TBM File Format Specification: Major Revision 2

Table of Contents

Introduction

This document is a formal specification of the file format for TrackerBoy module files, or *.tbm files. This document is for major revision 2 of the file format.

A Module file is a serialized form of the Module object.

The libtrackerboy library provides a reference implementation for serializing and deserialing module files with this format. See the io module for more documentation.

NOTE All multi-byte fields in this specification are stored in little-endian byte order.

Changes

Major 2 introduces the following changes to the format:

- Added a `Float32` type to the specification.
- The `customFramerate` field in the header is now a `Float32` instead of `Uint16`.
- Added `systemOverride` and `customFramerateOverride` fields to the `SongFormat` record.
- Replaced the initial envelope setting with a sequence in INST blocks
    - Added a new sequence to INST blocks, `skEnvelope`
    - Replaced the `InstrumentFormat` record with a single `Uint8` byte being the channel setting

This allows for two new features: per-song timing and envelope sequences.

Definitions

Module

A container of songs, instruments and waveforms. Instruments and waveforms are shared between all songs. A module can store up to 256 songs, 64 instruments and 64 waveforms.

Module Piece

An individual part of the module as a song, instrument or waveform. This individual part can be stored as a separate file.

Order

Song order data, a list of OrderRows that determine each pattern in the song.

OrderRow

A track id assignment for each channel. Determines the tracks which make a pattern.

Pattern

A collection of Tracks, one for each channel.

Track

Song data for a single channel. A Track is a list of TrackRows.

File extensions

It is recommended that the following file extensions are used for files adhering to this specification:

***.tbm**

TrackerBoy Module file

***.tbi**

TrackerBoy Instrument file

***.tbs**

TrackerBoy Song file

***.tbw**

TrackerBoy Waveform file

Basic Types

Below are basic data types used throughout the format

Type name	Size (bytes)	Description
Uint8	1	Unsigned 8-bit integer (0-255)
BiasedUint8	1	Unsigned 8-bit integer, biased form (1-256)
Char	1	8-bit Character type
Bool	1	Boolean as an uint8 (0 for false, nonzero for true)
Uint16	2	Unsigned 16-bit integer, in little endian (0-65536)
Uint32	4	Unsigned 32-bit integer, in little endian
LString	2 + *length*	Length-prefixed UTF-8 string, string data prefixed by a Uint16 *length*
Float32	4	IEEE 754 single-precision floating point number

Error Codes

Below is a list of error codes possible when deserialing/serializing a module file. After processing a file, one of these error codes, or Format Result (fr), is given.

Identifier	Code	Description
frNone	0	No error, format is acceptable
frInvalidSignature	1	File has an invalid signature
frInvalidRevision	2	File has an unrecognize revision, possibly from a newer version of the format
frCannotUpgrade	3	An older revision file could not be upgraded to the current revision
frInvalidSize	4	A payload block was incorrectly sized
frInvalidCount	5	The icount and/or wcount in the header was too big
frInvalidBlock	6	An unknown identifier was used in a payload block
frInvalidChannel	7	The format contains an invalid channel in a payload block
frInvalidSpeed	8	The format contains an invalid speed in a SONG block
frInvalidRowCount	9	A TrackFormat’s rows field exceeds the Song’s track size
frInvalidRowNumber	10	A RowFormat’s rowno field exceeds the Song’s track size
frInvalidId	11	An INST or WAVE block contains an invalid id
frDuplicatedId	12	Two INST blocks or two WAVE blocks have the same id
frInvalidTerminator	13	The file has an invalid terminator
frReadError	14	An read error occurred during processing
frWriteError	15	A write error occurred during processing

Structure

A TrackerBoy module consists of a Header, Payload and Terminator.

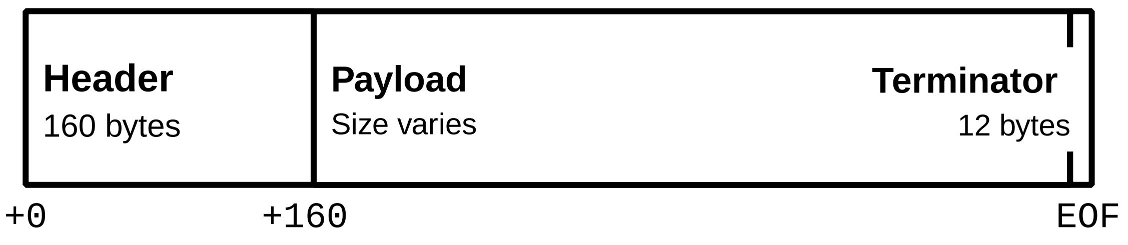

Figure 1. TBM File structure

Header

The figure below defines the Header structure used in all file types. All multi-byte fields are stored in little-endian. The size of the header is a fixed 160 bytes, with any unused space marked as reserved. Reserved fields can be utilized for future revisions of the format. Reserved fields should be set to zero, but this is not enforced.

The layout of the header depends on the header revision, located in offset 24. The current revision of the header is shown below.

```
    +0         +1         +2        +3
0   +-------------------------------------------+
    |                                           |
    | signature ( TRACKERBOY )                  |
    |                                           |
12  +-------------------------------------------+
    | version major                             |
16  +-------------------------------------------+
    | version minor                             |
20  +-------------------------------------------+
    | version patch                             |
24  +----------+----------+---------------------+
    | m. rev   | n. rev   | reserved            |
28  +----------+----------+---------------------+
    |                                           |
    |                                           |
    |                                           |
    | title                                     |
    |                                           |
    |                                           |
    |                                           |
    |                                           |
60  +-------------------------------------------+
    |                                           |
    |                                           |
    |                                           |
    | artist                                    |
    |                                           |
    |                                           |
    |                                           |
    |                                           |
92  +-------------------------------------------|
    |                                           |
    |                                           |
    |                                           |
    | copyright                                 |
    |                                           |
    |                                           |
    |                                           |
    |                                         |
124 +-----------------------------------------+
    | icount   | scount   | wcount   | system   |
128 +----------+----------+----------+----------+
    | customFramerate                         |
    +-----------------------------------------+
    |                                         |
    |                                         |
    |                                         |
    | reserved                                |
    |                                         |
    |                                         |
    |                                         |
    |                                         |
160 +-----------------------------------------+
```

Offset	Size	Type	Field name
+0	12	Char[12]	signature
+12	4	Uint32	versionMajor
+16	4	Uint32	versionMinor
+20	4	Uint32	versionPatch
+24	1	Uint8	m. rev
+25	1	Uint8	n. rev
+26	2	Char[2]	reserved
+28	32	Char[32]	title
+60	32	Char[32]	artist
+92	32	Char[32]	copyright
+124	1	Uint8	icount
+125	1	BiasedUint8	scount
+126	1	Uint8	wcount
+127	1	Uint8	system
+128	4	Float32	customFramerate
+132	28	Char[28]	reserved

Signature

Every TrackerBoy file begins with this signature:

Table 1. TBM Signature format

+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
`'\0'`	`'T'`	`'R'`	`'A'`	`'C'`	`'K'`	`'E'`	`'R'`	`'B'`	`'O'`	`'Y'`	`'\0'`

in order to identify the file as a TrackerBoy file.

Version (major, minor, patch)

Version information is stored as three 4-byte words. This information determines which version of trackerboy that created the file. Versioning is maintained by keeping a major and minor version, followed by a patch number. For example, if the trackerboy version is v1.0.2, then the header's version fields will contain `0x1` `0x0` and `0x2` for major, minor and patch, respectively.

Major revision (m. rev)

This version number indicates a breaking change for the file format. Starts at 0 and is incremented whenever the layout of the header or payload changes. TrackerBoy will not attempt to read modules with a newer major version, but can attempt to read older versions (backwardscompatible).

Examples of breaking changes:

- Modifying the layout of the Header structure
- Adding/removing blocks to the payload
- Modifying the format of a payload block

Minor revision (n. rev)

This version number indicates a change in the format that is forward-compatible with older versions. Changes such as utilizing a reserved field in the header.

NOTE TrackerBoy can read any module file as long as its major revision is less than or equal to the current revision. Saving always uses the current revision, so saving an older major version is a one-way upgrade.

Author information (title, artist, copyright)

These fields in the header are fixed 32 byte strings. Assume ASCII encoding. Any unused characters in the string should be set to 0 or `\0`. Since these strings are fixed, null-termination is not needed.

NOTE The size and naming of these strings are identical to the ones in *.gbs file format. This is intentional, as exporting to gbs is a planned feature.

icount, scount and wcount

icount
: instrument count

scount
: song count

wcount
: waveform count

These counter fields determine the number of INST, SONG and WAVE blocks present in the payload, respectively. `icount` and `wcount` can range from 0-64 and is unbiased. `scount` can range from 0-255 and is biased (a value of 0 means there is 1 SONG block).

System

The system field determines which Game Boy model this module is for. Since the driver is typically updated every vblank, the system field determines the framerate, tick rate or vblank interval for the driver. The available choices are listed in the following table:

Table 2. Valid system values

Identifier	Value	System name	Tick rate
systemDmg	0	DMG	59.7 Hz
systemSgb	1	SGB	61.1 Hz
systemCustom	2	N/A	varies

If the system is `systemCustom`, then a custom tick rate is used instead of the system's vblank. The custom tick rate is stored in the `customFramerate` field of the header. This custom tick rate must be a positive number and nonzero. The implementation should default to 30 fps when the custom tick rate does not meet this criteria.

If the system does not match any of these values, then the implementation should treat the system as the default, `systemDmg`.

Reserved

The remainder of the header is the `reserved` field(s). This is an unused section that may be utilized in future versions of the format if needed.

This field **should** be zeroed but the specification does not require it. The field can be safely ignored when processing module files.

Payload

The payload is located right after the header (offset 160). It contains a variable number of "blocks" or tagged data with a size.

Blocks

A block in the payload contains three parts: the id, the length and the data. The format of the block is shown below:

Table 3. Format of a payload block.

Offset	Size	Description
0	4	Identifier
4	4	Length
8	*Length*	Data

Block types

Each block has an identifier, which determines the type of data present in the block. The table below lists all recognized identifiers in the payload.

Table 4. Block types used in TBM files

Identifier	Value (Uint32)	Description
"COMM"	0x4D4D4F43	User set comment data for a module.
"SONG"	0x474E4F53	Container for a single song.
"INST"	0x54534E49	Container for a single instrument.
"WAVE"	0x45564157	Container for a single waveform.

Block ordering

Blocks are stored categorically by type in the following order:

Table 5. Order and total number of blocks in the payload

Order	Identifier	Count
1	COMM	1
2	SONG	1-256
3	INST	0-64
4	WAVE	0-64

A payload will always have exactly one COMM block, at least one SONG block, and 0 or more INST and WAVE blocks.

COMM block format

The COMM block just contains a UTF-8 string that is the user’s comment data. The string is not null-terminated since the length of the string is the length of the block. If the user has no comment set, then this block is empty (length = 0).

SONG block format

A SONG block contains the data for a single song in the module. Songs are stored in the same order as they were in the module's song list. The first song block is song #0 and so on.

Song data is composed of the following, in this order:

1. The name, `LString`
2. A `SongFormat` record
3. The song order, as an array of `OrderRow` or `Char[4][patternCount]`
4. The track data, as a sequence of `TrackFormat` and `RowFormat` records

Song name

The first part of a SONG block is the song's name, as an `LString`.

SongFormat

Following the name is a `SongFormat` record:

Offset	Size	Type	Field name
+0	1	BiasedUint8	rowsPerBeat
+1	1	BiasedUint8	rowsPerMeasure
+2	1	Uint8	speed
+3	1	BiasedUint8	patternCount
+4	1	BiasedUint8	rowsPerTrack
+5	2	Uint16	numberOfTracks
+7	1	Uint8	numberOfEffects
+8	1	Uint8	systemOverride
+9	4	Float32	customFramerateOverride

rowsPerBeat
: number of rows that make up a beat, used by the front end for highlighting and tempo calculation.

rowsPerMeasure
: number of rows that make up a measure, used by the front end for highlighting.

speed
: Initial speed setting for the song in Q4.4 format

patternCount
: number of patterns for the song

rowsPerTrack

the size, in rows, of a track (all tracks have the same size).

numberOfTracks

number of tracks stored in this song block.

numberOfEffects

this byte contains the number of effect columns visible for each channel. Each count ranges from 1-3 and is stored as a 2 bit number in this byte. Bits 0-1 are the count for CH1, bits 2-3 are the count for CH2 and so on.

NOTE This value is for UI purposes only and has no effect on playback!

systemOverride

This byte determines if the song should use a different tick rate than the one specified in the module. The following table shows all possible values:

Value	System	Description
0	N/A	Use the module's system setting (default)
1	systemDmg	Force DMG system
2	systemSgb	Force SGB system
3	systemCustom	Force custom framerate using `customFramerateOverride` field.

customFramerateOverride

The custom framerate to use if `systemOverride` was set to `3`. This value must be a positive number.

Song Order

Next is the song order, an array of `OrderRow` records with the dimension being the patternCount field from the song format record. An `OrderRow` record is a set of 4 `Uint8` track ids, with the first being the track id for channel 1 and the last being the id for channel 4.

Track Data

Finally, the rest of the block contains the pattern data for every track in the song. Each track gets its own `TrackFormat` record and an array of `RowFormat` records.

The `TrackFormat` record:

Offset	Size	Type	Field name
+0	1	Uint8	channel
+1	1	Uint8	trackId
+2	1	BiasedUint8	rows

channel

determines which channel the track is for. *Valid values 0-3*.

trackId

determines the track id to use for this track

rows

the number of RowFormat records that follow this structure

The `RowFormat` record:

Offset	Size	Type	Field name
+0	1	Uint8	rowno
+1	8	TrackRow	rowdata

rowno

the index in the track's row array to set

rowdata

the data to set at this index, where the `TrackRow` type is:

Offset	Size	Type	Field name
+0	1	Uint8	note
+1	1	Uint8	instrument
+2	6	Effect[3]	effects

with `Effect` being:

Offset	Size	Type	Field name
+0	1	Uint8	effectType
+1	1	Uint8	effectParam

and `effectType` should be any of the following:

Value	Effect Syntax	Identifier
0	---	etNoEffect
1	Bxx	etPatternGoto
2	C00	etPatternHalt
3	Dxx	etPatternSkip
4	Fxy	etSetTempo
5	Txx	etSfx
6	Exx	etSetEnvelope

7	V0x	etSetTimbre
8	I0x	etSetPanning
9	Hxx	etSetSweep
10	Sxx	etDelayedCut
11	Gxx	etDelayedNote
12	L00	etLock
13	0xy	etArpeggio
14	1xx	etPitchUp
15	2xx	etPitchDown
16	3xx	etAutoPortamento
17	4xy	etVibrato
18	5xx	etVibratoDelay
19	Pxx	etTuning
20	Qxy	etNoteSlideUp
21	Rxy	etNoteSlideDown
22	Jxy	etSetGlobalVolume

The last `RowFormat` record for the last track ends the `SONG` block.

INST block format

An INST block contains the data for a single instrument. The data is structured in this order:

1. The instrument’s id, `Uint8`
2. The instrument’s name, `LString`
3. The instrument’s channel, `Uint8`
4. 5 sequences each composed of:
    a. A `SequenceFormat` record
    b. The sequence’s data

Id and Name

The `INST` block data begins with a 1 byte id (0-63), followed by an `LString` name.

NOTE `WAVE` blocks also begin with an id and name in the same format.

Channel

Next is a single byte representing the instrument’s channel setting. This byte must be either 0, 1, 2 or 3 for CH1, CH2, CH3, CH4, respectively.

NOTE Instruments can be used on any channel, this setting is only used for previewing on the set channel, as well as organizational purposes.

Sequence data

Following the InstrumentFormat record is the sequence data for each of the instrument’s sequences. Data for a sequence is structured as a `SequenceFormat` record followed by the sequence data. There are five kinds of sequences for every instrument. The kind of sequence the data is for is determined by its order in the block:

Order	SequenceKind
0	skArp
1	skPanning
2	skPitch
3	skTimbre
4	skEnvelope

The `SequenceFormat` record:

Offset	Size	Type	Field name
+0	2	Uint16	length

+2	1	Bool	loopEnabled
+3	1	Uint8	loopIndex

length

The length of the sequence data. Following this record will be this number of `Uint8` bytes that are the sequence's data. *Valid values 0-256*.

loopEnabled

Determines if this sequence has a loop index.

loopIndex

The index of the loop point.

WAVE block format

A WAVE block contains the data for a single waveform. The data is structured in this order:

1. The waveform's id, `Uint8`
2. The waveform's name, `LString`
3. The waveform's data, a 16 byte array of packed 4-bit PCM samples

id and name

Same as `INST` blocks, the `WAVE` block's data begins with the waveform's id and name.

Waveform data

Next is the waveform's data, a 16 byte array of 32 4-bit PCM samples, with the same layout as the Game Boy's CH3 Wave RAM. The first sample in the waveform is the upper nibble of the first byte in the array, whereas the last sample is the lower nibble of the last byte in the array.

The waveform data ends the `WAVE` block.

Terminator

A terminator follows the payload, it is the signature, reversed.

Table 6. TBM Terminator format

+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
`'\0'`	`'Y'`	`'O'`	`'B'`	`'R'`	`'E'`	`'K'`	`'C'`	`'A'`	`'R'`	`'T'`	`'\0'`

EOF

The module file should be at end of file (EOF) after the terminator.

Module Piece Format

Module piece files contain a single part of a module, for easy reuse and sharing. A piece file can contain either a song (*.tbs), an instrument (*.tbi) or a waveform (*.tbw).

Structure

A piece file consists of a header followed by a payload. The payload is a single INST, SONG or WAVE block. A terminator is not used since there is only one block present in the payload.

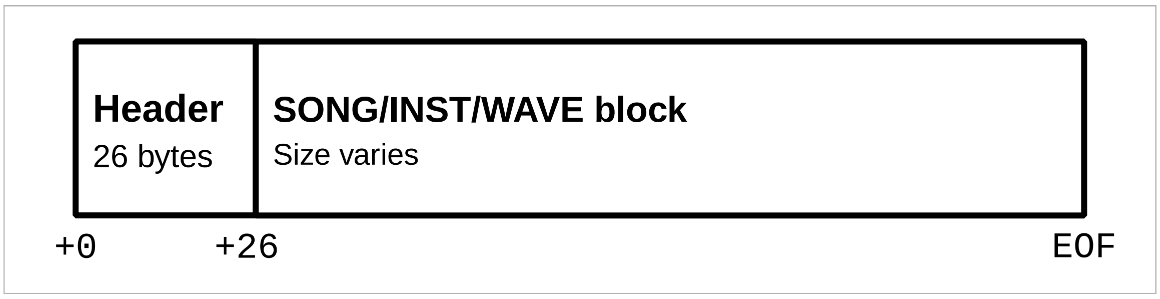

Figure 2. Structure of a module piece file

Header

The Header for a piece file is the same as the module one, but a reduced variant. This reduced header contains only the signature, version and revision fields, or bytes 0-25. The fields are exactly the same as the module format, see the previously defined Header format for more info.

Payload

The payload is a single INST block for *.tbi files, a single SONG block for *.tbs files or a single WAVE block for *.tbw files.

The format of these blocks are the same as the ones used in the module file format except for one key detail: the id is omitted for INST and WAVE blocks.

NOTE INST and WAVE blocks will be 1 byte less than their module counterpart, since the Id is omitted.

Upgrading

When upgrading a major 1 module to major 2, use the following guidelines:

- The header's customFramerate field can be safely casted to a Float32 if set.
- Existing SONG blocks should assume values of `0` for `systemOverride` and `0.0f` for `customFramerateOverride` fields in the `SongFormat` record.
- The initial envelope setting in INST blocks can easily be converted to an envelope sequence via:
    - an empty sequence when `initEnvelope` was `0`
    - a sequence with 1 value when `initEnvelope` was `1`, with that value being `envelope`.