Rekordbox Export Structure Analysis

James Elliott
Deep Symmetry, LLC

March 6, 2019

Abstract

The files written to external media by rekordbox for use in player hardware contain a wealth of information that can be used in place of queries to the remotedb server on the players, which is important because they can be obtained from the players' NFS servers, even if there are four players in use sharing the same media. Under those circumstances, remotedb queries are impossible. This article documents what has been learned so far about the files, and how to interpret them.

Contents

1 Database Exports 3
1.1 File Header . . . 3
1.2 Table Pages . . . 6
1.3 Table Rows . . . 8
1.3.1 Album Rows . . . 8
1.3.2 Artist Rows . . . 9
1.3.3 Artwork Rows . . . 10
1.3.4 Color Rows . . . 10
1.3.5 Genre Rows . . . 10
1.3.6 Key Rows . . . 12
1.3.7 Label Rows . . . 12
1.3.8 Playlist Tree Rows . . . 12
1.3.9 Playlist Entry Rows . . . 13
1.3.10 Track Rows . . . 13
1.4 DeviceSQL Strings . . . 16
1.4.1 Long ASCII Strings . . . 17
1.4.2 Long UTF-16 Big-Endian Strings . . . 17
1.4.3 Short ASCII Strings . . . 17

2 Analysis Files **18**
2.1 Analysis File Header 18
2.2 Analysis File Sections 19
2.2.1 Beat Grid Tag 20
2.2.2 Cue List Tag 21

3 Crate Digger **23**

List of Figures **23**

List of Tables **24**

1 Database Exports

The starting point for finding track metadata from a player is the database export file, which can be found within rekordbox media at the path `PIONEER/rekordbox/export.pdb` (if you are using the Crate Digger `FileFetcher` to request this file, use that path as the `filePath` argument, and use a `mountPath` value of `/B/` if you want to read it from the SD slot, or `/C/` to obtain it from the USB slot).

The file is a relational database format designed to be efficiently used by very low power devices (there were deployments on 16 bit devices with 32K of RAM). Today you are most likely to encounter it within the Pioneer Professional DJ ecosystem, because it is the format that their rekordbox software uses to write USB and SD media which can be mounted in DJ controllers and used to play and mix music.

The file consists of a series of fixed size pages. The first page contains a file header which defines the page size and the locations of database tables of different types, by the index of their first page. The rest of the pages consist of the data pages for all of the tables identified in the header.

Each table is made up of a series of rows which may be spread across any number of pages. The pages start with a header describing the page and linking to the next page. The rest of the page is used as a heap: rows are scattered around it, and located using an index structure that builds backwards from the end of the page. Each row of a given type has a fixed size structure which links to any variable-sized strings by their offsets within the page.

As changes are made to the table, some records may become unused, and there may be gaps within the heap that are too small to be used by other data. There is a bit map in the row index that identifies which rows are actually present. Rows that are not present must be ignored: they do not contain valid (or even necessarily well-formed) data.

The majority of the work in reverse-engineering this format was performed by Henry Betts[1] and Fabian Lesniak[2], to whom I am hugely grateful.

1.1 File Header

Unless otherwise stated, all multi-byte numbers in the file are stored in little-endian byte order. Field names used in the Figures match the IDs assigned to them in the Kaitai Struct specification[3], unless that is

[1]`https://github.com/henrybetts/Rekordbox-Decoding`

[2]`https://github.com/flesniak/python-prodj-link`

[3]`https://github.com/Deep-Symmetry/crate-digger/blob/master/src/main/kaitai/rekordbox_pdb.ksy`

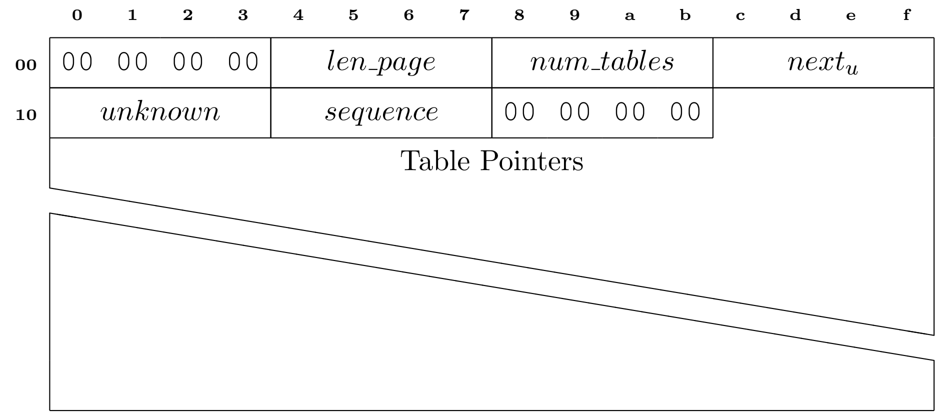

Figure 1: File Header

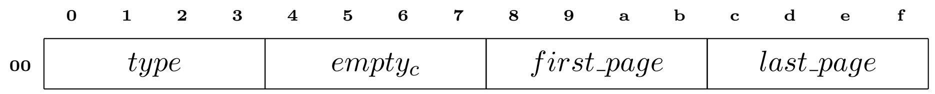

Figure 2: Table Pointer

too long to fit, in which case a subscripted abbreviation is used, and the text will mention the actual struct field name.

The first page begins with the file header, shown in Figure 1. The header starts with four zero bytes, followed by a four-byte integer, *len_page* at byte `04`, that establishes the size of each page (including this first one), in bytes. This is followed by another four-byte integer, *num_tables* at byte `08`, which reports the number of different tables that are present in the file. Each table will have a table pointer entry in the "Table pointers" section of the file header, described below, that identifies and locates the table.

The four-byte integer $next_u$ at byte `0c` has an unknown purpose, but Mr. Lesniak named it `next_unused_page` and said "Not used as any `empty_candidate`, points past the end of the file." The four-byte integer *sequence*, at byte `14`, was described "Always incremented by at least one, sometimes by two or three." and I assume this means it reflects a version number that rekordbox updates when synchronizing to the exported media.

Finally, there is another series of four zero bytes, and then the header ends with the list of table pointers which begins at byte `1c`. There are as many of these as specified by *num_tables*, and each has the structure shown in Figure 2.

Each Table Pointer is a series of four four-byte integers. The first, *type*, identifies the type of table being defined. The known table types are shown in Table 1. The second value, at byte `04` of the Table Pointer, was called `empty_candidate` by Mr. Lesniak. It may link to a chain of empty pages if the database is ever garbage collected, but this is speculation on my part.

Type	**Name**	**Row Content**
0	tracks	Track metadata: title, artist, genre, artwork ID, playing time, etc., Section 1.3.10.
1	genres	Musical genres, for reference by tracks and searching, Section 1.3.5.
2	artists	Artists, for reference by tracks and searching, Section 1.3.2.
3	albums	Albums, for reference by tracks and searching, Section 1.3.1.
4	labels	Music labels, for reference by tracks and searching, Section 1.3.7.
5	keys	Musical keys, for reference by tracks, searching, and key matching, Section 1.3.6.
6	colors	Color labels, for reference by tracks and searching, Section 1.3.4.
7	playlist_tree	Describes the hierarchical tree structure of available playlists and folders grouping them, Section 1.3.8.
8	playlist_entries	Links tracks to playlists, in the right order, Section 1.3.9.
13	artwork	File paths of album artwork images, Section 1.3.3.
16	columns	Details not yet confirmed.
19	history	Records the tracks played during performances.

Table 1: Table Types

Other than the type, the two important values are $first\_page$ at byte `08` and *last_page* at byte `0c`. These tell us how to find the table. They are page indices, where the page containing the file header has index 0, the page with index 1 begins at byte *len_page*, and so on. In other words, the first page of the table identified by the current Table Pointer can be found within the file starting at the byte *len_page* $\times$ $first\_page$.

The table is a linked list of pages: each page contains the index of the next page after it. However, you need to keep track of the

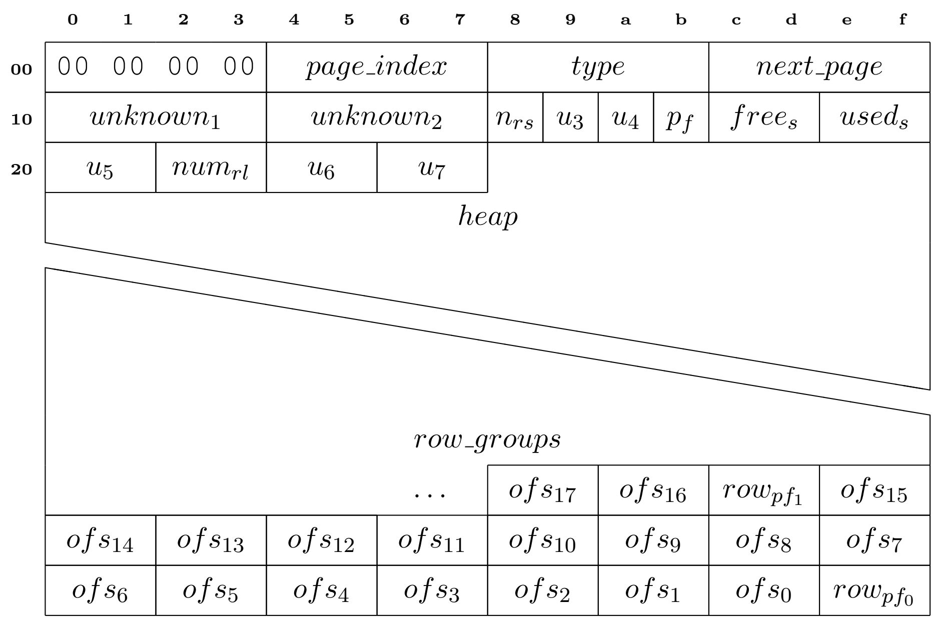

Figure 3: Table Page

last_page value for the table, because it tells you not to try to follow the next page link once you reach the page with that index. (If you do keep going, you will start reading pages of some different table.) The structure of the table pages themselves are described in the next section.

As far as we know, the remainder of the first page after the table pointers is unused.

1.2 Table Pages

The table header is followed by the table pages themselves. These each have the size specified by *len_page* in Figure 1, and the structure shown in Figure 3.

Data pages all seem to have the header structure described here, but not all of them actually store data. Some of them are “strange” and we have not yet figured out why. The discussion below describes how to recognize a strange page, and avoid trying to read it as a data page.

The first four bytes of a table page always seem to be zero. This is followed by a four-byte value *page_index* which identifies the index of this page within the list of table pages (the header has index 0, the first actual data page the index 1, and so on). This value seems to be redundant, because it can be calculated by dividing the offset of the start of the page by *len_page*, but perhaps it serves as a sanity check.

This is followed by another four-byte value, *type*, which identifies the type of the page, using the values shown in Table 1. This again seems redundant because the table header which was followed to reach this page also identified the table type, but perhaps it is another sanity check, or an alternate way to tell, when following page links, that you have reached the end of the table you are interested in. Speaking of which, the next four-byte value, *next_page*, is that link: it identifies the index at which the next page of this table can be found, as long as we have not already reached the final page of the table, as described in Section 1.1.

The exact meaning of $unknown_1$ is unclear. Mr. Flesinak said "sequence number (0→1: 8→13, 1→2: 22, 2→3: 27)" but I don't know how to interpret that. Even less is known about $unknown_2$. But *num_rows_small* at byte `18` within the page (abbrviated n_{rs} in Figure 3) holds the number of rows that are present in the page, unless *num_rows_large* (below) holds a value that is larger (but not equal to `1fff`). This seems like a strange mechanism for dealing with the fact that some tables (like playlist entries) have a lot of very small rows, too many to count with a single byte. But then why not just always use *num_rows_large*?

The purpose of the next two bytes are is also unclear. Of u_3 Mr. Flesniak said "a bitmask (first track: 32)", and he described u_4 as often 0, sometimes larger, especially for pages with a high number of rows (e.g. 12 for 101 rows).

Byte `1b` is called $page\_flags$ (abbrviated p_f in Figure 3). According to Mr. Flesniak, "strange" (non-data) pages will have the value `44` or `64`, and other pages have had the values `24` or `34`. Crate Digger considers a page to be a data page if $page\_flags\&40 = 0$.

Bytes `1c-1d` are called $free\_size$ (abbrviated $free_s$ in Figure 3), and store the amount of unused space in the page heap (excluding the row index which is built backwards from the end of the page); *used_size* at bytes `1c-1d` (abbrviated $used_s$) stores the number of bytes that are in use in the page heap.

Bytes `20-21`, u_5, are of unclear purpose. Mr. Flesniak labeled them "(0→1: 2)."

Bytes `20-21`, *num_rows_large* (abbrviated num_{rl} in Figure 3) hold the number of entries in the row index at the end of the page when that value is too large to fit into *num_rows_small* (as mentioned above), and that situation seems to be indicated when this value is larger than *num_rows_small*, but not equal to `1fff`.

u_6 at bytes `24-25` seems to have the value `1004` for strange pages, and `0000` for data pages. And Mr. Flesniak describes u_7 at bytes `26-27` as "always 0 except 1 for history pages, num entries for strange pages?"

After these header fields comes the page heap. Rows are allocated within this heap starting at byte `28`. Since rows can be different sizes, there needs to be a way to locate them. This takes the form of a row index, which is built from the end of the page backwards, in groups of up to sixteen row pointers along with a bitmask saying which of those rows are still part of the table (they might have been deleted).

The number of row index entries is determined by the value of either *num_rows_small* or *num_rows_large* as described above.

The bit mask for the first group of up to sixteen rows, labeled row_{pf_0} in Figure 3 (meaning "row presence flags group 0"), is found in the last two bytes of the page. The low order bit of this value will be set if row 0 is really present, the next bit if row 1 is really present, and so on. The two bytes before these flags, labeled ofs_0, store the offset of the first row in the page. This offset is the number of bytes past the end of the page header at which the row itself can be found. So if row 0 begins at the very beginning of the heap, at byte `28` in the page, ofs_0 would have the value `0000`.

As more rows are added to the page, space is allocated for them in the heap, and additional index entries are added at the end of the heap, growing backwards. Once there have been sixteen rows added, all of the bits in row_{pf_0} are accounted for, and when another row is added, before its offset entry ofs_{16} can be added, another row bit-mask entry row_{pf_1} needs to be allocated. And so the row index grows backwards towards the rows that are being added forwards, and once they are too close for a new row to fit, the page is full, and another page gets allocated to the table.

1.3 Table Rows

The structure of the rows themselves is determined by the *type* of the table, using the values shown in Table 1.

1.3.1 Album Rows

Album rows hold an album name and ID along with an artist association, with the structure shown in Figure 4. The unknown value at bytes `00-01` seems to usually have the values `80 00`. It is followed by a two-byte value Mr. Flesniak called $index\_shift$, although I don't know what that means, and another four bytes of unknown purpose. But at bytes `08-0b` we finally find a value we have a use for: *artist_id* holds the ID of an artist row associated with this track row. This is followed by *id*, the ID of this track row itself, at bytes `0c-0f`. We assume that there are index tables somewhere that would let us locate the page and row index of a record given its table type and ID, but we have not yet found and figured them out.

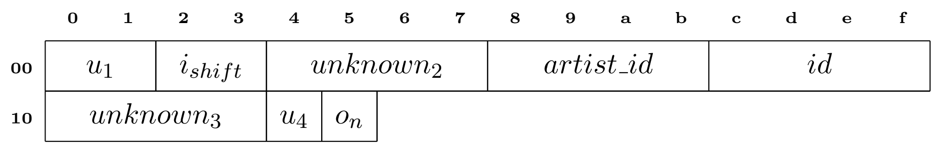

Figure 4: Album Row

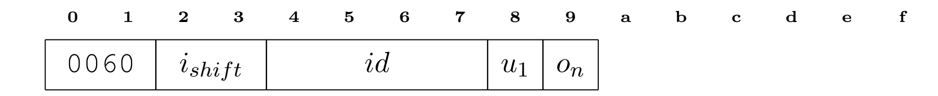

Figure 5: Artist Row with Nearby Name

This is followed by five more bytes with unknown meaning, and the final byte in the row, ofs_name is a pointer to the track name (labeled o_n in Figure 4. To find the location of the name, add ofs_name bytes to the address of the start of the track row itself. The name itself is encoded in a surprisingly baroque way, explained in Section 1.4

1.3.2 Artist Rows

Artist rows hold an Artist name and ID, with the structure shown in Figure 5 or Figure 6. The *subtype* value at bytes `00-01` determines which variant is used. If the artist name was allocated close enough to the row to be reached by a single byte offset, offset, *subtype* has the value `0060`, and the row has the structure in Figure 5. If the name is too far away for that, *subtype* has the value `0064` and the row has the structure in Figure 6.

In either case, *subtype* is followed by the unexplained two-byte value found in many row types that Mr. Flesniak called $index\_shift$, and then by *id*, the ID of this artist row itself, at bytes `04-07`, an unknown value, and ofs_name_near (labeled o_n in Figure 5), the one-byte name offset used only in the first variant.

If *subtype* is `0064`, the value of ofs_name_near is ignored, and instead the two-byte value ofs_name_far (labeled o_{far} in Figure 6) is used.

Whichever name offset is used, it is a pointer to the artist name. To find the location of the name, add the value of the offset to the address of the start of the artist row itself. This gives the address of a DeviceSQL string holding the name, with the structure explained in Section 1.4

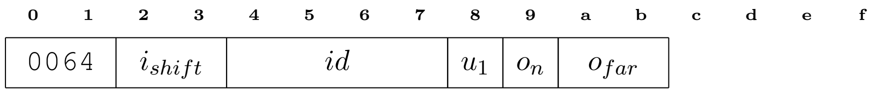

Figure 6: Artist Row with Far Name

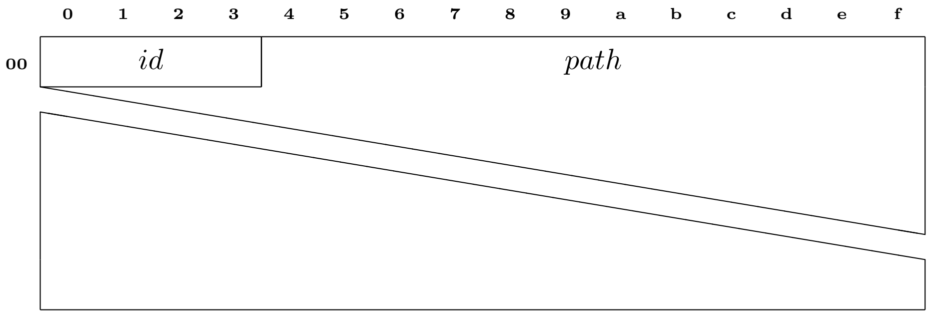

Figure 7: Artwork Row

1.3.3 Artwork Rows

Artwork rows hold an *id* (which tracks refer to) and the path at which the corresponding album art image file can be found, with the structure shown in Figure 7. Note that in this case, the DeviceSQL string *path* is embedded directly into the row itself, rather than being located elsewhere in the heap through an offset. The structure of the string itself is still as described in Section 1.4.

1.3.4 Color Rows

Color rows hold a numeric color *id* (which controls the actual color displayed on the player interface) at bytes `05-06` and a text label or *name* starting at byte `08` which is a DeviceSQL string shown in the information panel for tracks that are assigned the color. The rows have the structure shown in Figure 8. There are several bytes in the row that are not yet known to have any meaning. The structure of *name* is described in Section 1.4.

1.3.5 Genre Rows

Genre rows hold a numeric genre *id* (which tracks can be assigned) at bytes `00-03` and a text *name* starting at byte `04` which is a DeviceSQL string. The rows have the structure shown in Figure 9. The structure of *name* is described in Section 1.4.

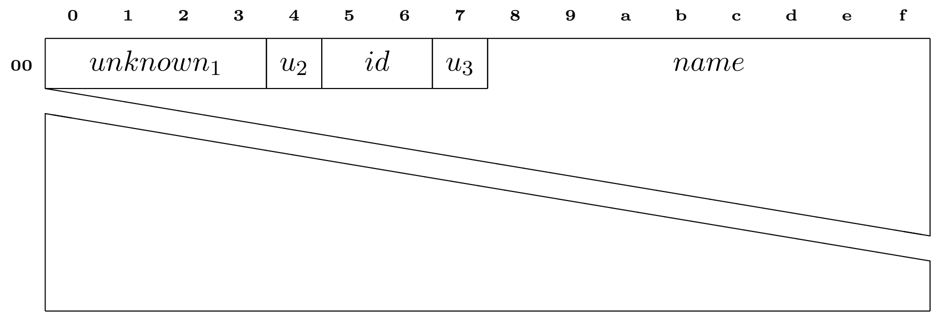

Figure 8: Color Row

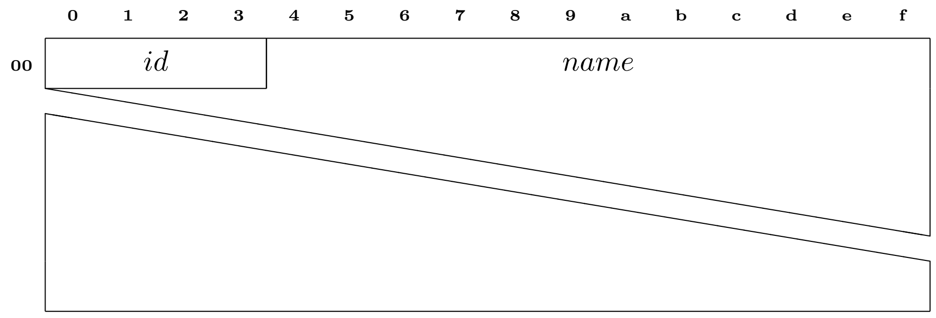

Figure 9: Genre or Label Row

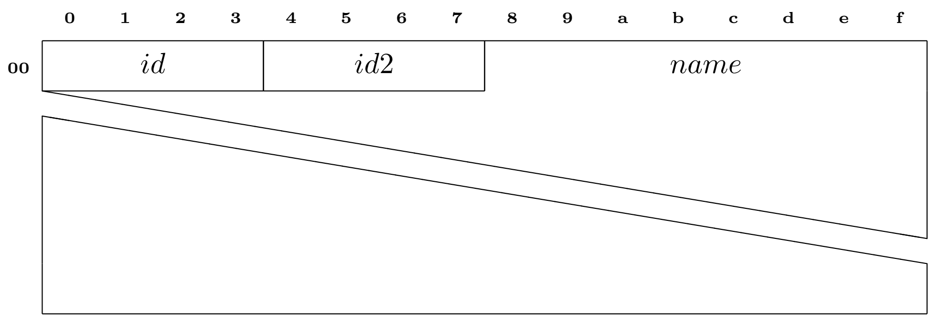

Figure 10: Key Row

1.3.6 Key Rows

Key rows represent musical keys. They hold a numeric *id* (which tracks can be assigned) at bytes `00-03` and a text *name* starting at byte `08` which is a DeviceSQL string. (There seems to be a second copy of the ID at bytes `04-07`.) The rows have the structure shown in Figure 10. The structure of *name* is described in Section 1.4.

1.3.7 Label Rows

Label rows represent record labels. They hold a numeric *id* (which tracks can be assigned) at bytes `00-03` and a text *name* starting at byte `04` which is a DeviceSQL string. The rows have the structure shown in Figure 9. The structure of *name* is described in Section 1.4.

1.3.8 Playlist Tree Rows

Playlist tree rows are used to organize the hierarchical structure of the playlist menu. There is probably an index somewhere that makes it possible to find the right rows directly when loading a playlist, but we have not yet figured out how indices work in DeviceSQL databases, so Crate Digger simply reads all the rows and builds its own in-memory index of the tree.

Playlist tree rows can either represent a playlist "folder" which contains other folders and playlists, or a regular playlist which holds only tracks. The rows are identified by an *id* at bytes `0c-0f`, and also contain a *parent_id* at bytes `00-03` which is how the hierarchical structure is represented: the contents of a folder are the other rows in this table whose *parent_id* folder is equal to the *id* of the folder.

Similarly, the tracks that make up a regular playlist are the Playlist Entry Rows (described in Section 1.3.9) whose *playlist_id* is equal to this row's *id*.

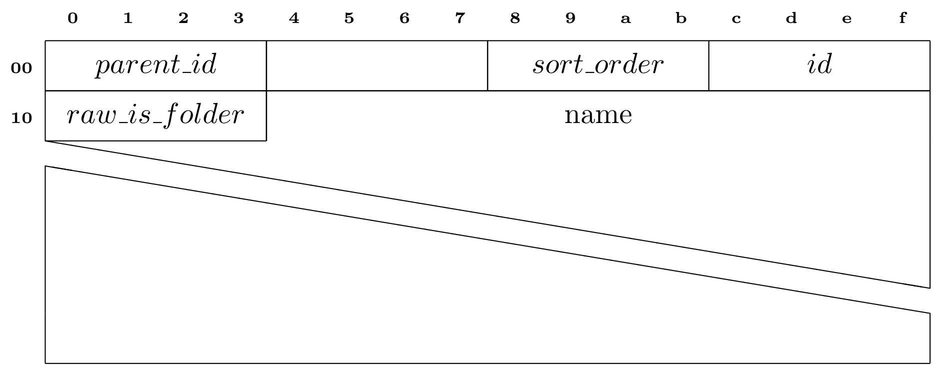

Figure 11: Playlist Tree Row

Each playlist tree row also has a text *name* starting at byte `14` which is a DeviceSQL string displayed when navigating the hierarchy, a *sort_order* indicator at bytes `08-0b` (this may be the same value used to select sort orders when requesting menus using the `dbserver` protocol, shown in Table 7 of the dysentery Protocol Analysis paper, but this has not yet been confirmed), and a value that specifies whether the row defines a folder or a playlist. In the Kaitai Struct, this value is called *raw_is_folder*, is found at bytes `10-13`, and has a non-zero value for folders. For convenience, the struct also defines a derived value, *is_folder*, which is a boolean.

The rows have the structure shown in Figure 11. The structure of *name* is described in Section 1.4.

1.3.9 Playlist Entry Rows

Playlist entry rows list the tracks that belong to a particular playlist, and also establish the order in which they should be played. They have a very simple structure, shown in Figure 12, containing only three values. The *entry_index* at bytes `00-03` specifies the position within the playlist at which this entry belongs. The *track_id* at bytes `04-07` identifies the track to be played at this position in the playlist, by corresponding to the *id* of a row in the Track table described in Section 1.3.10, and the *playlist_id* at bytes `08-0b` identifies the playlist to which it belongs, by corresponding to the *id* of a row in the Playlist Tree described in Section 1.3.8.

1.3.10 Track Rows

Track rows describe audio tracks that can be played from the media export, and provide many details about the music inluding links to

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
entry_index				*track_id*				*playlist_id*							

Figure 12: Playlist Entry Row

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	u_1		i_{shift}		*bitmask*				*sample_rate*				*composer_id*			
10	*file_size*				u_2				u_3		u_4		*artwork_id*			
20	*key_id*				*orig_artist_id*				*label_id*				*remixer_id*			
30	*bitrate*				*track_number*				*tempo*				*genre_id*			
40	*album_id*				*artist_id*				*id*				$disc_n$		$play_c$	
50	*year*		s_{depth}		*dur*		u_5		c_{id}	*r*	u_6		u_7		ofs_0	
60	ofs_1		ofs_2		ofs_3		ofs_4		ofs_5		ofs_6		ofs_7		ofs_8	
70	ofs_9		ofs_{10}		ofs_{11}		ofs_{12}		ofs_{13}		ofs_{14}		ofs_{15}		ofs_{16}	
70	ofs_{17}		ofs_{18}		ofs_{19}		ofs_{20}									

Figure 13: Track Row

other tables like artists, albums, keys, and others. They have the structure shown in Figure 13.

The first two bytes, labeled u_1, have an unknown purpose; they usually are `24` followed by `00`. They are followed by the unexplained two-byte value found in many row types that Mr. Flesniak called *index_shift*, and a four-byte value he called *bitmask*, although we do not know what the bits mean. The value *sample_rate* at bytes `08-0b` is the first one we have a solid understanding of: it holds the playback sample rate of the audio file, in samples per second.

Bytes `0c-0f` hold the value *composer_id* which identifies the composer of the track, if known, as a non-zero *id* value of an Artist row, as discussed in Section 1.3.2. The size of the audio file, in bytes, is found in *file_size* at bytes `10-13`. This is followed by an unknown four-byte value, u_2, which may be another ID, and two unknown two-byte values, u_3 (about which Mr. Flesniak says "always 19048?") and u_4 ("always 30967?").

If there is cover art for the track, there will be a non-zero value in *artwork_id* (bytes `1c-1f`), identifying the *id* of an Artwork row, as discussed in section 1.3.3.

If a dominant musical key was identified for the track there will be a non-zero value in *key_id* (bytes `20-23`), which represents the *id* of a Key row, as discussed in Section 1.3.6. If the track is known to be a remake, the non-zero Artist row *id* (Section 1.3.2) of the original performer will be found in *original_artist_id* at bytes `24-27`. If there is a known record label for the track, the non-zero value in *label_id* will link to the *id* of a Label row *id* as described in Section 1.3.7. Similarly, if there is a known remixer, there will be a non-zero value in *remixer_id* (bytes `2c-2f`) linking to the *id* of an Artist row (Section 1.3.2).

The field *bitrate* at bytes `30-33` stores the playback bit rate of the track, and *track_number* at bytes `34-37` holds the position of the track within its album. *tempo* at bytes `38-3b` holds the playback tempo of the start of the track in beats per minute, multiplied by 100 (in order to support a precision of $\frac{1}{100}$ BPM). If there is a known genre for the track, there will be a non-zero value in *genre_id* at bytes `3c-3f`, representing the *id* of a Genre row as discussed in Section 1.3.5.

If the track is part of an album, there will be a non-zero value in *album_id* at bytes `40-43`, and this will be the *id* of an Album row as discussed in Section 1.3.1. The Artist row *id* (Section 1.3.2) of the primary performer associated with the track is found in *artist_id* at bytes `44-47`. And the *id* of the track itself is found in *id* at bytes `48-4b`. If the album is known to consist of multiple discs, the disc number on which this track is found will be in *disc_number* at bytes `4c-4d`. And the number of times the track has been played is found in *play_count* (bytes `4e-4f`).

The year in which the track was recorded, if known, is in *year* at bytes `50-51`. The sample depth of the track audio file (bits per sample) is in *sample_depth* at bytes `52-53`. The playback time of the track (in seconds, at normal speed) is in *duration* at bytes `54-55`. The purpose of the next two bytes, labeled u_5, is unknown, they seem to always hold the value 41.

Byte `58`, *color_id* (labeled c_{id} in Figure 13), holds the color assigned to the track in rekordbox, as the *id* of a Color row (described in Section 1.3.4), or zero if no color has been assigned. Byte `59`, *rating* (labeled r in Figure 13) holds the rating (0 to 5 stars) assigned the track. The next two bytes, labeled u_6, have an unknown purpose, and seem to always have the value 1. The two bytes after them, labeled u_7, are also unknown; Mr. Flesniak said "alternating 2 and 3".

The rest of the track row is an array of 21 two-byte offsets that point to DeviceSQL strings. To find the start of the string, add the address of the start of the track row to the offset. The purpose of each string is described in Table 2, and the structure of the strings themselves is explained in Section 1.4.

For convenience, the strings can be accessed as Kaitai Struct instance values with the names shown in the table.

Index	Name	Content
0	*unknown_string_1*	Unknown, so far always empty.
1	*texter*	Unknown, named by `@flesniak`.
2	*unknown_string_2*	Unknown, "thought track number, wrong".
3	*unknown_string_3*	Unknown, "Strange strings."[4].
4	*unknown_string_4*	"Strange strings" (as above).
5	*messsage*	Unknown, named by `@flesniak`.
6	*kuvo_public*	Empty or `"ON"`.[5]
7	*autoload_hotcues*	Empty or `"ON"`.[6]
8	*unknown_string_5*	Unknown.
9	*unknown_string_6*	Unknown, usually empty.
10	*date_added*	When track was added to collection.
11	*release_date*	When track was released.
12	*mix_name*	Name of the track remix.
13	*unknown_string_7*	Unknown, usually empty.
14	*analyze_path*	File path of track analysis, see Section 2.
15	*analyze_date*	When track analysis was performed.
16	*comment*	Track comment assigned by DJ.
17	*title*	Track title.
18	*unknown_string_8*	Unknown, usually empty.
19	*filename*	Name of track audio file.
20	*file_path*	File path of track audio.

Table 2: Track Offset Strings

1.4 DeviceSQL Strings

Many row types store string values, sometimes by directly embedding them, but more often by storing an offset to a location elsewhere in the heap. In either case the string itself uses the strange structure described in this section. Strings can be stored in a variety of formats. The first byte of the structure, labeled *length_and_kind* in the parsed Kaitai Struct, identifies the encoding type and, when the value is odd, also the length (for short ASCII strings), as detailed in Section 1.4.3.

[4]Often zero length, sometimes low binary values, ASCII `01` or `02` as content.

[5]Apparently used rather than a simple bit flag to control whether the track information is visible on Kuvo.

[6]Apparently used rather than a simple bit flag to control whether hot cues are automatically loaded for the track.

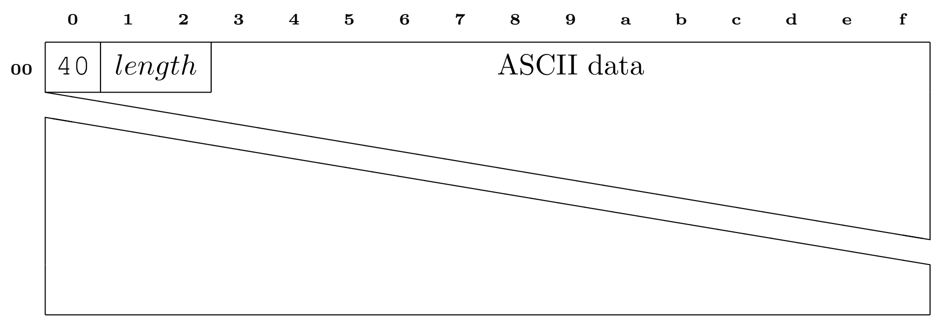

Figure 14: Long ASCII DeviceSQL String

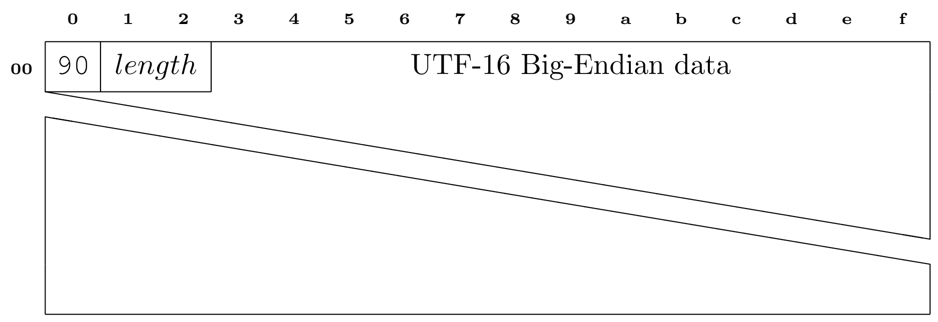

Figure 15: Long UTF-16-BE DeviceSQL String

1.4.1 Long ASCII Strings

If *length_and_kind* has the value `40`, it is followed by a two-byte *length* field, and then followed by that many bytes of ASCII-encoded string data, as shown in Figure 14.

1.4.2 Long UTF-16 Big-Endian Strings

If *length_and_kind* has the value `90`, it is followed by a two-byte *length* field, and then followed by that many bytes of UTF-16 big-endian encoded string data, as shown in Figure 15.

1.4.3 Short ASCII Strings

If *length_and_kind* has an odd value it is a *mangled_length*, labeled m_l in Figure 16. This means we are dealing with a short ASCII DeviceSQL string. To find the length of the string data (which immediately follows this byte), subtract 1 from *mangled_length*, divide it by 2, and subtract

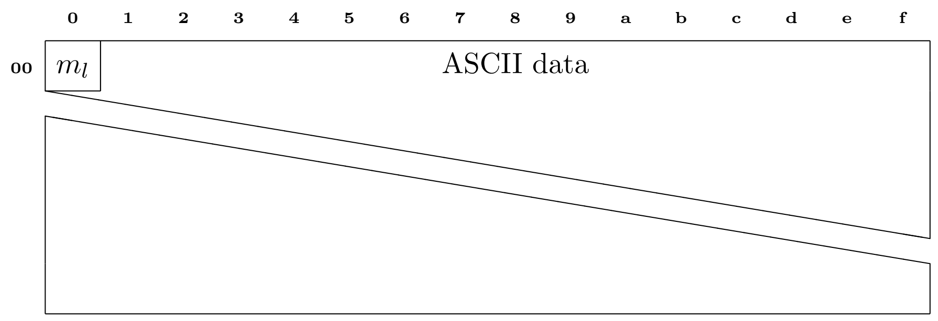

Figure 16: Short ASCII DeviceSQL String

1 again.

2 Analysis Files

When rekordbox analyzes tracks there is some data that is too big to fit in the database itself. We have already seen some of that (the album art images, and of course the track audio is left in the filesystem as well). The other analysis data is organized into “anlz” files, whose path can be found in the DeviceSQL string pointed to by index 14 in the string offsets found at the end of the corresponding track row (see Table 2 in Section 1.3.10). These files have names like `ANLZ0001.DAT` and their structure is described in this section.

The files are “tagged type” files, where there is an overall file header section, and then each entry in the file has its own header which identifies the type and length of that section.

Later player hardware added support for things like colored and higher-resolution waveforms. Apparently these were deemed too large to fit in the `.DAT` files, so another file was introduced, which shares the same base filename as the `.DAT` file, but uses an extension of `.EXT` instead. Both kinds of file share the same structure, but different sets of tags can be found in each.

2.1 Analysis File Header

For some reason the analysis files store their numbers in big-endian byte order, the opposite of the `export.pdb` database file. Field names used in the Figures match the IDs assigned to them in the Kaitai Struct specification[7], unless that is too long to fit, in which case a subscripted

[7] `https://github.com/Deep-Symmetry/crate-digger/blob/master/src/main/kaitai/rekordbox_anlz.ksy`

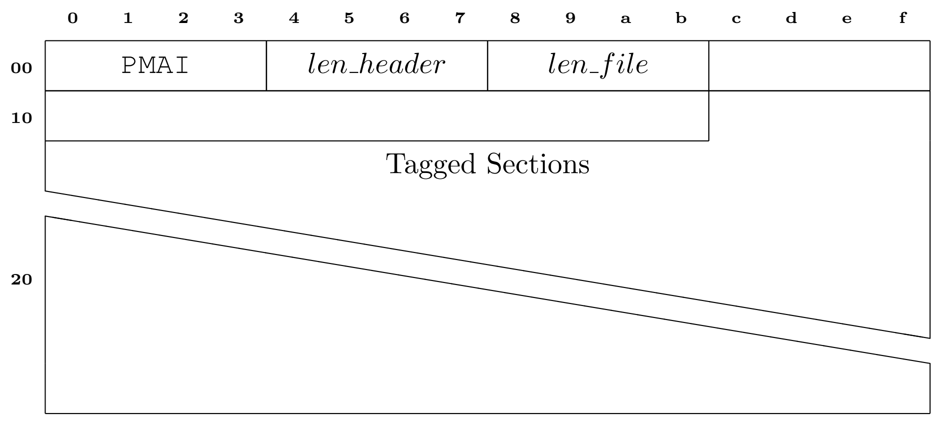

Figure 17: Analysis File Structure

abbreviation is used, and the text will mention the actual struct field name.

The file itself starts with the four-character code `PMAI` that identifies its format. This file format identifier is followed a four-byte value at bytes `04-07`, *len_header*, that specifies the length of the file header in bytes. This is followed by another four-byte value, *len_file*, at bytes `08-0b` that specifies the length of the whole file in bytes.

The header seems to usually be `1c` bytes long, though we do not yet know the purpose of any of the header values that come after *len_file*. After the header, the file consists of a series of tagged sections, each with their own four-character code identifying the seciton type, followed by a header and the section content. This overall structure is illustrated in Figure 17, and the structure of the known tag types is described next.

2.2 Analysis File Sections

The structure of each tagged section has an "envelope" that can be understood even if the internal structure of the section is unknown, making it easy to navigate through the file looking for the section you need. This structure is very similar to the file itself, and is illustrated in Figure 18.

Every section begins with a four-character code, *fourcc*, identifying its specific structure and content, as described in the sections below. This is followed by a four-byte value, *len_header*, which specifies how many bytes there are in the section header, and another four-byte value, *len_tag*, which specifies the length of the entire tagged section (including the header), in bytes. This value can be added to the address

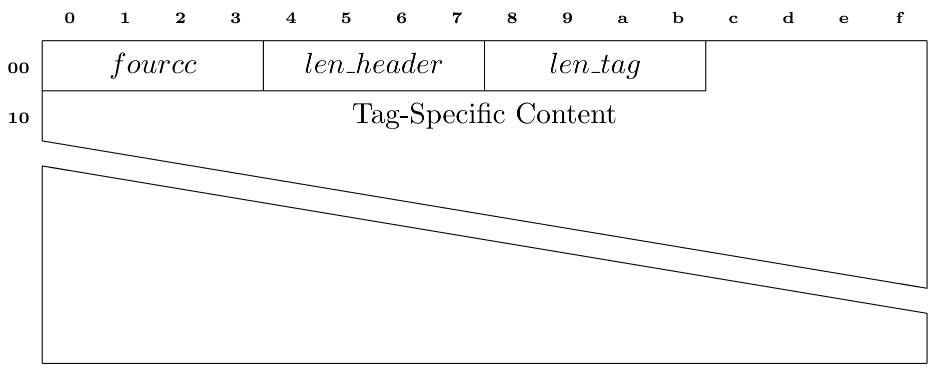

Figure 18: Tagged Section Structure

of the start of the tag to find the start of the next tag.

There is not much value to *len_header*. If you study the structure of each type of tagged section, you can see some sense of where the "header-like stuff" ends, and "content-like stuff" begins, and this seems to line up with the value of *len_header*. But because there are important values in each tag's header, and those always start immediately after *len_tag*, it is simply easier to ignore *len_header*, and model the tag body as beginning at byte `0c` of the tag. To show where the boundary occurs, in the diagrams that follow, values that fall inside the byte range of the header are colored yellow.

2.2.1 Beat Grid Tag

This kind of section holds a list of all beats found within the track, recording their bar position, the time at which they occur, and the tempo at that point. It is identified by the four-character code `PQTZ`, which may stand for "Pioneer Quantization". It has the structure shown in Figure 19. *len_header* is `18`. The tag-specific content starts with two unknown values, although Mr. Flesniak says that $unknown_2$ seems to always have the value `00800000`.

len_beats at bytes `14-17` specifies the number of beats were found in the track, and thus the number of beat entries that will be present in this section. The beat entries come next, and each has the structure shown in Figure 20.

Each beat entry is eight bytes long. It starts with a two-byte number, *beat_number* (abbreviated b_{num} in Figure 20) which specifies where the beat falls within its measure. So the value is always 1, 2, 3, or 4. This is followed by a two-byte *tempo* value, which records the track tempo at the point where this beat occurs, in beats per minute

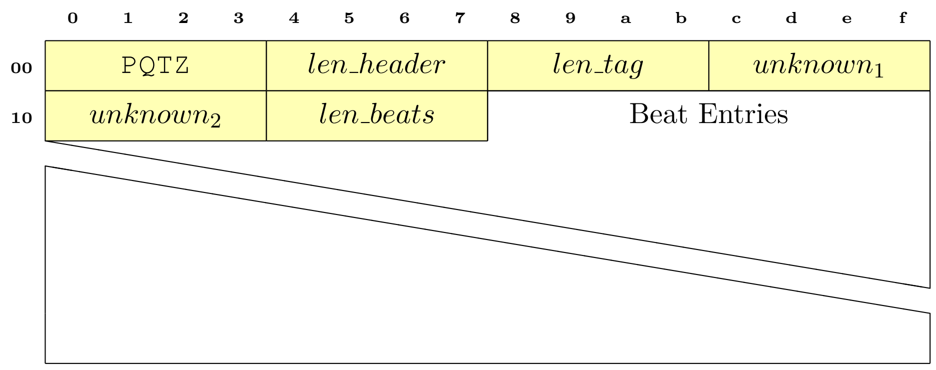

Figure 19: Beat Grid Tag

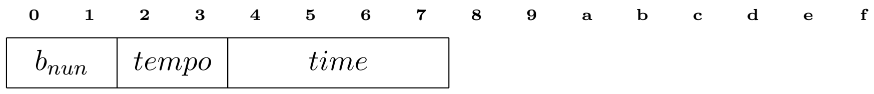

Figure 20: Beat Grid Beat

multiplied by 100 (to allow a precision of $\frac{1}{100}$ BPM). Finally, there is a four-byte *time* value, which specifies the time at which this beat would occur, in milliseconds, when playing the track at its normal speed.

As noted above, there will be as many beat entries as *len_beats* specifies. They continue to the end of the tag.

2.2.2 Cue List Tag

This kind of section holds either a list of ordinary memory cues and loop points, or a list of hot cues and loop points. It is identified by the four-character code `PCOB`, and has the structure shown in Figure 21. *len_header* is `18`.

The *type* value at bytes `0c-0f` determines whether this section holds memory cues (if *type* is `0`) or hot cues (if *type* is `1`). The number of cue entries present in the section is reported in *len_cues* at bytes `10-13`, and we don't yet know the meaning of *memory_count* at bytes `14-17`. The remainder of the section, from byte `18` through *len_tag* holds the cue entries themselves, with the structure shown in Figure 22.

Each cue entry is `38` bytes long. It is structured as its own miniature tag for unknown reasons, starting with the four-character code `PCPT` (Pioneer Cue Point?), and its own internal four-byte *len_header* and *len_entry* values (`1c` and `38` respectively).

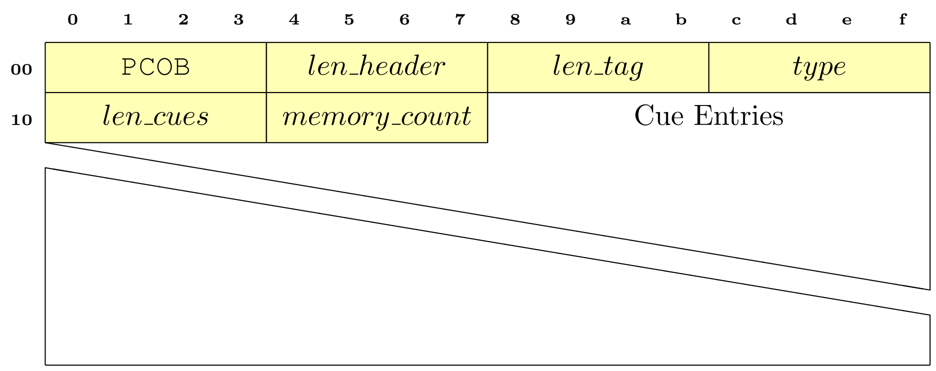

Figure 21: Cue List Tag

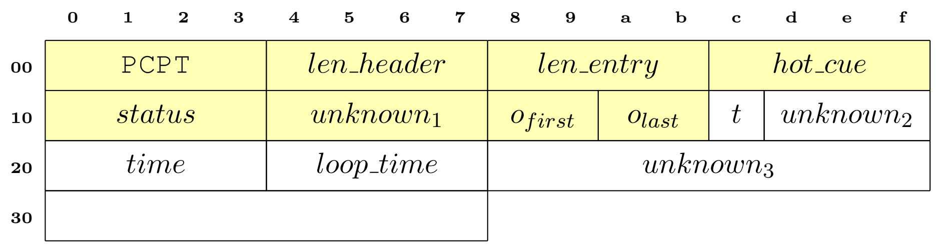

Figure 22: Cue List Entry

If the cue is an ordinary memory point, *hot_cue* at bytes `0c-0f` will be zero, otherwise it identifies the number of the hot cue that this entry represents (Hot Cue A is number 1, B is 2, and so on). The *status* value at bytes `10-13` seems to be a deletion indicator; if it is zero, the entry is ignored. Cues which the players pay attention to have the value 1 here.

The next four bytes have an unknown purpose, but seem to always have the value `00100000`. They are followed by two two-byte values, which seem to be for sorting the cues in the proper order in some strange way. *order_first* at bytes `1a-1b` (labeled o_{first} in Figure 22) has the value `ffff` for the first cue, `0000` for the second, then 2, 3 and on. *order_last* at bytes `1a-1b` (labeled o_{last}) has the value 1 for the first cue, 2 for the second, and so on, but `ffff` for the last. It would seem that the cues could be perfectly well sorted by just one of these fields, or indeed, by their *time* values.

The first "non-header" field is *time* at byte `1c` (labeled t in Figure 22), and it specifies whether the entry records a simple position (if it has the value 1) or a loop (if it has the value 2). The next three bytes have an unknown purpose, but seem to always have the values `0003e8`, or decimal 1000.

The value *time* at bytes `20-23` records the position of the cue within the track, as a number of milliseconds (representing when the cue would occur if the track is being played at normal speed). If *type* is 2, so this cue stores a loop, then *loop_time* at bytes `24-27` stores the track time in milliseconds at which the player should loop back to *time*.

We do not know what, if anything, is stored in the remaining bytes of the cue entry.

To be continued...

3 Crate Digger

You can find a Java library that can parse the structures described in this research, and that can retrieve them from players' NFS servers, at: `https://github.com/deep-symmetry/crate-digger`

The project also contains Kaitai Struct specifications for the file structures, which were used to automatically generate Java classes to parse them, and which can be used to generate equivalent code for a variety of other programming languages.

There are also ONC RPC specification files which were similarly used to generate Java classes to communicate with the NFSv2 servers in the players, and which can likely be used to generate structures for other languages as well.

List of Figures

1 File Header 4
2 Table Pointer 4
3 Table Page 6
4 Album Row 9
5 Artist Row with Nearby Name 9
6 Artist Row with Far Name 10
7 Artwork Row 10
8 Color Row 11
9 Genre or Label Row 11
10 Key Row 12
11 Playlist Tree Row 13
12 Playlist Entry Row 14
13 Track Row 14
14 Long ASCII DeviceSQL String 17
15 Long UTF-16-BE DeviceSQL String 17
16 Short ASCII DeviceSQL String 18
17 Analysis File Structure 19
18 Tagged Section Structure 20
19 Beat Grid Tag 21
20 Beat Grid Beat 21
21 Cue List Tag 22
22 Cue List Entry 22

List of Tables

1 Table Types 5
2 Track Offset Strings 16

deep Symmetry

`http://deepsymmetry.org`