MIDI

This information is gleaned from various text files in the public domain.

Source documents:
The USENET MIDI Primer
MIDI File Format 1.1
M Garvin

Channel usage
Standard patch assignments
Standard key assignments

Compiled and adapted by: Neil Rowland, Jr.

Channel usage for Windows

Windows docs numbers channels 1-16, but everywhere else it's 0-15, particularly in the MIDI file format. I'll stick to the zero-based numbering.

Channels **0 thru 8**: extended melodic tracks, 16 notes polyphony. Not all synths support these.

Channel **9**: extended percussion track, 16 "notes" polyphony. Not all synths support these.

Channels **10 and 11**: unused.

Channels **12 thru 14**: base-level melodic tracks, 6 notes polyphony.

Channel **15**: base-level percussion track, 3 "notes" polyphony.

See:
DATA FORMAT
VOICE MESSAGES

Standard patch assignments for Windows

Patch numbers are divided by category into 16 groups of 8 values each...

0 - 7	**Piano**
8 - 15	**Chromatic Percussion**
16 - 23	**Organ**
24 - 31	**Guitar**
32 - 39	**Bass**
40 - 47	**Strings**
48 - 55	**Ensemble**
56 - 63	**Brass**
64 - 71	**Reed**
72 - 79	**Pipe**
80 - 87	**Synth Lead**
88 - 95	**Synth Pad**
96 - 103	**Synth Effects**
104-111	**Ethnic**
112-119	**Percussive**
120-127	**Sound Effects**

Standard key assignments for Windows

For notes, middle C = 60.
A below middle C = 57.

For percussion:

35	acoustic bass drum
36	bass drum 1
36	side stick
38	acoustic snare
39	hand clap
40	electric snare
41	low floor tom
42	closed high hat
43	high floor tom
44	pedal high hat
45	low tom
46	open high hat
47	low-mid tom
48	high-mid tom
49	crash cymbal 1
50	high tom
51	ride cymbal 1
52	chines cymbal
53	ride bell
54	tambourine
55	splash cymbal
56	cowbell
57	crash cymbal 2
58	vibraslap
59	ride cymbal 2
60	high bongo
61	low bongo
62	mute high conga
63	open high conga
64	low conga
65	high tymbale
66	low timbale
67	high apogo
68	low apogo
69	cabasa
70	maracas
71	short whistle
72	long whistle
73	short guiro
74	long guiro
75	claves
76	high wood block
77	low wood block
78	mute cuica
79	open cuica
80	mute triangle
81	open triangle

See:

VOICE MESSAGES

Standard MIDI-File Format Spec. 1.1

Distributed by:
The International MIDI Association
5316 W. 57th St.
Los Angeles, CA 90056
(213) 649-6434

0 - Introduction

The document outlines the specification for MIDI Files. The purpose of MIDI Files is to provide a way of interchanging time-stamped MIDI data between different programs on the same or different computers. One of the primary design goals is compact representation, which makes it very appropriate for disk-based file format, but which might make it inappropriate for storing in memory for quick access by a sequencer program. (It can be easily converted to a quickly-accessible format on the fly as files are read in or written out.) It is not intended to replace the normal file format of any program, though it could be used for this purpose if desired.

MIDI Files contain one or more MIDI streams, with time information for each event. Song, sequence, and track structures, tempo and time signature information, are all supported. Track names and other descriptive information may be stored with the MIDI data. This format supports multiple tracks and multiple sequences so that if the user of a program which supports multiple tracks intends to move a file to another one, this format can allow that to happen.

This spec defines the 8-bit binary data stream used in the file. The data can be stored in a binary file, nibbilized, 7-bit-ized for efficient MIDI transmission, converted to Hex ASCII, or translated symbolically to a printable text file. This spec addresses what's in the 8-bit stream. It does not address how a MIDI File will be transmitted over MIDI. It is the general feeling that a MIDI transmission protocol will be developed for files in general and MIDI Files will use this scheme.

1 - Sequences, Tracks, Chunks: File Block Structure

CONVENTIONS
In this document, bit 0 means the least significant bit of a byte, and bit 7 is the most significant.

Some numbers in MIDI Files are represented is a form called **VARIABLE-LENGTH QUANTITY**. These numbers are represented 7 bits per byte, most significant bits first. All bytes except the last have bit 7 set, and the last byte has bit 7 clear. If the number is between 0 and 127, it is thus represented exactly as one byte.

Here are some examples of numbers represented as variable-length quantities:

00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
00004000	81 80 00
00100000	C0 80 00
001FFFFF	FF FF 7F
00200000	81 80 80 00
08000000	C0 80 80 00
0FFFFFFF	FF FF FF 7F

The largest number which is allowed is 0FFFFFFF so that the variable-length representations must fit in 32 bits in a routine to write variable-length numbers. Theoretically, larger numbers are possible, but 2 x 10^8 96ths of a beat at a fast tempo of 500 beats per minute is four days, long enough for any delta-time!

FILES
To any file system, a MIDI File is simply a series of 8-bit bytes. On the Macintosh, this byte stream is stored in the data fork of a file (with file type 'MIDI'), or on the Clipboard (with data type 'MIDI'). Most other computers store 8-bit byte streams in files -- naming or storage conventions for those computers will be defined as required.

CHUNKS
MIDI Files are made up of -chunks-. Each chunk has a **4-character type** and a**32-bit length**, which is the number of bytes in the chunk. This structure allows future chunk types to be designed which may be easily be ignored if encountered by a program written before the chunk type is introduced. Your programs should EXPECT alien chunks and treat them as if they weren't there.

Each chunk begins with a 4-character ASCII type. It is followed by a 32-bit length, **most significant byte first** (a length of 6 is stored as 00 00 00 06). This length refers to the number of bytes of data which follow: the eight bytes of type and length are not included. Therefore, a chunk with a length of 6 would actually occupy 14 bytes in the disk file.

This chunk architecture is similar to that used by Electronic Arts' IFF format, and the chunks described herin could easily be placed in an IFF file. The MIDI File itself is not an IFF file: it contains no nested chunks, and chunks are not constrained to be an even number of bytes long. Converting it to an IFF file is as easy as padding odd length chunks, and sticking the whole thing inside a FORM chunk.

MIDI Files contain two types of chunks: **header chunks** and **track chunks**. A -header-chunk provides a minimal amount of information pertaining to the entire MIDI file. A -track-chunk contains a sequential stream of MIDI data which may contain information for up to 16 MIDI **channels**. The concepts of multiple tracks, multiple MIDI outputs, patterns, sequences, and songs may all be implemented using several **track chunks**.

A MIDI File always starts with a header chunk, and is followed by one or more **track chunks**.

MThd <length of header data>
<header data>
MTrk <length of track data>
<track data>
MTrk <length of track data>
<track data>
. . .

HEADER CHUNKS

The header chunk at the beginning of the file specifies some basic information about the data in the file. Here's the syntax of the complete chunk:

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

As described above, <chunk type> is the four ASCII characters 'MThd'; <length> is a 32-bit representation of the number 6 (high byte first).

The data section contains three 16-bit words, stored most-significant byte first.

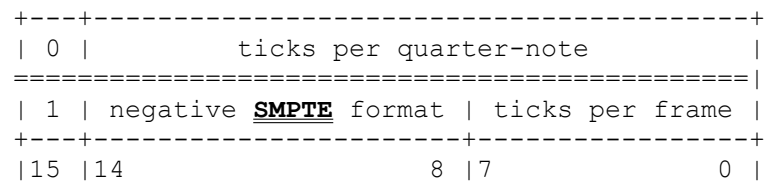
The first word, **<format>**, specifies the overall organization of the file. Only three values of <format> are specified:

0-the file contains a single multi-channel track
1-the file contains one or more simultanious tracks (or MIDI outputs) of a sequence
2-the file contains one or more sequentially independant single-track patterns

More information about these formats is provided below.
(**FORMATS 0, 1, AND 2**)

The next word, **<ntrks>**, is the number of **track chunks** in the file. It will always be 1 for a format 0 file.

The third word, **<division>**, specifies the meaning of the delta-times. It has two formats, one for metrical time, and one for time-code-based time:



If bit 15 of <division> is zero, the bits 14 thru 0 represent the number of delta time "ticks" which make up a quarter-note. For instance, if division is 96, then a time interval of an eighth-note between two events in the file would be 48.

If bit 15 of <division> is a one, delta times in a file correspond to subdivisions of a second, in a way consistent with **SMPTE** and MIDI Time Code. Bits 14 thru 8 contain one of the four values -24, -25, -29, or -30, corresponding to the four standard **SMPTE** and MIDI Time Code formats (-29 corresponds to 30 drop frome), and represents the number of frames per second. These negative numbers are stored in two's compliment form. The second byte (stored positive) is the resolution within a frame: typical values may be 4 (MIDI Time Code resolution), 8, 10, 80 (bit resolution), or 100. This stream allows exact specifications of time-code-based tracks, but also allows milisecond-based tracks by specifying 25|frames/sec and a resolution of 40 units per frame. If the events in a file are stored with a bit resolution of thirty-framel time code, the division word would be E250 hex.

FORMATS 0, 1, AND 2

A Format 0 file has a header chunk followed by one **track chunk**. It is the most interchangable representation of data. It is very useful for a simple single-track player in a program which needs to make synthesizers make sounds, but which is primarily concerened with something else such as mixers or sound effect boxes. It is very desirable to be able to produce such a format, even if your program is track-based, in order to work with these simple programs. On the other hand, perhaps someone will write a format conversion from format 1 to format 0 which might be so easy to use in some setting that it would save you the trouble of putting it into your program.

A Format 1 or 2 file has a **header chunk** followed by one or more track chunks. programs which support several simultanious tracks should be able to save and read data in format 1, a vertically one-dementional form, that is, as a collection of tracks. Programs which support several independant patterns should be able to save and read data in format 2, a horizontally one-dementional form. Providing these minimum capabilities will ensure maximum interchangability.

In a MIDI system with a computer and a **SMPTE** synchronizer which uses Song Pointer and Timing Clock, tempo maps (which describe the tempo throughout the track, and may also include time signature information, so that the bar number may be derived) are generally created on the computer. To use them with the synchronizer, it is necessary to transfer them from the computer. To make it easy for the synchronizer to extract this data from a MIDI File, tempo information should always be stored in the first **MTrk chunk**. For a format 0 file, the tempo will be scattered through the track and the tempo map reader should ignore the intervening events; for a format 1 file, the tempo map must be stored as the first track. It is polite to a tempo map reader to offerr your user the ability to make a format 0 file with just the tempo, unless you can use format 1.

All MIDI Files should specify tempo and **time signature**. If they don't, the time signature is assumed to be 4/4, and the tempo 120 beats per minute. In format 0, these **meta-events** should occur at least at the beginning of the single multi-channel track. In format 1, these meta-events should be contained i| the first track. In format 2, each of the temporally independant patterns should contain at least initial time signature and tempo information.

We may decide to define other format IDs to support other structures. A program encountering an unknown format ID may still read other **MTrk chunks** it finds from the file, as format 1 or 2, if its user can make sense of them and arrange them into some other structure if appropriate. Also, more parameters may be added to the MThd chunk in the future: it is important to read and honor the length, even if it is longer than 6.

TRACK CHUNKS

The track chunks (type **MTrk**) are where actual song data is stored. Each track chunk is simply a stream of MIDI events (and non-MIDI events), preceded by delta-time values. The format for Track Chunks (described below) is exactly the same for all three formats (0, 1, and 2: see "Header Chunk" above) of MIDI Files.

Here is the syntax of an MTrk chunk (the + means "one or more": at least one MTrk event must be present):

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

The syntax of an MTrk event is very simple:

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

<delta-time> is stored as a **variable-length quantity**. It represents the amount of time before the following event. If the first event in a track occurs at the very beginning of a track, or if two events occur simultaneously, a delta-time of zero is used. Delta-times are always present. (Not storing delta-times of 0 requires at least two bytes for any other value, and most delta-times aren't zero.) Delta-time is in some fraction of a beat (or a second, for recording a track with **SMPTE** times), as specified in the **header chunk**.

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

<MIDI event> is any MIDI channel message. **Running status** is used: status bytes of MIDI channel messages may be omitted if the preceding event is a MIDI channel message with the same status. The first event in each MTrk chunk must specify status. Delta-time is not considered an event itself: it is an integral part of the syntax for an MTrk event. Notice that **running status** occurs across delta-times.

<sysex event> is used to specify a MIDI system exclusive message, either as one unit or in packets, or as an "escape" to specify any arbitrary bytes to be transmitted. A normal complete system exclusive message is stored in a MIDI File in this way:

F0 <length> <bytes to be transmitted after F0>

The length is stored as a variable-length quantity. It specifies the number of bytes which follow it, not including the F0 or the length itself. For instance, the transmitted message F0 43 12 00 07 F7 would be stored in a MIDI File as F0 05 43 12 00 07 F7. It is required to include the F7 at the end so that the reader of the MIDI File knows that it has read the entire message.

Another form of sysex event is provided which does not imply that an F0 should be transmitted. This may be used as an "escape" to provide for the transmission of things which would not otherwise be legal, including system realtime messages, song pointer or select, MIDI Time Code, etc. This uses the F7 code:

F7 <length> <all bytes to be transmitted>

Unfortunately, some synthesizer manufacturers specify that their system exclusive messages are to be transmitted as little packets. Each packet is only part of an entire syntactical system exclusive message, but the times they are transmitted are important. Examples of this are the bytes sent in a CZ patch dump, or the FB-01's "system exclusive

mode" in which microtonal data can be transmitted. The F0 and F7 sysex events may be used together to break up syntactically complete system exclusive messages into timed packets.

An F0 sysex event is used for the first packet in a series -- it is a message in which the F0 should be transmitted. An F7 sysex event is used for the remainder of the packets, which do not begin with F0. (Of course, the F7 is not considered part of the system exclusive message).

A syntactic system exclusive message must always end with an F7, even if the real-life device didn't send one, so that you know when you've reached the end of an entire sysex message without looking ahead to the next event in the MIDI File. If it's stored in one compllete F0 sysex event, the last byte must be an F7. There also must not be any transmittable MIDI events in between the packets of a multi-packet system exclusive message. This principle is illustrated in the paragraph below.

Here is a MIDI File of a multi-packet system exclusive message: suppose the bytes F0 43 12 00 were to be sent, followed by a 200-tick delay, followed by the bytes 43 12 00 43 12 00, followed by a 100-tick delay, followed by the bytes 43 12 00 F7, this would be in the MIDI File:

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

When reading a MIDI File, and an F7 sysex event is encountered without a preceding F0 sysex event to start a multi-packet system exclusive message sequence, it should be presumed that the F7 event is being used as an "escape". In this case, it is not necessary that it end with an F7, unless it is desired that the F7 be transmitted.

<**meta-event**> specifies non-MIDI information useful to this format or to sequencers, with this syntax:

FF <type> <length> <bytes>

All meta-events begin with FF, then have an event type byte (which is always less than 128), and then have the length of the data stored as a variable-length quantity, and then the data itself. If there is no data, the length is 0. As with chunks, future meta-events may be designed which may not be known to existing programs, so programs must properly ignore meta-events which they do not recognize, and indeed should expect to see them. Programs must never ignore the length of a meta-event which they do not recognize, and they shouldn't be surprized if it's bigger than expected. If so, they must ignore everything past what they know about. However, they must not add anything of their own to the end of the meta-event.

Sysex events and **meta events** cancel any **running status** which was in effect. **Running status** does not apply to and may not be used for these messages.

See also:
McQueer on messages

3 - Meta-Events

A few meta-events are defined herin. It is not required for every program to support every meta-event.

In the syntax descriptions for each of the meta-events a set of conventions is used to describe parameters of the events. The FF which begins each event, the type of each event, and the lengths of events which do not have a variable amount of data are given directly in hexadecimal. A notation such as dd or se, which consists of two lower-case letters, mnemonically represents an 8-bit value. Four identical lower-case letters such as wwww mnemonically refer to a 16-bit value, stored most-significant-byte first. Six identical lower-case letters such as tttttt refer to a 24-bit value, stored most-significant-byte first. The notation len refers to the length portion of the meta-event syntax, that is, a number, stored as a variable- length quantity, which specifies how many bytes (possibly text) data were just specified by the length.

In general, meta-events in a track which occur at the same time may occur in any order. If a copyright event is used, it should be placed as early as possible in the file, so it will be noticed easily. Sequence Number and Sequence/Track Name events, if present, must appear at time 0. An end-of- track event must occur as the last event in the track.

Meta-events initially defined include:

`FF 00 02` **Sequence Number**

This optional event, which must occur at the beginning of a track, before any nonzero delta-times, and before any transmittable MIDI events, specifies the number of a sequence. In a format 2 MIDI File, it is used to identify each "pattern" so that a "song" sequence using the Cue message to refer to the patterns. If the ID numbers are omitted, the sequences' lacations in order in the file are used as defaults. In a format 0 or 1 MIDI File, which only contain one sequence, this number should be contained in the first (or only) track. If transfer of several multitrack sequences is required, this must be done as a group of format 1 files, each with a different sequence number.

Meta-event types 01 through 0F are reserved for various types of **text events**, each of which meets the specification of text events but is used for a different purpose.

`FF 20 01 cc` **MIDI Channel Prefix**

The MIDI **channel** (0-15) contained in this event may be used to associate a MIDI channel with all events which follow, including System exclusive and meta-events. This channel is "effective" until the next normal MIDI event (which contains a channel) or the next MIDI Channel Prefix meta-event. If MIDI channels refer to "tracks", this message may into a format 0 file, keeping their non-MIDI data associated with a track. This capability is also present in Yamaha's ESEQ file format.

`FF 2F 00` **End of Track**

This event is **not optional**. It is included so that an exact ending point may be specified for the track, so that an exect length, which is necessary for tracks which are looped or concatenated.

`FF 51 03 tttttt` **Set Tempo**

`FF 54 05 hr mn se fr ff` **SMPTE Offset**

This event, if present, designates the **SMPTE** time at which the track chunk is supposed to start. It should be present at the beginning of the track, that is, before any nonzero delta-times, and before any transmittable MIDI events. the hour must be encoded with the **SMPTE** format, just as it is in MIDI Time Code. In a format 1 file, the **SMPTE** Offset must be stored with the tempo map, and has no meaning in any of the other tracks. The ff field contains fractional frames, in 100ths of a frame, even in **SMPTE**-based tracks which specify a different frame subdivision for delta-times.

`FF 58 04 nn dd cc bb` **Time Signature**

`FF 59 02 sf mi` **Key Signature**

```
    sf = -7: 7 flats
    sf = -1: 1 flat
    sf =  0: key of C
    sf =  1: 1 sharp
    sf =  7: 7 sharps

    mi =  0: major key
    mi =  1: minor key
```

`FF 7F len data` **Sequencer Specific Meta-Event**

Special requirements for particular sequencers may use this event type: the first byte or bytes of data is a manufacturer ID (these are one byte, or if the first byte is 00, three bytes). As with MIDI System Exclusive, manufacturers who define something using this meta-event should publish it so that others may be used by a sequencer which elects to use this as its only file format; sequencers with their established feature-specific formats should probably stick to the standard features when using this format.

Text Meta-Events

FF 01 len text **Text Event**

Any amount of text describing anything. It is a good idea to put a text event right at the beginning of a track, with the name of the track, a description of its intended orchestration, and any other information which the user wants to put there. Text events may also occur at other times in a track, to be used as lyrics, or descriptions of cue points. The text in this event should be printable ASCII characters for maximum interchange. However, other characters codes using the high-order bit may be used for interchange of files between different programs on the same computer which supports an extended character set. Programs on a computer which does not support non-ASCII characters should ignore those characters.

FF 02 len text **Copyright Notice**

Contains a copyright notice as printable ASCII text. The notice should contain the characters (C), the year of the copyright, and the owner of the copyright. If several pieces of music are in the same MIDI File, all of the copyright notices should be placed together in this event so that it will be at the beginning of the file. This event should be the first event in the **track_chunk**, at time 0.

FF 03 len text **Sequence/Track Name**

If in a format 0 track, or the first track in a format 1 file, the name of the sequence. Otherwise, the name of the track.

FF 04 len text **Instrument Name**

A description of the type of instrumentation to be used in that track. May be used with the MIDI Prefix meta-event to specify which MIDI channel the description applies to, or the channel may be specified as text in the event itself.

FF 05 len text **Lyric**

A lyric to be sung. Generally, each syllable will be a seperate lyric event which begins at the event's time.

FF 06 len text **Marker**

Normally in a format 0 track, or the first track in a format 1 file. The name of that point in the sequence, such as a rehersal letter or section name ("First Verse", etc.)

FF 07 len text **Cue Point**

A description of something happening on a film or video screen or stage at that point in the musical score ("Car crashes into house", "curtain opens", "she slaps his face", etc.)

Set Tempo meta-event

FF 51 03 tttttt **Set Tempo** (in microseconds per MIDI quarter-note)

This event indicates a tempo change. Another way of putting "microseconds per quarter-note" is "24ths of a microsecond per MIDI clock". Representing tempos as time per beat instead of beat per time allows absolutly exact long-term synchronization with a time-based sync protocol such as **SMPTE** time code or MIDI time code. This amount of accuracy provided by this tempo resolution allows a four-minute piece at 120 beats per minute to be accurate within 500 usec at the end of the piece. Ideally, these events should only occur where MIDI clocks would be located -- this convention is intended to guarantee, or at least increase the likelihood, of compatibility with other synchronization devices so that a time signature/tempo map stored in this format may easily be transfered to another device.

["Six identical lower-case letters such as **tttttt** refer to a 24-bit value, stored most-significant-byte first."]

[So 4/4 time, 120 beats per minute would be: **FF 51 03 07 A1 20** which would translate to 48 MIDI clocks per second, 500,000 microseconds per quarter note, 2 quarter notes per second]

Time Signature meta-event

`FF 58 04 nn dd cc bb` **Time Signature**

The time signature is expressed as four numbers. **nn and dd** represent the numerator and denominator of the time signature as it would be notated. The denominator is a neqative power of two: 2 represents a quarter-note, 3 represents an eighth-note, etc. The **cc** parameter expresses the number of MIDI clocks in a metronome click. The **bb** parameter expresses the number of notated 32nd-notes in a MIDI quarter-note (24 MIDI clocks). This was added because there are already multiple programs which allow a user to specify that what MIDI thinks of as a quarter-note (24 clocks) is to be notated as, or related to in terms of, something else.

Therefore, the complete event for 6/8 time, where the metronome clicks every three eighth-notes, but there are 24 clocks per quarter-note, 72 to the bar, would be (in hex):

FF 58 04 06 03 24 08

That is, 6/8 time (8 is 2 to the 3rd power, so this is 06 03), 36 MIDI clocks per dotted-quarter (24 hex!), and eight notated 32nd-notes per quarter-note.

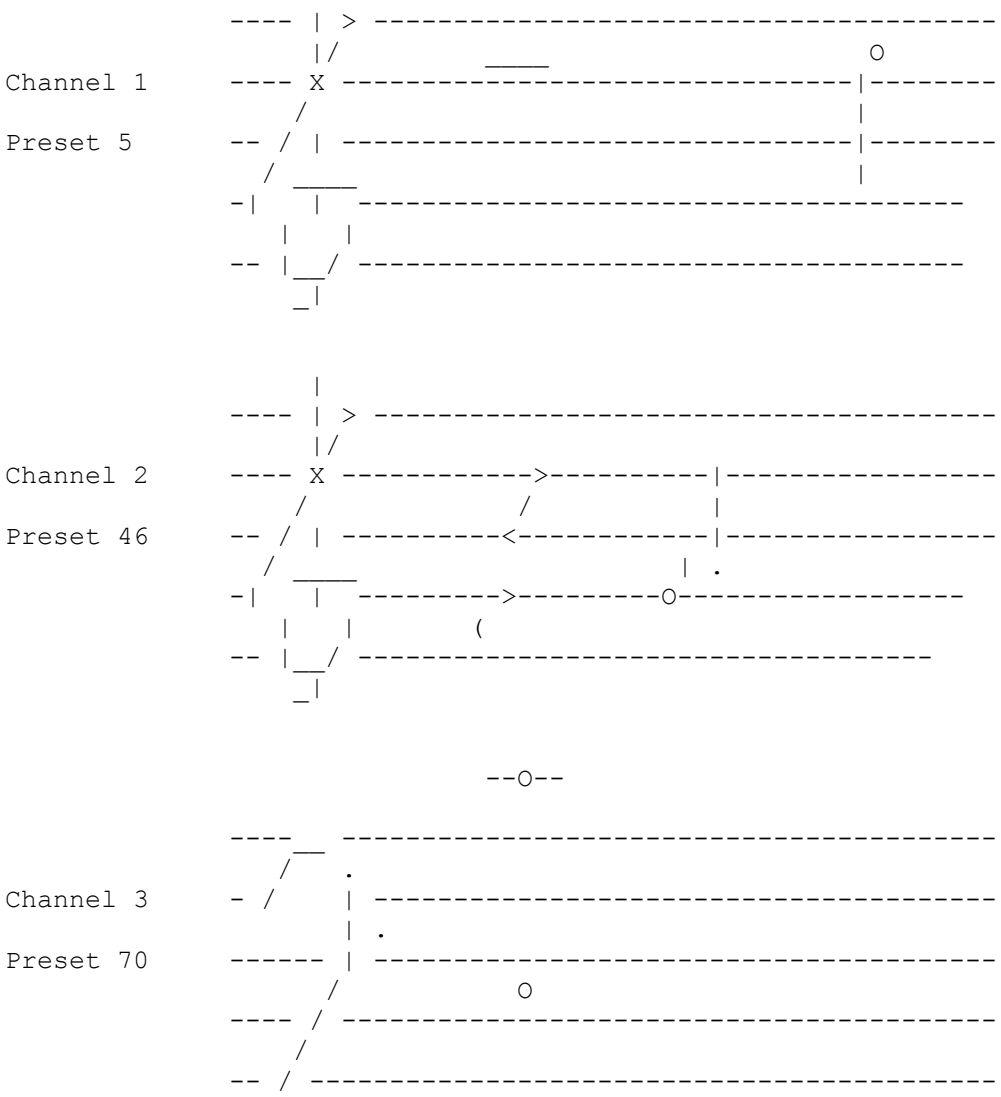
[So 4/4 time, 120 beats per second would be: **FF 58 04 04 02 18 08**]

4 - Program Fragments and Example MIDI Files

Here are some of the routines to read and write variable-length numbers in MIDI Files. These routines are in C, and use getc and putc, which read and write single 8-bit characters from/to the files infile and outfile.

```
WriteVarLen (value)
register long value;
(
   register long buffer;

   buffer = value & 0x7f;
   while ((value >>= 7) > 0)
   (
      buffer <<= 8;
      buffer |= 0x80;
      buffer += (value & 0x7f);
   )

   while (TRUE)
   (
      putc(buffer,outfile);
      if (buffer & 0x80)
         buffer >>= 8;
      else
         break;
   )
)

doubleword ReadVarLen ()
(
   register doubleword value;
   register byte c;

   if ((value = getc(infile)) & 0x80)
   (
      value &= 0x7f;
      do
      (
         value = (value << 7) + ((c = getc(infile))) & 0x7f);
      ) while (c & 0x80);
   )
   return (value);
)
```

As an example, MIDI Files for the following excerpt are shown below. First, a format 0 file is shown, with all information intermingled; then, a format 1 file is shown with all data seperated into four tracks: one for tempo and time signature, and three for the notes. A resolution of 96 "ticks" per quarter note is used. A time signature of 4/4 and a tempo of 120, though implied, are explicitly stated.



The contents of the MIDI stream represented by this example are broken down here:

Delta-Time (decimal)	Event-Code (hex)	Other Bytes (decimal)	Comment
0	FF 58	04 04 02 24 08	4 bytes; 4/4 time; 24 MIDI clocks/click, 8 32nd notes/ 24 MIDI clocks
0	FF 51	03 500000	3 bytes: 500,000 usec/ quarter note
0	C0	5	Ch.1 Program Change 5
0	C1	46	Ch.2 Program Change 46
0	C2	70	Ch.3 Program Change 70
0	92	48 96	Ch.3 Note On C2, forte
0	92	60 96	Ch.3 Note On C3, forte
96	91	67 64	Ch.2 Note On G3, mezzo-forte
96	90	76 32	Ch.1 Note On E4, piano
192	82	48 64	Ch.3 Note Off C2, standard
0	82	60 64	Ch.3 Note Off C3, standard

```
0           81          67 64          Ch.2 Note Off G3, standard
0           80          76 64          Ch.1 Note Off E4, standard
0           FF 2F       00             Track End
```

The entire format 0 MIDI file contents in hex follow. First, the **header chunk**:

```
            40 54 68 64             MThd
            00 00 00 06             chunk length
            00 00                   format 0
            00 01                   one track
            00 60                   96 per quarter-note
```

Then the **track_chunk**. Its header followed by the events (notice the running status is used in places):

```
            4D 54 72 6B             MTrk
            00 00 00 3B             chunk length (59)

Delta-Time  Event                   Comments
----------  ----------------------  ------------------------------
00          FF 58 04 04 02 18 08    time signature
00          FF 51 03 07 A1 20       tempo
00          C0 05
00          C1 2E
00          C2 46
00          92 30 60
00          3C 60                   running status
60          91 43 40
60          90 4C 20
81 40       82 30 40                two-byte delta-time
00          3C 40                   running status
00          81 43 40
00          80 4C 40
00          FF 2F 00                end of track
```

A format 1 representation of the file is slightly different. Its **header chunk**:

```
            4D 54 68 64             MThd
            00 00 00 06             chunk length
            00 01                   format 1
            00 04                   four tracks
            00 60                   96 per quarter note
```

First, the **track_chunk** for the time signature/tempo track. Its header, followed by the events:

```
            4D 54 72 6B             MTrk
            00 00 00 14             chunk length (20)

Delta-Time  Event                   Comments
----------  ----------------------  ------------------------------
00          FF 58 04 04 02 18 08    time signature
00          FF 51 03 07 A1 20       tempo
83 00       FF 2F 00                end of track
```

Then, the **track_chunk** for the first music track. The MIDI convention for note on/off **running status** is used in this example:

```
              4D 54 72 6B             MTrk
              00 00 00 10             chunk length (16)

Delta-Time    Event                   Comments
----------    ----------------------  --------------------------------
00            C0 05
81 40         90 4C 20
81 40         4C 00                   Running status: note on, vel=0
00            FF 2F 00
```

Then, the **track_chunk** for the second music track:

```
              4D 54 72 6B             MTrk
              00 00 00 0F             chunk length (15)

Delta-Time    Event                   Comments
----------    ----------------------  --------------------------------
00            C1 2E
60            91 43 40
82 20         43 00                   running status
00            FF 2F 00                end of track
```

Then, the **track_chunk** for the third music track:

```
              4D 54 72 6B             MTrk
              00 00 00 15             chunk length (21)

Delta-Time    Event                   Comments
----------    ----------------------  --------------------------------
00            C2 46
00            92 30 60
00            3C 60                   running status
83 00         30 00                   two-byte delta-time, running status
00            3C 00                   running status
00            FF 2F 00                end of track
```

The USENET MIDI Primer

The official MIDI 1.0 specification is available from:
IMA
the International MIDI Association
11857 Hartsook St.
North Holllywoood, CA 91607
(818) 505-8964

The complete MIDI spec as developed by the Japanese manufacturers and adopted by the "World" at the Summer '85 NAMM show is available to IMA members ($40/yr) foor $30 non-members $35.

The sketchy hardware and byte definitions are free with membership.

The following is an expanded MIDI definition (sort of in-between IMA MIDI 1.0 and the new $35 booklet) entered into the public domain on USENET net.musi.synth by an altruistic musically inclined engineer:

Bob McQueer
22 Bcy, 3151

All rites reversed. Reprint what you like.

The USENET MIDI Primer
Bob McQueer

PURPOSE

It seems as though many people in the USENET community have an interest in the Musical Instrument Digital Interface (MIDI), but for one reason or another have only obtained word of mouth or fragmentary descriptions of the specification. Basic questions such as "what's the baud rate?", "is it EIA?" and the like seem to keep surfacing in about half a dozen newsgroups. This article is an attempt to provide the basic data to the readers of the net.

REFERENCE

The major written reference for this article is version 1.0 of the MIDI specification, published by the International MIDI Association, copyright 1983. There exists an expanded document. This document, which I have not seen, is simply an expansion of the 1.0 spec. to contain more explanatory material, and fill in some areas of hazy explanation. There are no radical departures from 1.0 in it. I have also heard of a "2.0" spec., but the IMA claims no such animal exists. In any event, backwards compatibility with the information I am presenting here should be maintained.

CONVENTIONS

I will give constants in C syntax, ie. 0x for hexadecimal. If I refer to bits by number, I number them starting with 0 for the low order (1's place) bit. The following notation:

>>

```
text
```

<<

will be used to delimit commentary which is not part of the "bare- bones" specification. A sentence or paragraph marked with a question mark in column 1 is a point I would kind of like to hear something about myself.

OK, let's give it a shot.

PHYSICAL CONNECTOR SPECIFICATION
ELECTRICAL SPECIFICATION
DATA FORMAT
VOICE MESSAGES
MODE MESSAGES
SYSTEM MESSAGES
REAL TIME MESSAGES
AND NOW, CLIMBING TO THE PULPIT

Primer: PHYSICAL CONNECTOR SPECIFICATION

The standard connectors used for MIDI are 5 pin DIN. Separate sockets are used for input and output, clearly marked on a given device. The spec. gives 50 feet as the maximum cable length. Cables are to be shielded twisted pair, with the shield connecting pin 2 at both ends. The pair is pins 4 and 5, pins 1 and 3 being unconnected:

A device may also be equipped with a "MIDI-THRU" socket which is used to pass the input of one device directly to output.

>>

I think this arrangement shows some of the original conception of MIDI more as a way of allowing keyboardists to control multiple boxes than an instrument to computer interface. The "daisy-chain" arrangement probably has advantages for a performing musician who wants to play "stacked" synthesizers for a desired sound, and has to be able to set things up on the road.

<<

ELECTRICAL SPECIFICATION

Primer: ELECTRICAL SPECIFICATION

Asynchronous serial interface. The baud rate is 31.25 Kbaud (+/- 1%). There are 8 data bits, with 1 start bit and 1 stop bit, for 320 microseconds per serial byte.

MIDI is current loop, 5 mA. Logic 0 is current ON. The specification states that input is to be opto-isolated, and points out that Sharp PC-900 and HP 6N138 optoisolators are satisfactory devices. Rise and fall time for the optoisolator should be less than 2 microseconds.

The specification shows a little circuit diagram for the connections to a UART. I am not going to reproduce it here. There's not much to it - I think the important thing it shows is +5 volt connection to pi 4 of the MIDI out with pin 5 going to the UART, through 220 ohm load resisors. It also shows that you're supposed to connect to the "in" side of the UART through an optoisolator, and to the MIDI-THRU on the UART side of the isolator.

>>

```
        I'm not much of a hardware person, and don't really know what
        I'm talking about in paragraphs like the three above.  I DO
        recognize that this is a "non-standard" specification, which
        won't work over serial ports intended for anything else.  People
        who do know about such things seem to either have giggling
        or gagging fits when they see it, depending on their dispos-
        itions, saying things like "I haven't seen current loop since
        the days of the old teletypes".  I also know the fast 31.25
        Kbaud pushes the edge for clocking commonly available UART's.
```

<<

See M Garvinon the subject.

DATA FORMAT

Primer: DATA FORMAT

For standard MIDI messages, there is a clear concept that one device is a "transmitter" or "master", and the other a "receiver" or "slave". Messages take the form of opcode bytes, followed by data bytes. Opcode bytes are commonly called "status" bytes, so we shall use this term.

>>

```
        very similar to handling a terminal via escape sequences.  There
        aren't ACK's or other handshaking mechanisms in the protocol.
```

<<

Status bytes are marked by bit 7 being 1. All data bytes must contain a 0 in bit 7, and thus lie in the range 0 - 127.

MIDI has a logical channel concept. There are 16 logical **channels**, encoded into bits 0 - 3 of the status bytes of messages for which a channel number is significant. Since bit 7 is taken over for marking the status byte, this leaves 3 opcode bits for message types with a logical channel. 7 of the possible 8 opcodes are used in this fashion, reserving the status bytes containing all 1's in the high nibble for "system" messages which don't have a channel number. The low order nibble in these remaining messages is really further opcode.

>>

```
        If you are interested in receiving MIDI input, look over the
        SYSTEM messages even if you wish to ignore them.  Especially the
         "system exclusive" and "real time" messages.  The real time
        messages may be legally inserted in the middle of other data,
        and you should be aware of them, even though many devices won't
        use them.
```

<<

VOICE MESSAGES
MODE MESSAGES
SYSTEM MESSAGES
REAL TIME MESSAGES

Primer: VOICE MESSAGES

I will cover the message with **channel** numbers first. The opcode determines the number of data bytes for a single message (see "running status byte", below). The specification divides these into "voice" and "mode" messages. The "mode" messages are for control of the logical channels, and the control opcodes are piggybacked onto the data bytes for the "parameter" message. I will go into this after describing the "voice messages". These messages are:

```
status byte   meaning        data bytes

0x80-0x8f     note off       2 - 1 byte pitch, followed by 1 byte velocity
0x90-0x9f     note on        2 - 1 byte pitch, followed by 1 byte velocity
0xa0-0xaf     key pressure   2 - 1 byte pitch, 1 byte pressure (after-touch)
0xb0-0xbf     parameter      2 - 1 byte parameter number, 1 byte setting
0xc0-0xcf     program        1 byte program selected  [patch assignment]
0xd0-0xdf     chan. pressure 1 byte channel pressure (after-touch)
0xe0-0xef     pitch wheel    2 bytes giving a 14 bit value, least
                             significant 7 bits first
```

Many explanations are necessary here:

For all of these messages, a convention called the "**running status** byte" may be used. If the transmitter wishes to send another message of the same type on the same channel, thus the same status byte, the status byte need not be present.

Also, a "note on" message with a velocity of zero is to be synonymous with a "note off". Combined with the previous feature, this is intended to allow long strings of notes to be sent without repeating status bytes.

```
>>
        From what I've seen, the "zero velocity note on" feature is very
        heavily used.  My six-trak sends these, even though it sends
        status bytes on every note anyway.  Roland stuff uses it.
<<
```

The pitch bytes of notes are simply number of half-steps, with **middle C = 60**.

```
>>
        On keyboard synthesizers, this usually simply means which
        physical key corresponds, since the patch selection will
        change the actual pitch range of the keyboard.  Most keyboards
        have one C key which is unmistakably in the middle of the
        keyboard.  This is probably note 60.
<<
```

The velocity bytes for velocity sensing keyboards are supposed to represent a logarithmic scale. "advisable" in the words of the spec. **Non-velocity sensing devices are supposed to send velocity 64**.

The pitch wheel value is an absolute setting, 0 - 0x3FFF. The 1.0 spec. says that the increment is determined by the receiver. 0x2000 is to correspond to a centered pitch wheel (unmodified notes)

```
>>
```

```
        I believe standard scale steps are one of the things discussed
        in expansions.  The six-trak pitch wheel is up/down about a third.
        I believe several makers have used this value, but I may be wrong.

        The "pressure" messages are for keyboards which sense the amount
        of pressure placed on an already depressed key, as opposed to
        velocity, which is how fast it is depressed or released.

?       I'm not really certain of how "channel" pressure works.  Yamaha
        is one maker that uses these messages, I know.
<<
```

Now, about those parameter messages.

Instruments are so fundamentally different in the various controls they have that no attempt was made to define a standard set, like say 9 for "Filter Resonance". Instead, it was simply assumed that these messages allow you to set "controller" dials, whose purposes are left to the given device, except as noted below. The first data bytes correspond to these "controllers" as follows:

data byte

```
0 - 31       continuous controllers 0 - 31, most significant byte
32 - 63      continuous controllers 0 - 31, least significant byte
64 - 95      on / off switches

96 - 121     unspecified, reserved for future.
122 - 127    the "channel mode" messages I alluded to above.  See
             below.
```

The second data byte contains the seven bit setting for the controller. The switches have data byte 0 = OFF, 127 = ON with 1 - 126 undefined. If a controller only needs seven bits of resolution, it is supposed to use the most significant byte. If both are needed, the order is specified as most significant followed by least significant. With a 14 bit controller, it is to be legal to send only the least significant byte if the most significant doesn't need to be changed.

```
>>
        This may of, course, wind up stretched a bit by a given manufacturer.
        The Six-Trak, for instance, uses only single byte values (LEFT
        justified within the 7 bits at that), and recognizes >32 parameters
<<
```

Controller number 1 IS standardized to be the modulation wheel.

```
?       Are there any other standardizations which are being followed by most
        manufacturers?
```

MODE MESSAGES

Primer: MODE MESSAGES

These are messages with status bytes 0xb0 through 0xbf, and leading data bytes 122 - 127. In reality, these data bytes function as further opcode data for a group of messages which control the combination of voices and channels to be accepted by a receiver.

An important point is that there is an implicit "basic" channel over which a given device is to receive these messages. The receiver is to ignore mode messages over any other channels, no matter what mode it might be in. The basic channel for a given device may be fixed or set in some manner outside the scope of the MIDI standard.

The meaning of the values 122 through 127 is as follows:

```
data byte



          second data byte
122       local control     0 = local control off, 127 = on
123       all notes off     0
124       omni mode off     0
125       omni mode on      0
126       monophonic mode   number of monophonic channels, or 0
                            for a number equal to receivers voices
127       polyphonic mode   0

124 - 127 also turn all notes off.
```

Local control refers to whether or not notes played on an instruments keyboard play on the instrument or not. With local control off, the host is still supposed to be able to read input data if desired, as well as sending notes to the instrument. Very much like "local echo" on a terminal, or "half duplex" vs. "full duplex".

The mode setting messages control what channels / how many voices the receiver recognizes. The "basic channel" must be kept in mind. "Omni" refers to the ability to receive voice messages on all channels. "Mono" and "Poly" refer to whether multiple voices are allowed. The rub is that the omni on/off state and the mono/poly state interact with each other. We will go over each of the four possible settings, called "modes" and given numbers in the specification:

```
mode 1 - Omni on / Poly - voice messages received on all channels and
         assigned polyphonically.  Basically, any notes it gets, it
         plays, up to the number of voices it's capable of.

mode 2 - Omni on / Mono - monophonic instrument which will receive
         notes to play in one voice on all channels.

mode 3 - Omni off / Poly - polyphonic instrument which will receive
         voice messages on only the basic channel.

mode 4 - Omni off / Mono - A useful mode, but "mono" is a misnomer.
         To operate in this mode a receiver is supposed to receive
         one voice per channel.  The number channels recognized will be
         given by the second data byte, or the maximum number of possible
         voices if this byte is zero.  The set of channels thus defined
```

```
is a sequential set, starting with the basic channel.
```

The spec. states that a receiver may ignore any mode that it cannot honor, or switch to an alternate - "usually" mode 1. Receivers are supposed to default to mode 1 on power up. It is also stated that power up conditions are supposed to place a receiver in a state where it will only respond to note on / note off messages, requiring a setting of some sort to enable the other message types.

>>

```
I think this shows the desire to "daisy-chain" devices for
performance from a single master again.  We can set a series
of instruments to different basic channels, tie 'em together,
and let them pass through the stuff they're not supposed to
play to someone down the line.

This suffers greatly from lack of acknowledgement concerning
modes and usable channels by a receiver.  You basically have
to know your device, what it can do, and what channels it can
do it on.

I think most makers have used the "system exclusive" message
(see below) to handle channels in a more sophisticated manner,
as well as changing "basic channel" and enabling receipt of
different message types under host control rather than by
adjustment on the device alone.

The "parameters" may also be usurped by a manufacturer for
mode control, since their purposes are undefined.


Another HUGE problem with the "daisy-chain" mental set of MIDI
is that most devices ALWAYS shovel whatever they play to their
MIDI outs, whether they got it from the keyboard or MIDI in.
This means that you have to cope with the instrument echoing
input back at you if you're trying to do an interactive session
with the synthesizer.  There is DRASTIC need for some MIDI flag
which specifically means that only locally generated data is to
go to MIDI out.  From device to device there are ways of coping
with this, none of them good.
```

<<

SYSTEM MESSAGES

Primer: SYSTEM MESSAGES

The status bytes 0xf0 - 0xf7 do not have channel numbers in the lower nibble. These bytes are used as follows:

byte	purpose	data bytes
0xf0	system exclusive	variable length
0xf1	undefined	
0xf2	song position	2 - 14 bit value, least significant byte first
0xf3	song select	1 - song number
0xf4	undefined	
0xf5	undefined	
0xf6	tune request	0
0xf7	EOX (terminator)	0

The status bytes 0xf8 - 0xff are the so-called **"real-time" messages**. I will discuss these after the accumulated notes concerning the first bunch.

Song position / song select are for control of sequencers. The song position is in beats, which are to be interpreted as every 6 MIDI clock pulses. These messages determine what is to be played upon receipt of a "start" real-time message (see below).

The "tune request" is a command to analog synthesizers to tune their oscillators.

The system exclusive message is intended for manufacturers to use to insert any specific messages they want to which apply to their own product. The following data bytes are all to be "data" bytes, that is they are all to be in the range 0 - 127. The system exclusive is to be terminated by the 0xf7 terminator byte. The first data byte is also supposed to be a "manufacturer's id", assigned by a MIDI standards committee. THE TERMINATOR BYTE IS OPTIONAL - a system exclusive may also be "terminated" by the status byte of the next message.

>>

> Yamaha, in particular, caused problems by not sending terminator bytes. As I understand it, the DX-7 sends a system exclusive at something like 80 msec. intervals when it has nothing better to do, just so you know it's still there, I guess. The messages aren't explicitly terminated, so if you want to handle the protocol (esp. in hardware), you should be aware that a DX-7
>
> will leave you in "waiting for EOX" state a lot, and be sending data even when it isn't doing anything. This is all word of mouth, since I've never personally played with a DX-7.

<<

some MIDI ID's:

Sequential Circuits	1	Bon Tempi	0x20	Kawai	0x40
Big Briar	2	S.I.E.L.	0x21	Roland	0x41
Octave / Plateau	3			Korg	0x42
Moog	4	SyntheAxe	0x23	Yamaha	0x43
Passport Designs	5				
Lexicon	6				

PAIA	0x11
Simmons	0x12
Gentle Electric	0x13
Fairlight	0x14

>>

Note the USA / Europe / Japan grouping of codes. Also note that Sequential Circuits snarfed id number 1 - Sequential Circuits was one of the earliest participators in MIDI, some people claim its originator.

Two large makers missing from the original lineup were Casio and Oberheim. I know Oberheim is on the bandwagon now, and Casio also, I believe. Oberheim had their own protocol previous to MIDI, and when MIDI first came out they were reluctant to
? go along with it. I wonder what we'd be looking at if Oberheim had pushed their ideas and made them the standard. From what I understand they thought THEIRS was better, and kind of sulked for a while until the market forced them to go MIDI.

? Nobody seems to care much about these ID numbers. I can only imagine them becoming useful if additions to the standard message set are placed into system exclusives, with the ID byte to let you know what added protocol is being used. Are any groups of manufacturers considering consolidating their efforts in a standard extension set via system exclusives?

<<

REAL TIME MESSAGES

Primer: REAL TIME MESSAGES

This is the final group of status bytes, 0xf8 - 0xff. These bytes are reserved for messages which are called "real-time" messages because they are allowed to be sent ANYPLACE. This includes in between data bytes of other messages. A receiver is supposed to be able to receive and process (or ignore) these messages and resume collection of the remaining data bytes for the message which was in progress. Realtime messages do not affect the "running status byte" which might be in effect.

```
?       Do any devices REALLY insert these things in the middle of
        other messages?
```

All of these messages have no data bytes following (or they could get interrupted themselves, obviously). The messages:

```
0xf8    timing clock
0xf9    undefined
0xfa    start
0xfb    continue
0xfc    stop
0xfd    undefined
0xfe    active sensing
0xff    system reset
```

The timing clock message is to be sent at the rate of 24 clocks per quarter note, and is used to sync. devices, especially drum machines.

Start / continue / stop are for control of sequencers and drum machines. The continue message causes a device to pick up at the next clock mark.

```
>>
        These things are also designed for performance, allowing control
        of sequencers and drum machines from a "master" unit which
        sends the messages down the line when its buttons are pushed.

        I can't tell you much about the trials and tribulations of drum
        machines.  Other folks can, I am sure.
<<
```

The active sensing byte is to be sent every 300 ms. or more often, if it is used. Its purpose is to implement a timeout mechanism for a receiver to revert to a default state. A receiver is to operate normally if it never gets one of these, activating the timeout mechanism from the receipt of the first one.

```
>>
       My impression is that active sensing is largely unused.
<<
```

The system reset initializes to power up conditions. The spec. says that it should be used "sparingly" and in particular not sent automatically on power up.

AND NOW, CLIMBING TO THE PULPIT

Primer: AND NOW, CLIMBING TO THE PULPIT ....

>> - from here on out.

There are many deficiencies with MIDI, but it IS a standard. As such, it will have to be grappled with.

The electrical specification leaves me with only one question - WHY? What was wanted was a serial interface, and a perfectly good RS232 specification was to be had. WHY wasn't it used? The baud rate is too fast to simply convert into something you can feed directly to your serial port via fairly dumb hardware, also. The "standard" baud rate step you would have to use would be 38.4 Kbaud which very few hardware interfaces accept. The other alternative is to buffer messages and send them out a slower baud rate - in fact buffering of characters by some kind of I/O processor is very helpful. Hence units like the MPU-401, which does a lot of other stuff, too of course.

The fast baud rate with MIDI was set for two reasons I believe:

```
        1) to allow daisy-chaining of a few devices with no noticeable
                end to end lag.

        2) to allow chords to be played by just sending all the notes down
                the pipe, the baud rate being fast enough that they will
                sound simultaneous.
```

It doesn't exactly work - I've heard gripes concerning end to end lag on three instrument chains. And consider chords - at two bytes (running status byte being used) per note, there will be a ten character lag between the trailing edges of the first and last notes of a six note chord. That's 3.2 ms., assuming no "dead air" between characters. It's still pretty fast, but on large chords with voices possessing distinctive attack characteristics, you may hear separate note beginnings.

I think MIDI could have used some means of packetizing chords, or having transaction markers. If a "chord" message were specified, you could easily break even on byte count with a few notes, given that we assume all notes of a chord at the same velocity. Transaction markers might be useful in any case, although I don't know if it would be worth taking over the remaining system message space for them. I would say yes. I would see having "start" and "end" transaction bytes. On receipt of a "start" a receiver buffers up but does not act on messages until receipt of the "end" byte. You could then do chords by sending the notes ahead of time, and precisely timing the "end" marker. Of course, the job of the hardware in the receiver has been complicated considerably.

The protocol is VERY keyboard oriented - take a look at the use of TWO of the opcodes in the limited opcode space for "pressure" messages, and the inability to specify semitones or glissando effects except through the pitch wheel (which took up yet ANOTHER of the opcodes). All keyboards I know of modify ALL playing notes when they receive pitch wheel data. Also, you have to use a continuous stream of pitch wheel messages to effect a slide, the pitch wheel step isn't standardized, and on a slide of a large number of tones you will overrun the range of the wheel.

```
?       Some of these problems would be addressed by a device which allowed
        its pitch wheel to have selective control - say modifying only
        the notes playing on the channel the pitch wheel message is
        received in, for instance.  The thing for a guitar synthesizer
        to do, then, would be to use mode 4, one channel per string, and
```

> bends would only affect the one note. You could play a chord
> on a voice with a lot of release, then bend a note and not have
> the entire still sounding chord bend. Any such devices?

I think some of the deficiencies in MIDI might be addressed by different communities of interest developing a standard set of system exclusives which answer the problem. One perfect area for this, I think, is a standard set for representation of "non- keyboard / drum machine" instruments which have continuous pitch capabilities. Like a pedal steel, for instance. Or non-western intervals. Like a sitar.

There is a crying need to do SOMETHING about the "loopback" problem. I would even vote for usurping a few more bytes in the mode messages to allow you to TURN OFF input echo by the receiver. With the local control message, you could then at least deal with something that would act precisely like a half or full duplex terminal. .More..Several patchwork solutions exist to this problem, but there OUGHT to be a standard way of doing it within the protocol. Another thought is to allow data bytes of other than 0 or 127 to control echo on the existing local control message.

The lack of acknowledgement is a problem. Another candidate for a standard system exclusive set would be a series of messages for mode setting with acknowledgement. This set could then also take care of the loopback problem.

The complete lack of ability to specify standardized waveforms is probably another source of intense disappointment to many readers. Trouble is, the standard lingo used by the synthesizer industry and most working musicians is something which hails back to the first days of synthesizer design, deals with envelope generators and filters and VCO / LFO hardware parameters, and is very damn difficult to relate to Fourier series expressing the harmonic content or any other abstractions some people interested in doing computer composition would like. The parameter set used by the average synthesizer manufacturer isn't anyplace close to orthogonal in any sense, and is bound to vary wildly n comparison to anybody elses. Ther are essentially no abstractions made by most of the industry from underlyin hardwre prameers. hat tandarizaton exits refects only the similarity in hardware. This is one quagmire that we have a long way to go to get out of, I think. It might be possible, eventually, to come up with translation tables describing the best way to approximate a desired sound on a given device in terms of its parameter set, but the difficulties are enormous. MIDI has chosen to punt on this one, foks.

Well that's abot it. Good luck with talking to your synthesizer.

Bob McQueer
22 Bcy, 3151

MIDI INTERFACING - DESIGNING A SOFTWARE-BASED MUSIC RECORDER

M.Garvin
Xymetric Productions
211 W.Broadway
New York,NY 10013

INTRO

Imagine designing products for a changing microcomputer market without hardware and software standards. Standard architectures such as IBM PC's and Apples have led to a proliferation of new microcomputer products, and the development of MIDI (Musical Instrument Digital Interface) has resulted in a similar revolution in the music industry. By providing common ground for communication, MIDI allows software engineers and musicians to access a wide range of synthesizers, computers and musical controllers.

MIDI has become virtually uncontested as a means to link synthesizers and computer equipment, and is now finding acceptance in many nearby industries such as lighting control, film editing, and automated audio mixing. By supporting real-time access to so many devices, MIDI has opened new dimensions for recording, composition and live performance. Now musicians can generate orchestral scores with banks of rack-mounted synthesizers, coordinate sound and visual effects, and even transmit stored musical data over phone lines.

In this article I will outline several applications of MIDI and I will suggest several ways to get started writing your own MIDI software. Some of the current products and manufacturers are listed, but these listings are by no means complete. They are given as a starting point for obtaining additional information.

Garvin: OVERVIEW

The MIDI specification has remained reasonably intact since it was first proposed by synthesizer manufacturer Sequential Circuits in 1982. It is sufficiently specialized to handle most direct musical communication, yet its generality has allowed it to adapt to applications which were not foreseen when it was originally drafted.

MIDI entails both a hardware and a software specification: the hardware consists of a relatively fast optically isolated serial loop with separate cables for send and receive; the software provides detailed methods for transmitting note-control data and looser specifications for handling interaction between products from different manufacturers. MIDI works in much the same way as modem protocol running on RS-232 lines, but it is optimized for musical data. Modern MIDI record/playback systems could be regarded as a type of multiprocessor network, since an intelligent master (keyboard or computer) controls a series of devices, each with its own onboard intelligence.

MIDI commands include NOTE-ON and NOTE-OFF, response MODE for filtering received signals, REAL-TIME messages for co-ordinating events, and SYSTEM COMMON and EXCLUSIVE commands for setting up songs or addressing a particular brand of synthesizer. This simple instruction set allows enough flexibility to accomodate most synthesizer architectures, while providing much-needed universal music commands. As more manufacturers have realized the advantages of communicating with a wide array of musical equipment, MIDI's popularity has mushroomed. There are very few synthesis instruments sold today without MIDI interfaces.

All MIDI documentation is now handled by the International MIDI User's Group (IMUG) -- contact them for specific information or the complete 'MIDI spec'.

PRODUCTS

Garvin: PRODUCTS

Fortunately, the high level of competition among musical instrument manufacturers has been offset by specialization: small companies can offer transmit-only devices such as high quality keyboards with no sound output, or receive-only devices such as sound generators which respond only to MIDI input. Undoubtedly, the broadest new field is that of software-based controllers. These usually connect between a keyboard (or other source) and a sound generator, where they monitor and control MIDI communication. Some of the newer MIDI-based products include guitar, voice, and even xylophone-to-MIDI converters (Roland, Fairlight), software-hardware retrofits for playing digitized notes from personal computers (Hybrid Arts), MIDI- controlled reverb and echo units (Lexicon, Korg), and MIDI-controlled audio mixing consoles (AKAI).

SYNTHESIS METHODS

Garvin: SYNTHESIS METHODS

Early synthesizers -- used for scoring so many old science-fiction movies -- were assembled from a number of modules which were interconnected manually by patchcords. These and other voltage- controlled instruments have now attained a certain vintage status. New instruments use computerized signal routing, and modern sound generating techniques range from additive synthesis (KAWAI), FM (Yamaha), phase-distortion synthesis (Casio), to actual digital recording, or 'sampling', of natural sounds (Sequential Circuits, EMU, Kurzweil). Additive synthesis uses the addition of a number of sine-wave components to produce an output waveform. Theoretically, any waveform can be broken down into sine-wave components by using a process known as Fourier analysis. It follows then, that any waveform may be recreated by adding these same sine-wave components. In actual application, the process is not so simple; the human ear quickly becomes bored with the 'static', or unchanging waveform which is created. It is this static characteristic of early synthesizers which contributed to the stereotyped monotonous or bland sound.

The waveforms generated by traditional acoustic instruments change as notes are being played, so recreating the natural timbres of these instruments requires real-time control over the amplitudes, or 'envelopes' of the sine-wave components. In early synthesizers, this was approximated by the use of an envelope generator which cycled through Attack-Decay-Sustain-Release ('ADSR') states in response to key-down and key-up events. Patching the ADSR generator into voltage-controlled filters and voltage-controlled amplifiers provided a primitive level of control over the harmonic structure of the waveform. Newer machines sometimes use a separate programmable envelope for each sine-wave component of the waveform. With sufficient control of the amplitudes, any conceivable sound can be recreated without having to use digitized wave samples. This is the objective of the 're-synthesis' or 'adaptive synthesis' machines, such the Roland digital piano.

FM synthesis uses a limited number of sine-wave oscillators with individual envelopes, so in this respect it bears some resemblance to re-synthesis methods. One significant departure is in the way the oscillators are configured: different 'algorithms' can be chosen to allow oscillators to intermodulate, creating rich, and sometimes non- harmonic frequencies. The resulting output can range from clangorous-sounding bells to human-sounding voices, but FM machines are relatively unpredictable and difficult to program.

Phase-distortion synthesis involves scanning a simple (usually sine) wave, and varying the scan rate as the wave is being replayed. In other words, the leading edge of the sine wave can be scanned rapidly so it appears to be a nearly vertical edge; the trailing edge can be scanned more slowly so it has a tapered slope. The resulting 'saw- tooth' waveform is much richer in harmonics than the flute-like sine wave. Dynamic variation of the scan rate can change the shape and timbre of the waveform as a note is being played.

Some of these methods of sound generation may seem to make the use of waveform sampling unnecessary, but in fact, samplers can usually do much more than recreate natural sounds. For example, precise (up to 16-bit) digitizations of orchestras, drums, waterfalls, or human voice can be altered and played back at any pitch. The elite of the sampling synthesizers (Fairlight, New England Digital) can cost hundreds of thousands of dollars, but they eliminate the need for a lot of expensive equipment in a recording studio. New England Digital even advertises a tapeless studio which records audio tracks directly to a high-capacity hard disk. Optical Media, Inc. offers pre-recorded sound libraries on 'CD roms'. Such systems are becoming more affordable as the cost of mass storage continues to drop.

PATCH EDITORS and LIBRARIANS

Garvin: PATCH EDITORS and LIBRARIANS

These diverse methods of sound synthesis have introduced a new class of problems: a few potentiometers on a front panel are no longer sufficient to program (that's right, 'program') a synthesizer. One manufacturer claims that if all of their front panel functions were to be made available at once, their synthesizer would be seventeen feet long. Instead of overlaying the limited set of front panel controls with multiple modes, synthesizer front panel functions can usually be accessed by sending a special set of MIDI 'system specific' commands from a computer outfitted with MIDI ports.

Synthesizer programming has become an art in the same sense as actual performance of music, and album covers frequently give credit to sound programmers, even if they do not perform on the album. The sound programs, which can make-or-break the sound of a synthesizer, are known as 'patches'. Good programmers frequently make a living solely by selling their patches, either in the form of instruction sheets which simply tell how to recreate the original sounds, or in a downloadable binary form on floppy disks. Disk-based systems require the use of a 'patch librarian' program to organize and access patch files. Most patch librarians allow two-way communication between the synthesizer and computer, so patches can be sent to a synthesizer, modified with the use of the synthesizer's controls, and sent back to be archived or backed up on disk. Some sophisticated librarians (Voyetra Technologies) can handle several different types of synthesizers, or even download rhythm patterns to electronic drum machines.

'Patch editor' programs differ from patch librarians in that they can actually alter the sound of a patch. Just as the term implies, MIDI system-specific messages usually ARE specific to a certain brand and type of synthesizer, so most patch editors are designed to work with only one or two types of machines. The system-specific messages which are sent by these editors are prefixed by an address code which tells other types of machines to ignore the data which follows, so patches can be sent to one device in a MIDI network without problems resulting from other machines misinterpreting the data. System specific codes are usually published along with other technical data in a synthesizer's instruction manual.

Some vendors, such as Bacchus Software, specialize in patch editor programs. Bacchus's IBM PC-based editor is a Side-Kick style memory- resident program which pops up over another running music program. Its main function is editing and archiving patches for the popular but difficult-to-program Yamaha DX-7 synthesizer. It is usually in the best interests of manufacturers to either write their own micro- based editors or to hire third-party software writers to support their products. For example, DigiDesign's waveform-editing programs allow data to be downloaded from sampling synthesizers, displayed, altered, and sent back to a sampler to be played. They support several brands of synthesizers, but the manufacturers value their support, since they make the samplers much more accessible and marketable. Some manufacturers even feature DigiDesign's software in their own ads and exhibits.

SEQUENCERS

Garvin: SEQUENCERS

One of the many advantages afforded by MIDI is the ability to intercept and record musical events. The recorded data can be manipulated in ways that were impossible using standard audio tape recorders. Transposition (pitch-shift), copying, and rearranging can be accomplished with software alone, and errors made during the recording process can be corrected without having to do 'retakes'. Compositions can even be replayed while changing synthesizer patches, so a part which was originally written for a cello-type voicing could be tried out with a piano sound.

MIDI-based recorders or composition programs are sometimes known as sequencers (a carry-over from old analog instruments). In the modern context, a sequencer might be visualized much like an audio tape machine. Concepts such as multitracking, fast-forward and rewind can be translated to software to ease the transition from conventional tape-oriented studios. New concepts such as 'quantization' (automatic timing correction), real-time transposition, and complex looping constructs would be nearly impossible with the use of audio tape alone.

Sequencers are currently available for the IBM-PC (Jim Miller, Roger Powell, Voyetra), MacIntosh (Southworth, Mark of the Unicorn), Amiga (Mimetics), Atari ST (Hybrid Arts) and for the Commodore 64 and Apple II (Doctor T, Passport). One of the original entries in the IBM field is Jim Miller's Personal Composer. It can record a performance in real-time or data can be entered with a mouse or keyboard. Musical data can then be rearranged and edited using traditional staff-line notation and scores can be printed on common dot matrix printers. Personal Composer includes a built-in patch editor for DX-7 synthesizers, a small graphics editor and even a user-accessible LISP interface.

PERSONAL COMPUTER INTERFACES

Garvin: PERSONAL COMPUTER INTERFACES

If you already own one of the computers mentioned, getting started in MIDI composition is as simple as purchasing the 'standard' interface. You will then have access to broad range of software packages which can be expanded and updated via disk (just like compilers and word processors). Some of the more common interfaces are: Passport (C-64 and Apple II), Opcode (MacIntosh), Mimetics (Amiga), and Roland (IBM and Apple II). The Atari ST series has a built-in MIDI interface. Most of these interfaces consist of little more than a UART for handling serial communication. The Roland MPU-401, however, includes an on-board microprocessor and timer for providing the user with preprocessed, buffered data packets.

WHAT MIDI DOES (and doesn't do)

Garvin: WHAT MIDI DOES (and doesn't do)

First of all, MIDI does not solve ALL existing problems in interfacing sound-generating equipment. For example, it is easy to send instructions for turning particular notes on and off, and sending a NOTE-ON command will usually result in a predictable pitch from any two synthesizers. There is no standard for 'a violin sound' or 'an oboe sound', however. **Patch numbers** can be requested via MIDI, but it is up to the individual programmer or manufacturer to devise the violin or oboe sound and assign a patch number to it. Manufacturers of lighting controls or mixing boards are on their own; system specific commands tell other devices to ignore codes they won't understand, but there is no standard spec for how lights should respond to music commands.

These are some parameters which ARE standardized by MIDI:

VOICE messages include NOTE-ON and NOTE-OFF events. **Notes** are assigned numbers from **0-7Fh** and simply turned on and off. Velocity information is included with the command and may be interpreted as loudness. **Channel addresses** designate a particular oscillator or synthesizer which will respond to the command. Bytes from **90h to 9Fh turn on notes** for channels 0-0Fh; bytes from **80h to 8Fh turn the same notes off**. The three-byte command to turn on note number 32h, on channel 3, at velocity of 22h is: 93h, 32h, 22h. The command to turn off the same note with a turn-off velocity of 10h is: 83h, 32h, 10h. Velocity is usually not relevant when turning a note off, so sometimes a NOTE-ON command with a velocity of zero is used as a NOTE OFF. Other voice commands include key pressure and pitch bend, but these are not sent as part of the NOTE-ON or NOTE-OFF message packets.

MODE messages control the response characteristics of the receiving device. Synthesizers can be told to 'listen' to a specific channel (OMNI-OFF) or to respond to messages on all channels (OMNI- ON). In addition, provision is made to address one oscillator per channel (MONO MODE) or to allow the synthesizer's internal software to assign its own voices so messages can be sent over one channel (POLY). Combinations of these yield four modes.

SYSTEM REAL-TIME and clock messages allow synchronization of all machines in the chain via 'software-clocking' over the MIDI bus.

SYSTEM SPECIFIC commands address synthesizers from a specific manufacturer. This covers patches and other data which would have no meaning to certain types of synthesizers, or lighting control commands which should only be received by the light controller. Of course, each manufacturer must apply for their own 'ID' byte (Sequential Circuits = 1, Kawai = 40h, etc.). An 'END OF EXCLUSIVE' (EOX) or another status byte tells deselected devices to resume listening to bus data.

SYSTEM COMMON messages address all synthesizers on line. They are used primarily for 'setup' information such as selecting songs or telling synthesizers to tune their oscillators.

The leading byte of all MIDI commands has its MSB set (high), so command bytes are always between 80h and 0FFh. Data bytes have their MSB's reset (low), so their range is 0 to 7Fh. If a given block of data contains values greater than 7Fh, all bytes in the block are broken into nybbles and sent in two parts. This keeps the MSB's reset so that receivers can always stay in synch -- even if a byte is lost.

MIDI HARDWARE SPECIFICATION

Garvin: MIDI HARDWARE SPECIFICATION

The original MIDI specification made some tradeoffs between speed, economy, and efficiency which inevitably resulted in compromises in performance. It is easy to point out shortcomings now that the interface has become widely known, but the low cost had a lot to do with its initial acceptance.

MIDI is a serial protocol which communicates on a 31.25 KHz, optically isolated current loop. The odd baud rate resulted from the reluctance of earlier manufacturers to install special crystals when they usually had a 1 to 4 MHz processor clock available. Use of a convenient single-chip binary divider yields the 31.25 KHz UART clock.

Optical isolation is required for eliminating ground loops and isolating sensitive audio equipment from the high frequencies in computer gear. An optically isolated parallel interface could have been specified, but serial interfacing decreases the cost for the isolators and simplifies cabling. The penalty, of course, is speed. Clock rates are limited by cable capacitance and by response times for economical opto's and UART's. Some manufacturers are now using 62.5K (double frequency) rates for downloading waveform samples or other data-intensive applications, but it is unlikely that the baud rate standard will change in the near future.

The opto's at the receiving end of each MIDI link require about 5 ma to turn on. Since the loop sends a current (like old Teletype machines) it is relatively immune to noise, as long as cables don't extend more than 50 feet. The built-in current limit resistors prevent star-network configurations (only one receiver may be hooked to a transmitter), so a third port (the 'MIDI THRU' port) is included on most synthesizers to allow daisy-chaining of receivers. The MIDI THRU port duplicates the data coming from the MIDI IN port and retransmits it to the next machine in the chain.

SPEED CONSIDERATIONS

Garvin: SPEED CONSIDERATIONS

Most MIDI NOTE-ON or NOTE-OFF commands require three bytes at 320 microseconds per byte, so turning on a note on the synthesizer takes approximately 1 millisecond. The human ear is more sensitive to starting ('attack') transients than to ending ('decay') timings -- for psychoacoustic reasons, and simply because the played notes usually start off sharply and taper off before their final decay. This means that even in best-case circumstances where no other events are being sent over the MIDI bus, starting transients for ten NOTE-ON events will be spread apart by 10 milliseconds. This approaches the threshold of audible delay, and additional notes may have a 'slap- echo' effect.

To avoid objectionable delays, some MIDI hardware now features multiple MIDI OUT ports. The computer sends parallel commands to each port to avoid daisy-chaining delays. These should not be confused with MIDI THRU ports, which track the MIDI IN port. Multiple MIDI IN ports are less common, but certainly helpful when recording events coming from more than one source. Since MIDI is a multi-byte protocol, merging and recording two sources can be fairly complex if only one input port is available.

DESIGNING A SEQUENCER

Garvin: DESIGNING A SEQUENCER

MIDI control software can be written in any language, but fast queuing of serial data is important. The majority of software writers that I know use a mixture of C and assembly language. It may be convenient to use a compiler such as Wizard C, which allows dropping into assembler for I/O access or speed.

Small computers can be used, but be aware that RAM can be used quickly. If storage for a single note uses eight bytes, an eight- finger chord will use 64 bytes, and playing eight of these chords in one measure will use half a K of RAM. Linear address space is easy to allocate and control but segmented architectures present no large problems, since a segment is usually more than enough to record any single sequence of note events (a 'track' in recording terminology). Bank-select RAM is usually a problem when several tracks are played back in parallel. If the tracks are stored in separate banks, the switching overhead may be cumbersome.

Most of my current designs use IBM PC's and AT's, so some of the examples will focus on 8088 designs, but the design principles will adapt easily to other computers.

Garvin: DESIGNING HARDWARE

Custom hardware affords some measure of software security and may provide functions not available on existing interfaces. Since some designers prefer using their own hardware, I will provide a few guidelines.

MIDI's serial protocol requires no hardware handshake signals, so 40- pin UART's are not necessary. The only small UART that may cause trouble is the Intel 8251. I have used Motorola 68B50's in several designs with good results. On 8088 systems, Motorola's E (enable) signal can be developed by 'anding' port read and write.

Timers get tricky when multiple devices are hooked to one interrupt line. I have used Intel 8253's, but I usually connect the output line to a flip-flop so that the output pulse can be trapped and identified. Flip-flops are not necessary if the timer interrupt is isolated, since the 8259 interrupt controller has edge-triggering.

Timers will not be required on IBM-PC's using the Roland interface, but if you are designing your own, try to include an on-board timer. The PC's internal timers do not provide the accuracy necessary to deal with high-resolution music timing. For low-resolution applications, IRQ 0 from the PC mother board can be readjusted to run at a multiple ('X') of its normal speed. Then, every 'X' pulses, the old interrupt is run. The interrupt acknowledge should be skipped when jumping to the 'old' routine. Remember to reset the interrupt vectors before exiting to DOS. Disk head loads use timer 0 for time-outs, so problems with this routine usually cause the drive light to stay on.

Slower clock rates for hardware timers will obviously result in decreased resolution. Not so obvious, though, is the way that tempo resolution and note-timing accuracy combine to make even tougher demands on the system timer's crystal frequency. I try to clock the timer at around 2 MHz so that I can record with resolution of about 96 divisions per quarter note while maintaining sufficient accuracy in specifying tempos.

In most applications, the divisor sent to the hardware timer is used to trim the tempo, which is set in increments of 'beats-per-minute'. An easy way to control tempo is to index into a table of divisors by the desired number of "BPM's". I normally generate the tables with a 'C' program which does the calculations and prints out the tables exactly as they should appear in the sequencer program. When the values are verified, I redirect the output of the program into a file which is then compiled.

INTERRUPTS

The interrupt service routine (ISR) is responsible for prioritizing interrupts and co-ordinating all incoming data. Obscure problems with the ISR can propagate through the entire system. A flow-chart will probably help to clarify possible timing errors or bottlenecks before the ISR is coded.

It might appear that a system using only serial ports would pose no problems in I/O handling, but MIDI baud rates are high, and there may be multiple ports. Input interrupts should be serviced as quickly as possible because losing a byte may result in losing an important NOTE-OFF message. When the recorded events are retransmitted on playback, the synthesizer will play a 'stuck note' until the operator can figure out a way to generate a NOTE OFF (sometimes leaving an embarrassed performer desperately groping for the power switch). Output interrupts are less critical; a momentary delay is

the only penalty for slow output response times. Be particularly careful how these two interrupt sources are handled when UART hardware lines are shared.

On the IBM PC, the first instruction in the ISR can be an STI (the BIOS does this) for re-enabling higher priority interrupts. Lower priority interrupts can be enabled by masking the present interrupt and sending an acknowledge (EOI) to the interrupt controller. The pending interrupt cannot be retriggered until the source of the interrupt is reset (UART is read, etc.). Make sure that the pending interrupt line is high (active) when sending the EOI, because obscure problems can result with the 8259 when it cannot find the source of the interrupt being acknowledged. It may also help to poll the 8259 registers to make sure that the interrupt line is low before exiting from the ISR. This will help to avoid difficulty with the PC's edge triggered hardware.

Timer interrupts are important, but they can generally be considered a lower priority than UART interrupts. The timer routine itself is usually short, consisting of little more than incrementing a series of software counters, but at some point compares must be made with 'target' values and a long series of events may be triggered.

The timer interrupt conveys no actual data aside from a flag, so normal queues are not necessary. The best way to 'enqueue' timer interrupts is with the use of a semaphore, which allows the ISR itself to be interrupted. When the timer 'tick' occurs, the semaphore is incremented and THEN timer interrupts are re-enabled. If the semaphore has been incremented to 1, no interrupts are nested and normal processing can resume. If the value is greater than 1, this means that the timer interrupts have stacked up, so the interrupt is exited, and control is returned to the timer ISR which was interrupted. Before the timer ISR is exited the semaphore is decremented, and if the value is still non-zero, the interrupts must have been nested. In this case, control is returned to the top of the timer ISR, which repeats until the semaphore returns to zero. This allows the routine to catch up with 'lost' interrupts without losing any timer pulses or UART interrupts.

The flow chart (fig 1) is simplified; it may help to refer to the code listing (listing 1) for more subtle details. Make sure the semaphore is initialized to zero at startup or the timer ISR will never run.

TIMING NOTES

The MIDI sequencer will usually use a hardware clock for its main timing reference, but it may be necessary to synchronize to an external software clock provided by a drum machine or timing converter. I will refer to these as internal and external synch, respectively. MIDI software clocks occur at a standard rate of 24 per quarter note, which provides musical resolution to within a sixty-fourth note triplet. This sounds like it would keep up with even the fastest musicians, but remember that we are dealing with timing EDGES. Trimming these edges to the nearest 24th of a quarter note may cause the recorded notes to sound too symmetrical, or mechanical.

Conversely, it may seem that slowing the system clock could correct the timing of inaccurate note values. This rounding ('quantization') is sometimes used to advantage, but overuse removes the human signature and creates a metronomic, mechanical sound. Uneven qualities are most often missed on parts such as solos which appear 'up front' in a composition. Obviously, all devices synchronized with MIDI real-time clock signals will be quantized to some extent.

TIME-STAMPS

In order to maintain very precise timing of events while allowing interrupts to procede at full speed, I store a 'time-stamp' with each event coming into the UART receive queue. Very simply, if a received byte is greater than 7Fh (MSB is set on commands), the current time is enqueued after the byte. This provides 'freeze-frame' timing; the time record travels with the received data until the program is ready to process it. Only the leading byte needs to be time-stamped. Follow-up data (with MSB's reset) are assumed to have been received at the same time. This technique can provide accuracy better than that obtainable by waiting for the complete record and processing it instantly. Usually the sending device intends that all bytes be received simultaneously, so stamping the leading byte will more accurately reflect the actual event timing, even if the transmitter lags in sending the follow-up data.

REAL-TIME

The MIDI specification calls for two basic types of software clocks. 'Clock-in-stop' (0FCh) allows receivers to phase-lock to the clock frequency prior to start-up. When the transmitter switches to 'clock-in-play' (0F8h), all synchronized receivers switch to their active state (usually playback or record). To maintain accuracy, real-time messages are transmitted at any time -- even in the middle of other multi-byte messages. Receivers must account for this possibility even if the real-time messages are not used. Both clocks are always sent at the rate of 24 per quarter note. Altering the clock frequency will change tempo, not accuracy.

Other real-time messages include 'START-FROM-BEGINNING' (0FAh) which resets internal song pointers, 'CONTINUE' (0FBh) which tells receivers to resume from the current location and 'ACTIVE SENSING' (0FEh) which just lets the receivers know that the transmitter is still there. The latter is optional and used notably by the Yamaha DX-7 synthesizer. When using a DX-7, you will probably want to discard the 0FEh bytes, since they will be received constantly -- even when not recording. They can fill up the input queues if the input interrupts are enabled.

The last real-time message, 'SYSTEM RESET' (0FFh) is dangerous because it could start a regenerating condition where every component in the system sends resets to each other. It is usually reserved for linkage to a hardware reset switch or used judiciously by the master controller.

Garvin: STORAGE FORMATS

There is no standard yet for either RAM or disk-based storage for MIDI events. I have heard rumors of a standard for disk storage which would allow one manufacturer's software to read files written by someone else, but intermediate RAM storage is another story. The storage formats used throughout the industry are diverse, and usually so complex that changing internal formats would require extensive rewrites.

This is out of date. See: Standard MIDI-File Format Spec. 1.1

Most internal storage methods fall into one of four categories which I will call 'end-point-relative', 'end-point-absolute', 'single- point-absolute', and 'bar-and-note' storage. All storage formats involve storing data in a linear data stream. Relative timing implies that timing will be encoded as a distance from the previous event. Absolute timing will use a global time reference, such as beats and bars. End-point storage refers to separate storage locations for NOTE-ON and NOTE-OFF events (usually with their own time-stamps). Single-point storage requires that a pointer be aimed at the NOTE-ON record in the data stream, and when the NOTE-OFF event is received, it is stored at the same location (better yet, the note DURATION can be computed and stored). The bar-and-note method parallels the way music is normally notated. Each method has advantages, and there is a lot of overlap between categories.

I will try to carry MIDI's philosophy of setting MSB's of leading bytes when encoding data for storage in the stream. This will allow resynching if a byte is missed, and will make data streams easier to edit. Many software writers use this method for internal storage.

END-POINT-RELATIVE STORAGE

MIDI data is received as a stream of bytes with high bits (MSB'S) set on commands and reset on data. Why not store the bytes just as they are received? Embedded MIDI clock messages provides the proper spacing for NOTE-ON, NOTE-OFF and other events. Replaying the data requires setting up a series of 'play pointers' into the data stream (one for each track to be played). When the start command is received, the data is sent to the output UART's just as it was received, waiting for a 24th of a beat every time a clock command is encountered in the stream. Unfortunately, this method uses RAM storage even if no events are being transmitted, and multiple channels store multiple copies of all unnecessary timing bytes.

```
 Ex:
- F8h --------- F8h 94h 46h 32h -------------- F8h 84h 46h 16h ---------- F8h
|wait 24th     |wait 24th, then |wait 24th, then
||    output a NOTE-ON on ch.4  |    output a NOTE-OFF on ch.4
|| for note no. 46h       | for note no. 46h
||     velocity = 32h     |     OFF velocity = 16h
```

In this example, it is assumed that an external MIDI clock provides the timing. Even if an internal timer is used, 0F8h bytes can be inserted into the input queue to simulate the MIDI software clock. A refinement of this method conserves storage by counting all received clock bytes. When an event is received, the accumulated count is stored BEFORE the event and then the counter is reset.

END-POINT-ABSOLUTE STORAGE

Time-stamping requires a system timer that is incremented every time a MIDI clock or timer interrupt is received. When MIDI data is enqueued, the system timer is copied into the queue as the time- stamp. When the data is dequeued for storage, the time-stamp is stored with the data record to provide an accurate absolute timing reference. Unlike relative timing, this allows the user to locate any spot in the data stream without counting all embedded timing bytes. To replay time-stamped data, restart the system timer and wait until it matches the timing bytes of the first item in the data stream. Then send the data (without the time-stamp) and advance the stream pointers.

```
  Ex:
  ----94h 09h 03h 46h 32h -------------- 84h 08h 04h 46h 16h -------
|output a NOTE-ON on ch.4   |output a NOTE-OFF on ch.4
|    on the ninth clock tick       |    on the eighth clock tick
| of the third beat  | of the fourth beat
|     for note no. 46h      |     for note no. 46h
|  velocity = 32h    |  OFF velocity = 16h
```

The example uses only two bytes for the time-stamp and for the system timer. More practical systems use three bytes, to allow over two hours recording time before the timer overflows (at 96 pulses per quarter). The low-order byte is incremented until it reaches 96, (or 24 for MIDI clock timing). Then the byte is reset and the count propagates through the other two timer bytes. The top timer bytes 'turn over' at 127 so the MSB's are always zero.

SINGLE-POINT-ABSOLUTE STORAGE

Single-point storage requires maintaining a list of pointers to access active notes in the data stream. When a NOTE-ON is received, it is enqueued and stored in the data stream in the same way as end- point-absolute storage, but zero's are stored afterward in a compartment reserved to keep track of the note's duration. A pointer to the zero-bytes is stored in a list to allow access to the duration. Every time a MIDI clock byte or timer interrupt is received, the list is checked, and the pointers are used to increment duration bytes. When the NOTE-OFF event is received, the pointer is removed from the list -- no other action is necessary. The duration will be frozen as part of the note record. Single-point-absolute storage provides advantages in editing (notes can be moved easily), and display (it is easy to tell whether a note is a quarter note, half note, etc.).

```
   Ex:
  --94h 09h 03h 46h 32h -------------- 04h 01h ---------------------
    | output a NOTE-ON on ch.4    These are duration bytes which
    |     on the ninth clock tick        travel with the note record and are
    |  of the third beat   incremented by each timer tick
    |      for note no. 46h       until NOTE-OFF is received
    |   velocity = 32h
```

During playback, NOTE-ON's are derived in the usual way (wait until the system timer

reaches the embedded time-stamp), but this time the duration bytes are retrieved and stored in a 'timeout' list. They are decremented with every timer tick, and when they reach zero, a NOTE-OFF is transmitted. The timeout list must keep a copy of the note number so that the proper note can be turned off. The single- point method affords another advantage: it is unlikely that notes will get stuck. NOTE-OFF events cannot be missed. Any reasonable value in the timeout list will eventually decrement to zero and cause a NOTE-OFF to be sent.

BAR-AND-NOTE STORAGE

So far, I have discussed playing notes, but not rests. Of course, rests are simply the spaces between notes, but the storage formats outlined do not provide a way of tracking down these spaces for correlation with written music. Rests can be stored in the same way as notes, but note numbers and velocities are not needed. This seems to be a reversion to the original relative timing method when you consider that rests are similar to the old embedded relative timing markers. Now the NOTE-ON time-stamps become redundant -- where one note-or-rest stops, the next will start. Without some frame of reference, though, it is difficult to find a designated spot in the data stream. I have borrowed another device from written music: bar lines. There is no MIDI equivalent for a bar line, so I will use 0BAh. The bar marker will be followed by one-or-two-byte bar numbers to allow absolute locations to be found. This combines some of the better features from all the methods outlined above. I will use 0A0h as the token for a rest.

```
Ex:
-0BAh 32h -----0A0h 02h 14h ---------- 94h 46h 32h ---------------01h 02h---
| At bar #32h | rest    | output a NOTE-ON on ch.4  duration
|      |     for two beats     |    for note no. 46h       is 1 beat
|      |  and 14h ticks | velocity = 32h     and 2 ticks
```

Some storage formats are better suited to certain approaches to editing or to certain looping constructs or display formats, etc. Choose a format that suits your application, but remember to take as general an approach as possible. You will undoubtedly want to expand later to incorporate new ideas.

QUANTIZATION

Quantization was first mentioned in the context of scaling down the system clock. If a section of music was recorded using a 96 pulse per quarter note clock, the system clock can simply be slowed to 24 pulses per quarter note. Each timer interrupt would then increment the system timer by 4 instead of 1. This method works but it has one main drawback. If there is no frame of reference (an unquantized track, for instance) it may not be noticed, but all notes will effectively be shifted LATE by the quantize interval. This is like truncating a number when the real intention is to round it.

To quantize events so that the notes fall ON the beat rather than AFTER the beat, the timing target must be ANTICIPATED by half the amount of the quantization period. This causes events to be processed in the center of a 'quantize window'. To

accomplish this, the system timer contents are copied to a 'look-ahead timer' and incremented so that it LEADS the actual system time by half the quantize interval. Using look-ahead timers for compares will trigger events ahead of time. Remember just to run the clock routine every Nth clock tick and to add N/2 to the current time to derive the look- ahead timer.

The same result can be accomplished by first running the clock routine N/2 times consecutively (This accounts for look-ahead). After this initial look-ahead, the clock cycle consists of waiting for N clock periods and then running the clock routine N times consecutively.

Quantization will clean up notes whose timing is slightly 'frayed at the edges', but some mistakes can actually be accentuated. If the mistake is severe enough to fall outside of the intended quantize window, note timing will be rounded in the wrong direction as in NOTE #3 of fig. 2.

It was mentioned that the trailing edges of notes are usually much less timing-critical than the leading edges. One of my favorite techniques for avoiding a mechanical sound is quantizing the leading edges of notes but leaving the lengths intact. This can be accomplished by using a look-ahead on the clock which pulls the NOTE- ON event and NOTE-LENGTH from the data stream. The unprocessed note length is stored in a timeout list where it is decremented on each tick of the HI-RESOLUTION clock. The appropriate NOTE-OFF is sent when it reaches zero.

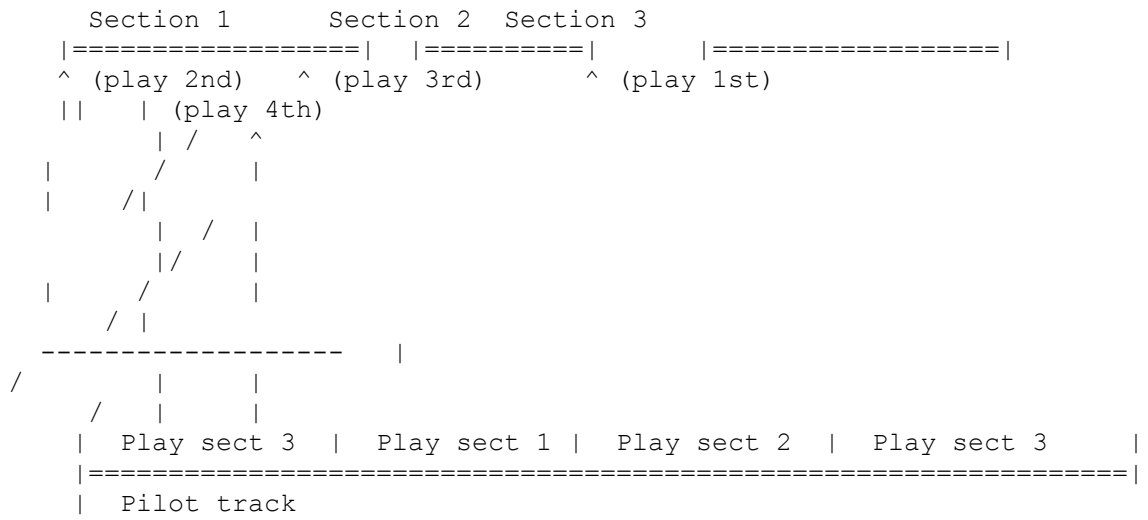
The timing of recorded music is easily altered or quantized, but random timing information from human input is difficult to add later (you COULD try it). Some synthesizer systems, such as Fairlight, use high-performance hardware to derive timing clocks as high as 384 clocks per quarter note, but be aware of micro-based software which claims this level of accuracy. By attempting performance beyond the capability of the machine, the software can actually sacrifice accuracy.

PILOT TRACK

Most musical compositions have verses, choruses and other types of sections. Sections are usually recorded or written separately. When all the sections are completed, they are rearranged, repeated, etc. by the use of what I will call a 'pilot' track. The pilot track in this type of system will operate 'outside' of the time frame of the composition. In fact it may be the only timing stream which moves in a linear fashion. Macro commands, such as 'play section 3, five times' will pilot the interpreter through the appropriate series of pointer loads, plays and reloads to accomplish the task.

The start of each section now becomes the main point of reference for note timing. At the end of a section the system timer is usually reset and the pilot track is consulted to find the next section to be played. Each section will be treated as if it is a separate composition, so each may continue up to the maximum length allowed by the system timer.

Some pilot track schemes use a FORTH-type stack to hold reiterative constructs and section or data references. I have even seen a sequencer which allowed English statements such as 'SECTION 3 = VERSE 1 + CHORUS'. In any case, the section is treated as a subroutine, with control returning to the pilot track when the section has completed.


Section 1
Section 2
Section 3
(play 2nd)
(play 3rd)
(play 1st)
(play 4th)
Play sect 3
Play sect 1
Play sect 2
Play sect 3
Pilot track

Garvin: ADVANCED FEATURES

If you've made it this far, you may be interested in some of the extensions to MIDI protocol to allow more rapid transmission of event triggers. '**Running status**' says that the receiver will keep the last command byte received. If additional data bytes are received without command bytes, the old command bytes will be used.

To transmit NOTE-ON's on channel 3 for note numbers 22h, 33h and 44h with velocities of 55h, 66h and 77h the following bytes could be sent:

```
93h--------------22h---55h---------33h---66h---------44h---77h------
|   |     |    |     |    |     |
NOTE-ON cmd      note#  vel       note#  vel  note# vel
Ch 3      first note       second note        third note
```

Duplicating the same series with an 83h as the first byte will turn the notes off again.

To make this feature more useful, the special condition 'NOTE-ON with velocity = 0' is reserved to signal a NOTE-OFF operation. Its function is identical to the normal NOTE-OFF but since it is actually a NOTE-ON command, the 'running status' rule applies. The same notes could be turned on, and off again by the following bytes:

```
93h-------22h--55h---33h--66h---44h--77h----22h---0----33h---0-----44h---0
|   |     |    |    |     |    |      |    |     |    |      |    |
NOTE-ON  note# vel  note# vel  note# vel    note# vel  note# vel  note# vel
Ch 3    first_note second_note third_note  first_note second_note third_note
```

Remember that NOTE-OFF's won't actually be sent immediately after NOTE-ON's. If any other command bytes are sent in between, the 'running status' is interrupted and the command byte must be retransmitted.

Garvin: DISPLAY METHODS

Storage methods can be related to display methods in that the note events can be displayed at one spot (as a conventional music note) or they can be displayed as a (usually horizontal) band on the screen with start and end points representing the start and end of the note. The latter method, sometimes known as piano-scroll notation, is analogous to the end-point storage method that I have outlined. So, do software designers who use single-point storage use conventional notation and designers using end-point storage use piano-scroll notation? Of course not. Strangely enough, some of the more popular software packages use exactly the opposite techniques as expected.

Both display methods have advantages: piano-scroll notation can be very 'visible' to musicians who are not accustomed to reading music, while conventional notation maintains high information density. Also, conventional notation always requires graphics capability. Piano-scroll can usually be done with text-mode graphics.

Displaying notes on staff lines requires at least 4 or 5 vertical pixels per line on the staff, for a total of 17 to 21 pixels for a complete staff line (four lines times 4-5 pixels plus an extra line). I have seen sheet music with notes printed as far as six spaces above or below the staff, so it is wise to allow a lot of blank space on each side of a staff line. Allowing 60 vertical pixels total per staff should yield five or six staff lines on a high-resolution screen.

Horizontal formats are usually best handled and allocated by the byte. A screen 640 pixels across would divide into 80 horizontal compartments with screen objects treated somewhat like ASCII characters but with variable widths (a G-Clef will require two or three character widths).

Despite the popularity of IBM computers, the CGA's unfortunate lack of adequate screen resolution has limited its use for staff-line oriented editors or forced earlier sequencers to use Hercules or other non-IBM graphics boards. The CGA screen produces very square- looking notes, so conventional notation on this screen may not be worthwhile. I find the EGA graphics board extremely slow to write, but color is a valuable tool for displaying music. Notes on a staff- line can be color-coded to designate channel numbers (to allow more than one channel per staff-line). Monochrome editors sometimes allow selection between two channels per staff by directing note stems up or down.

Display formats will depend largely on the display devices you have on hand or wish to support. While EGA boards have finally made the IBM PC competitive with other computers when displaying color, you may only want to enter musical notes and then print them out on paper. Some music transcribers are using Jim Miller's software without ever buying MIDI hardware for their computers. When doing black and white printouts, obviously a Hercules mono graphics card will suffice. The Hercules board has 720 horizontal pixels, which ends up looking a lot longer than the EGA's 640 pixels (sometimes a full bar of music).

Editors for note events take many different forms, but list-oriented editors are the easiest to design, followed by piano-scroll editors. List editors can simply convert a list of note numbers into NOTE-ON and NOTE-OFF events. This may be adequate if the main function of the software is to record live music being played on a synthesizer.

Most piano-scroll editors use strict binding between screen position and note events. For example, the screen is sometimes partitioned into an even bar of music. The X-axis position of the cursor will then relate directly to the music's time domain.

Staff-line editors usually require a complex series of pointers to correlate records in the data stream with locations on the screen. Using a cursor to locate and change a point within the music data stream can be a difficult task because the screen spacing may be non- linear when related to the time domain (a bar consisting of a single whole-note will be shorter than a bar which holds a series of sixteenth-notes). Many staff-line editors DO require vertical alignment of synchronized events, such as left and right hand piano parts which are written on separate staffs. This allows the screen to be swept from left to right with a single X-axis pointer, but strange timing errors will be introduced by the converter when vertical alignment is not maintained. If you are writing this type of editor/converter, I recommend keeping a separate X-axis pointer for each staff line. Staff-oriented editors are further complicated by the need for multiple data representations. The on-screen symbols usually must be transformed into a format which is quite different in order to play them as MIDI notes.

A piano-scroll editor is a good starting point for an experienced software writer who has limited knowledge of traditional musical concepts. Only the most experienced writers should attempt to write a staff-line editor, as it requires a thorough knowledge of music theory as well as programming.

Garvin: WRITING YOUR OWN PATCH EDITOR

One of my current projects is a patch editor for the Kawai K-3 synthesizer, so I can explain exactly how patch editing works. The K-3 generates sound by building waveforms from sine wave harmonics. These waveforms can be manipulated from the K-3's front panel, but the addition of a computer screen offers an enormous advantage in visualizing sounds as waveforms are being altered. In my wave editor, color-coded bars move to indicate harmonic numbers and amplitudes. As the operator tailors the waveform by adjusting the on-screen representation, an internal table of values is also adjusted. Just as in a text editor, different versions of the data can be saved to disk as the waveforms are being edited. The K-3 holds only one internal user-defined wave, but I hold up to a hundred waves within one file so they can be compared and interchanged.

When a patch or wave is ready to be sent to the K-3, a MIDI system exclusive command tells the synthesizer to expect new patch information. No acknowledge is necessary -- the harmonic numbers and amplitudes are sent out immediately. Transmitted data can be of any length but 8-bit data must be sent as two separate nybbles to ensure that the MSB's will be reset (0). Because of the potentially long data stream, KAWAI requires a checksum for confirmation, but this is entirely up to the manufacturer. The data packet is closed by an end-of-exclusive byte which lets the synthesizer get back to music processing.

The bytes sent to the K-3 to set up a wave look something like this:

```
F0h-40h-0h-20h-0h-1h-64h----0h-1h--0h-5h-...........--1h-9h--1h-0Fh----0F7h-
 |    |  |   |  |  |  | |      ||  |      |  |EX | CHAN | GRP | INTRN  HARM=1
AMP=5 (more-data) HARM=19h AMP=1Fh   EOX
KAWAI FUNC#  K3  WAVE
```

Data formats for send and for receive are identical, so two synthesizers can be hooked together for trading waveforms, or waves can be sent to the computer, modified, and then sent back to the K-3.

Every manufacturer and synthesizer has a different format for system specific commands. Consult synthesizer manuals for details.

Garvin: CHOOSING A SYNTHESIZER

If you will be testing your MIDI software by recording music in real- time, you will need a keyboard or other source of MIDI note messages. If cost is a factor, look at Casio's CZ-101 (miniature keyboard) or CZ-1000 (larger version). In a slightly higher price bracket, the Kawai K-3 has a very good performance-to-cost ratio. Yamaha has developed a wide range of FM instruments, and their DX-7 series is one of the best-selling synthesizers in the over-$1000 range. Korg also makes several affordable machines.

If you are interested only in patch editors, or if you already own a MIDI keyboard, consider using a stand-alone sound generator. These are becoming more popular, since one MIDI keyboard can control a number of slaved sound generators. Kawai has introduced a keyboard- less version of their K-3 synthesizer, and Yamaha has just introduced a module (the FB-01) for under $400. Low-cost rack-mounted sampling units include the AKAI model S612 and the Ensoniq Mirage.

The eight-voiced FB-01 and the four-voiced Casio CZ-101 can both assign separate tone patches to each voice. In other words, a piano sound, a violin sound, and a horn sound may all be played at once. This is an advantage to the software designer, since the module can be made to respond like multiple synthesizers. It is difficult to assign or 'prioritize' multiple voices with a single keyboard, so the Casio allows this 'MONO-MODE' operation only when sounds are played and assigned by an external computer. The FB-01, of course, has no keyboard.

Synthesizer keyboards generally have non-weighted plastic keys. If you prefer a keyboard which feels more like an acoustic piano, you may want to invest in one of the higher-quality MIDI keyboard- controllers, such as the Roland MKB-1000 or Yamaha KX-88. Since these are output-only keyboards (they have no sound-generation electronics), they must be used in conjunction with an external MIDI- controlled sound generator.

The sound of a synthesizer depends both on its electronics and on its programming, so it is impossible to categorize every type of instrument. There are some guidelines, however. Oscillators may be analog or digital. The difference is somewhat like comparing records and compact discs: digital oscillators are very precise and they can produce a wide range of timbres, but some say that they lack the warmth of analog oscillators. Analog machines, such as the Roland, Moog, or Oberheim synthesizers, are known for producing rich string patches or resonant brass sounds. Digital machines, such as the Yamaha line, excel at more percussive sounds, like pianos or bells. Samplers can capture very breathy, human or flute-like voices. Musicians often use different types of synthesizers to cover different ranges of tonalities, but the ranges overlap quite a bit. Listen to as many patches as possible before making your choice.

Garvin: SMPTE (and other time codes)

Computerized recording is becoming very popular, but the accepted medium for interchange of completed music is still audio tape. Tape is necessary for recording voices, guitars, and acoustic instruments, and for transporting sounds between studios which have different types of synthesizers or drum machines. Synchronizing tape-based and computer-based recording mediums can be difficult. Fortunately, a solution to many of the problems already exists in the form of 'time- code'.

Time codes of various types have been in use for many years. They are used to 'lock' audio recorders to video machines for movie sound tracks and for time-stamping video tape for TV news. One of the most popular forms of time code was devised by NASA as a simple, fixed time reference for their experiments. They used recordable audio- range pulses in a format known as bi-phase modulation. Bi-phase uses clock transitions, rather than states of polarity, to encode binary data. This means that the output will never be a non-recordable DC voltage. The 80-bit serial data stream encodes time as hours, minutes, seconds, frames, and sub-frames. The last two increments are arbitrary values which vary, depending on usage, but even this vague specification was sufficient to merit acceptance in a wide range of applications, especially video film. It has come to be known as SMPTE code, after the Society for Motion Picture and Television Engineers.

SMPTE code is based on a fixed time-of-day clock, rather than a variable rate, so it is not the ideal code for resolving the fine nuances of musical timing. Hardware synchronizers, incorporating complex frequency multipliers and phase-locking schemes, must be used to correlate MIDI tempo timing and SMPTE absolute timing. Some of the first 'synch boxes' for SMPTE-MIDI conversion were from Roland (SBX-80) and from Garfield Electronics. It is difficult to calculate rates and match-up points for the two time codes so both of these units take the more practical approach of building a map of alignments as the time codes are received. The map is then stored to tape or to disk via MIDI.

An important step in the development of SMPTE-to-MIDI standards are the map formats being proposed by SMPTE synchronizer manufacturers such as Adams-Smith. Adams-Smith's new Zeta 3 system will allow commands from MIDI or RS-232 to control a tape machine, or time codes from the tape machine can be translated back to MIDI format. Two tape transports and a variety of sequencers and computers can be operated from a single Zeta 3 synchronizer. In actual operation, machines are synchronized by 'striping' one of the tape tracks on each machine with SMPTE code. The tape controller reads these tracks and fine-tunes motor speeds on the transports. The Zeta 3 controller also outputs MIDI timing bytes to keep sequencers in step with the tape.

Other types of time codes which are in common use include 'FSK', or frequency shift keying, which encodes 0's and 1's as two different frequencies. A major drawback to FSK is the lack of enough resolution to provide any form of embedded absolute time reference. FSK tapes must always be started from a known reference point, since FSK is a RELATIVE timing reference.

Incorporation of SMPTE-control or provision for some kind of sync-to- tape can be a big advantage when marketing software. It may only be necessary to stay compatible with support hardware marketed by other companies.

Garvin: SUMMARY

By providing a bridge between the music and computer industries, MIDI has sparked new interest in the design of innovative musical instruments. It has, in fact, created its own industry. Many competent software engineers are becoming interested in music because of this accessibility, and better products are being introduced every day.

Make sure you look at a few of the commercially available sequencers or patch editors before you start writing software. If you see something conspicuously absent on all the sequencers you encounter (such as a built-in universal patch editor), chances are that it is difficult to design. There are some very imaginative designers working with MIDI and some 'impossible' things may be accomplished with a new approach or just a lot of work. Visit one of the larger music stores to find out what is currently on the market.

Garvin: Bibliography

IMUG (International MIDI Users Group): PO Box 593, Los Altos, CA 94022
Membership: 8426 Vine Valley Dr., Sun Valley, CA 91352

SMPTE (Society of Motion Picture and Television Engineers):
595 W.Hartsdale Ave., White Plains, NY 10607

PERIODICALS

Electronic Musician: 2608 Ninth St., Berkeley, CA 94710
Subscription Dept.: 5615 Cermak Road, Cicero, IL 60650
Keyboards Computers and Software: 299 Main St., Northport, NY 11768
Keyboard Magazine - subscription dept.: Box 2110, Cupertino, CA 95015

MANUFACTURERS

Adams - Smith: 34 Tower Street, Hudson, MA 01749
AKAI Professional Div.: PO Box 2344 Fort Worth, TX 76113
Allen & Heath Brennel, Ltd.: 5 Connair Rd., Orange, CT 06477
Bacchus Software Systems: 2210 Wilshire Blvd, Suite 330, Santa Monica,CA 90403
Brocktron-X: 5 East 22nd St., Suite #21M, New York, NY 10010
Casio, Inc.: 15 Gardner Rd., PO Box 1386, Fairfield, NJ
Digidesign Inc.: 100 S. Ellsworth, 9th Fl., San Mateo, CA 94401
Digital Keyboards, Inc. (Synergy): 105 Fifth Ave. Garden City Park, NY 11040
Doctor T's Music Software: 24 Lexington St. Watertown, MA 02172
E-mu Systems, Inc.: 1600 Green Hills Rd., Scotts Valley, CA 95066
Ensoniq Corp.: 263 Great Valley Parkway, Malvern, PA 19355
Fairlight Instruments: 1610 Butler Ave., West Los Angeles, CA 90025
Garfield Electronics: PO Box 1941, Burbank, CA 91507
Grey Matter Response: 15916 Haven Ave Tinley Park, IL 60477
Hybrid Arts, Inc.: 11920 West Olympic Boulevard, Los Angeles, CA 90064
Jim Miller (Personal Composer): PO Box 648, Honaunau, HI 96726
J L Cooper Electronics: 1931 Pontius Avenue, West Los Angeles, CA 90025
Kawai America Corp.: 24200 S. Vermont Ave. PO Box 0438, Harbor City, CA 90710
Key Clique Inc.: 3960 Laurel Canyon Blvd, suite 374,Studio City, CA 91604
Korg USA, Inc.: 89 Frost St., Westbury, New York 11590
Kurzweil Music Systems, Inc.: 411 Waverly Oaks Road, Waltham, MA 02154-8464
Lexicon, Inc.: 60 Turner St., Waltham, Massachusetts 02154
Mark of the Unicorn: 222 Third Street, Cambridge, MA 02142
Mimetics: PO Box 60238 Station A, Palo Alto, CA 94306
Moog Music and Electronics: 2500 Walden Ave., Buffalo, NY 14225
New England Digital Corporation: White River Junction, Vermont 05001
Optical Media International: PO Box 2107, Aptos, CA 95001
Oberheim: 11650 W.Olympic Blvd., Los Angeles, CA 90064
Opcode Systems: 707 Urban Lane, Palo Alto, CA 94301
Passport Designs: 116 North Cabrillo Hwy., Half Moon Bay, CA 94019
Roland Corp US: 7200 Dominion Circle, Los Angeles, CA 90040-3647
Sequential Circuits, Inc.: 3051 North First St., San Jose, CA 95134
Southworth Music Systems, Inc.: Box 275, R.D. 1, Harvard, MA 10451

Syntech Corp: 5699 Kanan Road, Agoura, CA 91301
360 Systems: 18730 Oxnard St. Tarzana CA 91356
Voyetra Technologies: 426 Mt. Pleasant Ave, Mamaroneck, NY 10543
Yamaha International Corp.: PO box 6600, Buena Park, CA 90622