SDC Specification

Version 1.0

---

1. Introduction

This document specifies, in detail, what the Simple Data Container binary format is, how to read and write it. Designed for versatility, it can be used in many ways - from saving data to a file to streaming a self-describing packet over a network socket.

Flag values are accessible in the *sdc.h* header with the SDC_ prefix. For example, the FLE flag can be accessed using SDC_FLE.

2. Format

Each data container is specified by a header, followed by a number of entries, each with their own data placed right after its respective header. This means that the stream of data cannot be simply indexed into, but walked one entry at a time. This is because the size of each step is dependent on the entry size, which is different for each data type.

Fig. 1: container with two unnamed[1] entries

In the image above, the h_entries field in the header is set to 2, which is followed by an entry with its data right after it. In the case of the data block having an odd size (for example, a string of length 11) has to place a single byte of padding after the data in order to align the next entry header to a 2-byte boundary.

[1] Naming entries is described in the entry section.

2.1. Main header

The main header is 10 bytes long, and is placed at the beginning of the container. It provides the format version of the file along with flags informing on the way to decode the data.

struct sdc_header

Field	*Size*	*Offset*	*Description*
h_magic	3	0	Always 0x54 0x44 0x43 (“SDC”).
h_version	1	3	Format version.
h_flags	1	4	Data flags.
h_extflags	1	5	Extensions in use.
h_userflags	2	6	User extensions.
h_entries	2	8	Number of entries.

- **h_magic**[2] - 3 byte magic value at the start of the stream, to confirm that the following data is encoded in the SDC format.
- **h_version**[3] - format version in use. Tools may choose to only support a single version, in which case this field must be checked (it should be checked anyways, in the case of any differences in the format). This is an 8-bit value, where the first 4 bits are the major version number and the last 4 bits are the minor version number.

  For example, the version 3.12 is represented as:

  ```
        minor version (12)
        v
  0011 1100
  ^
  major version (3)
  ```

  Giving the hex number 0x3b.

[2] SDC_MAGIC in *sdc.h*
[3] SDC_VERSION in *sdc.h*

- **h_flags** - an 8-bit number with information about the data format.

Flag	*Flag name*	*Description*
00	FLE	Integers are little-endian
01	BLE	Integers are big-endian

- **h_extflags** - an 8-bit number with the list of extensions[4] in use. Note that only officially specified extensions can be used here - the user may define their own in the user flags.

Flag	*Extension name*	*Description*
01	FEXTCOMPACT	Entries with well-defined data sizes are more compact.

- **h_userflags** - a 16-bit number with custom user extensions. Official tools do not check this field, so it can contain some additional meta-information for the user to use.

2.2. Entry

The entry is either 4 or 6 bytes long, depending if the ESIZE32 flag is set, meaning there needs to be an additional 16-bit integer to store the additional 16 bits of the size.

struct sdc_entry

Field	*Size*	*Offset*	*Description*
e_type	1	0	Type of the data.
e_flags	1	1	Entry flags.
e_size	2	2	Size of the data block, or element count.

This structure is always 2-byte aligned in the file or stream. It declares the type of the data stored, some flags about it and the size of the data block.

- **e_type** - an 8-bit number with the type ID. Depending on the type declared here, the data is read differently. See detailed type information later in the document.

[4] See “Extensions” section.

ID	*Type*	*Description*
00	NULL	Null type, stores nothing.
01	INT	32-bit signed integer.
02	LONG	64-bit signed integer.
03	UINT	32-bit unsigned integer.
04	ULONG	64-bit unsigned integer.
05	BOOL	Boolean value.
06	STRING	String value.
07	ARRAY	Array of entries.
08	BYTES	Raw bytes.

- **e_flags** - 8-bit number with information about the entry.

Flag	*Flag name*	*Description*
01	ENAMED	The entry is named.
02	ESIZE32	The 32-bit entry is used instead.

- **e_size** - 16-bit number storing the length of the data block, or the amount of entries when the type is ARRAY. Note that the size stored here does not count the padding byte if the size is odd, to align the next entry structure. In the case that ESIZE32 is used, this field becomes **e_sizelow**, and the next 2 bytes after the structure store the **e_sizehigh** integer. This is then used to calculate the real data block length, by shifting the e_sizehigh integer 16 bits to the left, and OR'ing it with e_sizelow.

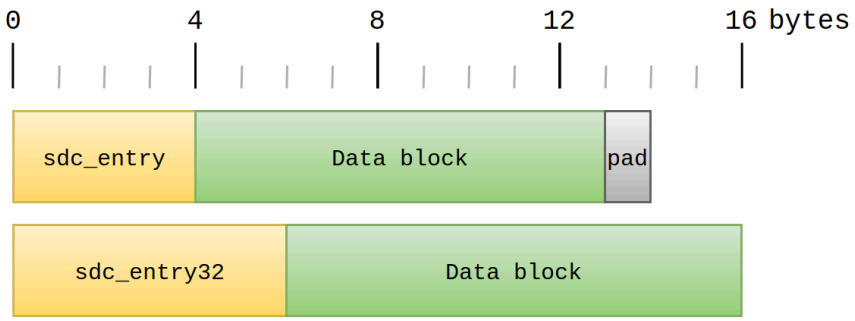

Fig. 2: example entries with their data blocks and padding

The image above shows two example entries, the upper one has a regular 16-bit size, and a single padding byte because the data block has an odd length (9 bytes).

An entry may be named, which means that it has a string right after the entry and the ENAMED flag set. The string itself is specially formatted with the length encoded in it, similar to what the DNS protocol does. Similar to the data block, if the length of the string block is odd, a single padding byte is added after it to align the data block to a 2-byte boundary.

Fig. 3: example named entry with string padding

For example, if the entry stores an integer named “thing”, the name string right after the entry will have the following bytes:

```
05 74 68 69 6e 67
   t  h  i  n  g
```

The string block is split known sizes, with each segment starting with an 8-bit length. In the example above, the string has a single segment which is 5 bytes long. Notice the lack of a null terminator at the end. In the case that the name is longer than 255 bytes, the string is split into multiple 255-byte segments. The last segment in the string block has a size smaller than 0xff.

For example, if the name is a 320 byte string, there are 2 segments of lengths 255 and 65 respectively:

```
ff <first 255 bytes> 41 <last 65 bytes>
```

In the case that there are a multiple of 255 bytes (255, 510, ...) in the name, there has to be an empty segment placed right after, to notify any tool to stop reading. For a 510 byte name:

```
ff <first 255 bytes> ff <last 255 bytes> 00
```

There is no upper limit on the amount of segments, but tools are required to accept at least 2 segments (510 byte names). This means tools *may* throw warnings or errors and stop parsing the container if this limit is reached.

3. Types

There are few real data types the format supports, but with the addition of the BYTES type and a tool that knows how to decode the byte stream, it can virtually support all possible types. This means that even a PNG file may be stored in an SDC file.

NULL
Self explanatory, it cannot contain any data, meaning e_size must be set to 0.

INT
Signed integer, 32-bit. Data size is 4.

LONG
Signed integer, 64-bit. Data size is 8.

UINT
Unsigned integer, 32-bit. Data size is 4.

ULONG
Unsigned integer, 64-bit. Data size is 8.

BOOL
Boolean value, represented as an 8-bit integer. A value of 0 is treated as false, and anything else is resolved to true. Data size is 1.

STRING
String value, which is not null-terminated. Data size is the length of the string.

ARRAY
Multi-type array of entries. This entry type has no data block, but rather notifies the user that the next n entries all belong to an array. Arrays can be nested. Data size is the count of entries in this array. Note that the count of entries does not count nested elements, meaning [1, 2, [4, 5, 6], 3] has a length of 4, not 6.

BYTES
Raw bytes. These bytes are not required to be formatted in any way, or have any escaped bytes. It is the users job to decode them for their particular use case. Data size is the length of the data block.

4. Extensions

Extensions may change how data is read or placed in the file or stream.

FEXTCOMPACT
Offers more compact entries for those applications where memory is sparse, by reusing the possibly redundant data block size field for the data itself. Note that mixing parts of integers into the e_size field may have performance related issues, as they have to be reconstructed into the original integer using multiple possibly unaligned reads.

Here is a list of types that have their data layout changed:

- **INT** & **UINT** - the first 16 bits are placed in the e_size field, and the last 16 bits are placed after the entry. This saves 2 bytes per integer.

- **LONG** & **ULONG** - the first 16 bits are placed in the e_size field, and the remaining 48 bits are placed right after. This saves 2 bytes per long integer.

- **BOOL** - the first byte of e_size is replaced with the 8-bit boolean integer. This saves 1 byte per boolean.