

The Unofficial XM File Format Specification


FastTracker II, ADPCM and Stripped Module Subformats

Vladimir Kameňar

The Unofficial XM File Format Specification

FastTracker II, ADPCM and Stripped Module Subformats

Vladimir Kameñar

 CelerSMS
Bogota, Colombia

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained herein.


CelerSMS S.A.S. is an independent publisher.
ISNI 0000 0005 0265 593X
Ringgold ID 598732
Publons publisher ID 18644

For more information, please contact:
(+57) 3023436167
info@celersms.com

Visit us on the web: www.celersms.com

Cataloging in Publication record:

Kameñar, Vladimir
The Unofficial XM File Format Specification / Vladimir Kameñar – 1st ed.
ISBN 978-958-53602-0-4
004.6

 This work is public domain.

ISBN 978-958-53602-0-4
OCLC 1262695345

Publication date: August 2021

Contents

| | |
|---------------------------------------|----|
| <i>Preface</i> | 5 |
| Introduction..... | 6 |
| General layout of the XM file..... | 8 |
| XM header | 9 |
| Pattern header | 12 |
| Instrument header | 13 |
| Sample header..... | 15 |
| Sample data..... | 15 |
| Pattern format | 17 |
| Annex A: Volumes and envelopes..... | 20 |
| Annex B: Periods and frequencies..... | 21 |

Preface

XM is one of the major module formats. Module files, also known as tracker modules or tracker music, are a family of music files originated in the times of the Amiga computer in the late 80s. They became an important part of the *demoscene* subculture. The modules are still popular among the professional musicians, demosceners and enthusiasts. Many module songs are available for free. For example, the Mod Archive¹ has one of the largest online collections of free tracker music.

Module files are different from the popular audio formats like MP3 or MIDI. The contents of a module file include both the song data (sequences of notes, loops, effects) and the sample data. The sample data describe the instruments used to reproduce the tracks. The module player software uses this data to recreate a virtual set of instruments and make them play the song. The MIDI files don't include the sample data. Therefore, only a standard set of instruments is available and the playback can sound different depending on the synthesizer hardware or software. On the other hand, the prerecorded audio formats like MP3 or OGG, when uncompressed, contain the actual PCM audio data. Therefore, it is possible to convert MIDI to module file and module file to prerecorded audio format like MP3. However, the opposite process is almost impossible.

Module files can be very compact. A full song can fit into less than 1Kb. For example, the song Minimal III by SofT MANiAC in XM format is just 905 bytes without compression.

There are module file players for any operating system, even for mobiles.

This specification explains the bits and the bytes of the XM file format. There are several open source XM player implementations in different languages, from low-level assembly to high-level C++ and Java and even JavaScript. Looking into the source code and reading the specification can help you understand how it works.

There are other popular module file formats, for example: MOD, IT, S3M, V2M.²

Vladimir Kameñar
Bogota, Colombia
August 2021

¹ *The Mod Archive* <https://modarchive.org>

² *Farbrausch demo tools* https://github.com/farbrausch/fr_public

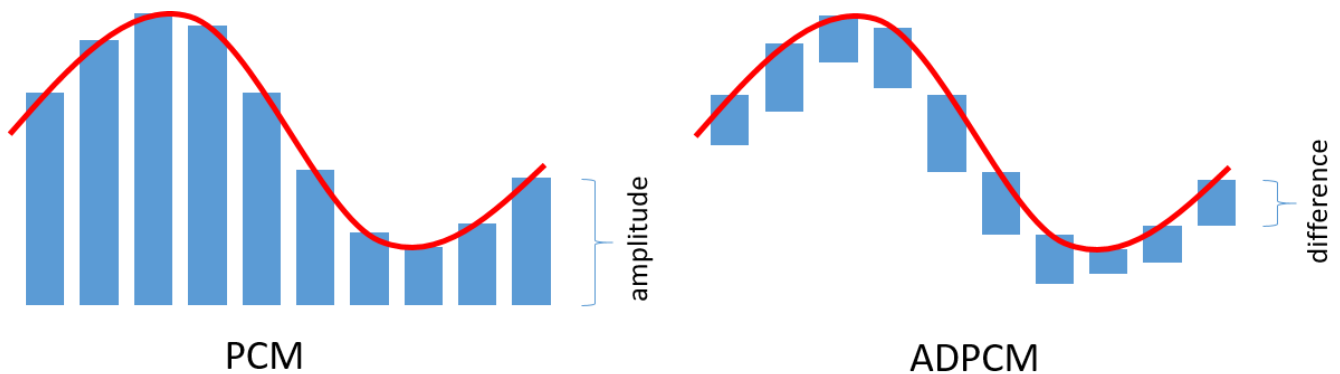
Introduction

This specification describes the structure of regular FastTracker II, ADPCM-compressed and Stripped Module files. It is unofficial because the author doesn't hold the copyright to the XM file format. There are many extensions to the original XM format, for example: ADPCM-compressed, OggVorbis-compressed, Stripped Module and others. Many XM files aren't compliant with the original standard because of accidental or intentional modifications. Therefore, it's hardly possible to describe a global XM standard. The original specification doesn't cover the extensions. That's why the µFMOD developers decided to write and maintain this unofficial specification.

The original FastTracker II file format was introduced in 1994 by Triton, a famous demoscene team.³ It is commonly referred as the "eXtended Module" (hence the file extension). The XM format is an extended version of the original MOD format introducing several new features, for example: multi-sample instruments,⁴ volume and panning envelopes,⁵ sample looping,⁶ portamento frequency tables, new extended effect commands, basic pattern compression, among other improvements.

Then, in the early 2000, ByteRaven and Wodan from TNT/NO-ID corrected and extended the original XM specification.

The ADPCM-compressed XM extension is a non-standard XM subformat introduced in the ModPlug tracker. It has the same XM file format structure. At least one of the samples is compressed in 4-bit ADPCM format. An ADPCM-compressed sample is almost 2 times smaller than its uncompressed equivalent. The drawback is the sound quality. The ADPCM compression acts like a low-pass filter. Therefore, it can cause distortion.



Another popular extension is OXM – OggVorbis-compressed XM.⁷ It preserves the original XM file structure. The instrument samples are compressed in OggVorbis stream format. There are at least 2 known OXM subformats. None of them are covered in this document. The Firelight FMOD library supports OXM files.

³ Mr.H, *The XM module format description for XM files version \$0104*, Triton, 1994

⁴ Sawyer, Ben et. al., *Game Developer's Marketplace*, Coriolis Group Books, ISBN 1576101770, 1998

⁵ Parekh, Ranjan, *Principles of Multimedia*, Tata McGraw-Hill Education, ISBN 9780070588332, 2006

⁶ Alves de Abreu, Valter Miguel, *Analysing trackers and their formats*, University of Porto, S2CID 192364225, 2018

⁷ Sweet, Michael, *MOD File Sequencing*, Addison-Wesley, ISBN 9780321961587, 2014

The Stripped Module file format is another non-standard XM subformat. It was introduced in μ FMOD in 2006. A Stripped Module file is smaller than a regular XM, because its headers are more compact. The Stripped Module format is a superset of the original XM. Therefore, a player supporting stripped modules can also support regular XM files. The audio content is unaffected while converting a regular XM file to stripped format. There is a free open source tool *XMStrip*,⁸ which converts regular XM to Stripped Module format and vice versa. The same tool can recover damaged or otherwise corrupt XM files.

There are more non-standard XM extensions. For example, some trackers use proprietary effect commands to trigger software events. There are other extensions adding Text2Speech (TTS) metadata, watermarks and so on. Unfortunately, very little or no documentation is provided to support these features or at least ignore them safely while loading a non-standard XM file.

This document describes only the original FastTracker II file format, the ADPCM extension and the Stripped Module. A comprehensive description of the effect commands is available in Thunder's MODFIL10.TXT.

⁸ *μ FMOD Guide* <https://ufmod.sourceforge.io/Win32/en.htm>

General layout of the XM file

XM header

header of 1st pattern

data of 1st pattern

header of 2nd pattern

data of 2nd pattern

...

header of last pattern

data of last pattern

1st part of 1st instrument header

If number of samples > 0 , then this data follows:

2nd part of 1st instrument header

1st sampleheader

2nd sampleheader follows (if any)

...

last sampleheader (if any)

If the sample size > 0 (1st sample header), then **sampledata of 1st sample** follows

If the sample size > 0 (2nd sample header), then **sampledata of 2nd sample** follows

...

If the sample size > 0 (last sample header), then **sampledata of last sample** follows

1st part of 2nd instrument header

... same layout as 1st instrument

...

1st part of last instrument header

... same layout as 1st instrument

Additional information

[Volume and Envelope Formulas](#)
[Periods and Frequencies](#)

XM header

| Offset | Length | Type | Ref | Example |
|--------|--------|-------|-----------------------|------------------------|
| 0 | 17 | char | ID text | 'Extended module: ' |
| 17 | 20 | char | Module name | 'Bellissima 99 (mix) ' |
| 37 | 1 | byte | 0x1A | 1A |
| 38 | 20 | char | Tracker name | 'FastTracker v2.00 ' |
| 58 | 2 | word | Version number | 04 01 |
| 60 | 4 | dword | Header size | 14 01 00 00 |
| 64 | 2 | word | Song length | 3E 00 (1..256) |
| 66 | 2 | word | Restart position | 00 00 |
| 68 | 2 | word | Number of channels | 20 00 (0..32/64) |
| 70 | 2 | word | Number of patterns | 37 00 (1..256) |
| 72 | 2 | word | Number of instruments | 12 00 (0..128) |
| 74 | 2 | word | Flags | 01 00 |
| 76 | 2 | word | Default tempo | 05 00 |
| 78 | 2 | word | Default BPM | 98 00 |
| 80 | ? | byte | Pattern order table | 00 01 02 03 ... |

ID text

Should read 'Extended module: ' in a normal XM file. In a Stripped XM this field usually contains just nulls. Some people clear or scramble this magic text in their XM files when embedding into an EXE to prevent others from ripping the track. Don't rely on this string when checking an XM file for validity.

Module name

Should be an ASCII string padded with spaces. Might be zero padded or empty as well (all spaces or all nulls). Some authors store random data here. Don't rely on Module name being a valid ASCII string.

0x1A

The hex value 0x1A in a normal XM file or 0x00 in a Stripped Module. Since most players check this field, the *XMStrip* tool clears it to prevent players not actually supporting the stripped format from incorrectly loading a Stripped XM. Apparently the value 0x1A has a special "escape" meaning. For example, if you print the contents of an XM file using the shell's cat command, it will stop after dumping "Extended module: " and the module's name. None of the following binary contents will be printed. That's how it was supposed to be if everybody respected the standards.

Tracker name

Should read 'FastTracker v2.00 ' or 'FastTracker II ' but some trackers (e.g. DigiTracker) use this field for other purposes (DigiTracker stores the Composer's name here). Should contain nulls in a Stripped XM. If this field doesn't contain valid data, it doesn't mean that the XM file is corrupt.

Version number

Hi-byte major and low-byte minor version numbers in a normal XM or 0x0000 in a Stripped XM. For example, 0x0104 means v1.4. None of the extensions use this field for identifying themselves. So, v1.4 doesn't mean that it's a standard XM file.

Header size

Total size of the following header data until the header of the 1st pattern. Using this value is the only way to locate the header of the 1st pattern. In a normal XM the minimal value is 20 + 256 =

0x00000114. This is so because the pattern order table in a normal XM has a fixed size of 256 bytes. For example, if the pattern order table consists of the following indexes:

0 1 4 1

a normal XM file would pad those values with zeroes:

```
00 01 04 01 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Instead of zeroes, some trackers store negative values. The actual pattern order table length is given in the [Song length] field (see below). So, you can easily identify the padding bytes. That's why these bytes are not stored in a Stripped XM file and the [Header size] could be lower than 0x00000114. The minimal value is 20 + 1 (at least 1 pattern should exist). In both normal and Stripped XM files [Header size] + 60 is the exact offset where the 1st pattern header begins. The only difference is that no padding exists in a Stripped XM. The same example pattern order in a Stripped XM would be:

00 01 04 01

Using only 4 bytes instead of 256. However, the formula is exactly the same in both cases.

Additional data may exist between the pattern order table and the 1st pattern header (normally you would just skip this additional data). Some trackers store the composer's name here, metadata and so on.

There is no practical maximum header size (except the obvious $2^{32}-1$ value), because nothing limits the amount of additional data. That's why you should always use the [Header size] value + 60 to jump directly to the 1st pattern header.

Song length

This is the pattern order table size in bytes, as mentioned earlier. Maximum value is 256. Minimum is 1, because a zero-length song doesn't make sense.

Restart position

Zero-based index to the pattern order table where the track should continue at after the end of the table is reached. Some trackers append an empty (silence) pattern at the end and set restart position to that pattern to prevent looping. When restart position holds an invalid index (greater than [Song length]), a zero value should be used instead.

Number of channels

The number of mixing channels. This has nothing to do with mono/stereo. Defines the number of columns in the pattern packets. In the original FastTracker II specification the maximum value was 32. However, XM files containing more than 32 channels do exist. μ FMOD suggests 64 as a more realistic maximum value.

Number of patterns

The number of patterns stored in the file. There should be at least 1 pattern. The maximum value is 256. Don't confuse this with [Song length]!

Number of instruments

In the range from 0 to 128.

Flags

Only LSB (bit 0) is used. It defines the frequency table:

```
0 = Amiga  
1 = Linear
```

See [below](#) for additional information.

Default tempo & Default BPM

The initial playback speed and BPM (Beats Per Minute).

Pattern order table

Defines the exact order for the XM patterns playback. The size of this table in bytes is the song length. For example:

```
05 08 03 08 ...
```

Pattern #5 will be the first one to play, if it exists in the file. If it doesn't exist, an empty pattern (dummy pattern) should be used instead. After #5 finishes playback, #8 will be triggered (once again, if it exists). Then, #3 will start playing. Then, #8 will be used once again and so on.

Pattern header

| Offset | Length | Type | Ref | Example |
|--------|--------|-------|---------------------------|--------------------|
| ? | 4 | dword | Pattern header length | 09 00 00 00 |
| +4 | 1 | byte | Packing type | 00 |
| +5 | 2 | word | Number of rows in pattern | 20 00 (1..256) |
| +7 | 2 | word | Packed pattern data size | 3B 02 |
| ? | ? | | Packed pattern data | 87 3F 01 1A 80 ... |

Pattern header length

The size of the pattern header in bytes. A typical value is 9, unless some overhead data is stored between the pattern header and the actual pattern data.

Packing type

Almost always 0. Doesn't mean anything.

Number of rows in pattern

In the range from 1 to 256.

Packed pattern data size

If it's set to 0, this pattern is **NOT** stored in the XM; in other words, you need to create an empty pattern if the module needs one. In fact, to be save, you'll always have to create a "standard" empty pattern: allocate $64 * (\text{number of channels})$ bytes and set them to value 0x80. Initialize the header of this pattern with the standard values:

```
pattern header length      = 9
Packing type               = 0
Number of rows in pattern = 64
Packed patterndata size    = 64 * (number of channels)
```

Also note that whenever a pattern number in the pattern sequence table is higher than the actual number of patterns (this is common for converted S3M's), you should play the standard empty pattern. If packed pattern data size is not 0, it might not match the exact size in bytes. This is more like a boolean value. So, it's recommended to parse the following pattern data until you read in a total of $(\text{number of rows}) * (\text{number of channels})$ patterns.

Instrument header

| Offset | Length | Type | Ref | Example |
|--------|--------|-------|-------------------|----------------|
| ? | 4 | dword | Instrument size | 07 01 00 00 |
| +4 | 22 | char | Instrument name | 'chip-lovers!' |
| +26 | 1 | byte | Instrument type | 00 |
| +27 | 2 | word | Number of samples | 01 00 |

If the number of samples is greater than 0, then this will follow:

| | | | | |
|------|----|-------|-----------------------------|--------------------|
| +29 | 4 | dword | Sample header size | 28 00 00 00 |
| +33 | 96 | byte | Sample keymap assignments | 00 00 00 00 00 ... |
| +129 | 48 | word | Points for volume envelope | 00 00 40 00 01 ... |
| +177 | 48 | word | Points for panning envelope | 00 00 00 00 00 ... |

Envelope points: x,y...x,y... in couples (2 words/point => a maximum of 12 points).

| | | | | |
|------|----|------|---|-----------------|
| +225 | 1 | byte | Number of volume points | 09 |
| +226 | 1 | byte | Number of panning points | 00 |
| +227 | 1 | byte | Volume sustain point | 02 |
| +228 | 1 | byte | Volume loop start point | 00 |
| +229 | 1 | byte | Volume loop end point | 00 |
| +230 | 1 | byte | Panning sustain point | 00 |
| +231 | 1 | byte | Panning loop start point | 00 |
| +232 | 1 | byte | Panning loop end point | 00 |
| +233 | 1 | byte | Volume type: bit 0: On; 1: Sustain; 2: Loop | 03 |
| +234 | 1 | byte | Panning type: bit 0: On; 1: Sustain; 2: Loop | 00 |
| +235 | 1 | byte | Vibrato type | 00 |
| +236 | 1 | byte | Vibrato sweep | 00 |
| +237 | 1 | byte | Vibrato depth | 00 |
| +238 | 1 | byte | Vibrato rate | 00 |
| +239 | 2 | word | Volume fadeout | 00 04 |
| +241 | 22 | word | Reserved | 00 00 00 00 ... |

Instrument size

The instrument header size in bytes. The standard minimal value is 29, meaning a totally empty instrument. Stripped minimal value is 4 and any of the following fields where the offset is greater than the instrument size should be set to zero.

Instrument name

A space-padded or zero-padded ASCII string, identifying the current instrument's name, but could be an arbitrary sequence of bytes.

Instrument type

Almost always 0. Doesn't mean anything.

Number of samples in instrument

If it's 0, none of the following values are stored.

Sample header size

The size of the sample header in bytes. You should use the "sample header size" (first field of second part of `instrumentheader`) to compute the total size of the 2 headers stored in the file.

Sample header

| Offset | Length | Type | Ref | Example |
|--------|--------|-------|--------------------|-------------|
| ? | 4 | dword | Sample length | 8C 0B 00 00 |
| +4 | 4 | dword | Sample loop start | 00 00 00 00 |
| +8 | 4 | dword | Sample loop length | 00 00 00 00 |

Note: If the sample loop length is 0, the sample is *NOT* a looping one, even if the "Forward loop" bit is set in the "TYPE" field.

| | | | | | |
|-----|----|------|---|---------|----------------|
| +12 | 1 | byte | Volume | 40 | |
| +13 | 1 | byte | Finetune | 00 | (-128..+127) |
| +14 | 1 | byte | Type: | 00 | |
| | | | Bit 0-1: 0 = No loop | | |
| | | | 1 = Forward loop | | |
| | | | 2 = Bidirectional loop (aka ping-pong) | | |
| | | | Bit 4 : 0 = 8-bit samples | | |
| | | | 1 = 16-bit samples | | |
| +15 | 1 | byte | Panning | 7C | (0..255) |
| +16 | 1 | byte | Relative note number | 1D | (signed value) |
| +17 | 1 | byte | Known values: | 00 | |
| | | | 00 = regular delta packed sample data | | |
| | | | AD = 4-bit ADPCM-compressed sample data | | |
| +18 | 22 | char | Sample name | '-----' | |

Sample data

| Offset | Length | Type | Ref | Example |
|--------|--------|------|-------------|--------------------|
| ? | ? | byte | Sample data | 00 00 DA 2D 08 ... |

Regular samples are stored as either 8- or 16-bit signed delta values. The size of the sample buffer in bytes is given in the [Sample header].[Sample length] field. To convert to real data:

```
old = 0;
for(i = 0; i < len; i++){
    old += sample[i];
    real_sample[i] = old;
}
```

For example, an 8-bit sample:

00 01 FF 02 01 FE ...

In decimal:

0 1 -1 2 1 -2 ...

Let's convert it to real data. The first sample value will be 0. The second one will be the previous (0) plus the delta value (1): $0 + 1 = 1$. Next one: $1 + (-1) = 0$ and so on:

0 1 0 2 3 1 ...

ADPCM-compressed samples are stored in the following format:

| | | | | |
|-----|----|------|------------------------|--------------------|
| ? | 16 | byte | Compression table | 00 01 02 04 08 ... |
| +16 | ? | byte | Compressed sample data | 00 08 00 18 31 ... |

The compression table is a 16-byte array, similar to a color palette in a bitmap file. Every byte in the array represents a delta value. Instead of storing the delta values directly in the sample data buffer, an ADPCM sample consists of 4-bit indexes into the compression table, where the actual deltas are stored. The contents of the compression table are almost always the same:

00 01 02 04 08 10 20 40 FF FE FC F8 F0 E0 D0 C0

or in decimal:

0 1 2 4 8 16 32 64 -1 -2 -4 -8 -16 -32 -48 -64

However, the actual values may be different (at least in theory). The compressed sample data is located at offset 16 just after the compression table. Its size should be calculated as follows:

```
len = ([Sample header].[Sample length] + 1) >> 1
```

To convert to real data:

```
old = 0;
for(i = 0, j = 0; i < len; i++, j += 2){
    index = sample[16 + i];
    old += sample[index & 0xF];
    real_sample[j] = old;
    old += sample[index >> 4];
    real_sample[j + 1] = old;
}
```

An ADPCM example:

00 01 02 04 08 10 20 40 FF FE FC F8 F0 E0 D0 C0 00 08 00 18 31 02 81 ...

Bytes in **red** are the compression table. The 4-bit indexes begin at offset 16. Every sample byte holds 2 indexes. For example, 0x31 holds index 1 in the low-order nibble and index 3 in the high-order nibble. Let's extract all indexes starting at byte offset 16 (low-order nibble goes first):

0 0 8 0 0 0 8 1 1 3 2 0 1 8 ...

The corresponding delta values (from the preceding compression table) are:

00 00 FF 00 00 00 FF 01 01 04 02 00 01 FF ...

In decimal:

0 0 -1 0 0 0 -1 1 1 4 2 0 1 -1 ...

Now, let's perform the delta conversion, as we did in the previous example:

0 0 -1 -1 -1 -1 -2 -1 0 4 6 6 7 6 ...

Pattern format

XM patterns are stored as following:

A pattern is a sequence of lines.

A line is a sequence of notes.

A note is stored as described below:

| Offset | Length | Type | Ref | Example | |
|--------|--------|------|--------------------|---------|--------------|
| ? | 1 | byte | Note | 01 | (0..96, 97) |
| +1 | 1 | byte | Instrument | 01 | (0..128) |
| +2 | 1 | byte | Volume column byte | 00 | (0..64, 255) |
| +3 | 1 | byte | Effect type | 05 | (0..26) |
| +4 | 1 | byte | Effect parameter | 1F | (0..255) |

Note

Possible values (when MSB = 0):

| | | |
|-----|---------|---------------------------|
| 0 | No note | |
| 1 | C-1 | Do |
| 2 | C#1 | Di (Sharp) |
| 3 | D-1 | Re |
| 4 | D#1 | Ri (Sharp) |
| 5 | E-1 | Mi |
| 6 | F-1 | Fa |
| 7 | F#1 | Fi (Sharp) |
| 8 | G-1 | Sol |
| 9 | G#1 | Si (Sharp) |
| 10 | A-1 | La |
| 11 | A#1 | Li (Sharp) |
| 12 | B-1 | Ti |
| 13 | C-2 | Do 2 nd octave |
| ... | | |
| 96 | B-8 | Ti 8 th octave |
| 97 | Key off | (aka 'Note off') |

A simple packing scheme is also adopted, so that the patterns don't become too large. The MSB in the note value is used for the compression. If the bit is set, then the other bits are interpreted as follows:

```
bit 0 set: Note follows
  1 set: Instrument follows
  2 set: Volume column byte follows
  3 set: Effect type follows
  4 set: Effect parameter follows
```

For example, the following 2 packets are equivalent:

```
01 01 00 00 00
```

and

```
83 01 01
```

0x83 in binary is 10000011. Since the MSB is set, the low-order nibble is interpreted as follows: note follows, instrument follows, volume column not present, effect type not present, effect parameter not present. So, the volume column, the effect type and the effect parameter should be cleared to 0.

Volume column byte

All effects in the volume column should work as the standard effects. The volume column is interpreted before the standard effects, so some standard effects may override volume column effects. The high-order nibble specifies one of the following commands: V, D, C, B, A, U, H, P, L, R, G. The low-order nibble specifies the command argument. The following values are defined:

| Value | Mnemonic | Command | Argument range |
|--------|----------|-------------------|--------------------|
| 00..0F | | <do nothing> | |
| 10..1F | V | Set volume | (0..15) |
| 20..2F | V | Set volume | (16..31) |
| 30..3F | V | Set volume | (32..47) |
| 40..4F | V | Set volume | (48..63) |
| 50 | V | Set volume | (64) |
| 51..5F | | <undefined> | |
| 60..6F | D | Volume slide down | (0..15) |
| 70..7F | C | Volume slide up | (0..15) |
| 80..8F | B | Fine volume down | (0..15) |
| 90..9F | A | Fine volume up | (0..15) |
| A0..AF | U | Vibrato speed | (0..15) |
| B0..BF | H | Vibrato deph | (0..15) |
| C0..CF | P | Set panning | (2, 6, 10, 14..62) |
| D0..DF | L | Pan slide left | (0..15) |
| E0..EF | R | Pan slide right | (0..15) |
| F0..FF | G | Tone portamento | (0..15) |

Effects

- 0 Arpeggio
- 1 (*) Porta up
- 2 (*) Porta down
- 3 (*) Tone porta
- 4 (*) Vibrato
- 5 (*) Tone porta+Volume slide
- 6 (*) Vibrato+Volume slide
- 7 (*) Tremolo
- 8 Set panning
- 9 Sample offset
- A (*) Volume slide
- B Position jump
- C Set volume
- D Pattern break
- E1 (*) Fine porta up
- E2 (*) Fine porta down
- E3 Set gliss control
- E4 Set vibrato control
- E5 Set finetune
- E6 Set loop begin/loop
- E7 Set tremolo control
- E9 Retrig note
- EA (*) Fine volume slide up
- EB (*) Fine volume slide down

| | |
|--------|------------------------------|
| EC | Note cut |
| ED | Note delay |
| EE | Pattern delay |
| F | Set tempo/BPM |
| G | (010h) Set global volume |
| H (*) | (011h) Global volume slide |
| I | (012h) Unused |
| J | (013h) Unused |
| K | (014h) Unused |
| L | (015h) Set envelope position |
| M | (016h) Unused |
| N | (017h) Unused |
| O | (018h) Unused |
| P (*) | (019h) Panning slide |
| Q | (01ah) Unused |
| R (*) | (01bh) Multi retrigger note |
| S | (01ch) Unused |
| T | (01dh) Tremor |
| U | (01eh) Unused |
| V | (01fh) Unused |
| W | (020h) Unused |
| X1 (*) | (021h) Extra fine porta up |
| X2 (*) | (021h) Extra fine porta down |

(*) = If the command byte is zero, the last nonzero byte for the command should be used.

Annex A: Volumes and envelopes

The volume formula:

$$FinalVol = \frac{FadeOutVol}{65536} \cdot \frac{EnvelopeVd}{64} \cdot \frac{GlobalVol}{64} \cdot \frac{Vol}{64} \cdot Scale$$

The panning formula:

$$FinalPan = Pan + (EnvelopePan - 32) \cdot \frac{(128 - |Pan - 128|)}{32}$$

If no envelope is active, use the value 32 instead of "EnvelopePan".

The envelopes are processed once per frame, instead of every frame where no new notes are read. This is also true for the instrument vibrato and the fadeout.

```
max x value: 0x144 = 324  
max y value: 0x40  = 64
```

Annex B: Periods and frequencies

```
PatternNote = 1..96      (1 = C-1, 96 = B-8)
FineTune    = -128..+127 (-128 = -1 halftone, +127 = +127/128 halftones)
RelativeTone = -96..95   (0 = C-4)
RealNote    = PatternNote + RelativeTone = 0..118 (0 = C-0, 118 = A#9)
```

Linear frequency table

$$\textit{Period} = 10 \cdot 12 \cdot 16 \cdot 4 - \textit{Note} \cdot 16 \cdot 4 - \textit{FineTune} / 2$$

$$\textit{Frequency} = 8363 \cdot 2^{\frac{6 \cdot 12 \cdot 16 \cdot 4 - \textit{Period}}{12 \cdot 16 \cdot 4}}$$

Don't attempt to implement the 2^N operation as a bit shift, because N is actually a real number! Most players use floating point arithmetic to implement the latter formula. If you can't or don't want to use floating point operations, you can use a 768 doubleword array, like ModPlug does. No other known way exists to compute the linear Frequency values.

When using linear Freq Tables, the distance between two octaves is 255 portamento units. e.g. the effect "1FF" at speed 1 will slide the pitch one octave up.

Amiga frequencies table

```
Pos      = (Note % 12) * 8 + FineTune / 16
Frac     = FineTune / 16 - Int(FineTune / 16)
Period   = (PeriodTab[Pos] * (1 - Frac) + PeriodTab[Pos+1] * Frac) * 16 / 2Note/12
```

The period is interpolated for finer finetune values.

```
Frequency = 8363 * 1712 / Period
```

```
WORD PeriodTab[] = {
    907, 900, 894, 887, 881, 875, 868, 862, 856, 850, 844, 838, 832, 826, 820, 814,
    808, 802, 796, 791, 785, 779, 774, 768, 762, 757, 752, 746, 741, 736, 730, 725,
    720, 715, 709, 704, 699, 694, 689, 684, 678, 675, 670, 665, 660, 655, 651, 646,
    640, 636, 632, 628, 623, 619, 614, 610, 604, 601, 597, 592, 588, 584, 580, 575,
    570, 567, 563, 559, 555, 551, 547, 543, 538, 535, 532, 528, 524, 520, 516, 513,
    508, 505, 502, 498, 494, 491, 487, 484, 480, 477, 474, 470, 467, 463, 460, 457
};
```