This page describes, in Unix manual page style, programs available for downloading from this site which translate MIDI music files into a human- and computer-readable CSV (Comma-Separated Value) format, suitable for manipulation by spreadsheet, database, or text processing utilities, and re-encoding processed CSV files into MIDI. No information is lost in transforming a MIDI file to CSV and back.

For decades, MIDI files have been the *lingua franca* of electronic musicians. Representing music at the level of a musical score, as opposed to a performance as do audio files such as MP3, they allow compositions to be manipulated in ways creative musicians do—orchestrated, assigned to different instruments, transposed to different keys, and adjusted in tempo, just to name a few. Most musicians work on MIDI files using special-purpose music editing programs called (for historical reasons) *sequencers*. These programs are ideally suited for capturing music as it is played and editing it into a final composition, but often lack the ability to easily apply transformations in bulk to large collections of music files (for example, changing instrument assignments for a set of files from those of one studio to another), algorithmically transforming music, or programmatically generating new music.

MIDI files are stored in a compact binary form intended to optimise speed (and hence reduce latency) on limited speed MIDI channels, while not overly taxing the processing power of the humble microcontrollers employed by instruments in the epoch when MIDI was developed. While well-suited to its intended purposes, MIDI files are somewhat difficult to read and write without a special-purpose library, particularly in commonly used text processing languages such as Perl and Python which are otherwise ideally suited to the kinds of transformations one often wishes to apply to MIDI-encoded music.

The **midicsv** and **csvmidi** programs allow you to use text processing tools to transform MIDI files or create them from scratch. **midicsv** converts a MIDI file into a Comma-Separated Value (CSV) text file, preserving all the information in the original MIDI file. The format of the CSV file is deliberately designed to facilitate processing by text manipulation tools. The **csvmidi** program reads CSV files in this format and creates an equivalent MIDI file. Processing a MIDI file through **midicsv** and then **csvmidi** will result in a completely equivalent (although, for technical reasons relating to options in MIDI file compression, not necessarily byte-for-byte identical) MIDI file.

MIDICSV Morsels

Odds are, if these utilities are what you're looking for, you've already thought of half a dozen ways to apply them to MIDI file manipulation. The following examples show how easy it is to transform MIDI files once they're expressed in CSV format. All of these examples are written in the Perl language; it's well suited to tasks of this type and it's the tool I reach for when faced with such problems. Any other language with comparable facilities for text manipulation is equally applicable. Source code for all of these Perl programs is included in the **midicsv** distribution. In the interest of brevity, I've elided comments in the program listings below and moved discussion to text preceding each example. If you don't have a hardware MIDI synthesiser, you can play MIDI files on most computers with the TiMidity++ software synthesiser, which can also create wave audio files from MIDI songs.

Transpose Key (`transpose.pl`)

All note events in the MIDI files, except those on channel 9, which the General MIDI standard dedicates to percussion, are shifted by the `$offset`, set in this example to −12, hence one octave lower. The Perl program simply filters the CSV representation of the MIDI file looking for note-on and note-off events. It excludes those for channel 9 (percussion), then applies the offset to the note number of the remaining events. This program might be used to transform a MIDI file with a pipeline like:

```
midicsv song.mid | perl transpose.pl | csvmidi >tsong.mid
```

```
$offset = -12;
$percussion = 9;

while ($a = <>) {
    if ($a =~ s/(\d+,\s*\d+,\s*Note_\w+,\s*(\d+),\s*)(\d+)//) {
        $n = $3;
        if ($2 != $percussion) {
            $n += $offset;
        }
        if ($n < 0) {
            next;
        }
        $a = "$1$n$a";
    }
    print($a);
}
```

Chorus (`chorus.pl`)

All note events in the MIDI files, except those on channel 9, which the General MIDI standard dedicates to percussion, are duplicated with the `$offset`, set in this example to −12, hence one octave lower. This is very similar to the transposition example above, but instead of replacing notes by offset notes, adds simultaneously sounded offset notes. When used with a negative `$offset` of one or more octave, this “fattens” the bass of a composition. Try it! This program may be used to transform a MIDI file with a pipeline like:

```
midicsv song.mid | perl chorus.pl | csvmidi >tsong.mid
```

```
$offset = -12;
$percussion = 9;

while ($a = <>) {
    print($a);
    if ($a =~ s/(\d+,\s*\d+,\s*Note_\w+,\s*(\d+),\s*)(\d+)//) {
        if ($2 != $percussion) {
            $n = $3;
            $n += $offset;
            if ($n < 0) {
                next;
            }
            $a = "$1$n$a";
            print($a);
        }
    }
}
```

Extract Channel (`exchannel.pl`)

Here's a program which extracts all MIDI events pertaining to a given channel (`$which_channel` in the program below), passing through the rest of the MIDI file structure. With `$which_channel` set to 9, as in this example, this program will extract the percussion track from a General MIDI file. You can run this program with a pipeline such as:

```
midicsv song.mid | perl exchannel.pl | csvmidi >tsong.mid

     $which_channel = 9;

     while ($a = <>) {
         if (!($a =~ m/\s*[\#\;]/)) {    # Ignore comment lines
             if ($a =~ m/\s*\d+\s*,\s*\d+\s*,\s*\w+_c\s*,\s*(\d+)/) {
                 if ($1 == $which_channel) {
                     print($a);
                 }
             } else {
                 print($a);
             }
         }
     }
```

Dumbest Drummer (`drummer.pl`)

This is about the simplest drum machine imaginable. You define a rudimentary rhythm track in an Perl array, then call the `&loop` function to play it a given number of times. It's so simpleminded, it doesn't even allow overlap of percussion hits—each note ends before the next sounds. A percussion note of 0 denotes a rest. This example includes the **general_midi.pl** file, included in the distribution, which defines two hashes that allow you to specify General MIDI patches (programs/instruments) and percussion note numbers by name. To run this program and create a ready-to-play MIDI file, use the pipeline:

```
perl drummer.pl | csvmidi >drumtrack.mid


     require 'general_midi.pl';

 #          Repeats, Note,
 #             Duration, Velocity
     @track = (4, $GM_Percussion{'Acoustic Bass Drum'},
                  480, 127,
               4, $GM_Percussion{'Low-Mid Tom'},
                  240, 127,
               1, 0, 120, 0,
               2,  $GM_Percussion{'Hand Clap'},
                  240, 127,
               1, 0, 240, 0
              );

     print << "EOD";
 0, 0, Header, 1, 1, 480
 1, 0, Start_track
 1, 0, Tempo, 500000
 EOD

     $time = 0;

     &loop(4, @track);

     print << "EOD";
 1, $time, End_track
 0, 0, End_of_file
 EOD

     sub note {  # &note($note_number, $duration [, $velocity])
         local ($which, $duration, $vel) = @_;

         if ($which > 0) {
             if (!defined($vel)) {
                 $vel = 127;
             }
             print("1, $time, Note_on_c, 9, $which, $vel\n");
         }
         $time += $duration;
         if ($which > 0) {
```

```
            print("1, $time, Note_off_c, 9, $which, 0\n");
        }
    }

    sub loop {  # &loop($ntimes, @track)
        local ($loops, @tr) = @_;
        local ($i, $r);

        for ($i = 0; $i < $loops; $i++) {
            local @t = @tr;
            while ($#t > 0) {
                local ($repeats, $note, $duration, $velocity) =
                    splice(@t, 0, 4);
                for ($r = 0; $r < $repeats; $r++) {
                    &note($note, $duration, $velocity);
                }
            }
        }
    }
```

Stoned Guitarist (`acomp.pl`)

Finally, we have this dumb-as-a-bag-of-hair algorithmic composer which simulates an around the bend musician noodling on an electric guitar. There's accompaniment by a cymbal, and a cymbal crash and utterly unwarranted applause at the end. This program isn't presented for its musical merit, but simply to illustrate how easy it is to generate arbitrary MIDI music files by writing CSV which is fed through **csvmidi**. This example also uses the definitions in the **general_midi.pl**. It may be run as follows:

```
perl acomp.pl | csvmidi >performance.mid
```

```
    require 'general_midi.pl';

    $instrument = $GM_Patch{'Distortion Guitar'};
    $tonespan = 32;
    $num_notes = 120;
    $percussion = $GM_Percussion{'Ride Cymbal 1'};
    $beat = 6;

    print << "EOD";
0, 0, Header, 1, 1, 480
1, 0, Start_track
1, 0, Tempo, 500000
1, 0, Program_c, 1, $instrument
EOD

    $time = 0;
    srand(time());

    for ($i = 0; $i < $num_notes; $i++) {
        $n = 60 + int((rand() * $tonespan) - int($tonespan / 2));
        $notelength = 120 + (60 * int(rand() * 6));
        &note(1, $n, $notelength, 127);
        if (($i % $beat) == 0) {
            print("1, $time, Note_on_c, 9, $percussion, 127\n");
        } elsif (($i % $beat) == ($beat - 1)) {
            print("1, $time, Note_off_c, 9, $percussion, 0\n");
        }
    }

    #   Cymbal crash at end
    $cymbal = $GM_Percussion{'Crash Cymbal 2'};
    print("1, $time, Note_on_c, 9, $cymbal, 127\n");
    $time += 480;
    print("1, $time, Note_off_c, 9, $cymbal, 0\n");

    #   Audience applause
    $time += 480;
    print("1, $time, Program_c, 1, $GM_Patch{'Applause'}\n");
```

```
        print("1, $time, Note_on_c, 1, 60, 100\n");
        for ($i = 16; $i <= 32; $i++) {
            $time += 120;
            $v = int(127 * ($i / 32));
            print("1, $time, Poly_aftertouch_c, 1, 60, $v\n");
        }
        for ($i = 32; $i >= 0; $i--) {
            $time += 240;
            $v = int(127 * ($i / 32));
            print("1, $time, Poly_aftertouch_c, 1, 60, $v\n");
        }
        print("1, $time, Note_off_c, 1, 60, 0\n");

        print << "EOD";
    1, $time, End_track
    0, 0, End_of_file
    EOD

        # &note($channel, $note_number, $duration [, $velocity])
        sub note {
            local ($channel, $which, $duration, $vel) = @_;

            if (!defined($vel)) {
                $vel = 127;
            }
            print("1, $time, Note_on_c, $channel, $which, $vel\n");
            $time += $duration;
            print("1, $time, Note_off_c, $channel, $which, 0\n");
        }
```

Download midicsv-1.1.tar.gz (Gzipped TAR archive)

The archive contains source code for the utilities, a Makefile for Unix systems, and ready-to-run executables for 32-bit Windows platforms. If you require only the Windows executables, you can download a Zipped archive containing just those files.

Manual Pages

midicsv

NAME

midicsv - translate MIDI file to CSV

SYNOPSIS

midicsv [**-u -v**] [*infile* [*outfile*]]

DESCRIPTION

midicsv reads a standard MIDI file and decodes it into a CSV (Comma-Separated Value) file which preserves all the information in the MIDI file. The ASCII CSV file may be loaded into a spreadsheet or database application, or processed by a program to transform the MIDI data (for example, to key transpose a composition or extract a track from a multi-track sequence). A CSV file in the format created by **midicsv** may be converted back into a standard MIDI file with the **csvmidi** program.

OPTIONS

-u
: Print how-to-call information.

-v
: Print verbose debugging information on standard error. The MIDI file header is dumped, along with the length of each track in the file.

FILES

If no *infile* is specified or *infile* is "**-**", **midicsv** reads its input from standard input; if no *outfile* is given or *outfile* is "**-**", CSV output is written to standard output. The input and output are processed in a strictly serial manner; consequently **midicsv** may be used in pipelines without restrictions.

BUGS

midicsv assumes its input is a well-formed standard MIDI file; while some error checking is performed, gross errors in the input file may cause **midicsv** to crash.

Please report problems to bugs **at** fourmilab.ch.

SEE ALSO

csvmidi(1), **midicsv**(5)

csvmidi

NAME

csvmidi - encode CSV file as MIDI

SYNOPSIS

csvmidi [**-u -v -x -z**] [*infile* [*outfile*]]

DESCRIPTION

csvmidi reads a CSV (Comma-Separated Value) file in the format written by **midicsv** and creates the equivalent standard MIDI file.

OPTIONS

-u
: Print how-to-call information.

-v
: Print verbose debugging information on standard error. The MIDI file header is dumped, along with the length of each track in the file.

-x
: MIDI streams support a rudimentary form of compression in which successive events with the same "status" (event type and channel) may omit the status byte. By default **csvmidi** avails itself of this compression. If the **-x** option is specified, the status byte is emitted for all events—it is never compressed even when the MIDI standard permits it to be.

-z
: Most errors detected in CSV records cause a warning message to be displayed on standard error and the record ignored. The **-z** option causes **csvmidi** to immediately terminate processing when the first error is detected.

EXIT STATUS

If no errors or warnings are detected **csvmidi** exits with status 0. A status of of 1 is returned if one or more errors were detected in the CSV input file, while a status of 2 indicates a syntax error on the command line or inability to open the input or output file.

FILES

If no *infile* is specified or *infile* is "**-**", **csvmidi** reads its input from standard input; if no *outfile* is given or *outfile* is "**-**", MIDI output is written to standard output. The input and output are processed in a strictly serial manner; consequently **csvmidi** may be used in pipelines without restrictions.

BUGS

csvmidi assumes its input is in the format written by **midicsv**. If supplied a CSV file with well-formed records which nonetheless makes no semantic sense as a MIDI file, the results will, in all likelihood, simply perplex any program or instrument to which it's sent. **csvmidi** checks for missing fields and range checks all numeric values, but does not perform higher-level consistency checking (for example, making sure that every note on event is paired with a subsequent note off). That level of verification, if required, should be done on the CSV file before it is processed by **csvmidi**.

Exporting a file to CSV with **midicsv** and then importing it with **csvmidi** is not guaranteed to create an identical MIDI file. MIDI files support compression modes which are not obligatory. A MIDI file exported to CSV and then re-imported should, however, be *equivalent* to the original file and should, if exported to CSV, be identical to the CSV exported from the original file.

Please report problems to bugs **at** fourmilab.ch.

SEE ALSO

midicsv(1), **midicsv**(5)

midicsv File Format

NAME

midicsv - MIDI Comma-Separated Value (CSV) file format

DESCRIPTION

The **midicsv** and **csvmidi** programs permit you to intertranslate standard MIDI files and comma-separated value (CSV) files. These CSV files preserve all information in the MIDI file, and may be loaded into spreadsheet and database programs or easily manipulated with text processing tools. This document describes the CSV representation of MIDI files written by **midicsv** and read by **csvmidi**. Readers are assumed to understand the structure, terminology, and contents of MIDI files—please refer to a MIDI file reference for details.

RECORD STRUCTURE

Each record in the CSV representation of a MIDI contains at least three fields:

Track
: Numeric field identifying the track to which this record belongs. Tracks of MIDI data are numbered starting at 1. Track 0 is reserved for file header, information, and end of file records.

Time
: Absolute time, in terms of MIDI clocks, at which this event occurs. Meta-events for which time is not meaningful (for example, song title, copyright information, etc.) have an absolute time of 0.

Type
: Name identifying the type of the record. Record types are text consisting of upper and lower case letters and the underscore ("_"), contain no embedded spaces, and are not enclosed in quotes. **csvmidi** ignores upper/lower case in the **Type** field; the specifications "**Note_on_c**", "**Note_On_C**", and "**NOTE_ON_C**" are considered identical.

Records in the CSV file are sorted first by the track number, then by time. Out of order records will be discarded with an error message from **csvmidi**. Following the three required fields are parameter fields which depend upon the **Type**; some **Type**s take no parameters. Each **Type** and its parameter fields is discussed below.

Any line with an initial nonblank character of "**#**" or "**;**" is ignored; either delimiter may be used to introduce comments in a CSV file. Only full-line comments are permitted; you cannot use these delimiters to terminate scanning of a regular data record. Completely blank lines are ignored.

File Structure Records

0, 0, Header, *format, nTracks, division*
: The first record of a CSV MIDI file is always the **Header** record. Parameters are *format*: the MIDI file type (0, 1, or 2), *nTracks*: the number of tracks in the file, and *division*: the number of clock pulses per quarter note. The **Track** and **Time** fields are always zero.

0, 0, End_of_file
: The last record in a CSV MIDI file is always an **End_of_file** record. Its **Track** and **Time** fields are always zero.

Track, **0, Start_track**
: A **Start_track** record marks the start of a new track, with the *Track* field giving the track number. All records between the **Start_track** record and the matching **End_track** will have the same *Track* field.

Track, Time, **End_track**
: An **End_track** marks the end of events for the specified *Track*. The *Time* field gives the total duration of the track, which will be identical to the *Time* in the last event before the **End_track**.

File Meta-Events

The following events occur within MIDI tracks and specify various kinds of information and actions. They may appear at any time within the track. Those which provide general information for which time is not relevant usually appear at the start of the track with **Time** zero, but this is not a requirement.

Many of these meta-events include a text string argument. Text strings are output in CSV records enclosed in ASCII double quote (") characters. Quote characters embedded within strings are represented by two consecutive quotes. Non-graphic characters in the ISO 8859-1 Latin-1 set are output as a backslash followed by their three digit octal character code. Two consecutive backslashes denote a literal backslash in the string. Strings in MIDI files can be extremely long, theoretically as many as $2^{28}-1$ characters; programs which process MIDI CSV files should take care to avoid buffer overflows or truncation resulting from lines containing long string items. All meta-events which take a text argument are identified by a suffix of "**_t**".

Track, Time, **Title_t,** *Text*
: The *Text* specifies the title of the track or sequence. The first **Title** meta-event in a type 0 MIDI file, or in the first track of a type 1 file gives the name of the work. Subsequent **Title** meta-events in other tracks give the names of those tracks.

Track, Time, **Copyright_t,** *Text*
: The *Text* specifies copyright information for the sequence. This is usually placed at time 0 of the first track in the sequence.

Track, Time, **Instrument_name_t,** *Text*
: The *Text* names the instrument intended to play the contents of this track, This is usually placed at time 0 of the track. Note that this meta-event is simply a description; MIDI synthesisers are not required (and rarely if ever) respond to it. This meta-event is particularly useful in sequences prepared for synthesisers which do not conform to the General MIDI patch set, as it documents the intended instrument for the track when the sequence is used on a synthesiser with a different patch set.

Track, Time, **Marker_t,** *Text*
: The *Text* marks a point in the sequence which occurs at the given *Time*, for example "Third Movement".

Track, Time, **Cue_point_t,** *Text*
: The *Text* identifies synchronisation point which occurs at the specified *Time*, for example, "Door slams".

Track, Time, **Lyric_t,** *Text*
: The *Text* gives a lyric intended to be sung at the given *Time*. Lyrics are often broken down into separate syllables to time-align them more precisely with the sequence.

Track, Time, **Text_t,** *Text*
: This meta-event supplies an arbitrary *Text* string tagged to the *Track* and *Time*. It can be used for textual information which doesn't fall into one of the more specific categories given above.

Track, **0, Sequence_number,** *Number*
: This meta-event specifies a sequence *Number* between 0 and 65535, used to arrange multiple tracks in a type 2 MIDI file, or to identify the sequence in which a collection of type 0 or 1 MIDI files should be played. The **Sequence_number** meta-event should occur at **Time** zero, at the start of the track.

Track, Time, **MIDI_port,** *Number*
: This meta-event specifies that subsequent events in the **Track** should be sent to MIDI port (bus) *Number*, between 0 and 255. This meta-event usually appears at the start of a track with **Time** zero, but may appear within a track should the need arise to change the port while the track is being played.

Track, Time, **Channel_prefix,** *Number*
: This meta-event specifies the MIDI channel that subsequent meta-events and **System_exclusive** events pertain to. The channel *Number* specifies a MIDI channel from 0 to 15. In fact, the *Number* may be as large as 255, but the consequences of specifying a channel number greater than 15 are undefined.

Track, Time, **Time_signature,** *Num, Denom, Click, NotesQ*
: The time signature, metronome click rate, and number of 32nd notes per MIDI quarter note (24 MIDI clock times) are given by the numeric arguments. *Num* gives the numerator of the time signature as specified on sheet music. *Denom* specifies the denominator as a negative power of two, for example 2 for a quarter note, 3 for an eighth note, etc. *Click* gives the number of MIDI clocks per metronome click, and *NotesQ* the number of 32nd notes in the nominal MIDI quarter note time of 24 clocks (8 for the default MIDI quarter note definition).

Track, Time, **Key_signature,** *Key, Major/Minor*
: The key signature is specified by the numeric *Key* value, which is 0 for the key of C, a positive value for each sharp above C, or a negative value for each flat below C, thus in the inclusive range −7 to 7. The *Major/Minor* field is a quoted string which will be **major** for a major key and **minor** for a minor key.

Track, Time, **Tempo,** *Number*
: The tempo is specified as the *Number* of microseconds per quarter note, between 1 and 16777215. A value of 500000 corresponds to 120 quarter notes ("beats") per minute. To convert beats per minute to a **Tempo** *value*, take the quotient from dividing 60,000,000 by the beats per minute.

Track, **0, SMPTE_offset,** *Hour, Minute, Second, Frame, FracFrame*
: This meta-event, which must occur with a zero **Time** at the start of a track, specifies the SMPTE time code at which it should start playing. The *FracFrame* field gives the fractional frame time (0 to 99).

Track, Time, **Sequencer_specific,** *Length, Data, ...*
: The **Sequencer_specific** meta-event is used to store vendor-proprietary data in a MIDI file. The *Length* can be any value between 0 and $2^{28}-1$, specifying the number of *Data* bytes (between 0 and 255) which follow. **Sequencer_specific** records may be very long; programs which process MIDI CSV files should be careful to protect against buffer overflows and truncation of these records.

Track, Time, **Unknown_meta_event,** *Type, Length, Data, ...*
: If **midicsv** encounters a meta-event with a code not defined by the standard MIDI file specification, it outputs an unknown meta-event record in which *Type* gives the numeric meta-event type code, *Length* the number of data bytes in the meta-event, which can be any value between 0 and $2^{28}-1$, followed by the *Data* bytes. Since meta-events include their own length, it is possible to parse them even if their type and meaning are unknown. **csvmidi** will reconstruct unknown meta-events with the same type code and content as in the original MIDI file.

Channel Events

These events are the "meat and potatoes" of MIDI files: the actual notes and modifiers that command the instruments to play the music. Each has a MIDI channel number as its first argument, followed by event-specific parameters. To permit programs which process CSV files to easily distinguish them from meta-events, names of channel events all have a suffix of "**_c**".

Track, Time, **Note_on_c,** *Channel, Note, Velocity*
: Send a command to play the specified *Note* (Middle C is defined as *Note* number 60; all other notes are relative in the MIDI specification, but most instruments conform to the well-tempered scale) on the given *Channel* with *Velocity* (0 to 127). A **Note_on_c** event with *Velocity* zero is equivalent to a **Note_off_c**.

Track, Time, **Note_off_c,** *Channel, Note, Velocity*
: Stop playing the specified *Note* on the given *Channel*. The *Velocity* should be zero, but you never know what you'll find in a MIDI file.

Track, Time, **Pitch_bend_c,** *Channel, Value*
: Send a pitch bend command of the specified *Value* to the given *Channel*. The pitch bend *Value* is a 14 bit unsigned integer and hence must be in the inclusive range from 0 to 16383. The value 8192 indicates no pitch bend; 0 the lowest pitch bend, and 16383 the highest. The actual change in pitch these values produce is unspecified.

Track, Time, **Control_c,** *Channel, Control_num, Value*
: Set the controller *Control_num* on the given *Channel* to the specified *Value*. *Control_num* and *Value* must be in the inclusive range 0 to 127. The assignment of *Control_num* values to effects differs from instrument to instrument. The General MIDI specification defines the meaning of controllers 1 (modulation), 7 (volume), 10 (pan), 11 (expression),

and 64 (sustain), but not all instruments and patches respond to these controllers. Instruments which support those capabilities usually assign reverberation to controller 91 and chorus to controller 93.

Track, Time, **Program_c,** *Channel, Program_num*
Switch the specified *Channel* to program (patch) *Program_num*, which must be between 0 and 127. The program or patch selects which instrument and associated settings that channel will emulate. The General MIDI specification provides a standard set of instruments, but synthesisers are free to implement other sets of instruments and many permit the user to create custom patches and assign them to program numbers.

Apparently, due to instrument manufacturers' skepticism about musicians' ability to cope with the number zero, many instruments number patches from 1 to 128 rather than the 0 to 127 used within MIDI files. When interpreting *Program_num* values, note that they may be one less than the patch numbers given in an instrument's documentation.

Track, Time, **Channel_aftertouch_c,** *Channel, Value*
When a key is held down after being pressed, some synthesisers send the pressure, repeatedly if it varies, until the key is released, but do not distinguish pressure on different keys played simultaneously and held down. This is referred to as "monophonic" or "channel" aftertouch (the latter indicating it applies to the *Channel* as a whole, not individual note numbers on that channel). The pressure *Value* (0 to 127) is typically taken to apply to the last note played, but instruments are not guaranteed to behave in this manner.

Track, Time, **Poly_aftertouch_c,** *Channel, Note, Value*
Polyphonic synthesisers (those capable of playing multiple notes simultaneously on a single channel), often provide independent aftertouch for each note. This event specifies the aftertouch pressure *Value* (0 to 127) for the specified *Note* on the given *Channel*.

System Exclusive Events

System Exclusive events permit storing vendor-specific information to be transmitted to that vendor's products.

Track, Time, **System_exclusive,** *Length, Data, ...*
The *Length* bytes of *Data* (0 to 255) are sent at the specified *Time* to the MIDI channel defined by the most recent **Channel_prefix** event on the *Track*, as a System Exclusive message. Note that *Length* can be any value between 0 and $2^{28}-1$. Programs which process MIDI CSV files should be careful to protect against buffer overflows and truncation of these records.

Track, Time, **System_exclusive_packet,** *Length, Data, ...*
The *Length* bytes of *Data* (0 to 255) are sent at the specified *Time* to the MIDI channel defined by the most recent **Channel_prefix** event on the *Track*. The *Data* bytes are simply blasted out to the MIDI bus without any prefix. This message is used by MIDI devices which break up long system exclusive message into small packets, spaced out in time to avoid overdriving their modest microcontrollers. Note that *Length* can be any value between 0 and $2^{28}-1$. Programs which process MIDI CSV files should be careful to protect against buffer overflows and truncation of these records.

EXAMPLES

The following CSV file defines the five-note motif from the film *Close Encounters of the Third Kind* using an organ patch from the General MIDI instrument set. When processed by **midicsv** and sent to a synthesiser which conforms to General MIDI, the sequence will be played.

```
0, 0, Header, 1, 2, 480
1, 0, Start_track
1, 0, Title_t, "Close Encounters"
1, 0, Text_t, "Sample for MIDIcsv Distribution"
1, 0, Copyright_t, "This file is in the public domain"
1, 0, Time_signature, 4, 2, 24, 8
1, 0, Tempo, 500000
1, 0, End_track
2, 0, Start_track
2, 0, Instrument_name_t, "Church Organ"
2, 0, Program_c, 1, 19
2, 0, Note_on_c, 1, 79, 81
2, 960, Note_off_c, 1, 79, 0
2, 960, Note_on_c, 1, 81, 81
2, 1920, Note_off_c, 1, 81, 0
2, 1920, Note_on_c, 1, 77, 81
2, 2880, Note_off_c, 1, 77, 0
2, 2880, Note_on_c, 1, 65, 81
2, 3840, Note_off_c, 1, 65, 0
2, 3840, Note_on_c, 1, 72, 81
2, 4800, Note_off_c, 1, 72, 0
2, 4800, End_track
```

```
0, 0, End_of_file
```

BUGS

The CSV representation of a MIDI file is simply a text-oriented encoding of its contents. If the input to **midicsv** contains errors which violate the MIDI standard, the resulting CSV file will faithfully replicate these errors. Similarly, the CSV input to **csvmidi** must not only consist of records which conform to the syntax given in this document, the input as a whole must also be a *semantically* correct MIDI file. Programs which wish to use **csvmidi** to generate MIDI files from scratch should be careful to conform to the structure required of MIDI files. When in doubt, use **midicsv** to dump a sequence comparable to the one your program will create and use its structure as a template for your own.

Please report errors to bugs **at** fourmilab.ch.

SEE ALSO

csvmidi(1), **midicsv**(1)

This software is in the public domain. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, without any conditions or restrictions. This software is provided “as is” without express or implied warranty.

---

by John Walker
February, 2004
Revised January, 2008

Fourmilab Home Page