标准 MIDI 文件 1.0

发表的:
The MIDI Manufacturers Association
Los Angeles, CA

RP001

1996 年 2 月修订

MMA
POB 3173
La Habra CA 90632-3173

标准 MIDI Files 1.0

介绍

该文档概述了 MIDI 文件的规范。MIDI 文件的目的是提供一种在相同或不同计算机上的不同程序之间交换带时间戳的 MIDI 数据的方法。主要的设计目标之一是紧凑的表示，这使得它非常适合基于磁盘的文件格式，但这可能使它不适合存储在内存中以供音序器程序快速访问。(当文件被读取或写入时，它可以很容易地转换为快速访问的格式。)它并不打算取代任何程序的正常文件格式，尽管如果需要的话，它可以用于此目的。

MIDI 文件包含一个或多个 MIDI 流，其中包含每个事件的时间信息。歌曲、序列和音轨结构、节奏和时间签名信息，都是支持的。曲目名称和其他描述性信息可能与 MIDI 数据一起存储。这种格式支持多音轨和多序列，因此如果一个支持多音轨的程序的用户打算将一个文件移动到另一个，这种格式可以允许这种情况发生。

该规范定义了文件中使用的 8 位二进制数据流。数据可以被存储在二进制文件中，被逐点化，被 7 位化以实现高效的 MIDI 传输，被转换成十六进制 ASCII，或者被符号化地翻译成可打印的文本文件。本规范解决了 8 位流中的内容。它没有解决 MIDI 文件如何通过 MIDI 传输的问题。一般的感觉是，将为一般文件开发一个 MIDI 传输协议，MIDI 文件将使用这个方案。

序列，轨道，块:文件块结构

约定

在本文档中，第 0 位表示字节的最低有效位，第 7 位表示最高有效位。

MIDI 文件中的一些数字以一种称为变长量的形式表示。这些数字以每字节 7 位的形式表示，最重要的位优先。除最后一个字节外的所有字节都设置了第 7 位，最后一个字节清除了第 7 位。如果数字在 0 到 127 之间，那么它就被精确地表示为一个字节。

下面是一些用变长量表示的数字的例子:

数字(十六进制)	表示(十六进制)
00000000	00
00000040	40
0000007 7	f
00000080	81 00
00002000	c0 00
00003fff	ff 7f
00004000	81 8000
00100000	c0 80000
001fffff	ff ff 7f
00200000	81 80 80 00
8000000	c0 80 80000
0ffffffff	ff ff 7f

允许的最大数字是 0FFFFFFF，因此在编写可变长度数字的例程中，可变长度表示必须适合 32 位。理论上，更大的数字是可能的，但是在每分钟 500 拍的快节奏下，2 × 108 96 拍是 4 天，对于任何增量时间来说都足够长了!

文件

对于任何文件系统，一个 MIDI 文件只是一系列 8 位字节。在 Macintosh 上，这个字节流存储在文件的数据分支中(文件类型为“Midi”)，或者存储在剪贴板中(数据类型为“Midi”)。大多数其他计算机将 8 位字节流存储在文件中——这些计算机的命名或存储约定将根据需要定义。
块

MIDI 文件由块组成。每个块有 4 个字符的类型和 32 位的长度，即块中的字节数。这种结构允许设计未来的块类型，如果在引入块类型之前编写的程序遇到这些类型，可能很容易被忽略。你的程序应该预料到外来的块，并把它们当作不存在一样对待。

每个块以一个 4 个字符的 ASCII 类型开头。它后面是一个 32 位长度，最高有效字节优先(6 的长度存储为 00 00 00 06)。这个长度指的是后面的数据字节数:type 和 length 的 8 个字节不包括在内。因此，长度为 6 的块实际上会在磁盘文件中占用 14 个字节。

这个块结构类似于 ea 的 IFF 格式，这里描述的块可以很容易地放在一个 IFF 文件中。MIDI File 本身并不是一个 IFF 文件:它不包含嵌套的块，并且块也没有被限制为偶数字节长。将其转换为 IFF 文件非常简单，只需填充奇数长度的块，并将整个内容粘贴到 FORM 块中即可。

MIDI 文件包含两种类型的块:header 块和 track 块。头块提供了与整个 MIDI 文件相关的最少量的信息。音轨块包含 MIDI 数据的顺序流，它可能包含多达 16 个 MIDI 通道的信息。多个音轨、多个 MIDI 输出、模式、序列和歌曲的概念都可以使用几个音轨块来实现。

MIDI 文件总是以标题块开始，然后是一个或多个音轨块。

MThd <头数据的长度>
<标题>数据
MTrk <轨道数据>的长度
< >跟踪数据
MTrk <轨道数据>长度<轨道数据>…

块描述

头块

文件开头的头块指定了文件中数据的一些基本信息。下面是完整块的语法:

<Header Chunk> = < Chunk type> <length> <format> <ntrks> <division>

如上所述，<chunktype>是四个 ASCII 字符'MThd';<length>是数字 6 的 32 位表示(高字节优先)。

数据段包含三个 16 位字，优先存储最高有效字节。

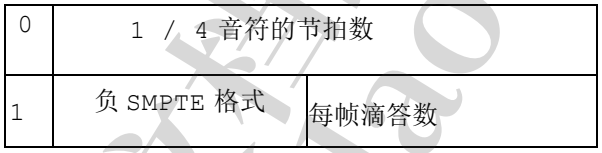
第一个字<format>，指定了文件的整体组织结构。<format>只指定了三个值:

0　　文件包含单个多声道音轨
1　　文件包含一个序列的一个或多个同步轨道(或 MIDI 输出)
2　　文件包含一个或多个顺序独立的单轨模式

下面提供了关于这些格式的更多信息。

下一个单词<ntrks>，是文件中音轨块的数量。对于格式为 0 的文件，它将始终为 1。

第三个字<division>，指定 delta-times 的含义。它有两种格式，一种用于格律时间，另一种用于基于时间代码的时间:



如果<division>的第 15 位是 0，则第 14 位到 0 位表示组成一个四分音符的增量时间“滴答”的个数。例如，如果<division>是 96，那么文件中两个事件之间的 8 个音符的时间间隔将是 48。

如果<division>的第 15 位是 1，则文件中的增量时间对应于秒的细分，以与 SMPTE 和 MIDI 时间代码一致的方式。第 14 位至第 8 位包含-24、-25、-29 或-30 四个值中的一个，对应于四种标准的 SMPTE 和 MIDI 时间码格式(-29 对应于 30 个降帧)，表示每秒帧数。这些负数以二的补码形式存储。第二个字节(存储为正数)是一帧内的分辨率:典型的值可能是 4 (MIDI 时间码分辨率)、8、10、80(位分辨率)或 100。这个系统允许精确地规范基于时间码的音轨，但也允许基于毫秒的音轨，通过指定 25 帧/秒和每帧 40 个单位的分辨率。如果文件中的事件以三十帧时间码的位分辨率存储，则分割字将是 E250 十六进制。

格式为 0、1 和 2

Format 0 文件有一个头块，后面跟着一个磁道块。它是最可互换的数据表示形式。对于需要让合成器发出声音的程序中的一个简单的单轨播放器来说，它非常有用，但它主要与混音器或音效盒等其他东西有关。为了配合这些简单的程序，能够产生这样的格式是非常可取的，即使你的程序是基于音轨的。另一方面，也许有人会编写一个从格式 1 到格式 0 的格式转换，它可能在某些设置中非常容易使用，从而省去了将其放入程序的麻烦。

格式 1 或格式 2 的文件有一个头块，后面跟着一个或多个磁道块。支持多个同步音轨的程序应该能够保存和读取 format 1 的数据，这是一种垂直的一维形式，即作为音轨的集合。支持几种独立模式的程序应该能够以格式 2(一种水平一维形式)保存和读取数据。提供这些最小的功能将确保最大的互换性。

在带有计算机和使用歌曲指针和定时时钟的 SMPTE 同步器的 MIDI 系统中，速度图(描述整个音轨的速度，也可能包括时间签名信息，以便可以导出条号)通常在计算机上创建。要与同步器一起使用它们，就必须将它们从计算机中传输出来。为了方便同步器从 MIDI File 中提取这些数据，节奏信息应该始终存储在第一个 MTrk 块中。对于 format 0 文件，节奏会分散在音轨中，节奏图阅读器应该忽略中间的事件;对于格式 1 的文件，节奏图必须存储为第一个音轨。对于一个节奏图阅读器来说，礼貌的做法是让你的用户能够制作一个只有节奏的格式 0 文件，除非你可以使用格式 1。

所有 MIDI 文件都应该指定节奏和时间签名。如果没有，则假定时间签名为 4/4，节奏为每分钟 120 拍。在格式 0 中，这些元事件至少应该出现在单个多声道音轨的开头。在格式 1 中，这些元事件应该包含在第一个音轨中。在格式 2 中，每个时间上独立的模式应该至少包含初始的时间签名和节奏信息。

我们可能会决定定义其他格式 id 来支持其他结构。遇到未知格式 ID 的程序仍然可以读取它从文件中找到的其他 MTrk 块，作为格式 1 或格式 2，如果它的用户可以理解它们并在适当的情况下将它们安排到其他结构中。此外，将来可能会向 MThd 块中添加更多参数:读取并遵守长度是很重要的，即使它长于 6。

跟踪块

曲目块(类型为 MTrk)是存储实际歌曲数据的地方。每个音轨块只是一个 MIDI 事件(和非 MIDI 事件)的流，前面有 delta-time 值。对于 MIDI 文件的所有三种格式(0、1 和 2:参见上面的“Header Chunk”)，Track Chunks 的格式(如下所述)完全相同。

下面是 MTrk 块的语法(+表示“一个或多个”:至少有一个 MTrk 事件必须存在):

```
<Track Chunk> = < Chunk type> <length> <MTrk event>+
```

MTrk 事件的语法非常简单:

```
<MTrk event> = <delta-time> <event>
```

<delta-time>被存储为一个可变长度的数量。它表示下一个事件发生前的时间量。如果赛道上的第一个事件发生在赛道的最开始，或者两个事件同时发生，则使用零的 delta-time。增量时间总是存在的。(不存储 delta-times 为 0 的任何其他值至少需要两个字节，并且大多数 delta-times 不为零。)Delta-time 的单位是标头块中指定的刻度。

```
<event> = <MIDI event> | <sysex event> | <meta-event>
```

<MIDI event> =任意 MIDI 通道消息。使用运行状态:如果前面的事件是具有相同状态的 MIDI 通道消息，则可以省略 MIDI 通道消息的状态字节。每个 MTrk 块中的第一个事件必须指定状态。Delta-time 本身不被认为是一个事件:它是 MTrk 事件语法的一个组成部分。请注意，运行状态发生在增量时间上。

<sysex event>用于指定 MIDI 系统独占消息，可以作为一个单元或包，也可以作为“转义”来指定要传输的任意字节。一个正常的完整的系统独占消息是这样存储在 MIDI File 中的:

```
F0 <length> <bytes 要在 F0>之后传输
```

该长度以变长数量的形式存储。它指定其后的字节数，不包括 F0 或长度本身。例如，传输的消息 F0 43 1200 07 F7 将被存储在 MIDI 文件中为 F0 05 43 1200 07 F7。它需要在末尾包含 F7，以便 MIDI 文件的读取器知道它已经读取了整个消息。

提供了另一种形式的 sysex 事件，它并不意味着应该传输 F0。这可以用作“escape”，以提供传输不合法的东西，包括系统实时消息，歌曲指针或选择，MIDI 时间代码等。这里使用的是 F7 代码:

```
F7 <length> <要传输的所有字节>
```

不幸的是，一些合成器制造商指定他们的系统独占消息作为小数据包传输。每个包只是整个语法系统独占消息的一部分，但是它们被传输的时间很重要。这方面的例子是 CZ 补丁转储中发送的字节，或者 FB-01 的“系统独占模式”，在这种模式下可以传输微音调数据。F0 和 F7 sysex 事件可以一起使用，将语法完整的系统独占消息分解成定时数据包。

F0 sysex 事件用于一个系列中的第一个数据包-它是一个应该传输 F0 的消息。F7 sysex 事件用于不以 F0 开头的其余数据包。(当然，F7 不被认为是系统独占消息的一部分)。

句法上的系统独占消息必须始终以 F7 结束，即使现实生活中的设备没有发送 F7，这样你就知道什么时候你已经到达了整个 sysex 消息的末尾，而不用提前查看 MIDI 文件中的下一个事件。如果它存储在一个完整的 F0 sysex 事件中，那么最后一个字节必须是 F7。如果它被分解成数据包，那么最后一个数据包的最后一个字节必须是 F7。在多包系统独占消息的包之间也不能有任何可传输的 MIDI 事件。这一原则将在下面的段落中进行说明。
这里有一个多包系统独占消息的例子:假设要发送的字节 F0 43 1200，后面是 200-tick 的延迟，后面是字节 43 1200 43 1200，后面是 100-tick 的延迟，后面是字节 43 1200 F7，这将在 MIDI 文件中:

```
 F0 03 43 1200
81 48                    200 嘀嗒 delta 时间
 F7 06 43 1200 43 1200
64                       100-tick delta-time
F7 04 43 12 00 F7
```

当读取 MIDI 文件时，如果遇到 F7 sysex 事件，而前面没有 F0 sysex 事件来启动多包系统独占消息序列，则应假定 F7 事件被用作“转义”。在这种情况下，它没有必要以 F7 结束，除非希望传输 F7。

<meta-event>指定对该格式或序列器有用的非 midi 信息，其语法如下:

```
FF <type> <length> <bytes>
```

所有元事件都以 FF 开头，然后有一个事件类型字节(总是小于 128)，然后将数据的长度存储为变长数量，然后是数据本身。如果没有数据，则长度为 0。与块一样，未来的元事件可能被设计为现有程序可能不知道的，因此程序必须适当忽略它们不识别的元事件，实际上，应该期望看到它们。程序绝不能忽略它们所识别的元事件的长度，如果它比它们预期的要大，它们也不应该感到惊讶。如果

是这样，它们必须忽略超出它们所知道的一切。然而，他们绝对不能在元事件的结尾添加任何自己的东西。

Sysex 事件和元事件会取消任何有效的运行状态。运行状态不适用于这些消息，也不能用于这些消息。

Meta-Events

这里定义了一些元事件。并不要求每个程序都支持每个元事件。

在每个元事件的语法描述中，使用了一组约定来描述事件的参数。每个事件开始的 FF，每个事件的类型，以及没有可变数据量的事件的长度直接以十六进制给出。像 dd 或 se 这样的表示法，由两个小写字母组成，容易记忆地表示一个 8 位的值。四个相同的小写字母，如 www，表示一个 16 位的值，首先存储最高有效字节。六个相同的小写字母，如 tttttt，指的是一个 24 位的值，首先存储的是最大有效字节。表法 len 指的是元事件语法的长度部分，也就是一个数字，以变长数量的形式存储，它指定在元事件中有多少个数据字节跟随它。符号 text 和 data 指的是长度指定了多少字节(可能是文本)的数据。

一般来说，轨迹中同时发生的元事件可以以任何顺序发生。如果使用版权事件，应该尽早将其放在文件中，这样才容易被注意到。Sequence Number 和 Sequence/Track Name 事件，如果存在，必须在时间 0 出现。曲目结束事件必须作为曲目中的最后一个事件出现。

最初定义的元事件包括:

FF 00 02 ssss　　Sequence Number

这个可选事件必须发生在音轨的开始，在任何非零增量时间之前，在任何可传输的 MIDI 事件之前，指定序列的编号。在格式 2 的 MIDI 文件中，它用于识别每个“模式”，以便使用 Cue 消息引用模式的“歌曲”序列。如果省略 ID 号，则序列在文件中的顺序位置作为默认值使用。在只包含一个序列的格式为 0 或 1 的 MIDI 文件中，这个数字应该包含在第一个(或唯一)音轨中。如果需要传输多个多音轨序列，则必须将其作为一组格式为 1 的文件来完成，每个文件都有不同的序列号。

FF 01 len text　　text Event

描述任何事物的任何数量的文本。在曲目的开头放置一个文本事件是一个好主意，带有曲目的名称，对其预期编排的描述，以及用户想要放在那里的任何其他信息。文本事件也可能出现在曲目的其他时间，用作歌词，或提示点的描述。此事件中的文本应该是可打印的 ASCII 字符，以便最大限度地交换。然而，使用高阶位的其他字符代码可用于在支持扩展字符集的同一台计算机上的不同程序之间交换文件。不支持非 ascii 字符的计算机上的程序应该忽略这些字符。

元事件类型 01 到 0F 是为各种类型的文本事件保留的，每种类型都符合文本事件的规范(上文)，但用于不同的目的:

FF 02 len text　　版权声明

包含可打印的 ASCII 文本形式的版权声明。该声明应包含字符(C)、版权年份和版权所有者。如果几首乐曲在同一个 MIDI 文件中，那么所有的版权声明应该放在一起，这样它就会出现在文件的开头。此事件应该是第一个音轨块中的第一个事件，时间为 0。

FF 03 len text　　Sequence/Track Name

如果在一个格式为 0 的音轨，或者是格式为 1 的文件中的第一个音轨，序列的名称。否则，为音轨的名称。

FF 04 len text　　Instrument Name

对该音轨中使用的仪器类型的描述。可以与 MIDI Prefix 元事件一起使用，以指定描述适用于哪个 MIDI 通道，或者通道可以在事件本身中指定为文本。

FF 05 len text　　Lyric

要唱的歌词。一般来说，每个音节都是一个独立的歌词事件，从事件发生的时间开始。

FF 06 len text　　Marker

通常为格式 0 的音轨，或格式 1 文件中的第一音轨。序列中该点的名称，如排演字母或分段名称(“First Verse”等)。

FF 07 len text　　Cue Point

对电影或视频屏幕或舞台上发生的事情的描述(“汽车撞进了房子”、“窗帘打开了”、“她扇了他的脸”等)。

FF 20 01 cc　　MIDI 通道前缀

此事件中包含的 MIDI 通道(0-15)可用于将 MIDI 通道与随后的所有事件关联，包括 System Exclusive 和 meta-events。这个通道一直“有效”到下一个正常的 MIDI 事件(其中包含一个通道)或下一个 MIDI channel Prefix 元事件。如果 MIDI 通道指的是“音轨”，这个消息可能有助于将几个音轨塞进一个 format 0 文件中，保持它们的非 MIDI 数据与一个音轨相关联。这种能力也存在于雅马哈的 ESEQ 文件格式中。

FF 2F 00　　End of Track

此事件不可选。包含它是为了可以为轨道指定一个精确的终点，这样它就有一个精确的长度，这对于循环或连接的轨道是必要的。

FF 51 03 tttttt　　设定节奏，每 MIDI 四分音符以微秒为单位

此事件表示节奏变化。“每四分音符微秒数”的另一种表达方式是“每 MIDI 时钟 24 微秒”。将节奏表示为每拍时间而不是每拍时间，可以与基于时间的同步协议(如 SMPTE 时间代码或 MIDI 时间代码)进行绝对精确的长期同步。这种速度分辨率提供的精确度允许每分钟 120 拍的四分钟作品在作品结束时精确到 500 usec 以内。理想情况下，这些事件应该只发生在 MIDI 时钟所在的地方——这种惯例旨在保证，或至少增加与其他同步设备兼容的可能性，以便以这种格式存储的时间签名/节奏图可以很容易地转移到另一个设备。

FF 54 05 hr mn se for FF　　SMPTE Offset

此事件(如果存在)指定轨道块应该开始的 SMPTE 时间。它应该出现在音轨的开始，也就是说，在任何非零增量时间之前，在任何可传输的 MIDI 事件之前。小时必须用 SMPTE 格式进行编码，就像在 MIDI 时间码中一样。在格式 1 文件中，SMPTE Offset 必须与节拍图一起存储，并且在其他任何音轨中都没有意义。ff 字段包含分数帧，每帧的 100 分之一，即使在基于 smpte 的轨道中，它为 delta-time 指定了不同的帧细分。

FF 58 04 nn dd cc bb　　时间签名

时间签名用 4 个数字表示。Nn 和 dd 表示时间特征的分子和分母，就像它被标记的那样。分母是 2 的负次方:2 代表一个四分音符，3 代表一个八分音符，等等。cc 参数表示节拍器滴答声中 MIDI 时钟的个数。bb 参数表示 MIDI 认为的一个四分音符(24 MIDI Clocks)中已记 32nd 音符的个数。之所以增加这个参数，是因为已经有多个程序允许用户指定 MIDI 认为的四分之一音符(24 个时钟)要被标记为其他东西，或者与其他东西相关。

因此，6/8 时间的完整事件，即节拍器每三个八分音符点击一次，但每个四分音符有 24 个时钟，72 个小节，将是(十六进制):

```
Ff 58 04 06 03 24 08
```

也就是说，6/8 的时间(8 是 2 的 3 次方，所以这是 06 03)，每点四分之一 36 个 MIDI 时钟(24 十六进制!)，每个 MIDI 四分之一音符有 8 个 32 音符。

FF 59 02 sf mi　　Key Signature

sf = - 7:7 降 sf= -1:1 降 sf = 0: C 调 sf= 1:1 升 sf = 7:7 升 mi= 0: 大调

Mi = 1:小调

FF 7F len data　　Sequencer-Specific Meta-Event

对特定测序器的特殊要求可能会使用这种事件类型:数据的第一个字节或几个字节是制造商 ID(这些是一个字节，或者，如果第一个字节是 00，则是三个字节)。与 MIDI System Exclusive 一样，使用此元事件定义某些东西的制造商应该发布它，以便其他人可能知道如何使用它。毕竟，这是一种交换格式。这种类型的事件可以被一个 sequencer 使用，它选择使用这个作为它唯一的文件格式;在使用此格式时，具有既定功能特定格式的音序器可能应该坚持使用标准功能。

程序片段和示例 MIDI 文件

下面是一些在 MIDI 文件中读写变长数字的例程。这些例程是用 C 语言编写的，使用 getc 和 putc，它们从/文件中读取和写入单个 8 位字符 infile 和 outfile。

```
WriteVarLen (value)寄存器长
值;
{
 注册长缓冲区;

 Buffer = value & 0x7f;
 While ((value >>= 7) > 0)
 {
```

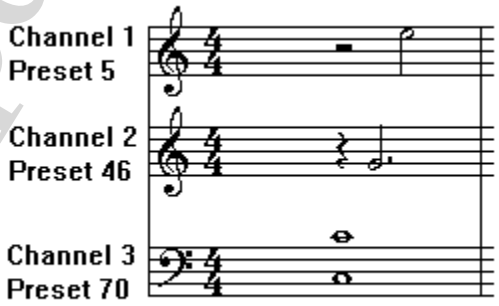
```
  缓冲区<<= 8;缓冲区|=
0x80;Buffer += (value &
0x7f);
 }

 而(真)
 {
  putc(缓冲区,输出文件);If
(buffer & 0x80)
buffer >>= 8;其他破坏;
 }
}

doubleword ReadVarLen ()
{
  注册双字值;寄存器字节 c;

 If ((value = getc(infile)) & 0x80)
 {
  值&= 0x7f;
  做
  {
   Value = (Value << 7) + ((c = getc(infile)) & 0x7f);
  } while (c & 0x80);
 }
 返回(价值);}
```

作为一个例子，下面摘录的 MIDI 文件如下所示。首先，显示的是一个格式为 0 的文件，所有信息混杂在一起;然后，显示一个格式 1 的文件，所有数据被分成四个音轨:一个用于速度和时间签名，三个用于音符。每个四分音符使用 96 个“节拍”的分辨率。4/4 的拍子和 120 的节奏，虽然是隐含的，但都是明确说明的。



这个例子所代表的 MIDI 流的内容分解如下:

Delta Time (dec)	Event Code (hex)	Other Bytes (dec)	Comment
0	FF 58	04 04 02 24 08	4 字节:4/4 时间，24 MIDI 时钟/点击，8 个 32 音符/24 个 MIDI 时钟
0	FF 51	03 500000	3 字节:每个四分音符 500,000μsec
0	C0	5	Ch. 1, Program Change 5
0	C1	46	第 2 章，方案变更 46
0	C2	70	第三章，方案变更 70
0	92	48 96	第 3 章第 48 号注释，强音
第 3 章		60	号音符，强音
96	91	67 64	第 2 章第 67 号音符，中音强音
96	90	76 32	第 1 章 76 号音符，钢琴
192	82	48 64	第 3 章音符关闭#48，标准
0	82	60 64	第 3 章音符关闭#60，标准
0	81	67 64	Ch. 2 Note Off #67，标准
0	80	76 64	Ch. 1 Note Off #76，标准
0	FF 2F	00	Track End

整个格式 0 MIDI 文件的十六进制内容如下。

首先，头块:

4d 54 68 64	MThd
00 00 00 06	块长度
00 00	格式 0
00 01	一个跟踪
00 60	96 /四分音符

然后是音轨块。它的标题，后面跟着事件(注意，跑步状态是用在地方的):

4D 54 72 6B	MTrk
00 00 00 3B	块长度(59)

增量时间	事件	注释
00	FF 58 04 04 02 18 08	时间签名
00	FF 51 03 07 A1 20	拍子
00 c00	05	
00	c1 2e	
00	c2 46	
00	92 30 60	
00	3C 60	运行状态
60	91 43 40	
60	90 4c 20	
81 40	82 30 40	两字节增量时间

00	3C 40	运行状态
00	81 43 40	
00	80 4c 40	
00	FF 2F 00	赛道终点

文件的格式 1 表示略有不同。

首先，它的头块:

4d 54 68 64	MThd
00 00 00 06	块长度
00 01	格式 1
00 04	四个轨道
00 60	96 /四分音符

然后音轨块为时间签名/节奏音轨。它的头，后面跟着事件:

4D 54 72 6B	MTrk
00 00 00 14	块长度(20)

Delta-time	Event	Comments
00	FF 58 04 04 02 18 08	时间签名
00	FF 51 03 07 A1 20	节奏
83 00	FF 2F 00	尾轨

然后，首首音乐曲目的音轨块。在这个例子中使用了音符开/关运行状态的 MIDI 约定:

4D 54 72 6B	MTrk
00 00 00 10	块长度(16)

Delta-time	Event	Comments
00 c00	05	
81 40	90 4c 20	
81 40	4C 00	运行状态:注开，vel = 0
00	FF 2F 00	轨道结束

然后，第二首音乐曲目的音轨块:

4D 54 72 6B	MTrk
00 00 00 0F	块长度(15)

Delta-time	Event	Comments
00	c1 2e	
60	91 43 40	

```
82 2043      00运行状态    00   FF 2F 00 轨道结束
```

然后，第三首音乐曲目的音轨块:

```
                4D 54 72 6B          MTrk
                00 00 00 15              块长度(21)
```

Delta-time	Event	Comments
00	c2 46	
00	92 30 60	
00	3C 60	运行状态
83 00	30 00	两字节增量时间，运行状态
00	3C 00	运行状态
00	FF 2F 00	轨道结束

计算 Delta Times:

下面是一个如何将累积 delta 时间转换为毫秒的例子。在最简单的情况下，需要 2 条信息:

1) SMF Header Chunk 定义了一个“除法”，即每个四分音符的 delta 节拍。(如。， 96 = 96 ppq) (Ref: pg. 4, SMF 1.0)
2) Tempo 设置，这是一个非 midi 数据 Meta Event，通常在 SMF 的第一个音轨的时间 delta 时间 0 处发现。如果未指定，则假定 tempo 为 120 bpm。节拍以每四分音符的微秒数表示。(如。， 500000 = 120 bpm)。(参考:
   动力分配。5,9, SMF 1.0)

要将增量时间转换为毫秒，只需进行简单的代数计算:

Time(单位 ms) = (Number of Ticks) * (Tempo (uS/qn) / Div (Ticks /qn)) / 1000

举个例子，如果 Set Tempo 的值是 500000 uS / qn，而 Division 是 96 ticks / qn，那么进入 SMF 的 6144 ticks 的时间量为:

Time = 6144 * (500000/96) / 1000
= 32000 毫秒

上面的例子是一个非常简单的情况。在实践中，smf 可以包含多个 Set Tempo Meta Events，它们间隔在整个文件中，为了计算任何 Tick 的正确经过时间，需要执行一个运行计算。

请注意，虽然执行上述计算不需要时间签名，但是，如果需要特定 Bar/Beat 值的运行时间，则需要时间签名。与 Set Tempo 变化一样，Time Signature 可以在整个 SMF 中变化，并且通常需要运行计算来确定任何 Bar/Beat 的正确流逝时间。