteMapping` child of the `footnotes` element defines the footnote marker to be its `to` attribute text for the footnote whose 1-based index is given in its `definesReference` attribute.

2.4.12 The style Element

```
style
   :color=color?
   :color2=color?
   :labelAngle=real?
   :border-bottom=(solid | thick | thin | double | none)?
   :border-top=(solid | thick | thin | double | none)?
   :border-left=(solid | thick | thin | double | none)?
   :border-right=(solid | thick | thin | double | none)?
   :border-bottom-color?
   :border-top-color?
   :border-left-color?
   :border-right-color?
   :font-family?
   :font-size?
   :font-weight=(regular | bold)?
   :font-style=(regular | italic)?
   :font-underline=(none | underline)?
   :margin-bottom=dimension?
   :margin-left=dimension?
   :margin-right=dimension?
   :margin-top=dimension?
   :textAlignment=(left | right | center | decimal | mixed)?
   :labelLocationHorizontal=(positive | negative | center)?
   :labelLocationVertical=(positive | negative | center)?
   :decimal-offset=dimension?
   :size?
   :width?
   :visible=bool?
=> EMPTY
```

A `style` element has an effect only when it is referenced by another element to set some aspect of the table's style. Most of the attributes are self-explanatory. The rest are described below.

`color` [Attribute]
: In some cases, the text color; in others, the background color.

`color2` [Attribute]
: Not used.

`labelAngle` [Attribute]
: Normally 0. The value -90 causes inner column or outer row labels to be rotated vertically.

`labelLocationHorizontal` [Attribute]
Not used.

`labelLocationVertical` [Attribute]
The value `positive` corresponds to vertically aligning text to the top of a cell, `negative` to the bottom, `center` to the middle.

2.4.13 The labelFrame Element

```
labelFrame :style=ref style => location+ label? paragraph?

paragraph :hangingIndent=dimension? => EMPTY
```

A `labelFrame` element specifies content and style for some aspect of a table. Only `labelFrame` elements that have a `label` child are important. The `purpose` attribute in the `label` determines what the `labelFrame` affects:

`title` The table's title and its style.

`subTitle` The table's caption and its style.

`footnote` The table's footnotes and the style for the footer area.

`layer` The style for the layer area.

`subSubTitle`
Ignored.

The `style` attribute references the style to use for the area.

The `label`, if present, specifies the text to put into the title or caption or footnotes. For footnotes, the label has two `text` children for every footnote, each of which has a `usesReference` attribute identifying the 1-based index of a footnote. The first, third, fifth, ... `text` child specifies the content for a footnote; the second, fourth, sixth, ... child specifies the marker. Content tends to end in a new-line, which the reader may wish to trim; similarly, markers tend to end in '`.`'.

The `paragraph`, if present, may be ignored, since it is always empty.

2.4.14 Legacy Properties

The detail XML format has features for styling most of the aspects of a table. It also inherits defaults for many aspects from structure XML, which has the following `tableProperties` element:

```
tableProperties
   :name?
=> generalProperties footnoteProperties cellFormatProperties borderProperties printing

generalProperties
   :hideEmptyRows=bool?
   :maximumColumnWidth=dimension?
   :maximumRowWidth=dimension?
   :minimumColumnWidth=dimension?
   :minimumRowWidth=dimension?
```

```
   :rowDimensionLabels=(inCorner | nested)?
=> EMPTY

footnoteProperties
   :markerPosition=(superscript | subscript)?
   :numberFormat=(alphabetic | numeric)?
=> EMPTY

cellFormatProperties => cell_style+

any[cell_style]
   :alternatingColor=color?
   :alternatingTextColor=color?
=> style

style
   :color=color?
   :color2=color?
   :font-family?
   :font-size?
   :font-style=(regular | italic)?
   :font-weight=(regular | bold)?
   :font-underline=(none | underline)?
   :labelLocationVertical=(positive | negative | center)?
   :margin-bottom=dimension?
   :margin-left=dimension?
   :margin-right=dimension?
   :margin-top=dimension?
   :textAlignment=(left | right | center | decimal | mixed)?
   :decimal-offset=dimension?
=> EMPTY

borderProperties => border_style+

any[border_style]
   :borderStyleType=(none | solid | dashed | thick | thin | double)?
   :color=color?
=> EMPTY

printingProperties
   :printAllLayers=bool?
   :rescaleLongTableToFitPage=bool?
   :rescaleWideTableToFitPage=bool?
   :windowOrphanLines=int?
   :continuationText?
   :continuationTextAtBottom=bool?
   :continuationTextAtTop=bool?
```

```
        :printEachLayerOnSeparatePage=bool?
    => EMPTY
```

The `name` attribute appears only in standalone `.stt` files (see Section 3.1 [SPSS TableLook STT Format], page 84).

3 SPSS TableLook File Formats

SPSS has a concept called a TableLook to control the styling of pivot tables in output. SPSS 15 and earlier used `.tlo` files with a special binary format to save TableLooks to disk; SPSS 16 and later use `.stt` files in an XML format to save them. Both formats expose roughly the same features, although the older `.tlo` format does have some features that `.stt` does not.

This chapter describes both formats.

3.1 The `.stt` Format

The `.stt` file format is an XML file that contains a subset of the SPV structure member format (see Section 2.1 [SPV Structure Member Format], page 29). Its root element is a `tableProperties` element (see Section 2.4.14 [SPV Detail Legacy Properties], page 81).

3.2 The `.tlo` Format

A `.tlo` file has a custom binary format. This section describes it using the syntax used previously for SPV binary members (see Section 2.2 [SPV Light Detail Member Format], page 40). There is one new convention: TLO files express colors as `int32` values in which the low 8 bits are the red component, the next 8 bits are green, and next 8 bits are blue, and the high bits are zeros.

TLO files support various features that SPV files do not. PSPP implements the SPV feature set, so it mostly ignores the added TLO features. The details of this mapping are explained below.

At the top level, a TLO file consists of five sections. The first four are always present and the last one is optional:

```
TableLook =>
   PTTableLook[tl]
   PVSeparatorStyle[ss]
   PVCellStyle[cs]
   PVTextStyle[ts]
   V2Styles?
```

Each section is described below.

3.2.1 PTTableLook

```
PTTableLook =>
   ff ff 00 00 "PTTableLook" (00|02)[version]
   int16[flags]
   00 00
   bool[nested-row-labels] 00
   bool[footnote-marker-subscripts] 00
   i54 i18
```

In PTTableLook, `version` is 00 or 02. The only difference is that version 00 lacks V2Styles (see Section 3.2.4 [V2Styles in SPSS TLO Files], page 87) and that version 02 includes it. Both TLO versions are seen in the wild.

`flags` is a bit-mapped field. Its bits have the following meanings:

0x2 If set to 1, hide empty rows and columns; otherwise, show them.

0x4 If set to 1, use numeric footnote markers; otherwise, use alphabetic footnote markers.

0x8 If set to 1, print all layers; otherwise, print only the current layer.

0x10 If set to 1, scale the table to fit the page width; otherwise, break it horizontally if necessary.

0x20 If set to 1, scale the table to fit the page length; otherwise, break it vertically if necessary.

0x40 If set to 1, print each layer on a separate page (only if all layers are being printed); otherwise, paginate layers naturally.

0x80 If set to 1, print a continuation string at the top of a table that is split between pages.

0x100 If set to 1, print a continuation string at the bottom of a table that is split between pages.

When `nested-row-labels` is 1, row dimension labels appear nested; otherwise, they are put into the upper-left corner of the pivot table.

When `footnote-marker-subscripts` is 1, footnote markers are shown as subscripts; otherwise, they are shown as superscripts.

3.2.2 PVSeparatorStyle

```
PVSeparatorStyle =>
   ff ff 00 00 "PVSeparatorStyle" 00
   Separator*4[sep1]
   03 80 00
   Separator*4[sep2]

Separator =>
   case(
       00 00
     | 01 00 int32[color] int16[style] int16[width]
   )[type]
```

PVSeparatorStyle contains eight Separators, in two groups. Each Separator represents a border between pivot table elements. TLO and SPV files have the same concepts for borders. See Section 2.2.5 [SPV Light Member Borders], page 44, for the treatment of borders in SPV files.

A Separator's `type` is 00 if the border is not drawn, 01 otherwise. For a border that is drawn, `color` is the color that it is drawn in. `style` and `width` have the following meanings:

`style` $= 0$ and $0 \leq$ `width` ≤ 3
: An increasingly thick single line. SPV files only have three line thicknesses. PSPP treats `width` 0 as a thin line, `width` 1 as a solid (normal width) line, and `width` 2 or 3 as a thick line.

`style` = 1 and $0 \leq$ `width` ≤ 1
: A doubled line, composed of normal-width (0) or thick (1) lines. SPV files only have "normal" width double lines, so PSPP maps both variants the same way.

`style` = 2
: A dashed line.

The first group, `sep1`, represents the following borders within the pivot table, by index:

0. Horizontal dimension rows
1. Vertical dimension rows
2. Horizontal category rows
3. Vertical category rows

The second group, `sep2`, represents the following borders within the pivot table, by index:

0. Horizontal dimension columns
1. Vertical dimension columns
2. Horizontal category columns
3. Vertical category columns

3.2.3 `PVCellStyle` and `PVTextStyle`

```
PVCellStyle =>
   ff ff 00 00 "PVCellStyle"
   AreaColor[title-color]

PVTextStyle =>
   ff ff 00 00 "PVTextStyle" 00
   AreaStyle[title-style] MostAreas*7[most-areas]

MostAreas =>
   06 80
   AreaColor[color] 08 80 00 AreaStyle[style]
```

These sections hold the styling and coloring for each of the 8 areas in a pivot table. They are conceptually similar to the area style information in SPV light members (see Section 2.2.4 [SPV Light Member Areas], page 44).

The styling and coloring for the title area is split between PVCellStyle and PVTextStyle: the former holds `title-color`, the latter holds `title-style`. The style for the remaining 7 areas is in `most-areas` in PVTextStyle, in the following order: layers, corner, row labels, column labels, data, caption, and footer.

```
AreaColor =>
   00 01 00 int32[color10] int32[color0] byte[shading] 00
```

AreaColor represents the background color of an area. TLO files, but not SPV files, describe backgrounds that are a shaded combination of two colors: `shading` of 0 is pure `color0`, `shading` of 10 is pure `color10`, and value in between mix pixels of the two different colors in linear degree. PSPP does not implement shading, so for $1 \leq$ `shading` ≤ 9 it interpolates RGB values between colors to arrive at an intermediate shade.

```
AreaStyle =>
```

```
int16[valign] int16[halign] int16[decimal-offset]
int16[left-margin] int16[right-margin] int16[top-margin] int16[bottom-margin]
00 00 01 00
int32[font-size] int16[stretch]
00*2
int32[rotation-angle]
00*4
int16[weight]
00*2
bool[italic] bool[underline] bool[strikethrough]
int32[rtf-charset-number]
byte[x]
byte[font-name-len] byte*[font-name-len][font-name]
int32[text-color]
00*2
```

AreaStyle represents style properties of an area.

`valign` is 0 for top alignment, 1 for bottom alginment, 2 for center.

`halign` is 0 for left alignment, 1 for right, 2 for center, 3 for mixed, 4 for decimal. For decimal alignment, `decimal-offset` is the offset of the decimal point in 20ths of a point.

`left-margin`, `right-margin`, `top-margin`, and `bottom-margin` are also measured in 20ths of a point.

`font-size` is negative 96ths of an inch, e.g. 9 point is -12 or 0xfffffff3.

`stretch` has something to do with font size or stretch. The usual value is 01 and values larger than that do weird things. A reader can safely ignore it.

`rotation-angle` is a font rotation angle. A reader can safely ignore it.

`weight` is 400 for a normal-weight font, 700 indicates bold. (This is a Windows API convention.)

`italic` and `underline` have the obvious meanings. So does `strikethrough`, which PSPP ignores.

`rtf-charset-number` is a character set number from RTF. A reader can safely ignore it.

The meaning of `x` is unknown. Values 12, 22, 31, and 32 have been observed.

The `font-name` is the name of a font, such as `Arial`. Only US-ASCII characters have been observed here.

`text-color` is the color of the text itself.

3.2.4 V2Styles

```
V2Styles =>
   Separator*11[sep3]
   byte[continuation-len] byte*[continuation-len][continuation]
   int32[min-col-width] int32[max-col-width]
   int32[min-row-height] int32[max-row-height]
```

This final, optional, part of the TLO file format contains some additional style information. It begins with `sep3`, which represents the following borders within the pivot table, by index:

0 Title.

1. . . 4 Left, right, top, and bottom inner frame.

5. . . 8 Left, right, top, and bottom outer frame.

9, 10 Left and top of data area.

When V2Styles is absent, the inner frame borders default to a solid line and the others listed above to no line.

`continuation` is the string that goes at the top or bottom of a table broken across pages. When V2Styles is absent, the default is `(Cont.)`.

`min-col-width` is the minimum width that a column will be assigned automatically. `max-col-width` is the maximum width that a column will be assigned to accommodate a long column label. `min-row-width` and `max-row-width` are a similar range for the width of row labels. All of these measurements are in points. When V2Styles is absent, the defaults are 36 for `min-col-width` and `min-row-height`, 72 for `max-col-width`, and 120 for `max-row-height`.

4 Encrypted File Wrappers

SPSS 21 and later can package multiple kinds of files inside an encrypted wrapper. The wrapper has a common format, regardless of the kind of the file that it contains.

> **Warning:** The SPSS encryption wrapper is poorly designed. When the password is unknown, it is much cheaper and faster to decrypt a file encrypted this way than if a well designed alternative were used. If you must use this format, use a 10-byte randomly generated password.

4.1 Common Wrapper Format

An encrypted file wrapper begins with the following 36-byte header, where *xxx* identifies the type of file encapsulated: `SAV` for a system file, `SPS` for a syntax file, `SPV` for a viewer file. PSPP code for identifying these files just checks for the `ENCRYPTED` keyword at offset 8, but the other bytes are also fixed in practice:

```
0000  1c 00 00 00 00 00 00 00  45 4e 43 52 59 50 54 45  |........ENCRYPTE|
0010  44 xx xx xx 15 00 00 00  00 00 00 00 00 00 00 00  |Dxxx............|
0020  00 00 00 00                                       |....|
```

Following the fixed header is essentially the regular contents of the encapsulated file in its usual format, with each 16-byte block encrypted with AES-256 in ECB mode.

To make the plaintext an even multiple of 16 bytes in length, the encryption process appends PKCS #7 padding, as specified in RFC 5652 section 6.3. Padding appends 1 to 16 bytes to the plaintext, in which each byte of padding is the number of padding bytes added. If the plaintext is, for example, 2 bytes short of a multiple of 16, the padding is 2 bytes with value 02; if the plaintext is a multiple of 16 bytes in length, the padding is 16 bytes with value 0x10.

The AES-256 key is derived from a password in the following way:

1. Start from the literal password typed by the user. Truncate it to at most 10 bytes, then append as many null bytes as necessary until there are exactly 32 bytes. Call this *password*.
2. Let *constant* be the following 73-byte constant:

```
0000  00 00 00 01 35 27 13 cc  53 a7 78 89 87 53 22 11
0010  d6 5b 31 58 dc fe 2e 7e  94 da 2f 00 cc 15 71 80
0020  0a 6c 63 53 00 38 c3 38  ac 22 f3 63 62 0e ce 85
0030  3f b8 07 4c 4e 2b 77 c7  21 f5 1a 80 1d 67 fb e1
0040  e1 83 07 d8 0d 00 00 01  00
```

3. Compute CMAC-AES-256(*password*, *constant*). Call the 16-byte result *cmac*.
4. The 32-byte AES-256 key is *cmac* `||` *cmac*, that is, *cmac* repeated twice.

Example

Consider the password '`pspp`'. *password* is:

```
0000  70 73 70 70 00 00 00 00  00 00 00 00 00 00 00 00  |pspp............|
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

cmac is:

```
0000  3e da 09 8e 66 04 d4 fd  f9 63 0c 2c a8 6f b0 45
```

The AES-256 key is:

```
0000  3e da 09 8e 66 04 d4 fd  f9 63 0c 2c a8 6f b0 45
0010  3e da 09 8e 66 04 d4 fd  f9 63 0c 2c a8 6f b0 45
```

4.1.1 Checking Passwords

A program reading an encrypted file may wish to verify that the password it was given is the correct one. One way is to verify that the PKCS #7 padding at the end of the file is well formed. However, any plaintext that ends in byte 01 is well formed PKCS #7, meaning that about 1 in 256 keys will falsely pass this test. This might be acceptable for interactive use, but the false positive rate is too high for a brute-force search of the password space.

A better test requires some knowledge of the file format being wrapped, to obtain a "magic number" for the beginning of the file.

- The plaintext of system files begins with `$FL2@(#)` or `$FL3@(#)`.
- Before encryption, a syntax file is prefixed with a line at the beginning of the form `* Encoding: `*`encoding`*`.`, where *encoding* is the encoding used for the rest of the file, e.g. `windows-1252`. Thus, `* Encoding` may be used as a magic number for system files.
- The plaintext of viewer files begins with 50 4b 03 04 14 00 08 (50 4b is `PK`).

4.2 Password Encoding

SPSS also supports what it calls "encrypted passwords." These are not encrypted. They are encoded with a simple, fixed scheme. An encoded password is always a multiple of 2 characters long, and never longer than 20 characters. The characters in an encoded password are always in the graphic ASCII range 33 through 126. Each successive pair of characters in the password encodes a single byte in the plaintext password.

Use the following algorithm to decode a pair of characters:

1. Let *a* be the ASCII code of the first character, and *b* be the ASCII code of the second character.
2. Let *ah* be the most significant 4 bits of *a*. Find the line in the table below that has *ah* on the left side. The right side of the line is a set of possible values for the most significant 4 bits of the decoded byte.

   ```
   2  ⇒ 2367
   3  ⇒ 0145
   47 ⇒ 89cd
   56 ⇒ abef
   ```

3. Let *bh* be the most significant 4 bits of *b*. Find the line in the second table below that has *bh* on the left side. The right side of the line is a set of possible values for the most significant 4 bits of the decoded byte. Together with the results of the previous step, only a single possibility is left.

   ```
   2  ⇒ 139b
   3  ⇒ 028a
   47 ⇒ 46ce
   56 ⇒ 57df
   ```

4. Let *al* be the least significant 4 bits of *a*. Find the line in the table below that has *al* on the left side. The right side of the line is a set of possible values for the least significant 4 bits of the decoded byte.

   `03cf` ⇒ `0145`
   `12de` ⇒ `2367`
   `478b` ⇒ `89cd`
   `569a` ⇒ `abef`

5. Let *bl* be the least significant 4 bits of *b*. Find the line in the table below that has *bl* on the left side. The right side of the line is a set of possible values for the least significant 4 bits of the decoded byte. Together with the results of the previous step, only a single possibility is left.

   `03cf` ⇒ `028a`
   `12de` ⇒ `139b`
   `478b` ⇒ `46ce`
   `569a` ⇒ `57df`

Example

Consider the encoded character pair '`-|`'. *a* is 0x2d and *b* is 0x7c, so *ah* is 2, *bh* is 7, *al* is 0xd, and *bl* is 0xc. *ah* means that the most significant four bits of the decoded character is 2, 3, 6, or 7, and *bh* means that they are 4, 6, 0xc, or 0xe. The single possibility in common is 6, so the most significant four bits are 6. Similarly, *al* means that the least significant four bits are 2, 3, 6, or 7, and *bl* means they are 0, 2, 8, or 0xa, so the least significant four bits are 2. The decoded character is therefore 0x62, the letter '`b`'.

5 Portable File Format

These days, most computers use the same internal data formats for integer and floating-point data, if one ignores little differences like big- versus little-endian byte ordering. However, occasionally it is necessary to exchange data between systems with incompatible data formats. This is what portable files are designed to do.

The portable file format is mostly obsolete. System files (see Chapter 1 [System File Format], page 2) are a better alternative.

Please note: This information is gleaned from examination of ASCII-formatted portable files only, so some of it may be incorrect for portable files formatted in EBCDIC or other character sets.

5.1 Portable File Characters

Portable files are arranged as a series of lines of 80 characters each. Each line is terminated by a carriage-return, line-feed sequence ("new-lines"). New-lines are only used to avoid line length limits imposed by some OSes; they are not meaningful.

Most lines in portable files are exactly 80 characters long. The only exception is a line that ends in one or more spaces, in which the spaces may optionally be omitted. Thus, a portable file reader must act as though a line shorter than 80 characters is padded to that length with spaces.

The file must be terminated with a '`Z`' character. In addition, if the final line in the file does not have exactly 80 characters, then it is padded on the right with '`Z`' characters. (The file contents may be in any character set; the file contains a description of its own character set, as explained in the next section. Therefore, the '`Z`' character is not necessarily an ASCII '`Z`'.)

For the rest of the description of the portable file format, new-lines and the trailing '`Z`'s will be ignored, as if they did not exist, because they are not an important part of understanding the file contents.

5.2 Portable File Structure

Every portable file consists of the following records, in sequence:

- File header.
- Version and date info.
- Product identification.
- Author identification (optional).
- Subproduct identification (optional).
- Variable count.
- Case weight variable (optional).
- Variables. Each variable record may optionally be followed by a missing value record and a variable label record.
- Value labels (optional).
- Documents (optional).

- Data.

Most records are identified by a single-character tag code. The file header and version info record do not have a tag.

Other than these single-character codes, there are three types of fields in a portable file: floating-point, integer, and string. Floating-point fields have the following format:

- Zero or more leading spaces.
- Optional asterisk (‘*’), which indicates a missing value. The asterisk must be followed by a single character, generally a period (‘.’), but it appears that other characters may also be possible. This completes the specification of a missing value.
- Optional minus sign (‘-’) to indicate a negative number.
- A whole number, consisting of one or more base-30 digits: ‘0’ through ‘9’ plus capital letters ‘A’ through ‘T’.
- Optional fraction, consisting of a radix point (‘.’) followed by one or more base-30 digits.
- Optional exponent, consisting of a plus or minus sign (‘+’ or ‘-’) followed by one or more base-30 digits.
- A forward slash (‘/’).

Integer fields take a form identical to floating-point fields, but they may not contain a fraction.

String fields take the form of an integer field having value *n*, followed by exactly *n* characters, which are the string content.

5.3 Portable File Header

Every portable file begins with a 464-byte header, consisting of a 200-byte collection of vanity splash strings, followed by a 256-byte character set translation table, followed by an 8-byte tag string.

The 200-byte segment is divided into five 40-byte sections, each of which represents the string `charset SPSS PORT FILE` in a different character set encoding, where *charset* is the name of the character set used in the file, e.g. `ASCII` or `EBCDIC`. Each string is padded on the right with spaces in its respective character set.

It appears that these strings exist only to inform those who might view the file on a screen, and that they are not parsed by SPSS products. Thus, they can be safely ignored. For those interested, the strings are supposed to be in the following character sets, in the specified order: EBCDIC, 7-bit ASCII, CDC 6-bit ASCII, 6-bit ASCII, Honeywell 6-bit ASCII.

The 256-byte segment describes a mapping from the character set used in the portable file to an arbitrary character set having characters at the following positions:

0–60

Control characters. Not important enough to describe in full here.

61–63

Reserved.

64–73
: Digits '`0`' through '`9`'.

74–99
: Capital letters '`A`' through '`Z`'.

100–125
: Lowercase letters '`a`' through '`z`'.

126
: Space.

127–130
: Symbols `.<(+`

131
: Solid vertical pipe.

132–142
: Symbols `&[]!$*);^-/`

143
: Broken vertical pipe.

144–150
: Symbols `,%_>?`:`

151
: British pound symbol.

152–155
: Symbols `@'="`.

156
: Less than or equal symbol.

157
: Empty box.

158
: Plus or minus.

159
: Filled box.

160
: Degree symbol.

161
: Dagger.

162
: Symbol '`~`'.

163
: En dash.

164
: Lower left corner box draw.

165
: Upper left corner box draw.

166
: Greater than or equal symbol.

167–176
: Superscript '0' through '9'.

177
: Lower right corner box draw.

178
: Upper right corner box draw.

179
: Not equal symbol.

180
: Em dash.

181
: Superscript '('.

182
: Superscript ')'.

183
: Horizontal dagger (?).

184–186
: Symbols '{}\'.

187
: Cents symbol.

188
: Centered dot, or bullet.

189–255
: Reserved.

Symbols that are not defined in a particular character set are set to the same value as symbol 64; i.e., to '0'.

The 8-byte tag string consists of the exact characters `SPSSPORT` in the portable file's character set, which can be used to verify that the file is indeed a portable file.

5.4 Version and Date Info Record

This record does not have a tag code. It has the following structure:

- A single character identifying the file format version. The letter A represents version 0, and so on.
- An 8-character string field giving the file creation date in the format YYYYMMDD.
- A 6-character string field giving the file creation time in the format HHMMSS.

5.5 Identification Records

The product identification record has tag code '1'. It consists of a single string field giving the name of the product that wrote the portable file.

The author identification record has tag code '2'. It is optional. If present, it consists of a single string field giving the name of the person who caused the portable file to be written.

The subproduct identification record has tag code '3'. It is optional. If present, it consists of a single string field giving additional information on the product that wrote the portable file.

5.6 Variable Count Record

The variable count record has tag code '4'. It consists of a single integer field giving the number of variables in the file dictionary.

5.7 Precision Record

The precision record has tag code '5'. It consists of a single integer field specifying the maximum number of base-30 digits used in data in the file.

5.8 Case Weight Variable Record

The case weight variable record is optional. If it is present, it indicates the variable used for weighting cases; if it is absent, cases are unweighted. It has tag code '6'. It consists of a single string field that names the weighting variable.

5.9 Variable Records

Each variable record represents a single variable. Variable records have tag code '7'. They have the following structure:

- Width (integer). This is 0 for a numeric variable, and a number between 1 and 255 for a string variable.
- Name (string). 1–8 characters long. Must be in all capitals.

  A few portable files that contain duplicate variable names have been spotted in the wild. PSPP handles these by renaming the duplicates with numeric extensions: *var*_1, *var*_2, and so on.
- Print format. This is a set of three integer fields:
  - Format type (see Section 1.3 [Variable Record], page 6).

- Format width. 1–40.
- Number of decimal places. 1–40.

A few portable files with invalid format types or formats that are not of the appropriate width for their variables have been spotted in the wild. PSPP assigns a default F or A format to a variable with an invalid format.

- Write format. Same structure as the print format described above.

Each variable record can optionally be followed by a missing value record, which has tag code '`8`'. A missing value record has one field, the missing value itself (a floating-point or string, as appropriate). Up to three of these missing value records can be used.

There is also a record for missing value ranges, which has tag code '`B`'. It is followed by two fields representing the range, which are floating-point or string as appropriate. If a missing value range is present, it may be followed by a single missing value record.

Tag codes '`9`' and '`A`' represent `LO THRU` *x* and *x* `THRU HI` ranges, respectively. Each is followed by a single field representing *x*. If one of the ranges is present, it may be followed by a single missing value record.

In addition, each variable record can optionally be followed by a variable label record, which has tag code '`C`'. A variable label record has one field, the variable label itself (string).

5.10 Value Label Records

Value label records have tag code '`D`'. They have the following format:

- Variable count (integer).
- List of variables (strings). The variable count specifies the number in the list. Variables are specified by their names. All variables must be of the same type (numeric or string), but string variables do not necessarily have the same width.
- Label count (integer).
- List of (value, label) tuples. The label count specifies the number of tuples. Each tuple consists of a value, which is numeric or string as appropriate to the variables, followed by a label (string).

A few portable files that specify duplicate value labels, that is, two different labels for a single value of a single variable, have been spotted in the wild. PSPP uses the last value label specified in these cases.

5.11 Document Record

One document record may optionally follow the value label record. The document record consists of tag code '`E`', following by the number of document lines as an integer, followed by that number of strings, each of which represents one document line. Document lines must be 80 bytes long or shorter.

5.12 Portable File Data

The data record has tag code '`F`'. There is only one tag for all the data; thus, all the data must follow the dictionary. The data is terminated by the end-of-file marker '`Z`', which is not valid as the beginning of a data element.

Data elements are output in the same order as the variable records describing them. String variables are output as string fields, and numeric variables are output as floating-point fields.

6 SPSS/PC+ System File Format

SPSS/PC+, first released in 1984, was a simplified version of SPSS for IBM PC and compatible computers. It used a data file format related to the one described in the previous chapter, but simplified and incompatible. The SPSS/PC+ software became obsolete in the 1990s, so files in this format are rarely encountered today. Nevertheless, for completeness, and because it is not very difficult, it seems worthwhile to support at least reading these files. This chapter documents this format, based on examination of a corpus of about 60 files from a variety of sources.

System files use four data types: 8-bit characters, 16-bit unsigned integers, 32-bit unsigned integers, and 64-bit floating points, called here `char`, `uint16`, `uint32`, and `flt64`, respectively. Data is not necessarily aligned on a word or double-word boundary.

SPSS/PC+ ran only on IBM PC and compatible computers. Therefore, values in these files are always in little-endian byte order. Floating-point numbers are always in IEEE 754 format.

SPSS/PC+ system files represent the system-missing value as -1.66e308, or `f5 1e 26 02 8a 8c ed ff` expressed as hexadecimal. (This is an unusual choice: it is close to, but not equal to, the largest negative 64-bit IEEE 754, which is about -1.8e308.)

Text in SPSS/PC+ system file is encoded in ASCII-based 8-bit MS DOS codepages. The corpus used for investigating the format were all ASCII-only.

An SPSS/PC+ system file begins with the following 256-byte directory:

```
uint32              two;
uint32              zero;
struct {
    uint32          ofs;
    uint32          len;
} records[15];
char                filename[128];
```

`uint32 two;`
`uint32 zero;`

Always set to 2 and 0, respectively.

These fields could be used as a signature for the file format, but the `product` field in record 0 seems more likely to be unique (see Section 6.1 [Record 0 Main Header Record], page 100).

`struct { ... } records[15];`

Each of the elements in this array identifies a record in the system file. The `ofs` is a byte offset, from the beginning of the file, that identifies the start of the record. `len` specifies the length of the record, in bytes. Many records are optional or not used. If a record is not present, `ofs` and `len` for that record are both are zero.

`char filename[128];`

In most files in the corpus, this field is entirely filled with spaces. In one file, it contains a file name, followed by a null bytes, followed by spaces to fill the remainder of the field. The meaning is unknown.

The following sections describe the contents of each record, identified by the index into the `records` array.

6.1 Record 0: Main Header Record

All files in the corpus have this record at offset 0x100 with length 0xb0 (but readers should find this record, like the others, via the `records` table in the directory). Its format is:

```
uint16          one0;
char            product[62];
flt64           sysmis;
uint32          zero0;
uint32          zero1;
uint16          one1;
uint16          compressed;
uint16          nominal_case_size;
uint16          n_cases0;
uint16          weight_index;
uint16          zero2;
uint16          n_cases1;
uint16          zero3;
char            creation_date[8];
char            creation_time[8];
char            label[64];
```

`uint16 one0;`
`uint16 one1;`
: Always set to 1.

`uint32 zero0;`
`uint32 zero1;`
`uint16 zero2;`
`uint16 zero3;`
: Always set to 0.

: It seems likely that one of these variables is set to 1 if weighting is enabled, but none of the files in the corpus is weighted.

`char product[62];`
: Name of the program that created the file. Only the following unique values have been observed, in each case padded on the right with spaces:

```
DESPSS/PC+ System File Written by Data Entry II
PCSPSS SYSTEM FILE.  IBM PC DOS, SPSS/PC+
PCSPSS SYSTEM FILE.  IBM PC DOS, SPSS/PC+ V3.0
PCSPSS SYSTEM FILE.  IBM PC DOS, SPSS for Windows
```

: Thus, it is reasonable to use the presence of the string '`SPSS`' at offset 0x104 as a simple test for an SPSS/PC+ data file.

`flt64 sysmis;`
: The system-missing value, as described previously (see Chapter 6 [SPSS/PC+ System File Format], page 99).

`uint16 compressed;`
: Set to 0 if the data in the file is not compressed, 1 if the data is compressed with simple bytecode compression.

`uint16 nominal_case_size;`
: Number of data elements per case. This is the number of variables, except that long string variables add extra data elements (one for every 8 bytes after the first 8). String variables in SPSS/PC+ system files are limited to 255 bytes.

`uint16 n_cases0;`
`uint16 n_cases1;`
: The number of cases in the data record. Both values are the same. Some files in the corpus contain data for the number of cases noted here, followed by garbage that somewhat resembles data.

`uint16 weight_index;`
: 0, if the file is unweighted, otherwise a 1-based index into the data record of the weighting variable, e.g. 4 for the first variable after the 3 system-defined variables.

`char creation_date[8];`
: The date that the file was created, in '`mm/dd/yy`' format. Single-digit days and months are not prefixed by zeros. The string is padded with spaces on right or left or both, e.g. '`_2/4/93_`', '`10/5/87_`', and '`_1/11/88`' (with '`_`' standing in for a space) are all actual examples from the corpus.

`char creation_time[8];`
: The time that the file was created, in '`HH:MM:SS`' format. Single-digit hours are padded on a left with a space. Minutes and seconds are always written as two digits.

`char file_label[64];`
: File label declared by the user, if any (see Section "FILE LABEL" in *PSPP Users Guide*). Padded on the right with spaces.

6.2 Record 1: Variables Record

The variables record most commonly starts at offset 0x1b0, but it can be placed elsewhere. The record contains instances of the following 32-byte structure:

```
uint32              value_label_start;
uint32              value_label_end;
uint32              var_label_ofs;
uint32              format;
char                name[8];
union {
    flt64           f;
    char            s[8];
} missing;
```

The number of instances is the `nominal_case_size` specified in the main header record. There is one instance for each numeric variable and each string variable with width 8 bytes or less. String variables wider than 8 bytes have one instance for each 8 bytes, rounding up. The first instance for a long string specifies the variable's correct dictionary information. Subsequent instances for a long string are generally filled with all-zero bytes, although the `missing` field contains the numeric system-missing value, and some writers also fill in `var_label_ofs`, `format`, and `name`, sometimes filling the latter with the numeric system-missing value rather than a text string. Regardless of the values used, readers should ignore the contents of these additional instances for long strings.

`uint32 value_label_start;`
`uint32 value_label_end;`

For a variable with value labels, these specify offsets into the label record of the start and end of this variable's value labels, respectively. See Section 6.3 [Record 2 Labels Record], page 103, for more information.

For a variable without any value labels, these are both zero.

A long string variable may not have value labels.

`uint32 var_label_ofs;`

For a variable with a variable label, this specifies an offset into the label record. See Section 6.3 [Record 2 Labels Record], page 103, for more information.

For a variable without a variable label, this is zero.

`uint32 format;`

The variable's output format, in the same format used in system files. See [System File Output Formats], page 8, for details. SPSS/PC+ system files only use format types 5 (F, for numeric variables) and 1 (A, for string variables).

`char name[8];`

The variable's name, padded on the right with spaces.

`union { ... } missing;`

A user-missing value. For numeric variables, `missing.f` is the variable's user-missing value. For string variables, `missing.s` is a string missing value. A variable without a user-missing value is indicated with `missing.f` set to the system-missing value, even for string variables (!). A Long string variable may not have a missing value.

In addition to the user-defined variables, every SPSS/PC+ system file contains, as its first three variables, the following system-defined variables, in the following order. The system-defined variables have no variable label, value labels, or missing values.

`$CASENUM` A numeric variable with format F8.0. Most of the time this is a sequence number, starting with 1 for the first case and counting up for each subsequent case. Some files skip over values, which probably reflects cases that were deleted.

`$DATE` A string variable with format A8. Same format (including varying padding) as the `creation_date` field in the main header record (see Section 6.1 [Record 0 Main Header Record], page 100). The actual date can differ from `creation_date` and from record to record. This may reflect when individual cases were added or updated.

`$WEIGHT` A numeric variable with format F8.2. This represents the case's weight; SPSS/PC+ files do not have a user-defined weighting variable. If weighting has not been enabled, every case has value 1.0.

6.3 Record 2: Labels Record

The labels record holds value labels and variable labels. Unlike the other records, it is not meant to be read directly and sequentially. Instead, this record must be interpreted one piece at a time, by following pointers from the variables record.

The `value_label_start`, `value_label_end`, and `var_label_ofs` fields in a variable record are all offsets relative to the beginning of the labels record, with an additional 7-byte offset. That is, if the labels record starts at byte offset `labels_ofs` and a variable has a given `var_label_ofs`, then the variable label begins at byte offset `labels_ofs` + `var_label_ofs` + 7 in the file.

A variable label, starting at the offset indicated by `var_label_ofs`, consists of a one-byte length followed by the specified number of bytes of the variable label string, like this:

```
uint8               length;
char                s[length];
```

A set of value labels, extending from `value_label_start` to `value_label_end` (exclusive), consists of a numeric or string value followed by a string in the format just described. String values are padded on the right with spaces to fill the 8-byte field, like this:

```
union {
    flt64           f;
    char            s[8];
} value;
uint8               length;
char                s[length];
```

The labels record begins with a pair of uint32 values. The first of these is always 3. The second is between 8 and 16 less than the number of bytes in the record. Neither value is important for interpreting the file.

6.4 Record 3: Data Record

The format of the data record varies depending on the value of `compressed` in the file header record:

0: no compression
: Data is arranged as a series of 8-byte elements, one per variable instance variable in the variable record (see Section 6.2 [Record 1 Variables Record], page 101). Numeric values are given in `flt64` format; string values are literal characters string, padded on the right with spaces when necessary to fill out 8-byte units.

1: bytecode compression
: The first 8 bytes of the data record is divided into a series of 1-byte command codes. These codes have meanings as described below:

  0 The system-missing value.

1
: A numeric or string value that is not compressible. The value is stored in the 8 bytes following the current block of command bytes. If this value appears twice in a block of command bytes, then it indicates the second group of 8 bytes following the command bytes, and so on.

2 through 255
: A number with value *code* - 100, where *code* is the value of the compression code. For example, code 105 indicates a numeric variable of value 5.

The end of the 8-byte group of bytecodes is followed by any 8-byte blocks of non-compressible values indicated by code 1. After that follows another 8-byte group of bytecodes, then those bytecodes' non-compressible values. The pattern repeats up to the number of cases specified by the main header record have been seen.

The corpus does not contain any files with command codes 2 through 95, so it is possible that some of these codes are used for special purposes.

Cases of data often, but not always, fill the entire data record. Readers should stop reading after the number of cases specified in the main header record. Otherwise, readers may try to interpret garbage following the data as additional cases.

6.5 Records 4 and 5: Data Entry

Records 4 and 5 appear to be related to SPSS/PC+ Data Entry.

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.