PSPP Developers Guide

GNU PSPP Statistical Analysis Software
Release 0.8.5

Ben Pfaff
John Darrington

This manual is for GNU PSPP version 0.8.5, software for statistical analysis.

Table of Contents

1 Introduction

This manual is a guide to PSPP internals. Its intended audience is developers who wish to modify or extend PSPP's capabilities. The use of PSPP is documented in a separate manual. See Section "Introduction" in *PSPP Users Guide*.

This manual is both a tutorial and a reference manual for PSPP developers. It is ultimately intended to cover everything that developers who wish to implement new PSPP statistical procedures and other commands should know. It is currently incomplete, partly because existing developers have not yet spent enough time on writing, and partly because the interfaces not yet documented are not yet mature enough to making documenting them worthwhile.

PSPP developers should have some familiarity with the basics of PSPP from a user's perspective. This manual attempts to refer to the PSPP user manual's descriptions of concepts that PSPP users should find familiar at the time of their first reference. However, it is probably a good idea to at least skim the PSPP manual before reading this one, if you are not already familiar with PSPP.

2 Basic Concepts

This chapter introduces basic data structures and other concepts needed for developing in PSPP.

2.1 Values

The unit of data in PSPP is a *value*.

Values are classified by *type* and *width*. The type of a value is either *numeric* or *string* (sometimes called alphanumeric). The width of a string value ranges from 1 to `MAX_STRING` bytes. The width of a numeric value is artificially defined to be 0; thus, the type of a value can be inferred from its width.

Some support is provided for working with value types and widths, in `data/val-type.h`:

`int MAX_STRING` [Macro]
: Maximum width of a string value, in bytes, currently 32,767.

`bool val_type_is_valid` (*enum val_type* `val_type`) [Function]
: Returns true if *val_type* is a valid value type, that is, either `VAL_NUMERIC` or `VAL_STRING`. Useful for assertions.

`enum val_type val_type_from_width` (*int* `width`) [Function]
: Returns `VAL_NUMERIC` if *width* is 0 and thus represents the width of a numeric value, otherwise `VAL_STRING` to indicate that *width* is the width of a string value.

The following subsections describe how values of each type are represented.

2.1.1 Numeric Values

A value known to be numeric at compile time is represented as a `double`. PSPP provides three values of `double` for special purposes, defined in `data/val-type.h`:

`double SYSMIS` [Macro]
: The *system-missing value*, used to represent a datum whose true value is unknown, such as a survey question that was not answered by the respondent, or undefined, such as the result of division by zero. PSPP propagates the system-missing value through calculations and compensates for missing values in statistical analyses. See Section "Missing Observations" in *PSPP Users Guide*, for a PSPP user's view of missing values.

  PSPP currently defines `SYSMIS` as `-DBL_MAX`, that is, the greatest finite negative value of `double`. It is best not to depend on this definition, because PSPP may transition to using an IEEE NaN (not a number) instead at some point in the future.

`double LOWEST` [Macro]
`double HIGHEST` [Macro]
: The greatest finite negative (except for `SYSMIS`) and positive values of `double`, respectively. These values do not ordinarily appear in user data files. Instead, they are used to implement endpoints of open-ended ranges that are occasionally permitted in PSPP syntax, e.g. `5 THRU HI` as a range of missing values (see Section "MISSING VALUES" in *PSPP Users Guide*).

2.1.2 String Values

A value known at compile time to have string type is represented as an array of `char`. String values do not necessarily represent readable text strings and may contain arbitrary 8-bit data, including null bytes, control codes, and bytes with the high bit set. Thus, string values are not null-terminated strings, but rather opaque arrays of bytes.

`SYSMIS`, `LOWEST`, and `HIGHEST` have no equivalents as string values. Usually, PSPP fills an unknown or undefined string values with spaces, but PSPP does not treat such a string as a special case when it processes it later.

`MAX_STRING`, the maximum length of a string value, is defined in `data/val-type.h`.

2.1.3 Runtime Typed Values

When a value's type is only known at runtime, it is often represented as a `union value`, defined in `data/value.h`. A `union value` does not identify the type or width of the data it contains. Code that works with `union values`s must therefore have external knowledge of its content, often through the type and width of a `struct variable` (see Section 2.5 [Variables], page 18).

`union value` has one member that clients are permitted to access directly, a `double` named '`f`' that stores the content of a numeric `union value`. It has other members that store the content of string `union value`, but client code should use accessor functions instead of referring to these directly.

PSPP provides some functions for working with `union value`s. The most useful are described below. To use these functions, recall that a numeric value has a width of 0.

`void value_init` (*union value* *`*value`*, *int* *`width`*) [Function]

Initializes *value* as a value of the given *width*. After initialization, the data in *value* are indeterminate; the caller is responsible for storing initial data in it.

`void value_destroy` (*union value* *`*value`*, *int* *`width`*) [Function]

Frees auxiliary storage associated with *value*, which must have the given *width*.

`bool value_needs_init` (*int* *`width`*) [Function]

For some widths, `value_init` and `value_destroy` do not actually do anything, because no additional storage is needed beyond the size of `union value`. This function returns true if *width* is such a width, which case there is no actual need to call those functions. This can be a useful optimization if a large number of `union value`s of such a width are to be initialized or destroyed.

This function returns false if `value_init` and `value_destroy` are actually required for the given *width*.

`double value_num` (*const union value* *`*value`*) [Function]

Returns the numeric value in *value*, which must have been initialized as a numeric value. Equivalent to *`value`*`->f`.

`const char * value_str` (*const union value* *`*value`*, *int* *`width`*) [Function]
`char * value_str_rw` (*union value* *`*value`*, *int* *`width`*) [Function]

Returns the string value in *value*, which must have been initialized with positive width *width*. The string returned is not null-terminated. Only *width* bytes of returned data may be accessed.

The two different functions exist only for `const`-correctness. Otherwise they are identical.

It is important that *width* be the correct value that was passed to `value_init`. Passing a smaller or larger value (e.g. because that number of bytes will be accessed) will not always work and should be avoided.

`void value_copy` (*union value* *`dst`, *const union value* *`src`, *int* `width`) [Function]

Copies the contents of `union value` *src* to *dst*. Both *dst* and *src* must have been initialized with the specified *width*.

`void value_set_missing` (*union value* *`value`, *int* `width`) [Function]

Sets *value* to `SYSMIS` if it is numeric or to all spaces if it is alphanumeric, according to *width*. *value* must have been initialized with the specified *width*.

`bool value_is_resizable` (*const union value* *`value`, *int* `old_width`, *int* `new_width`) [Function]

Determines whether *value*, which must have been initialized with the specified *old_width*, may be resized to *new_width*. Resizing is possible if the following criteria are met. First, *old_width* and *new_width* must be both numeric or both string widths. Second, if *new_width* is a short string width and less than *old_width*, resizing is allowed only if bytes *new_width* through *old_width* in *value* contain only spaces.

These rules are part of those used by `mv_is_resizable` and `val_labs_can_set_width`.

`void value_resize` (*union value* *`value`, *int* `old_width`, *int* `new_width`) [Function]

Resizes *value* from *old_width* to *new_width*, which must be allowed by the rules stated above. *value* must have been initialized with the specified *old_width* before calling this function. After resizing, *value* has width *new_width*.

If *new_width* is greater than *old_width*, *value* will be padded on the right with spaces to the new width. If *new_width* is less than *old_width*, the rightmost bytes of *value* are truncated.

`bool value_equal` (*const union value* *`a`, *const union value* *`b`, *int* `width`) [Function]

Compares of *a* and *b*, which must both have width *width*. Returns true if their contents are the same, false if they differ.

`int value_compare_3way` (*const union value* *`a`, *const union value* *`b`, *int* `width`) [Function]

Compares of *a* and *b*, which must both have width *width*. Returns -1 if *a* is less than *b*, 0 if they are equal, or 1 if *a* is greater than *b*.

Numeric values are compared numerically, with `SYSMIS` comparing less than any real number. String values are compared lexicographically byte-by-byte.

`size_t value_hash` (*const union value* *`value`, *int* `width`, *unsigned int* `basis`) [Function]

Computes and returns a hash of *value*, which must have the specified *width*. The value in *basis* is folded into the hash.

2.2 Input and Output Formats

Input and output formats specify how to convert data fields to and from data values (see Section "Input and Output Formats" in *PSPP Users Guide*). PSPP uses `struct fmt_spec` to represent input and output formats.

Function prototypes and other declarations related to formats are in the `<data/format.h>` header.

`struct fmt_spec` [Structure]

An input or output format, with the following members:

`enum fmt_type type`
: The format type (see below).

`int w`
: Field width, in bytes. The width of numeric fields is always between 1 and 40 bytes, and the width of string fields is always between 1 and 65534 bytes. However, many individual types of formats place stricter limits on field width (see [fmt_max_input_width], page 7, [fmt_max_output_width], page 7).

`int d`
: Number of decimal places, in character positions. For format types that do not allow decimal places to be specified, this value must be 0. Format types that do allow decimal places have type-specific and often width-specific restrictions on `d` (see [fmt_max_input_decimals], page 7, [fmt_max_output_decimals], page 7).

`enum fmt_type` [Enumeration]

An enumerated type representing an input or output format type. Each PSPP input and output format has a corresponding enumeration constant prefixed by '`FMT`': `FMT_F`, `FMT_COMMA`, `FMT_DOT`, and so on.

The following sections describe functions for manipulating formats and the data in fields represented by formats.

2.2.1 Constructing and Verifying Formats

These functions construct `struct fmt_spec`s and verify that they are valid.

`struct fmt_spec fmt_for_input` (*enum fmt_type* `type`, *int* `w`, *int* `d`) [Function]
`struct fmt_spec fmt_for_output` (*enum fmt_type* `type`, *int* `w`, *int* `d`) [Function]

Constructs a `struct fmt_spec` with the given *type*, *w*, and *d*, asserts that the result is a valid input (or output) format, and returns it.

`struct fmt_spec fmt_for_output_from_input` (*const struct fmt_spec* *`input`) [Function]

Given *input*, which must be a valid input format, returns the equivalent output format. See Section "Input and Output Formats" in *PSPP Users Guide*, for the rules for converting input formats into output formats.

`struct fmt_spec fmt_default_for_width` (*int* `width`) [Function]

Returns the default output format for a variable of the given *width*. For a numeric variable, this is F8.2 format; for a string variable, it is the A format of the given *width*.

The following functions check whether a `struct fmt_spec` is valid for various uses and return true if so, false otherwise. When any of them returns false, it also outputs an explanatory error message using `msg`. To suppress error output, enclose a call to one of these functions by a `msg_disable`/`msg_enable` pair.

`bool fmt_check` (*const struct fmt_spec* ***format***, *bool* ***for_input***) [Function]
`bool fmt_check_input` (*const struct fmt_spec* ***format***) [Function]
`bool fmt_check_output` (*const struct fmt_spec* ***format***) [Function]

Checks whether *format* is a valid input format (for `fmt_check_input`, or `fmt_check` if *for_input*) or output format (for `fmt_check_output`, or `fmt_check` if not *for_input*).

`bool fmt_check_type_compat` (*const struct fmt_spec* ***format***, *enum val_type* ***type***) [Function]

Checks whether *format* matches the value type *type*, that is, if *type* is `VAL_NUMERIC` and *format* is a numeric format or *type* is `VAL_STRING` and *format* is a string format.

`bool fmt_check_width_compat` (*const struct fmt_spec* ***format***, *int* ***width***) [Function]

Checks whether *format* may be used as an output format for a value of the given *width*.

`fmt_var_width`, described in the following section, can be also be used to determine the value width needed by a format.

2.2.2 Format Utility Functions

These functions work with `struct fmt_spec`s.

`int fmt_var_width` (*const struct fmt_spec* ***format***) [Function]

Returns the width for values associated with *format*. If *format* is a numeric format, the width is 0; if *format* is an A format, then the width `format->w`; otherwise, *format* is an AHEX format and its width is `format->w / 2`.

`char *fmt_to_string` (*const struct fmt_spec* ***format***, *char* ***s****[FMT_STRING_LEN_MAX + 1]*) [Function]

Converts *format* to a human-readable format specifier in *s* and returns *s*. *format* need not be a valid input or output format specifier, e.g. it is allowed to have an excess width or decimal places. In particular, if *format* has decimals, they are included in the output string, even if *format*'s type does not allow decimals, to allow accurately presenting incorrect formats to the user.

`bool fmt_equal` (*const struct fmt_spec* ***a***, *const struct fmt_spec* ***b***) [Function]

Compares *a* and *b* memberwise and returns true if they are identical, false otherwise. *format* need not be a valid input or output format specifier.

`void fmt_resize` (*struct fmt_spec* ***fmt***, *int* ***width***) [Function]

Sets the width of *fmt* to a valid format for a `union value` of size *width*.

2.2.3 Obtaining Properties of Format Types

These functions work with `enum fmt_type`s instead of the higher-level `struct fmt_spec`s. Their primary purpose is to report properties of each possible format type, which in turn allows clients to abstract away many of the details of the very heterogeneous requirements of each format type.

The first group of functions works with format type names.

`const char` **fmt_name* (*enum fmt_type* ***type***) [Function]

> Returns the name for the given *type*, e.g. `"COMMA"` for `FMT_COMMA`.

`bool` **`fmt_from_name`** (*const char **`name`, *enum fmt_type **`type`) [Function]

> Tries to find the `enum fmt_type` associated with *name*. If successful, sets `*type` to the type and returns true; otherwise, returns false without modifying `*type`.

The functions below query basic limits on width and decimal places for each kind of format.

`bool fmt_takes_decimals` (*enum fmt_type* `type`) [Function]

> Returns true if a format of the given *type* is allowed to have a nonzero number of decimal places (the `d` member of `struct fmt_spec`), false if not.

`int fmt_min_input_width` (*enum fmt_type* `type`) [Function]
`int fmt_max_input_width` (*enum fmt_type* `type`) [Function]
`int fmt_min_output_width` (*enum fmt_type* `type`) [Function]
`int fmt_max_output_width` (*enum fmt_type* `type`) [Function]

> Returns the minimum or maximum width (the `w` member of `struct fmt_spec`) allowed for an input or output format of the specified *type*.

`int fmt_max_input_decimals` (*enum fmt_type* `type`, *int* `width`) [Function]
`int fmt_max_output_decimals` (*enum fmt_type* `type`, *int* `width`) [Function]

> Returns the maximum number of decimal places allowed for an input or output format, respectively, of the given *type* and *width*. Returns 0 if the specified *type* does not allow any decimal places or if *width* is too narrow to allow decimal places.

`int fmt_step_width` (*enum fmt_type* `type`) [Function]

> Returns the "width step" for a `struct fmt_spec` of the given *type*. A `struct fmt_spec`'s width must be a multiple of its type's width step. Most format types have a width step of 1, so that their formats' widths may be any integer within the valid range, but hexadecimal numeric formats and AHEX string formats have a width step of 2.

These functions allow clients to broadly determine how each kind of input or output format behaves.

`bool fmt_is_string` (*enum fmt_type* `type`) [Function]
`bool fmt_is_numeric` (*enum fmt_type* `type`) [Function]

> Returns true if *type* is a format for numeric or string values, respectively, false otherwise.

enum fmt_category *fmt_get_category* (*enum fmt_type* **type**) [Function]

Returns the category within which *type* falls.

enum fmt_category [Enumeration]

A group of format types. Format type categories correspond to the input and output categories described in the PSPP user documentation (see Section "Input and Output Formats" in *PSPP Users Guide*).

Each format is in exactly one category. The categories have bitwise disjoint values to make it easy to test whether a format type is in one of multiple categories, e.g.

```
if (fmt_get_category (type) & (FMT_CAT_DATE | FMT_CAT_TIME))
  {
    /* ...type is a date or time format... */
  }
```

The format categories are:

`FMT_CAT_BASIC`
: Basic numeric formats.

`FMT_CAT_CUSTOM`
: Custom currency formats.

`FMT_CAT_LEGACY`
: Legacy numeric formats.

`FMT_CAT_BINARY`
: Binary formats.

`FMT_CAT_HEXADECIMAL`
: Hexadecimal formats.

`FMT_CAT_DATE`
: Date formats.

`FMT_CAT_TIME`
: Time formats.

`FMT_CAT_DATE_COMPONENT`
: Date component formats.

`FMT_CAT_STRING`
: String formats.

The PSPP input and output routines use the following pair of functions to convert `enum fmt_type`s to and from the separate set of codes used in system and portable files:

int fmt_to_io (*enum fmt_type* **type**) [Function]

Returns the format code used in system and portable files that corresponds to *type*.

bool fmt_from_io (*int* **io**, *enum fmt_type* ***type**) [Function]

Converts *io*, a format code used in system and portable files, into a `enum fmt_type` in `*type`. Returns true if successful, false if *io* is not valid.

These functions reflect the relationship between input and output formats.

`enum fmt_type` *fmt_input_to_output* (*enum fmt_type* `type`) [Function]

Returns the output format type that is used by default by DATA LIST and other input procedures when *type* is specified as an input format. The conversion from input format to output format is more complicated than simply changing the format. See [fmt_for_output_from_input], page 5, for a function that performs the entire conversion.

`bool fmt_usable_for_input` (*enum fmt_type* `type`) [Function]

Returns true if *type* may be used as an input format type, false otherwise. The custom currency formats, in particular, may be used for output but not for input.

All format types are valid for output.

The final group of format type property functions obtain human-readable templates that illustrate the formats graphically.

`const char` **fmt_date_template* (*enum fmt_type* `type`) [Function]

Returns a formatting template for *type*, which must be a date or time format type. These formats are used by `data_in` and `data_out` to guide parsing and formatting date and time data.

`char *fmt_dollar_template` (*const struct fmt_spec **`format`) [Function]

Returns a string of the form `$#,###.##` according to *format*, which must be of type `FMT_DOLLAR`. The caller must free the string with `free`.

2.2.4 Numeric Formatting Styles

Each of the basic numeric formats (F, E, COMMA, DOT, DOLLAR, PCT) and custom currency formats (CCA, CCB, CCC, CCD, CCE) has an associated numeric formatting style, represented by `struct fmt_number_style`. Input and output conversion of formats that have numeric styles is determined mainly by the style, although the formatting rules have special cases that are not represented within the style.

`struct fmt_number_style` [Structure]

A structure type with the following members:

`struct substring neg_prefix`
`struct substring prefix`
`struct substring suffix`
`struct substring neg_suffix`

A set of strings used a prefix to negative numbers, a prefix to every number, a suffix to every number, and a suffix to negative numbers, respectively. Each of these strings is no more than `FMT_STYLE_AFFIX_MAX` bytes (currently 16) bytes in length. These strings must be freed with `ss_dealloc` when no longer needed.

`decimal` The character used as a decimal point. It must be either '`.`' or '`,`'.

`grouping` The character used for grouping digits to the left of the decimal point. It may be '`.`' or '`,`', in which case it must not be equal to `decimal`, or it may be set to 0 to disable grouping.

The following functions are provided for working with numeric formatting styles.

`void fmt_number_style_init` (*struct fmt_number_style* ***style***) [Function]

Initialises a `struct fmt_number_style` with all of the prefixes and suffixes set to the empty string, '`.`' as the decimal point character, and grouping disables.

`void fmt_number_style_destroy` (*struct fmt_number_style* ***style***) [Function]

Destroys *style*, freeing its storage.

`struct fmt_number_style *fmt_create` (*void*) [Function]

A function which creates an array of all the styles used by pspp, and calls fmt_number_style_init on each of them.

`void fmt_done` (*struct fmt_number_style* ***styles***) [Function]

A wrapper function which takes an array of `struct fmt_number_style`, calls fmt_number_style_destroy on each of them, and then frees the array.

`int fmt_affix_width` (*const struct fmt_number_style* ***style***) [Function]

Returns the total length of *style*'s `prefix` and `suffix`.

`int fmt_neg_affix_width` (*const struct fmt_number_style* ***style***) [Function]

Returns the total length of *style*'s `neg_prefix` and `neg_suffix`.

PSPP maintains a global set of number styles for each of the basic numeric formats and custom currency formats. The following functions work with these global styles:

`const struct fmt_number_style * fmt_get_style` (*enum fmt_type* ***type***) [Function]

Returns the numeric style for the given format *type*.

`const char * fmt_name` (*enum fmt_type* ***type***) [Function]

Returns the name of the given format *type*.

2.2.5 Formatted Data Input and Output

These functions provide the ability to convert data fields into `union value`s and vice versa.

`bool data_in` (*struct substring* ***input***, *const char* ***encoding***, *enum fmt_type* ***type***, *int* ***implied_decimals***, *int* ***first_column***, *const struct dictionary* ***dict***, *union value* ***output***, *int* ***width***) [Function]

Parses *input* as a field containing data in the given format *type*. The resulting value is stored in *output*, which the caller must have initialized with the given *width*. For consistency, *width* must be 0 if *type* is a numeric format type and greater than 0 if *type* is a string format type. *encoding* should be set to indicate the character encoding of *input*. *dict* must be a pointer to the dictionary with which *output* is associated.

If *input* is the empty string (with length 0), *output* is set to the value set on SET BLANKS (see Section "SET BLANKS" in *PSPP Users Guide*) for a numeric value, or to all spaces for a string value. This applies regardless of the usual parsing requirements for *type*.

If *implied_decimals* is greater than zero, then the numeric result is shifted right by *implied_decimals* decimal places if *input* does not contain a decimal point character or an exponent. Only certain numeric format types support implied decimal places; for string formats and other numeric formats, *implied_decimals* has no effect. DATA LIST FIXED is the primary user of this feature (see Section "DATA LIST FIXED" in *PSPP Users Guide*). Other callers should generally specify 0 for *implied_decimals*, to disable this feature.

When *input* contains invalid input data, `data_in` outputs a message using `msg`. If *first_column* is nonzero, it is included in any such error message as the 1-based column number of the start of the field. The last column in the field is calculated as *first_column* + *input* − 1. To suppress error output, enclose the call to `data_in` by calls to `msg_disable` and `msg_enable`.

This function returns true on success, false if a message was output (even if suppressed). Overflow and underflow provoke warnings but are not propagated to the caller as errors.

This function is declared in `data/data-in.h`.

`char *` *data_out* (*const union value* *`*input`*, *const struct fmt_spec* *`*format`*) [Function]

`char *` *data_out_legacy* (*const union value* *`*input`*, *const char* *`*encoding`*, *const struct fmt_spec* *`*format`*) [Function]

Converts the data pointed to by *input* into a string value, which will be encoded in UTF-8, according to output format specifier *format*. Format must be a valid output format. The width of *input* is inferred from *format* using an algorithm equivalent to `fmt_var_width`.

When *input* contains data that cannot be represented in the given *format*, `data_out` may output a message using `msg`, although the current implementation does not consistently do so. To suppress error output, enclose the call to `data_out` by calls to `msg_disable` and `msg_enable`.

This function is declared in `data/data-out.h`.

2.3 User-Missing Values

In addition to the system-missing value for numeric values, each variable has a set of user-missing values (see Section "MISSING VALUES" in *PSPP Users Guide*). A set of user-missing values is represented by `struct missing_values`.

It is rarely necessary to interact directly with a `struct missing_values` object. Instead, the most common operation, querying whether a particular value is a missing value for a given variable, is most conveniently executed through functions on `struct variable`. See Section 2.5.3 [Variable Missing Values], page 19, for details.

A `struct missing_values` is essentially a set of `union value`s that have a common value width (see Section 2.1 [Values], page 2). For a set of missing values associated with a variable (the common case), the set's width is the same as the variable's width.

Function prototypes and other declarations related to missing values are declared in `data/missing-values.h`.

`struct missing_values` [Structure]

Opaque type that represents a set of missing values.

The contents of a set of missing values is subject to some restrictions. Regardless of width, a set of missing values is allowed to be empty. A set of numeric missing values may contain up to three discrete numeric values, or a range of numeric values (which includes both ends of the range), or a range plus one discrete numeric value. A set of string missing values may contain up to three discrete string values (with the same width as the set), but ranges are not supported.

In addition, values in string missing values wider than `MV_MAX_STRING` bytes may contain non-space characters only in their first `MV_MAX_STRING` bytes; all the bytes after the first `MV_MAX_STRING` must be spaces. See [mv_is_acceptable], page 15, for a function that tests a value against these constraints.

`int MV_MAX_STRING` [Macro]

Number of bytes in a string missing value that are not required to be spaces. The current value is 8, a value which is fixed by the system file format. In PSPP we could easily eliminate this restriction, but doing so would also require us to extend the system file format in an incompatible way, which we consider a bad tradeoff.

The most often useful functions for missing values are those for testing whether a given value is missing, described in the following section. Several other functions for creating, inspecting, and modifying `struct missing_values` objects are described afterward, but these functions are much more rarely useful.

2.3.1 Testing for Missing Values

The most often useful functions for missing values are those for testing whether a given value is missing, described here. However, using one of the corresponding missing value testing functions for variables can be even easier (see Section 2.5.3 [Variable Missing Values], page 19).

`bool mv_is_value_missing` (*const struct missing_values* *`*mv`*, *const union value* *`*value`*, *enum mv_class* `class`) [Function]

`bool mv_is_num_missing` (*const struct missing_values* *`*mv`*, *double* `value`, *enum mv_class* `class`) [Function]

`bool mv_is_str_missing` (*const struct missing_values* *`*mv`*, *const char* `value`[], *enum mv_class* `class`) [Function]

Tests whether *value* is in one of the categories of missing values given by *class*. Returns true if so, false otherwise.

mv determines the width of *value* and provides the set of user-missing values to test.

The only difference among these functions in the form in which *value* is provided, so you may use whichever function is most convenient.

The *class* argument determines the exact kinds of missing values that the functions test for:

`enum mv_class` [Enumeration]

`MV_USER` Returns true if *value* is in the set of user-missing values given by *mv*.

`MV_SYSTEM`
: Returns true if *value* is system-missing. (If *mv* represents a set of string values, then *value* is never system-missing.)

`MV_ANY`
`MV_USER | MV_SYSTEM`
: Returns true if *value* is user-missing or system-missing.

`MV_NONE`
: Always returns false, that is, *value* is never considered missing.

2.3.2 Creation and Destruction

These functions create and destroy `struct missing_values` objects.

`void mv_init` (*struct missing_values* `*mv`, *int* `width`) [Function]
: Initializes *mv* as a set of user-missing values. The set is initially empty. Any values added to it must have the specified *width*.

`void mv_destroy` (*struct missing_values* `*mv`) [Function]
: Destroys *mv*, which must not be referred to again.

`void mv_copy` (*struct missing_values* `*mv`, *const struct missing_values* `*old`) [Function]
: Initializes *mv* as a copy of the existing set of user-missing values *old*.

`void mv_clear` (*struct missing_values* `*mv`) [Function]
: Empties the user-missing value set *mv*, retaining its existing width.

2.3.3 Changing User-Missing Value Set Width

A few PSPP language constructs copy sets of user-missing values from one variable to another. When the source and target variables have the same width, this is simple. But when the target variable's width might be different from the source variable's, it takes a little more work. The functions described here can help.

In fact, it is usually unnecessary to call these functions directly. Most of the time `var_set_missing_values`, which uses `mv_resize` internally to resize the new set of missing values to the required width, may be used instead. See [var_set_missing_values], page 20, for more information.

`bool mv_is_resizable` (*const struct missing_values* `*mv`, *int* `new_width`) [Function]
: Tests whether *mv*'s width may be changed to *new_width* using `mv_resize`. Returns true if it is allowed, false otherwise.

  If *mv* contains any missing values, then it may be resized only if each missing value may be resized, as determined by `value_is_resizable` (see [value_is_resizable], page 4).

`void mv_resize` (*struct missing_values* `*mv`, *int* `width`) [Function]
: Changes *mv*'s width to *width*. *mv* and *width* must satisfy the constraints explained above.

  When a string missing value set's width is increased, each user-missing value is padded on the right with spaces to the new width.

2.3.4 Inspecting User-Missing Value Sets

These functions inspect the properties and contents of `struct missing_values` objects.

The first set of functions inspects the discrete values that sets of user-missing values may contain:

`bool mv_is_empty` (*const struct missing_values *`mv`*) [Function]

Returns true if *mv* contains no user-missing values, false if it contains at least one user-missing value (either a discrete value or a numeric range).

`int mv_get_width` (*const struct missing_values *`mv`*) [Function]

Returns the width of the user-missing values that *mv* represents.

`int mv_n_values` (*const struct missing_values *`mv`*) [Function]

Returns the number of discrete user-missing values included in *mv*. The return value will be between 0 and 3. For sets of numeric user-missing values that include a range, the return value will be 0 or 1.

`bool mv_has_value` (*const struct missing_values *`mv`*) [Function]

Returns true if *mv* has at least one discrete user-missing values, that is, if `mv_n_values` would return nonzero for *mv*.

`const union value * mv_get_value` (*const struct missing_values *`mv`, int `index`*) [Function]

Returns the discrete user-missing value in *mv* with the given *index*. The caller must not modify or free the returned value or refer to it after modifying or freeing *mv*. The index must be less than the number of discrete user-missing values in *mv*, as reported by `mv_n_values`.

The second set of functions inspects the single range of values that numeric sets of user-missing values may contain:

`bool mv_has_range` (*const struct missing_values *`mv`*) [Function]

Returns true if *mv* includes a range, false otherwise.

`void mv_get_range` (*const struct missing_values *`mv`, double *`low`, double *`high`*) [Function]

Stores the low endpoint of *mv*'s range in `*low` and the high endpoint of the range in `*high`. *mv* must include a range.

2.3.5 Modifying User-Missing Value Sets

These functions modify the contents of `struct missing_values` objects.

The next set of functions applies to all sets of user-missing values:

`bool mv_add_value` (*struct missing_values *`mv`, const union value *`value`*) [Function]
`bool mv_add_str` (*struct missing_values *`mv`, const char `value`[]*) [Function]
`bool mv_add_num` (*struct missing_values *`mv`, double `value`*) [Function]

Attempts to add the given discrete *value* to set of user-missing values *mv*. *value* must have the same width as *mv*. Returns true if *value* was successfully added, false if the set could not accept any more discrete values or if *value* is not an acceptable user-missing value (see `mv_is_acceptable` below).

These functions are equivalent, except for the form in which *value* is provided, so you may use whichever function is most convenient.

`void mv_pop_value` (*struct missing_values* ***mv***, *union value* ***value***) [Function]

Removes a discrete value from *mv* (which must contain at least one discrete value) and stores it in *value*.

`bool mv_replace_value` (*struct missing_values* ***mv***, *const union value* ***value***, *int* ***index***) [Function]

Attempts to replace the discrete value with the given *index* in *mv* (which must contain at least *index* + 1 discrete values) by *value*. Returns true if successful, false if *value* is not an acceptable user-missing value (see `mv_is_acceptable` below).

`bool mv_is_acceptable` (*const union value* ***value***, *int* ***width***) [Function]

Returns true if *value*, which must have the specified *width*, may be added to a missing value set of the same *width*, false if it cannot. As described above, all numeric values and string values of width `MV_MAX_STRING` or less may be added, but string value of greater width may be added only if bytes beyond the first `MV_MAX_STRING` are all spaces.

The second set of functions applies only to numeric sets of user-missing values:

`bool mv_add_range` (*struct missing_values* ***mv***, *double* ***low***, *double* ***high***) [Function]

Attempts to add a numeric range covering *low*. . .*high* (inclusive on both ends) to *mv*, which must be a numeric set of user-missing values. Returns true if the range is successful added, false on failure. Fails if *mv* already contains a range, or if *mv* contains more than one discrete value, or if *low* > *high*.

`void mv_pop_range` (*struct missing_values* ***mv***, *double* ****low***, *double* ****high***) [Function]

Given *mv*, which must be a numeric set of user-missing values that contains a range, removes that range from *mv* and stores its low endpoint in `*low` and its high endpoint in `*high`.

2.4 Value Labels

Each variable has a set of value labels (see Section "VALUE LABELS" in *PSPP Users Guide*), represented as `struct val_labs`. A `struct val_labs` is essentially a map from `union value`s to strings. All of the values in a set of value labels have the same width, which for a set of value labels owned by a variable (the common case) is the same as its variable.

Sets of value labels may contain any number of entries.

It is rarely necessary to interact directly with a `struct val_labs` object. Instead, the most common operation, looking up the label for a value of a given variable, can be conveniently executed through functions on `struct variable`. See Section 2.5.4 [Variable Value Labels], page 20, for details.

Function prototypes and other declarations related to missing values are declared in `data/value-labels.h`.

`struct val_labs` [Structure]
: Opaque type that represents a set of value labels.

The most often useful function for value labels is `val_labs_find`, for looking up the label associated with a value.

`char * val_labs_find` (*const struct val_labs* ***val_labs***, *union value* ***value***) [Function]
: Looks in *val_labs* for a label for the given *value*. Returns the label, if one is found, or a null pointer otherwise.

Several other functions for working with value labels are described in the following section, but these are more rarely useful.

2.4.1 Creation and Destruction

These functions create and destroy `struct val_labs` objects.

`struct val_labs * val_labs_create` (*int* ***width***) [Function]
: Creates and returns an initially empty set of value labels with the given *width*.

`struct val_labs * val_labs_clone` (*const struct val_labs* ***val_labs***) [Function]
: Creates and returns a set of value labels whose width and contents are the same as those of *var_labs*.

`void val_labs_clear` (*struct val_labs* ***var_labs***) [Function]
: Deletes all value labels from *var_labs*.

`void val_labs_destroy` (*struct val_labs* ***var_labs***) [Function]
: Destroys *var_labs*, which must not be referenced again.

2.4.2 Value Labels Properties

These functions inspect and manipulate basic properties of `struct val_labs` objects.

`size_t val_labs_count` (*const struct val_labs* ***val_labs***) [Function]
: Returns the number of value labels in *val_labs*.

`bool val_labs_can_set_width` (*const struct val_labs* ***val_labs***, *int* ***new_width***) [Function]
: Tests whether *val_labs*'s width may be changed to *new_width* using `val_labs_set_width`. Returns true if it is allowed, false otherwise.

  A set of value labels may be resized to a given width only if each value in it may be resized to that width, as determined by `value_is_resizable` (see [value_is_resizable], page 4).

`void val_labs_set_width` (*struct val_labs* ***val_labs***, *int* ***new_width***) [Function]
: Changes the width of *val_labs*'s values to *new_width*, which must be a valid new width as determined by `val_labs_can_set_width`.

2.4.3 Adding and Removing Labels

These functions add and remove value labels from a `struct val_labs` object.

`bool val_labs_add` (*struct val_labs* ***`val_labs`***, *union value* **`value`**, *const char* ***`label`***) [Function]

Adds *label* to in *var_labs* as a label for *value*, which must have the same width as the set of value labels. Returns true if successful, false if *value* already has a label.

`void val_labs_replace` (*struct val_labs* ***`val_labs`***, *union value* **`value`**, *const char* ***`label`***) [Function]

Adds *label* to in *var_labs* as a label for *value*, which must have the same width as the set of value labels. If *value* already has a label in *var_labs*, it is replaced.

`bool val_labs_remove` (*struct val_labs* ***`val_labs`***, *union value* **`value`**) [Function]

Removes from *val_labs* any label for *value*, which must have the same width as the set of value labels. Returns true if a label was removed, false otherwise.

2.4.4 Iterating through Value Labels

These functions allow iteration through the set of value labels represented by a `struct val_labs` object. They may be used in the context of a `for` loop:

```
struct val_labs val_labs;
const struct val_lab *vl;

...

for (vl = val_labs_first (val_labs); vl != NULL;
     vl = val_labs_next (val_labs, vl))
  {
    ...do something with vl...
  }
```

Value labels should not be added or deleted from a `struct val_labs` as it is undergoing iteration.

`const struct val_lab * val_labs_first` (*const struct val_labs* ***`val_labs`***) [Function]

Returns the first value label in *var_labs*, if it contains at least one value label, or a null pointer if it does not contain any value labels.

`const struct val_lab * val_labs_next` (*const struct val_labs* ***`val_labs`***, *const struct val_labs_iterator* ****`vl`***) [Function]

Returns the value label in *var_labs* following *vl*, if *vl* is not the last value label in *val_labs*, or a null pointer if there are no value labels following *vl*.

`const struct val_lab ** val_labs_sorted` (*const struct val_labs* ***`val_labs`***) [Function]

Allocates and returns an array of pointers to value labels, which are sorted in increasing order by value. The array has `val_labs_count (val_labs)` elements. The caller is responsible for freeing the array with `free` (but must not free any of the `struct val_lab` elements that the array points to).

The iteration functions above work with pointers to `struct val_lab` which is an opaque data structure that users of `struct val_labs` must not modify or free directly. The following functions work with objects of this type:

`const union value * val_lab_get_value` (*const struct val_lab* *`*vl`*) [Function]

> Returns the value of value label *vl*. The caller must not modify or free the returned value. (To achieve a similar result, remove the value label with `val_labs_remove`, then add the new value with `val_labs_add`.)
>
> The width of the returned value cannot be determined directly from *vl*. It may be obtained by calling `val_labs_get_width` on the `struct val_labs` that *vl* is in.

`const char * val_lab_get_label` (*const struct val_lab* *`*vl`*) [Function]

> Returns the label in *vl* as a null-terminated string. The caller must not modify or free the returned string. (Use `val_labs_replace` to change a value label.)

2.5 Variables

A PSPP variable is represented by `struct variable`, an opaque type declared in `data/variable.h` along with related declarations. See Section "Variables" in *PSPP Users Guide*, for a description of PSPP variables from a user perspective.

PSPP is unusual among computer languages in that, by itself, a PSPP variable does not have a value. Instead, a variable in PSPP takes on a value only in the context of a case, which supplies one value for each variable in a set of variables (see Section 2.8 [Cases], page 36). The set of variables in a case, in turn, are ordinarily part of a dictionary (see Section 2.6 [Dictionaries], page 29).

Every variable has several attributes, most of which correspond directly to one of the variable attributes visible to PSPP users (see Section "Attributes" in *PSPP Users Guide*).

The following sections describe variable-related functions and macros.

2.5.1 Variable Name

A variable name is a string between 1 and `ID_MAX_LEN` bytes long that satisfies the rules for PSPP identifiers (see Section "Tokens" in *PSPP Users Guide*). Variable names are mixed-case and treated case-insensitively.

`int ID_MAX_LEN` [Macro]

> Maximum length of a variable name, in bytes, currently 64.

Only one commonly useful function relates to variable names:

`const char * var_get_name` (*const struct variable* *`*var`*) [Function]

> Returns *var*'s variable name as a C string.

A few other functions are much more rarely used. Some of these functions are used internally by the dictionary implementation:

`void var_set_name` (*struct variable* *`*var`*, *const char* *`*new_name`*) [Function]

> Changes the name of *var* to *new_name*, which must be a "plausible" name as defined below.
>
> This function cannot be applied to a variable that is part of a dictionary. Use `dict_rename_var` instead (see Section 2.6.5 [Dictionary Renaming Variables], page 32).

`enum dict_class var_get_dict_class` (*const struct variable* ***var***) [Function]
: Returns the dictionary class of *var*'s name (see Section 2.5.9 [Dictionary Class], page 24).

2.5.2 Variable Type and Width

A variable's type and width are the type and width of its values (see Section 2.1 [Values], page 2).

`enum val_type var_get_type` (*const struct variable* ***var***) [Function]
: Returns the type of variable *var*.

`int var_get_width` (*const struct variable* ***var***) [Function]
: Returns the width of variable *var*.

`void var_set_width` (*struct variable* ***var***, *int* ***width***) [Function]
: Sets the width of variable *var* to *width*. The width of a variable should not normally be changed after the variable is created, so this function is rarely used. This function cannot be applied to a variable that is part of a dictionary.

`bool var_is_numeric` (*const struct variable* ***var***) [Function]
: Returns true if *var* is a numeric variable, false otherwise.

`bool var_is_alpha` (*const struct variable* ***var***) [Function]
: Returns true if *var* is an alphanumeric (string) variable, false otherwise.

2.5.3 Variable Missing Values

A numeric or short string variable may have a set of user-missing values (see Section "MISSING VALUES" in *PSPP Users Guide*), represented as a `struct missing_values` (see Section 2.3 [User-Missing Values], page 11).

The most frequent operation on a variable's missing values is to query whether a value is user- or system-missing:

`bool var_is_value_missing` (*const struct variable* ***var***, *const union value* ***value***, *enum mv_class* ***class***) [Function]
`bool var_is_num_missing` (*const struct variable* ***var***, *double* ***value***, *enum mv_class* ***class***) [Function]
`bool var_is_str_missing` (*const struct variable* ***var***, *const char* ***value***[], *enum mv_class* ***class***) [Function]
: Tests whether *value* is a missing value of the given *class* for variable *var* and returns true if so, false otherwise. `var_is_num_missing` may only be applied to numeric variables; `var_is_str_missing` may only be applied to string variables. *value* must have been initialized with the same width as *var*.

: `var_is_`*type*`_missing (`*var*`, `*value*`, `*class*`)` is equivalent to `mv_is_`*type*`_missing (var_get_missing_values (`*var*`), `*value*`, `*class*`)`.

In addition, a few functions are provided to work more directly with a variable's `struct missing_values`:

const struct missing_values * var_get_missing_values (*const struct variable* *var*) [Function]

Returns the struct missing_values associated with *var*. The caller must not modify the returned structure. The return value is always non-null.

void var_set_missing_values (*struct variable* *var*, *const struct missing_values* *miss*) [Function]

Changes *var*'s missing values to a copy of *miss*, or if *miss* is a null pointer, clears *var*'s missing values. If *miss* is non-null, it must have the same width as *var* or be resizable to *var*'s width (see [mv_resize], page 13). The caller retains ownership of *miss*.

void var_clear_missing_values (*struct variable* *var*) [Function]

Clears *var*'s missing values. Equivalent to var_set_missing_values (*var*, NULL).

bool var_has_missing_values (*const struct variable* *var*) [Function]

Returns true if *var* has any missing values, false if it has none. Equivalent to mv_is_empty (var_get_missing_values (*var*)).

2.5.4 Variable Value Labels

A numeric or short string variable may have a set of value labels (see Section "VALUE LABELS" in *PSPP Users Guide*), represented as a struct val_labs (see Section 2.4 [Value Labels], page 15). The most commonly useful functions for value labels return the value label associated with a value:

const char * var_lookup_value_label (*const struct variable* *var*, *const union value* *value*) [Function]

Looks for a label for *value* in *var*'s set of value labels. *value* must have the same width as *var*. Returns the label if one exists, otherwise a null pointer.

void var_append_value_name (*const struct variable* *var*, *const union value* *value*, *struct string* *str*) [Function]

Looks for a label for *value* in *var*'s set of value labels. *value* must have the same width as *var*. If a label exists, it will be appended to the string pointed to by *str*. Otherwise, it formats *value* using *var*'s print format (see Section 2.2 [Input and Output Formats], page 5) and appends the formatted string.

The underlying struct val_labs structure may also be accessed directly using the functions described below.

bool var_has_value_labels (*const struct variable* *var*) [Function]

Returns true if *var* has at least one value label, false otherwise.

const struct val_labs * var_get_value_labels (*const struct variable* *var*) [Function]

Returns the struct val_labs associated with *var*. If *var* has no value labels, then the return value may or may not be a null pointer.

The variable retains ownership of the returned struct val_labs, which the caller must not attempt to modify.

`void var_set_value_labels` (*struct variable* *`*var`*, *const struct val_labs* *`*val_labs`*) [Function]

Replaces *var*'s value labels by a copy of *val_labs*. The caller retains ownership of *val_labs*. If *val_labs* is a null pointer, then *var*'s value labels, if any, are deleted.

`void var_clear_value_labels` (*struct variable* *`*var`*) [Function]

Deletes *var*'s value labels. Equivalent to `var_set_value_labels (`*`var`*`, NULL)`.

A final group of functions offers shorthands for operations that would otherwise require getting the value labels from a variable, copying them, modifying them, and then setting the modified value labels into the variable (making a second copy):

`bool var_add_value_label` (*struct variable* *`*var`*, *const union value* *`*value`*, *const char* *`*label`*) [Function]

Attempts to add a copy of *label* as a label for *value* for the given *var*. *value* must have the same width as *var*. If *value* already has a label, then the old label is retained. Returns true if a label is added, false if there was an existing label for *value*. Either way, the caller retains ownership of *value* and *label*.

`void var_replace_value_label` (*struct variable* *`*var`*, *const union value* *`*value`*, *const char* *`*label`*) [Function]

Attempts to add a copy of *label* as a label for *value* for the given *var*. *value* must have the same width as *var*. If *value* already has a label, then *label* replaces the old label. Either way, the caller retains ownership of *value* and *label*.

2.5.5 Variable Print and Write Formats

Each variable has an associated pair of output formats, called its *print format* and *write format*. See Section "Input and Output Formats" in *PSPP Users Guide*, for an introduction to formats. See Section 2.2 [Input and Output Formats], page 5, for a developer's description of format representation.

The print format is used to convert a variable's data values to strings for human-readable output. The write format is used similarly for machine-readable output, primarily by the WRITE transformation (see Section "WRITE" in *PSPP Users Guide*). Most often a variable's print and write formats are the same.

A newly created variable by default has format F8.2 if it is numeric or an A format with the same width as the variable if it is string. Many creators of variables override these defaults.

Both the print format and write format are output formats. Input formats are not part of `struct variable`. Instead, input programs and transformations keep track of variable input formats themselves.

The following functions work with variable print and write formats.

`const struct fmt_spec * var_get_print_format` (*const struct variable* *`*var`*) [Function]

`const struct fmt_spec * var_get_write_format` (*const struct variable* *`*var`*) [Function]

Returns *var*'s print or write format, respectively.

`void var_set_print_format` (*struct variable* *`var`, *const struct fmt_spec* *`format`) [Function]

`void var_set_write_format` (*struct variable* *`var`, *const struct fmt_spec* *`format`) [Function]

`void var_set_both_formats` (*struct variable* *`var`, *const struct fmt_spec* *`format`) [Function]

Sets *var*'s print format, write format, or both formats, respectively, to a copy of *format*.

2.5.6 Variable Labels

A variable label is a string that describes a variable. Variable labels may contain spaces and punctuation not allowed in variable names. See Section "VARIABLE LABELS" in *PSPP Users Guide*, for a user-level description of variable labels.

The most commonly useful functions for variable labels are those to retrieve a variable's label:

`const char * var_to_string` (*const struct variable* *`var`) [Function]

Returns *var*'s variable label, if it has one, otherwise *var*'s name. In either case the caller must not attempt to modify or free the returned string.

This function is useful for user output.

`const char * var_get_label` (*const struct variable* *`var`) [Function]

Returns *var*'s variable label, if it has one, or a null pointer otherwise.

A few other variable label functions are also provided:

`void var_set_label` (*struct variable* *`var`, *const char* *`label`) [Function]

Sets *var*'s variable label to a copy of *label*, or removes any label from *var* if *label* is a null pointer or contains only spaces. Leading and trailing spaces are removed from the variable label and its remaining content is truncated at 255 bytes.

`void var_clear_label` (*struct variable* *`var`) [Function]

Removes any variable label from *var*.

`bool var_has_label` (*const struct variable* *`var`) [Function]

Returns true if *var* has a variable label, false otherwise.

2.5.7 GUI Attributes

These functions and types access and set attributes that are mainly used by graphical user interfaces. Their values are also stored in and retrieved from system files (but not portable files).

The first group of functions relate to the measurement level of numeric data. New variables are assigned a nominal level of measurement by default.

`enum measure` [Enumeration]

Measurement level. Available values are:

`MEASURE_NOMINAL`

Numeric data values are arbitrary. Arithmetic operations and numerical comparisons of such data are not meaningful.

`MEASURE_ORDINAL`
: Numeric data values indicate progression along a rank order. Arbitrary arithmetic operations such as addition are not meaningful on such data, but inequality comparisons (less, greater, etc.) have straightforward interpretations.

`MEASURE_SCALE`
: Ratios, sums, etc. of numeric data values have meaningful interpretations.

PSPP does not have a separate category for interval data, which would naturally fall between the ordinal and scale measurement levels.

`bool measure_is_valid` (*enum measure* ***measure***) [Function]

Returns true if *measure* is a valid level of measurement, that is, if it is one of the `enum measure` constants listed above, and false otherwise.

`enum measure` *var_get_measure* (*const struct variable* *`var`) [Function]
`void var_set_measure` (*struct variable* *`var`, *enum measure* ***measure***) [Function]

Gets or sets *var*'s measurement level.

The following set of functions relates to the width of on-screen columns used for displaying variable data in a graphical user interface environment. The unit of measurement is the width of a character. For proportionally spaced fonts, this is based on the average width of a character.

`int var_get_display_width` (*const struct variable* *`var`) [Function]
`void var_set_display_width` (*struct variable* *`var`, *int* `display_width`) [Function]

Gets or sets *var*'s display width.

`int var_default_display_width` (*int* `width`) [Function]

Returns the default display width for a variable with the given *width*. The default width of a numeric variable is 8. The default width of a string variable is *width* or 32, whichever is less.

The final group of functions work with the justification of data when it is displayed in on-screen columns. New variables are by default right-justified.

`enum alignment` [Enumeration]

Text justification. Possible values are `ALIGN_LEFT`, `ALIGN_RIGHT`, and `ALIGN_CENTRE`.

`bool alignment_is_valid` (*enum alignment* `alignment`) [Function]

Returns true if *alignment* is a valid alignment, that is, if it is one of the `enum alignment` constants listed above, and false otherwise.

`enum alignment` *var_get_alignment* (*const struct variable* *`var`) [Function]
`void var_set_alignment` (*struct variable* *`var`, *enum alignment* `alignment`) [Function]

Gets or sets *var*'s alignment.

2.5.8 Variable Leave Status

Commonly, most or all data in a case come from an input file, read with a command such as DATA LIST or GET, but data can also be generated with transformations such as COMPUTE. In the latter case the question of a datum's "initial value" can arise. For example, the value of a piece of generated data can recursively depend on its own value:

```
COMPUTE X = X + 1.
```

Another situation where the initial value of a variable arises is when its value is not set at all for some cases, e.g. below, `Y` is set only for the first 10 cases:

```
DO IF #CASENUM <= 10.
+ COMPUTE Y = 1.
END IF.
```

By default, the initial value of a datum in either of these situations is the system-missing value for numeric values and spaces for string values. This means that, above, X would be system-missing and that Y would be 1 for the first 10 cases and system-missing for the remainder.

PSPP also supports retaining the value of a variable from one case to another, using the LEAVE command (see Section "LEAVE" in *PSPP Users Guide*). The initial value of such a variable is 0 if it is numeric and spaces if it is a string. If the command '`LEAVE X Y`' is appended to the above example, then X would have value 1 in the first case and increase by 1 in every succeeding case, and Y would have value 1 for the first 10 cases and 0 for later cases.

The LEAVE command has no effect on data that comes from an input file or whose values do not depend on a variable's initial value.

The value of scratch variables (see Section "Scratch Variables" in *PSPP Users Guide*) are always left from one case to another.

The following functions work with a variable's leave status.

`bool` **`var_get_leave`** (*const struct variable* *`var`) [Function]

Returns true if *var*'s value is to be retained from case to case, false if it is reinitialized to system-missing or spaces.

`void` **`var_set_leave`** (*struct variable* *`var`, *bool* `leave`) [Function]

If *leave* is true, marks *var* to be left from case to case; if *leave* is false, marks *var* to be reinitialized for each case.

If *var* is a scratch variable, *leave* must be true.

`bool` **`var_must_leave`** (*const struct variable* *`var`) [Function]

Returns true if *var* must be left from case to case, that is, if *var* is a scratch variable.

2.5.9 Dictionary Class

Occasionally it is useful to classify variables into *dictionary classes* based on their names. Dictionary classes are represented by `enum dict_class`. This type and other declarations for dictionary classes are in the `<data/dict-class.h>` header.

`enum` **`dict_class`** [Enumeration]

The dictionary classes are:

`DC_ORDINARY`
: An ordinary variable, one whose name does not begin with '$' or '#'.

`DC_SYSTEM`
: A system variable, one whose name begins with '$'. See Section "System Variables" in *PSPP Users Guide*.

`DC_SCRATCH`
: A scratch variable, one whose name begins with '#'. See Section "Scratch Variables" in *PSPP Users Guide*.

The values for dictionary classes are bitwise disjoint, which allows them to be used in bit-masks. An extra enumeration constant `DC_ALL`, whose value is the bitwise-*or* of all of the above constants, is provided to aid in this purpose.

One example use of dictionary classes arises in connection with PSPP syntax that uses `a TO b` to name the variables in a dictionary from *a* to *b* (see Section "Sets of Variables" in *PSPP Users Guide*). This syntax requires *a* and *b* to be in the same dictionary class. It limits the variables that it includes to those in that dictionary class.

The following functions relate to dictionary classes.

enum dict_class dict_class_from_id (*const char* ***name***) [Function]

Returns the "dictionary class" for the given variable *name*, by looking at its first letter.

const char * dict_class_to_name (*enum dict_class* ***dict_class***) [Function]

Returns a name for the given *dict_class* as an adjective, e.g. `"scratch"`.

This function should probably not be used in new code as it can lead to difficulties for internationalization.

2.5.10 Variable Creation and Destruction

Only rarely should PSPP code create or destroy variables directly. Ordinarily, variables are created within a dictionary and destroying by individual deletion from the dictionary or by destroying the entire dictionary at once. The functions here enable the exceptional case, of creation and destruction of variables that are not associated with any dictionary. These functions are used internally in the dictionary implementation.

struct variable * var_create (*const char* ***name***, *int* ***width***) [Function]

Creates and returns a new variable with the given *name* and *width*. The new variable is not part of any dictionary. Use `dict_create_var`, instead, to create a variable in a dictionary (see Section 2.6.2 [Dictionary Creating Variables], page 30).

name should be a valid variable name and must be a "plausible" variable name (see Section 2.5.1 [Variable Name], page 18). *width* must be between 0 and `MAX_STRING`, inclusive (see Section 2.1 [Values], page 2).

The new variable has no user-missing values, value labels, or variable label. Numeric variables initially have F8.2 print and write formats, right-justified display alignment, and scale level of measurement. String variables are created with A print and write formats, left-justified display alignment, and nominal level of measurement. The initial display width is determined by `var_default_display_width` (see [var_default_display_width], page 23).

The new variable initially has no short name (see Section 2.5.11 [Variable Short Names], page 26) and no auxiliary data (see Section 2.5.13 [Variable Auxiliary Data], page 28).

`struct variable * var_clone (`*const struct variable ****old_var***`)` [Function]

Creates and returns a new variable with the same attributes as *old_var*, with a few exceptions. First, the new variable is not part of any dictionary, regardless of whether *old_var* was in a dictionary. Use `dict_clone_var`, instead, to add a clone of a variable to a dictionary.

Second, the new variable is not given any short name, even if *old_var* had a short name. This is because the new variable is likely to be immediately renamed, in which case the short name would be incorrect (see Section 2.5.11 [Variable Short Names], page 26).

Finally, *old_var*'s auxiliary data, if any, is not copied to the new variable (see Section 2.5.13 [Variable Auxiliary Data], page 28).

`void var_destroy (`*struct variable ****var***`)` [Function]

Destroys *var* and frees all associated storage, including its auxiliary data, if any. *var* must not be part of a dictionary. To delete a variable from a dictionary and destroy it, use `dict_delete_var` (see Section 2.6.3 [Dictionary Deleting Variables], page 31).

2.5.11 Variable Short Names

PSPP variable names may be up to 64 (`ID_MAX_LEN`) bytes long. The system and portable file formats, however, were designed when variable names were limited to 8 bytes in length. Since then, the system file format has been augmented with an extension record that explains how the 8-byte short names map to full-length names (see Section B.11 [Long Variable Names Record], page 68), but the short names are still present. Thus, the continued presence of the short names is more or less invisible to PSPP users, but every variable in a system file still has a short name that must be unique.

PSPP can generate unique short names for variables based on their full names at the time it creates the data file. If all variables' full names are unique in their first 8 bytes, then the short names are simply prefixes of the full names; otherwise, PSPP changes them so that they are unique.

By itself this algorithm interoperates well with other software that can read system files, as long as that software understands the extension record that maps short names to long names. When the other software does not understand the extension record, it can produce surprising results. Consider a situation where PSPP reads a system file that contains two variables named RANKINGSCORE, then the user adds a new variable named RANKINGSTATUS, then saves the modified data as a new system file. A program that does not understand long names would then see one of these variables under the name RANKINGS—either one, depending on the algorithm's details—and the other under a different name. The effect could be very confusing: by adding a new and apparently unrelated variable in PSPP, the user effectively renamed the existing variable.

To counteract this potential problem, every `struct variable` may have a short name. A variable created by the system or portable file reader receives the short name from that data file. When a variable with a short name is written to a system or portable file, that variable receives priority over other long names whose names begin with the same 8 bytes but which were not read from a data file under that short name.

Variables not created by the system or portable file reader have no short name by default.

A variable with a full name of 8 bytes or less in length has absolute priority for that name when the variable is written to a system file, even over a second variable with that assigned short name.

PSPP does not enforce uniqueness of short names, although the short names read from any given data file will always be unique. If two variables with the same short name are written to a single data file, neither one receives priority.

The following macros and functions relate to short names.

`SHORT_NAME_LEN` [Macro]

Maximum length of a short name, in bytes. Its value is 8.

`const char * var_get_short_name` (*const struct variable* **`*var`**) [Function]

Returns *var*'s short name, or a null pointer if *var* has not been assigned a short name.

`void var_set_short_name` (*struct variable* **`*var`**, *const char* **`*short_name`**) [Function]

Sets *var*'s short name to *short_name*, or removes *var*'s short name if *short_name* is a null pointer. If it is non-null, then *short_name* must be a plausible name for a variable. The name will be truncated to 8 bytes in length and converted to all-uppercase.

`void var_clear_short_name` (*struct variable* **`*var`**) [Function]

Removes *var*'s short name.

2.5.12 Variable Relationships

Variables have close relationships with dictionaries (see Section 2.6 [Dictionaries], page 29) and cases (see Section 2.8 [Cases], page 36). A variable is usually a member of some dictionary, and a case is often used to store data for the set of variables in a dictionary.

These functions report on these relationships. They may be applied only to variables that are in a dictionary.

`size_t var_get_dict_index` (*const struct variable* **`*var`**) [Function]

Returns *var*'s index within its dictionary. The first variable in a dictionary has index 0, the next variable index 1, and so on.

The dictionary index can be influenced using dictionary functions such as dict_reorder_var (see [dict_reorder_var], page 32).

`size_t var_get_case_index` (*const struct variable* **`*var`**) [Function]

Returns *var*'s index within a case. The case index is an index into an array of `union value` large enough to contain all the data in the dictionary.

The returned case index can be used to access the value of *var* within a case for its dictionary, as in e.g. `case_data_idx (case, var_get_case_index (var))`, but ordinarily it is more convenient to use the data access functions that do variable-to-index translation internally, as in e.g. `case_data (case, var)`.

2.5.13 Variable Auxiliary Data

Each `struct variable` can have a single pointer to auxiliary data of type `void *`. These functions manipulate a variable's auxiliary data.

Use of auxiliary data is discouraged because of its lack of flexibility. Only one client can make use of auxiliary data on a given variable at any time, even though many clients could usefully associate data with a variable.

To prevent multiple clients from attempting to use a variable's single auxiliary data field at the same time, we adopt the convention that use of auxiliary data in the active dataset dictionary is restricted to the currently executing command. In particular, transformations must not attach auxiliary data to a variable in the active dataset in the expectation that it can be used later when the active dataset is read and the transformation is executed. To help enforce this restriction, auxiliary data is deleted from all variables in the active dataset dictionary after the execution of each PSPP command.

This convention for safe use of auxiliary data applies only to the active dataset dictionary. Rules for other dictionaries may be established separately.

Auxiliary data should be replaced by a more flexible mechanism at some point, but no replacement mechanism has been designed or implemented so far.

The following functions work with variable auxiliary data.

`void * var_get_aux` (*const struct variable* *`var`) [Function]

Returns *var*'s auxiliary data, or a null pointer if none has been assigned.

`void * var_attach_aux` (*const struct variable* *`var`, *void* *`aux`, *void* (*`aux_dtor`) (*struct variable* *)) [Function]

Sets *var*'s auxiliary data to *aux*, which must not be null. *var* must not already have auxiliary data.

Before *var*'s auxiliary data is cleared by `var_clear_aux`, *aux_dtor*, if non-null, will be called with *var* as its argument. It should free any storage associated with *aux*, if necessary. `var_dtor_free` may be appropriate for use as *aux_dtor*:

`void` *var_dtor_free* (*struct variable* *`var`) [Function]

Frees *var*'s auxiliary data by calling `free`.

`void var_clear_aux` (*struct variable* *`var`) [Function]

Removes auxiliary data, if any, from *var*, first calling the destructor passed to `var_attach_aux`, if one was provided.

Use `dict_clear_aux` to remove auxiliary data from every variable in a dictionary.

`void * var_detach_aux` (*struct variable* *`var`) [Function]

Removes auxiliary data, if any, from *var*, and returns it. Returns a null pointer if *var* had no auxiliary data.

Any destructor passed to `var_attach_aux` is not called, so the caller is responsible for freeing storage associated with the returned auxiliary data.

2.5.14 Variable Categorical Values

Some statistical procedures require a list of all the values that a categorical variable takes on. Arranging such a list requires making a pass through the data, so PSPP caches categorical values in `struct variable`.

When variable auxiliary data is revamped to support multiple clients as described in the previous section, categorical values are an obvious candidate. The form in which they are currently supported is inelegant.

Categorical values are not robust against changes in the data. That is, there is currently no way to detect that a transformation has changed data values, meaning that categorical values lists for the changed variables must be recomputed. PSPP is in fact in need of a general-purpose caching and cache-invalidation mechanism, but none has yet been designed and built.

The following functions work with cached categorical values.

`struct cat_vals * var_get_obs_vals` (*const struct variable ****var***) [Function]
: Returns *var*'s set of categorical values. Yields undefined behavior if *var* does not have any categorical values.

`void var_set_obs_vals` (*const struct variable ****var***, *struct cat_vals ****cat_vals***) [Function]
: Destroys *var*'s categorical values, if any, and replaces them by *cat_vals*, ownership of which is transferred to *var*. If *cat_vals* is a null pointer, then *var*'s categorical values are cleared.

`bool var_has_obs_vals` (*const struct variable ****var***) [Function]
: Returns true if *var* has a set of categorical values, false otherwise.

2.6 Dictionaries

Each data file in memory or on disk has an associated dictionary, whose primary purpose is to describe the data in the file. See Section "Variables" in *PSPP Users Guide*, for a PSPP user's view of a dictionary.

A data file stored in a PSPP format, either as a system or portable file, has a representation of its dictionary embedded in it. Other kinds of data files are usually not self-describing enough to construct a dictionary unassisted, so the dictionaries for these files must be specified explicitly with PSPP commands such as DATA LIST.

The most important content of a dictionary is an array of variables, which must have unique names. A dictionary also conceptually contains a mapping from each of its variables to a location within a case (see Section 2.8 [Cases], page 36), although in fact these mappings are stored within individual variables.

System variables are not members of any dictionary (see Section "System Variables" in *PSPP Users Guide*).

Dictionaries are represented by `struct dictionary`. Declarations related to dictionaries are in the `<data/dictionary.h>` header.

The following sections describe functions for use with dictionaries.

2.6.1 Accessing Variables

The most common operations on a dictionary simply retrieve a `struct variable *` of an individual variable based on its name or position.

`struct variable * dict_lookup_var` (*const struct dictionary **`dict`, *const char **`name`) [Function]

`struct variable * dict_lookup_var_assert` (*const struct dictionary **`dict`, *const char **`name`) [Function]

Looks up and returns the variable with the given *name* within *dict*. Name lookup is not case-sensitive.

`dict_lookup_var` returns a null pointer if *dict* does not contain a variable named *name*. `dict_lookup_var_assert` asserts that such a variable exists.

`struct variable * dict_get_var` (*const struct dictionary **`dict`, *size_t* `position`) [Function]

Returns the variable at the given *position* in *dict*. *position* must be less than the number of variables in *dict* (see below).

`size_t dict_get_var_cnt` (*const struct dictionary **`dict`) [Function]

Returns the number of variables in *dict*.

Another pair of functions allows retrieving a number of variables at once. These functions are more rarely useful.

`void dict_get_vars` (*const struct dictionary **`dict`, *const struct variable ****`vars`, *size_t **`cnt`, *enum dict_class* `exclude`) [Function]

`void dict_get_vars_mutable` (*const struct dictionary **`dict`, *struct variable ****`vars`, *size_t **`cnt`, *enum dict_class* `exclude`) [Function]

Retrieves all of the variables in *dict*, in their original order, except that any variables in the dictionary classes specified *exclude*, if any, are excluded (see Section 2.5.9 [Dictionary Class], page 24). Pointers to the variables are stored in an array allocated with `malloc`, and a pointer to the first element of this array is stored in `*vars`. The caller is responsible for freeing this memory when it is no longer needed. The number of variables retrieved is stored in `*cnt`.

The presence or absence of `DC_SYSTEM` in *exclude* has no effect, because dictionaries never include system variables.

One additional function is available. This function is most often used in assertions, but it is not restricted to such use.

`bool dict_contains_var` (*const struct dictionary **`dict`, *const struct variable **`var`) [Function]

Tests whether *var* is one of the variables in *dict*. Returns true if so, false otherwise.

2.6.2 Creating Variables

These functions create a new variable and insert it into a dictionary in a single step.

There is no provision for inserting an already created variable into a dictionary. There is no reason that such a function could not be written, but so far there has been no need for one.

The names provided to one of these functions should be valid variable names and must be plausible variable names.

If a variable with the same name already exists in the dictionary, the non-`assert` variants of these functions return a null pointer, without modifying the dictionary. The `assert` variants, on the other hand, assert that no duplicate name exists.

A variable may be in only one dictionary at any given time.

`struct variable * dict_create_var` (*struct dictionary *`dict`, const char *`name`, int `width`*) [Function]

`struct variable * dict_create_var_assert` (*struct dictionary *`dict`, const char *`name`, int `width`*) [Function]

Creates a new variable with the given *name* and *width*, as if through a call to `var_create` with those arguments (see [var_create], page 25), appends the new variable to *dict*'s array of variables, and returns the new variable.

`struct variable * dict_clone_var` (*struct dictionary *`dict`, const struct variable *`old_var`*) [Function]

`struct variable * dict_clone_var_assert` (*struct dictionary *`dict`, const struct variable *`old_var`*) [Function]

Creates a new variable as a clone of *var*, inserts the new variable into *dict*, and returns the new variable. Other properties of the new variable are copied from *old_var*, except for those not copied by `var_clone` (see [var_clone], page 26).

var does not need to be a member of any dictionary.

`struct variable * dict_clone_var_as` (*struct dictionary *`dict`, const struct variable *`old_var`, const char *`name`*) [Function]

`struct variable * dict_clone_var_as_assert` (*struct dictionary *`dict`, const struct variable *`old_var`, const char *`name`*) [Function]

These functions are similar to `dict_clone_var` and `dict_clone_var_assert`, respectively, except that the new variable is named *name* instead of keeping *old_var*'s name.

2.6.3 Deleting Variables

These functions remove variables from a dictionary's array of variables. They also destroy the removed variables and free their associated storage.

Deleting a variable to which there might be external pointers is a bad idea. In particular, deleting variables from the active dataset dictionary is a risky proposition, because transformations can retain references to arbitrary variables. Therefore, no variable should be deleted from the active dataset dictionary when any transformations are active, because those transformations might reference the variable to be deleted. The safest time to delete a variable is just after a procedure has been executed, as done by DELETE VARIABLES.

Deleting a variable automatically removes references to that variable from elsewhere in the dictionary as a weighting variable, filter variable, SPLIT FILE variable, or member of a vector.

No functions are provided for removing a variable from a dictionary without destroying that variable. As with insertion of an existing variable, there is no reason that this could not be implemented, but so far there has been no need.

`void dict_delete_var` (*struct dictionary *`dict`*, *struct variable *`var`*) [Function]

Deletes *var* from *dict*, of which it must be a member.

`void dict_delete_vars` (*struct dictionary *`dict`*, *struct variable *const *`vars`*, *size_t `count`*) [Function]

Deletes the *count* variables in array *vars* from *dict*. All of the variables in *vars* must be members of *dict*. No variable may be included in *vars* more than once.

`void dict_delete_consecutive_vars` (*struct dictionary *`dict`*, *size_t `idx`*, *size_t `count`*) [Function]

Deletes the variables in sequential positions *idx*. . . *idx* + *count* (exclusive) from *dict*, which must contain at least *idx* + *count* variables.

`void dict_delete_scratch_vars` (*struct dictionary *`dict`*) [Function]

Deletes all scratch variables from *dict*.

2.6.4 Changing Variable Order

The variables in a dictionary are stored in an array. These functions change the order of a dictionary's array of variables without changing which variables are in the dictionary.

`void dict_reorder_var` (*struct dictionary *`dict`*, *struct variable *`var`*, *size_t `new_index`*) [Function]

Moves *var*, which must be in *dict*, so that it is at position *new_index* in *dict*'s array of variables. Other variables in *dict*, if any, retain their relative positions. *new_index* must be less than the number of variables in *dict*.

`void dict_reorder_vars` (*struct dictionary *`dict`*, *struct variable *const *`new_order`*, *size_t `count`*) [Function]

Moves the *count* variables in *new_order* to the beginning of *dict*'s array of variables in the specified order. Other variables in *dict*, if any, retain their relative positions.

All of the variables in *new_order* must be in *dict*. No duplicates are allowed within *new_order*, which means that *count* must be no greater than the number of variables in *dict*.

2.6.5 Renaming Variables

These functions change the names of variables within a dictionary. The `var_set_name` function (see [var_set_name], page 18) cannot be applied directly to a variable that is in a dictionary, because `struct dictionary` contains an index by name that `var_set_name` would not update. The following functions take care to update the index as well. They also ensure that variable renaming does not cause a dictionary to contain a duplicate variable name.

`void dict_rename_var` (*struct dictionary *`dict`*, *struct variable *`var`*, *const char *`new_name`*) [Function]

Changes the name of *var*, which must be in *dict*, to *new_name*. A variable named *new_name* must not already be in *dict*, unless *new_name* is the same as *var*'s current name.

bool **dict_rename_vars** (*struct dictionary* **dicT*, *struct variable* ***vars*, *char* ***new_names*, *size_t* *count*, *char* ***err_name*) [Function]

Renames each of the *count* variables in *vars* to the name in the corresponding position of *new_names*. If the renaming would result in a duplicate variable name, returns false and stores one of the names that would be be duplicated into `*err_name`, if *err_name* is non-null. Otherwise, the renaming is successful, and true is returned.

2.6.6 Weight Variable

A data set's cases may optionally be weighted by the value of a numeric variable. See Section "WEIGHT" in *PSPP Users Guide*, for a user view of weight variables.

The weight variable is written to and read from system and portable files.

The most commonly useful function related to weighting is a convenience function to retrieve a weighting value from a case.

double **dict_get_case_weight** (*const struct dictionary* **dict*, *const struct ccase* **case*, *bool* **warn_on_invalid*) [Function]

Retrieves and returns the value of the weighting variable specified by *dict* from *case*. Returns 1.0 if *dict* has no weighting variable.

Returns 0.0 if *c*'s weight value is user- or system-missing, zero, or negative. In such a case, if *warn_on_invalid* is non-null and `*warn_on_invalid` is true, `dict_get_case_weight` also issues an error message and sets `*warn_on_invalid` to false. To disable error reporting, pass a null pointer or a pointer to false as *warn_on_invalid* or use a `msg_disable`/`msg_enable` pair.

The dictionary also has a pair of functions for getting and setting the weight variable.

struct variable * **dict_get_weight** (*const struct dictionary* **dict*) [Function]

Returns *dict*'s current weighting variable, or a null pointer if the dictionary does not have a weighting variable.

void **dict_set_weight** (*struct dictionary* **dict*, *struct variable* **var*) [Function]

Sets *dict*'s weighting variable to *var*. If *var* is non-null, it must be a numeric variable in *dict*. If *var* is null, then *dict*'s weighting variable, if any, is cleared.

2.6.7 Filter Variable

When the active dataset is read by a procedure, cases can be excluded from analysis based on the values of a *filter variable*. See Section "FILTER" in *PSPP Users Guide*, for a user view of filtering.

These functions store and retrieve the filter variable. They are rarely useful, because the data analysis framework automatically excludes from analysis the cases that should be filtered.

struct variable * **dict_get_filter** (*const struct dictionary* **dict*) [Function]

Returns *dict*'s current filter variable, or a null pointer if the dictionary does not have a filter variable.

void **dict_set_filter** (*struct dictionary* **dict*, *struct variable* **var*) [Function]

Sets *dict*'s filter variable to *var*. If *var* is non-null, it must be a numeric variable in *dict*. If *var* is null, then *dict*'s filter variable, if any, is cleared.

2.6.8 Case Limit

The limit on cases analyzed by a procedure, set by the N OF CASES command (see Section "N OF CASES" in *PSPP Users Guide*), is stored as part of the dictionary. The dictionary does not, on the other hand, play any role in enforcing the case limit (a job done by data analysis framework code).

A case limit of 0 means that the number of cases is not limited.

These functions are rarely useful, because the data analysis framework automatically excludes from analysis any cases beyond the limit.

`casenumber dict_get_case_limit` (*const struct dictionary* ***`dict`**) [Function]

Returns the current case limit for *dict*.

`void dict_set_case_limit` (*struct dictionary* ***`dict`**, *casenumber* **`limit`**) [Function]

Sets *dict*'s case limit to *limit*.

2.6.9 Split Variables

The user may use the SPLIT FILE command (see Section "SPLIT FILE" in *PSPP Users Guide*) to select a set of variables on which to split the active dataset into groups of cases to be analyzed independently in each statistical procedure. The set of split variables is stored as part of the dictionary, although the effect on data analysis is implemented by each individual statistical procedure.

Split variables may be numeric or short or long string variables.

The most useful functions for split variables are those to retrieve them. Even these functions are rarely useful directly: for the purpose of breaking cases into groups based on the values of the split variables, it is usually easier to use `casegrouper_create_splits`.

`const struct variable *const * dict_get_split_vars` (*const struct dictionary* ***`dict`**) [Function]

Returns a pointer to an array of pointers to split variables. If and only if there are no split variables, returns a null pointer. The caller must not modify or free the returned array.

`size_t dict_get_split_cnt` (*const struct dictionary* ***`dict`**) [Function]

Returns the number of split variables.

The following functions are also available for working with split variables.

`void dict_set_split_vars` (*struct dictionary* ***`dict`**, *struct variable *const* ***`vars`**, *size_t* **`cnt`**) [Function]

Sets *dict*'s split variables to the *cnt* variables in *vars*. If *cnt* is 0, then *dict* will not have any split variables. The caller retains ownership of *vars*.

`void dict_unset_split_var` (*struct dictionary* ***`dict`**, *struct variable* ***`var`**) [Function]

Removes *var*, which must be a variable in *dict*, from *dict*'s split of split variables.

2.6.10 File Label

A dictionary may optionally have an associated string that describes its contents, called its file label. The user may set the file label with the FILE LABEL command (see Section "FILE LABEL" in *PSPP Users Guide*).

These functions set and retrieve the file label.

`const char * dict_get_label` (*const struct dictionary* ***dict***) [Function]

Returns *dict*'s file label. If *dict* does not have a label, returns a null pointer.

`void dict_set_label` (*struct dictionary* ***dict***, *const char* ***label***) [Function]

Sets *dict*'s label to *label*. If *label* is non-null, then its content, truncated to at most 60 bytes, becomes the new file label. If *label* is null, then *dict*'s label is removed.

The caller retains ownership of *label*.

2.6.11 Documents

A dictionary may include an arbitrary number of lines of explanatory text, called the dictionary's documents. For compatibility, document lines have a fixed width, and lines that are not exactly this width are truncated or padded with spaces as necessary to bring them to the correct width.

PSPP users can use the DOCUMENT (see Section "DOCUMENT" in *PSPP Users Guide*), ADD DOCUMENT (see Section "ADD DOCUMENT" in *PSPP Users Guide*), and DROP DOCUMENTS (see Section "DROP DOCUMENTS" in *PSPP Users Guide*) commands to manipulate documents.

`int DOC_LINE_LENGTH` [Macro]

The fixed length of a document line, in bytes, defined to 80.

The following functions work with whole sets of documents. They accept or return sets of documents formatted as null-terminated strings that are an exact multiple of `DOC_LINE_LENGTH` bytes in length.

`const char * dict_get_documents` (*const struct dictionary* ***dict***) [Function]

Returns the documents in *dict*, or a null pointer if *dict* has no documents.

`void dict_set_documents` (*struct dictionary* ***dict***, *const char* ***new_documents***) [Function]

Sets *dict*'s documents to *new_documents*. If *new_documents* is a null pointer or an empty string, then *dict*'s documents are cleared. The caller retains ownership of *new_documents*.

`void dict_clear_documents` (*struct dictionary* ***dict***) [Function]

Clears the documents from *dict*.

The following functions work with individual lines in a dictionary's set of documents.

`void dict_add_document_line` (*struct dictionary* ***dict***, *const char* ***content***) [Function]

Appends *content* to the documents in *dict*. The text in *content* will be truncated or padded with spaces as necessary to make it exactly `DOC_LINE_LENGTH` bytes long. The caller retains ownership of *content*.

If *content* is over `DOC_LINE_LENGTH`, this function also issues a warning using `msg`. To suppress the warning, enclose a call to one of this function in a `msg_disable`/`msg_enable` pair.

`size_t dict_get_document_line_cnt` (*const struct dictionary* **dict*) [Function]
Returns the number of line of documents in *dict*. If the dictionary contains no documents, returns 0.

`void dict_get_document_line` (*const struct dictionary* **dict*, *size_t* *idx*, *struct string* **content*) [Function]
Replaces the text in *content* (which must already have been initialized by the caller) by the document line in *dict* numbered *idx*, which must be less than the number of lines of documents in *dict*. Any trailing white space in the document line is trimmed, so that *content* will have a length between 0 and `DOC_LINE_LENGTH`.

2.7 Coding Conventions

Every `.c` file should have '`#include <config.h>`' as its first non-comment line. No `.h` file should include `config.h`.

This section needs to be finished.

2.8 Cases

This section needs to be written.

2.9 Data Sets

This section needs to be written.

2.10 Pools

This section needs to be written.

3 Parsing Command Syntax

4 Processing Data

Developer's Guide

Proposed outline:

```
* Introduction
* Basic concepts
** Data sets
** Variables
** Dictionaries
** Coding conventions
** Pools
* Syntax parsing
* Data processing
** Reading data
*** Casereaders generalities
*** Casereaders from data files
*** Casereaders from the active dataset
*** Other casereaders
** Writing data
*** Casewriters generally
*** Casewriters to data files
*** Modifying the active dataset
**** Modifying cases obtained from active dataset casereaders has no real effect
**** Transformations; procedures that transform
** Transforming data
*** Sorting and merging
*** Filtering
*** Grouping
**** Ordering and interaction of filtering and grouping
*** Multiple passes over data
*** Counting cases and case weights
** Best practices
*** Multiple passes with filters versus single pass with loops
*** Sequential versus random access
*** Managing memory
*** Passing cases around
*** Renaming casereaders
*** Avoiding excessive buffering
*** Propagating errors
*** Avoid static/global data
*** Don't worry about null filters, groups, etc.
*** Be aware of reference counting semantics for cases
```

5 Presenting Output

6 Internationalisation

Internationalisation in pspp is complicated. The most annoying aspect is that of character-encoding. This chapter attempts to describe the problems and current ways in which they are addressed.

6.1 The working locales

Pspp has three "working" locales:

- The locale of the user interface.
- The locale of the output.
- The locale of the data. Only the character encoding is relevant.

Each of these locales may, at different times take separate (or identical) values. So for example, a French statistician can use pspp to prepare a report in the English language, using a datafile which has been created by a Japanese researcher hence uses a Japanese character set.

It's rarely, if ever, necessary to interrogate the system to find out the values of the 3 locales. However it's important to be aware of the source (destination) locale when reading (writing) string data. When transfering data between a source and a destination, the appropriate recoding must be performed.

6.1.1 The user interface locale

This is the locale which is visible to the person using pspp. Error messages and confidence indications are written in this locale. For example "Cannot open file" will be written in the user interface locale.

This locale is set from the environment of the user who starts pspp{ire} or from the system locale if not set.

6.1.2 The output locale

This locale is the one that should be visible to the person reading a report generated by pspp. Non-data related strings (Eg: "Page number", "Standard Deviation" etc.) will appear in this locale.

6.1.3 The data locale

This locale is the one associated with the data being analysed with pspp. The only important aspect of this locale is the character encoding.[1] The dictionary pertaining to the data contains a field denoting the encoding. Any string data stored in a `union value` will be encoded in the dictionary's character set.

6.2 System files

`*.sav` files contain a field which is supposed to identify the encoding of the data they contain (see Section B.6 [Machine Integer Info Record], page 62). However, many files produced by early versions of spss set this to "2" (ASCII) regardless of the encoding of the data.

[1] It might also be desirable for the LC_COLLATE category to be used for the purposes of sorting data.

Later versions contain an additional record (see Section B.13 [Character Encoding Record], page 69) describing the encoding. When a system file is read, the dictionary's encoding is set using information gleened from the system file. If the encoding cannot be determined or would be unreliable, then it remains unset.

6.3 GUI

The psppire graphic user interface is written using the Gtk+ api, for which all strings must be encoded in UTF8. All strings passed to the GTK+/GLib library functions (except for filenames) must be UTF-8 encoded otherwise errors will occur. Thus, for the purposes of the programming psppire, the user interface locale should be assumed to be UTF8, even if setlocale and/or nl_langinfo indicates otherwise.

6.3.1 Filenames

The GLib API has some special functions for dealing with filenames. Strings returned from functions like gtk_file_chooser_dialog_get_name are not, in general, encoded in UTF8, but in "filename" encoding. If that filename is passed to another GLib function which expects a filename, no conversion is necessary. If it's passed to a function for the purposes of displaying it (eg. in a window's title-bar) it must be converted to UTF8 — there is a special function for this: g_filename_display_name or g_filename_basename. If however, a filename needs to be passed outside of GTK+/GLib (for example to fopen) it must be converted to the local system encoding.

6.4 Existing locale handling functions

The major aspect of locale handling which the programmer has to consider is that of character encoding.

The following function is used to recode strings:

`char *` *recode_string* (*const char* *`to`, *const char* *`from`, *const char* *`text`, *int* `len`); [Function]

> Converts the string *text*, which is encoded in *from* to a new string encoded in *to* encoding. If *len* is not -1, then it must be the number of bytes in *text*. It is the caller's responsibility to free the returned string when no longer required.

In order to minimise the number of conversions required, and to simplify design, PSPP attempts to store all internal strings in UTF8 encoding. Thus, when reading system and portable files (or any other data source), the following items are immediately converted to UTF8 encoding:

- Variable names
- Variable labels
- Value labels

Conversely, when writing system files, these are converted back to the encoding of that system file.

String data stored in union values are left in their original encoding. These will be converted by the data_in/data_out functions.

6.5 Quirks

For historical reasons, not all locale handling follows posix conventions. This makes it difficult (impossible?) to elegantly handle the issues. For example, it would make sense for the gui's datasheet to display numbers formatted according to the LC_NUMERIC category of the data locale. Instead however there is the `data_out` function (see Section 2.2.3 [Obtaining Properties of Format Types], page 7) which uses the `settings_get_decimal_char` function instead of the decimal separator of the locale. Similarly, formatting of monetary values is displayed in a pspp/spss specific fashion instead of using the LC_MONETARY category.

7 Function Index

*

A

C

D

F

H

I

L

M

S

V

8 Concept Index

M

N

S

V

W

Appendix A 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.

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.

A.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.

A.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 a integer field having value *n*, followed by exactly *n* characters, which are the string content.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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 B.3 [Variable Record], page 57).

  - 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).

A.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.

A.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.

A.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.

Appendix B System File Format

A system file encapsulates a set of cases and dictionary information that describes how they may be interpreted. This chapter describes the format of a system file.

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 B.11 [Long Variable Names Record], page 68) and very long string records (see Section B.12 [Very Long String Record], page 68) 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 56).

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 57).

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 B.13 [Character Encoding Record], page 69), if present, and failing that the `character_code` in the machine integer info record (see Section B.6 [Machine Integer Info Record], page 62). 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 B.13 [Character Encoding Record], page 69).

B.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.
- 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.

B.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.

`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 Appendix B [System File Format], page 54).

`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, system files written by some systems set this value to -1. 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 58). 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 Appendix B [System File Format], page 54). 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.

B.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 its offset in the set of variable records, including dummy variable records for long string variables. The first variable record has a dictionary index of 0, the second has a dictionary index of 1, 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 B.12 [Very Long String Record], page 68, 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 B.15 [Long String Missing Values Record], page 71).

`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 B.12 [Very Long String Record], page 68). 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`

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.

B.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 B.14 [Long String Value Labels Record], page 70).

The value label record has the following format:

```
int32               rec_type;
int32               label_count;

/* Repeated label_cnt 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 dictionary indexes of variables to which to apply the value labels (see [Dictionary Index], page 58). There are `var_count` elements.

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

B.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.

`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.

B.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 B.13 [Character Encoding Record], page 69).

B.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 Appendix B [System File Format], page 54, 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.

B.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.
- 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?)

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

- 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
```

B.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.

B.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 type of the variable:

  1 Nominal Scale

  2 Ordinal Scale

  3 Continuous Scale

  SPSS sometimes writes a `measure` of 0. PSPP interprets this as nominal scale.

`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

B.11 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 B.3 [Variable Record], page 57). 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.

B.12 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 B.3 [Variable Record], page 57). 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.

B.13 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 63).

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.

B.14 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.

B.15 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;
long_string_missing_value   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.)

`long_string_missing_value values[];`
: The missing values themselves. This array contains exactly `n_missing_values` elements, each of which has the following substructure:

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

`int32 value_len;`
: The length of the 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 value[];`
: The missing value string, exactly `value_len` bytes, without any padding or null terminator.

B.16 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 type 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 type 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 type 17.

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                                             |.)              |
```

B.16.1 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.

B.17 Extended Number of Cases Record

The file header record expresses the number of cases in the system file as an int32 (see Section B.2 [File Header Record], page 56). 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.

B.18 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:

5 A set of grouped variables (according to Aapi Hämäläinen).

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.

B.19 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.

B.20 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 B.3 [Variable Record], page 57). 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. One file has been seen written by SPSS 14 that contained such a code in a *string* field with the value 0 (after the bias is subtracted) as a way of encoding null bytes.

  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_ofs - 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_ofs - 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.

B.21 Encrypted System Files

SPSS 21 and later support an encrypted system file format.

> **Warning:** The SPSS encrypted file format is poorly designed. 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.

Encrypted File Format

Encrypted system files begin with the following 36-byte fixed header:

```
0000  1c 00 00 00 00 00 00 00  45 4e 43 52 59 50 54 45  |........ENCRYPTE|
0010  44 53 41 56 15 00 00 00  00 00 00 00 00 00 00 00  |DSAV............|
0020  00 00 00 00                                       |....|
```

Following the fixed header is a complete system file in the usual format, except that each 16-byte block is encrypted with AES-256 in ECB mode. 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 (between 1 and 22) null bytes 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
```

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`'.

Appendix C 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 C.1 [Record 0 Main Header Record], page 81).

`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.

C.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 Appendix C [SPSS/PC+ System File Format], page 80).

`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.

C.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 C.3 [Record 2 Labels Record], page 84, 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 C.3 [Record 2 Labels Record], page 84, 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 59, 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 C.1 [Record 0 Main Header Record], page 81). 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.

C.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.

C.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 C.2 [Record 1 Variables Record], page 82). 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.

C.5 Records 4 and 5: Data Entry

Records 4 and 5 appear to be related to SPSS/PC+ Data Entry.

Appendix D q2c Input Format

PSPP statistical procedures have a bizarre and somewhat irregular syntax. Despite this, a parser generator has been written that adequately addresses many of the possibilities and tries to provide hooks for the exceptional cases. This parser generator is named `q2c`.

D.1 Invoking q2c

```
q2c input.q output.c
```

`q2c` translates a '`.q`' file into a '`.c`' file. It takes exactly two command-line arguments, which are the input file name and output file name, respectively. `q2c` does not accept any command-line options.

D.2 q2c Input Structure

`q2c` input files are divided into two sections: the grammar rules and the supporting code. The *grammar rules*, which make up the first part of the input, are used to define the syntax of the statistical procedure to be parsed. The *supporting code*, following the grammar rules, are copied largely unchanged to the output file, except for certain escapes.

The most important lines in the grammar rules are used for defining procedure syntax. These lines can be prefixed with a dollar sign ('`$`'), which prevents Emacs' CC-mode from munging them. Besides this, a bang ('`!`') at the beginning of a line causes the line, minus the bang, to be written verbatim to the output file (useful for comments). As a third special case, any line that begins with the exact characters `/* *INDENT` is ignored and not written to the output. This allows `.q` files to be processed through `indent` without being munged.

The syntax of the grammar rules themselves is given in the following sections.

The supporting code is passed into the output file largely unchanged. However, the following escapes are supported. Each escape must appear on a line by itself.

`/* (header) */`
: Expands to a series of C `#include` directives which include the headers that are required for the parser generated by `q2c`.

`/* (decls` *scope*`) */`
: Expands to C variable and data type declarations for the variables and `enums` input and output by the `q2c` parser. *scope* must be either `local` or `global`. `local` causes the declarations to be output as function locals. `global` causes them to be declared as `static` module variables; thus, `global` is a bit of a misnomer.

`/* (parser) */`
: Expands to the entire parser. Must be enclosed within a C function.

`/* (free) */`
: Expands to a set of calls to the `free` function for variables declared by the parser. Only needs to be invoked if subcommands of type `string` are used in the grammar rules.

D.3 Grammar Rules

The grammar rules describe the format of the syntax that the parser generated by q2c will understand. The way that the grammar rules are included in q2c input file are described above.

The grammar rules are divided into tokens of the following types:

Identifier (ID)
: An identifier token is a sequence of letters, digits, and underscores ('_'). Identifiers are *not* case-sensitive.

String (STRING)
: String tokens are initiated by a double-quote character ('"') and consist of all the characters between that double quote and the next double quote, which must be on the same line as the first. Within a string, a backslash can be used as a "literal escape". The only reasons to use a literal escape are to include a double quote or a backslash within a string.

Special character
: Other characters, other than white space, constitute tokens in themselves.

The syntax of the grammar rules is as follows:

```
grammar-rules ::= command-name opt-prefix : subcommands .
command-name ::= ID
             ::= STRING
opt-prefix ::=
           ::= ( ID )
subcommands ::= subcommand
            ::= subcommands ; subcommand
```

The syntax begins with an ID token that gives the name of the procedure to be parsed. For command names that contain multiple words, a STRING token may be used instead, e.g. '"FILE HANDLE"'. Optionally, an ID in parentheses specifies a prefix used for all file-scope identifiers declared by the emitted code.

The rest of the syntax consists of subcommands separated by semicolons (';') and terminated with a full stop ('.').

```
subcommand ::= default-opt arity-opt ID sbc-defn
default-opt ::=
            ::= *
arity-opt ::=
          ::= +
          ::= ^
sbc-defn ::= opt-prefix = specifiers
         ::= [ ID ] = array-sbc
         ::= opt-prefix = sbc-special-form
```

A subcommand that begins with an asterisk ('*') is the default subcommand. The keyword used for the default subcommand can be omitted in the PSPP syntax file.

A plus sign ('+') indicates that a subcommand can appear more than once. A caret ('^') indicate that a subcommand must appear exactly once. A subcommand marked with neither character may appear once or not at all, but not more than once.

The subcommand name appears after the leading option characters.

There are three forms of subcommands. The first and most common form simply gives an equals sign (‘=’) and a list of specifiers, which can each be set to a single setting. The second form declares an array, which is a set of flags that can be individually turned on by the user. There are also several special forms that do not take a list of specifiers.

Arrays require an additional ID argument. This is used as a prefix, prepended to the variable names constructed from the specifiers. The other forms also allow an optional prefix to be specified.

```
array-sbc ::= alternatives
          ::= array-sbc , alternatives
alternatives ::= ID
             ::= alternatives | ID
```

An array subcommand is a set of Boolean values that can independently be turned on by the user, listed separated by commas (‘,’). If an value has more than one name then these names are separated by pipes (‘|’).

```
specifiers ::= specifier
           ::= specifiers , specifier
specifier ::= opt-id : settings
opt-id ::=
       ::= ID
```

Ordinary subcommands (other than arrays and special forms) require a list of specifiers. Each specifier has an optional name and a list of settings. If the name is given then a correspondingly named variable will be used to store the user’s choice of setting. If no name is given then there is no way to tell which setting the user picked; in this case the settings should probably have values attached.

```
settings ::= setting
         ::= settings / setting
setting ::= setting-options ID setting-value
setting-options ::=
                ::= *
                ::= !
                ::= * !
```

Individual settings are separated by forward slashes (‘/’). Each setting can be as little as an ID token, but options and values can optionally be included. The ‘*’ option means that, for this setting, the ID can be omitted. The ‘!’ option means that this option is the default for its specifier.

```
setting-value ::=
              ::= ( setting-value-2 )
              ::= setting-value-2
setting-value-2 ::= setting-value-options setting-value-type : ID
setting-value-options ::=
                      ::= *
setting-value-type ::= N
                   ::= D
                   ::= S
```

Settings may have values. If the value must be enclosed in parentheses, then enclose the value declaration in parentheses. Declare the setting type as 'n', 'd', or 's' for integer, floating-point, or string type, respectively. The given ID is used to construct a variable name. If option '*' is given, then the value is optional; otherwise it must be specified whenever the corresponding setting is specified.

```
sbc-special-form ::= VAR
                 ::= VARLIST varlist-options
                 ::= INTEGER opt-list
                 ::= DOUBLE opt-list
                 ::= PINT
                 ::= STRING (the literal word STRING)
                 ::= CUSTOM
varlist-options ::=
                ::= ( STRING )
opt-list ::=
         ::= LIST
```

The special forms are of the following types:

`VAR`
: A single variable name.

`VARLIST`
: A list of variables. If given, the string can be used to provide `PV_*` options to the call to `parse_variables`.

`INTEGER`
: A single integer value.

`INTEGER LIST`
: A list of integers separated by spaces or commas.

`DOUBLE`
: A single floating-point value.

`DOUBLE LIST`
: A list of floating-point values.

`PINT`
: A single positive integer value.

`STRING`
: A string value.

`CUSTOM`
: A custom function is used to parse this subcommand. The function must have prototype `int custom_name (void)`. It should return 0 on failure (when it has already issued an appropriate diagnostic), 1 on success, or 2 if it fails and the calling function should issue a syntax error on behalf of the custom handler.

Appendix E GNU Free Documentation License

Version 1.3, 3 November 2008

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.