Chapter 13
MIDI Toolbox

Most Important Information

The commands for the direct transmission and reception of **MIDI** information are described in this chapter. MIDI provides a standard way to communicate between synthesizers and computers. The most important output words are **MIDI.NOTEON** , **MIDI.NOTEOFF** , **MIDI.PRESET** and **MIDI.CHANNEL!** . The low level words that are most important are **MIDI.XMIT** , **MIDI.FLUSH** and **MIDI.RECV**. The **MIDI Parser** receives incoming MIDI information and then passes it to your user written functions. The most important MIDI Parser words are **MP.RESET** , **MP-ON-VECTOR** , **MP-OFF-VECTOR** and **MIDI.PARSE**. Tools for reading and writing MIDI Files are also described in this chapter.

Overview

MIDI is an industry standard for the electronic exchange of musical information among a variety of computer assisted musical devices, such as synthesizers, samplers, notation and sequencing software, computers, mixers, signal processors, storage media, and performance sensors. It is a byte-oriented serial interface that provides fairly complete control over traditional music synthesizers, samplers, processors, mixers, computer software, and so on. For a good description of MIDI, refer to the International MIDI Association MIDI Standard description or any of the MIDI overview articles from *Keyboard* magazine, *Computer Music Journal*, or Craig Anderton's excellent book *MIDI for Musicians*.

HMSL contains a complete implementation of the MIDI standard, which we call the *low level MIDI driver*. This low level MIDI driver is used by HMSL to support MIDI instruments. These words can also be used directly for simple experiments in MIDI, or used from HMSL morphs by embedding MIDI calls in productions, interpreters, etc. If a particular instrument-specific MIDI operation is not supported here, it should be fairly simple, by referring to the source code, to write the driver yourself.

MIDI *channel allocation* is done automatically by instruments. Users may make use of the **MIDI-ALLOCATOR** object (see instruments) in their own allocation procedures, or may of course change channels directly in productions and actions, using the simple MIDI routines listed below.

One of the main uses of these routines for HMSL users will be in defining custom MIDI instruments.

Tutorial 1: Simple MIDI Output

In this tutorial we will play a few notes manually. We will also verify that MIDI is connected properly. First make sure that the OUTPUT of your MIDI interface is connected to the INPUT of your synthesizer. If you have a keyboard MIDI synthesizer, test to see if the audio connections are made properly by playing a few notes on the keyboard.

MIDI information is sent over 16 channels. A typical synthesizer can be assigned to respond to information on one or more channels. This allows a MIDI controller, eg. a computer, sequencer or keyboard, to control several different synthesizers at once. Set your synthesizer to respond to MIDI channel 1. (It probably already is.) Now tell HMSL to send its information over MIDI channel 1. Do this by entering:

```
1   MIDI.CHANNEL!    ( no space between L and ! )
```

Now let's send a "note-on" message over this channel. For the purposes of this exercise, select a preset that sustains well. Organ, flute or string sounds will do fine. Save the drums, piano, and bells for later.

MIDI "note-on" messages consist of two data values, a note index and a velocity. The note index corresponds to a particular note, or key, on the synthesizer. The lowest 'C' note is usually 36. A MIDI note index of 50 is a C note two octaves up. The maximum allowable note is 127 but the top note on a "5 octave + high C"

synthesizer would be 96. The velocity controls the loudness of the note and typically is a measure of how hard a key was struck. Velocities can range from 0 to 127. Enter:

```
50  100  MIDI.NOTEON   ( note=50, velocity=100 )
```

You should now hear a sustaining note. If not, check your MIDI connections, audio connections, and volume levels and repeat the command.

Eventually you will want to turn this note off. Enter:

```
50  71  MIDI.NOTEOFF
```

You may be wondering why the OFF command has a velocity. This relates to how fast you "pull your finger off the key", and is part of the MIDI standard. Only a few synthesizers can use this number, an example being the Kawai K4.

Now let's overlap two notes (if your synthesizer is monophonic, this won't work). Enter:

```
54    70  MIDI.NOTEON
59   110  MIDI.NOTEON  ( hear 2 notes, 2nd one should be loudest)
54     0  MIDI.NOTEOFF ( first note turns off )
59     0  MIDI.NOTEOFF ( silence )
```

There is a handy command that will turn off the last note played without you having to type it in again. Enter:

```
62  90  MIDI.NOTEON
MIDI.LASTOFF
```

For convenience, let's make a word that will play a note then turn it off.

Enter:

```
: BANG     ( note  -- ,  play a note for 1/4 second )
   100  MIDI.NOTEON     ( play it with velocity value of 100 )
   250  MSEC      ( wait roughly 250 milliseconds )
   MIDI.LASTOFF   ( turn off note )
;
56  BANG    ( test it, should play 1 short note )
```

Now let's use this word in another program. Let's write a program to play a random note every quarter second.

```
:  RAND.NOTES   ( -- , play random notes )
   BEGIN      ( start looping )
      96   36  WCHOOSE  ( select random note between 96 and 36 )
      BANG        ( play it )
      ?TERMINAL   ( key hit?)
   UNTIL
;
RAND.NOTES
```

(Note: Refer to the chapter of this manual called "HMSL Utilities (Distribution Functions)" for a complete description of **WCHOOSE**, and the related word **CHOOSE**).

Let's now write a word called MIDI.SCALE, which will produce an ascending scale using a different patch for each note. Try editing this in a file so you can experiment with changing it.

```
ANEW TASK-MIDI_SCALE
VARIABLE TEMPO
: MIDI.SCALE ( start end -- , play ascending scale )
   1+ SWAP   ( put in proper order for DO )
   DO
      I 100 MIDI.NOTEON ( turn on )
```

```
      TEMPO @ DELAY  ( wait )
      I 0 MIDI.NOTEOFF  ( turn off same note )
      TEMPO @ DELAY
      I MIDI.PRESET  ( change patch )
   LOOP
;
10 TEMPO ! (1/6 of a second )
40 60 MIDI.SCALE
```

Note that in the above example, the indices of the DO ... LOOP should not exceed the number of presets available on the synthesizer in use.

Tutorial 2: Pitch Bend

In many acoustic instruments, it is possible to bend pitches up or down as a performance gesture. Guitar players, violinists, trombonists and others often change slightly the pitch of the note(s) being played. Keyboard players (pianists, organists, etc.) have generally not had a way to bend pitch. Now, however, almost all MIDI keyboards have a lever or a wheel next to the keys that can be used to bend pitch. Moving this wheel will cause a MIDI *Pitch Bend* message to be sent. These messages are channel specific. Thus you could have a different amount of pitch bend for each channel. Here is an experiment. Select a preset/program/voice on your synthesiser that has a sustained sound, eg. flute. Now enter:

```
60 64 MIDI.NOTEON  ( play middle C )
```

While this is sounding, enter:

```
HEX
800 MIDI.PITCH.BEND \ pitch goes up
1000 MIDI.PITCH.BEND \ pitch goes up more
0 MIDI.PITCH.BEND     \ pitch should be back to normal
-1000 MIDI.PITCH.BEND  \ pitch goes lower
DECIMAL
```

[Note: there is a lower level word called MIDI.BEND that takes raw pitch bend values. With MIDI.BEND, a value of $2000 means no pitch bend.]

Alternative Tuning using Pitch Bend

The amount that the pitch raises or lowers for a given pitch bend value can be set in the synthesiser. For information on how to set the range, look in your synthesiser's user manual. For this next experiment, set the pitch bend range to one (1) semitone. This will allow us to use the full pitch bend range to move up or down one semitone. Using this technique we can produce many pitches between the normal 12 tone equal tempered pitches. Thus we can experiment with *alternative tunings* or *microtuning*. One simple tuning is called *quarter tone tuning.* This could also be referred to as *24 tone equal temperament*.

To play a note that is halfway between two "regular" notes, we can send a pitch bend value that is halfway between zero and the maximum. The maximum is hexadecimal $1FFF so we can use $1000 as a halfway value. $1000 is 4096 decimal. Enter the following words into a file so that you can experiment with them.

```
ANEW TASK-QTR_TUNING

$ 2000 2/ CONSTANT BEND/I

: I>NOTE+BEND  ( note-index -- note bend )
   2 /MOD  ( -- remainder quotient=note )
   SWAP BEND/I *
;
: QTR.NOTEON ( index velocity -- , quarter note on )
   SWAP I>NOTE+BEND  ( -- velocity note bend )
   MIDI.PITCH.BEND
   SWAP MIDI.NOTEON
```

```
;
: QTR.NOTEOFF ( index velocity -- , quarter note off )
   SWAP I>NOTE+BEND
   MIDI.PITCH.BEND
   SWAP MIDI.NOTEOFF
;
```

Compile this code, then test it interactively. Enter:

```
0SP 120 I>NOTE+BEND .S  ( prints: 60 0 )
```

Thus 120 corresponds exactly to a middle C. Now enter:

```
0SP 121 I>NOTE+BEND .S  ( prints: 60 4096 )
0SP 122 I>NOTE+BEND .S  ( prints: 61 0 )
```

Notice how this word calculates the "normal" MIDI note value and pitch bend value needed to generate indexed quarter tones. To hear these, enter:

```
120 QTR.NOTEON  ( listen )
120 QTR.NOTEOFF ( turn it off )
121 QTR.NOTEON ( listen )
121 QTR.NOTEOFF ( turn it off )
```

To hear a quarter note scale, add this word to your file with the other words.

```
: QTR.SCALE  ( start end -- )
   1+ SWAP
   DO I 64 QTR.NOTEON
      10 DELAY
      I 64 QTR.NOTEOFF
   LOOP
;
```

Compile the file. To hear an octave of quarter tones, enter:

```
120 145 QTR.SCALE
```

Remember that your synthesiser must be set to 1 one semitone bend range for this to be in tune.

There are some limitations to this technique. Because the pitch bend affects every note on the channel, this only works with one note per channel. If you want microtonal polyphony, you will need to use multiple channels. Many synthesizers have support for alternative tunings, eg. Yamaha TX81Z, Yamaha FB-01, Ensoniq EPS, etc. Most of these are accessed via system exclusive messages. Carter Scholz has proposed a standard for tuning which has been adopted and which we hope will be supported by the manufacturers. 12 tone equal temper has become very common in western music mainly because it is convenient. Electronic instruments will allow musicians to rediscover alternative tunings which are often preferable. The various forms of just intonation, for example, generally sound more harmonious to the ear.

Hear are some experiments you may want to try.

1) Modify the above example to implement a 1/8th tone scale (48 tone equal temperament).

2) Use multiple channels to play microtonal chords.

3) Write an interpreter that plays quarter tones, or define a new MIDI Instrument class that plays quarter tones.

Tutorial 3: Control Changes, Modulation, Sustain

Pitch bend is not the only way to affect the sound of an instrument. Other techniques include modulation wheels, sustain pedals, breath controllers, pedals, portamento time, etc. These are implemented as *MIDI Controls*.There are 128 MIDI controls numbered from 0 to 127. They can have values ranging from 0 to 127. Controls are channel specific. Some of the controls have been standardized. For a complete list of MIDI controls refer to the MIDI Standard specification. Here is a partial list:

```
2 = modulation wheel
5 = portamento time
```

```
 7 = volume
64 = sustain pedal , 0=off, 127 = on
65 = portamento pedal, 0=off, 127 = on
```

To send a control change use this word:

```
MIDI.CONTROL  ( control# value -- )
```

For example, to send a mod wheel value of 50, enter:

```
2 50 MIDI.CONTROL
```

Some presets may not respond to various controllers. Try different presets, different controls, and different synthesizers until you get something to work. Some synthesizers will let you arbitrarily assign a controller to some timbral parameter. These are often called X controls. Check the user manual of your synthesiser for details on how it responds to controller messages.

To change the volume of a channel, enter:

```
7 20 MIDI.CONTROL   ( really quiet )
60 64 MIDI.NOTEON
MIDI.LASTOFF
7 127 MIDI.CONTROL ( full volume again )
60 64 MIDI.NOTEON
MIDI.LASTOFF
```

Here is a fun word that randomly turns on and off the sustain pedal while playing random notes. It will occasionally build up clouds of notes then turn them all off. Play this on a polyphonic synthesiser. You may want to enter this in a file.

```
: MIDI.SUSTAIN  ( flag -- , pedal on or off )
   IF 127 \ map flag to value
   ELSE 0
   THEN
   64 SWAP MIDI.CONTROL
;
: RAND.SUSTAIN  ( -- )
   BEGIN
      24 CHOOSE 60 + \ two octave range above middle C
      64 MIDI.NOTEON  \ default velocity = 64
      10 DELAY MIDI.LASTOFF
\
\ about every 1/4th note, randomize sustain pedal
      4 CHOOSE 0=
      IF 2 CHOOSE MIDI.SUSTAIN
      THEN
      ?TERMINAL
   UNTIL
;
```

There are also other performance parameters you can send. Pressure is a message that corresponds to how hard you press on the keys. This is sometimes called *aftertouch.* Some keyboards, notably the Ensoniq EPS, support a special type of aftertouch called *polyphonic key pressure.* Instead of having one pressure value for the entire keyboard, each key can send an individual pressure value. This allows you to, for example, modulate some notes in a chord more than others. Experiment with the MIDI words in the glossary to learn what is possible.

Reference

Words that correspond to standard messages defined in the MIDI Specification are in the following glossary. Look also at the MIDI Utilities glossary for more handy but non-standard words. If you need to implement new standard messages, see the Low Level MIDI glossary.

MIDI.ALLOFF (-- , turn off all notes on current channel)

Any note that was turned on with MIDI.NOTEON and not turned off with MIDI.NOTEOFF stays on! Sometimes, if things go wrong, notes can be left on after a piece finishes. This word is useful in this embarrassing situation. This word derives the current channel from a variable called MIDI-CHANNEL. Many MIDI synthesizers, for example, the CZ-101, do not respond to this command.

MIDI.BEND (bend -- , send pitch wheel change command.)

Sends pitch bend value for current MIDI channel. This can be used for traditional articulation or for experimenting with different tuning systems. The amount that the pitch changes for a given change in this number depends on the synthesizer used and how the "bend range" is set. Bend values can range from hexadecimal 0 to 3FFF, the maximum 14 bit number. Hexadecimal 2000 is considered NO bend. For example:

```
: BENT.NOTE   ( note -- , play a note then bend it )
   100 MIDI.NOTEON ( turn on note, use a preset that sustains)
   [  HEX   ]   ( use hexadecimal numbers for input )
   100 0
   DO I 20 * 2000 + ( numbers from 2000 to 3Fe0 )
      MIDI.BEND 5 MSEC
   LOOP
   MIDI.LASTOFF
   [ DECIMAL ]  ( go back to decimal )
;
50 BENT.NOTE
```

See: MIDI.PITCH.BEND

MIDI.CHANNEL! (channel -- , sets channel for midi output)

Sets current MIDI-CHANNEL variable for use in multi-channel work. *Number should be between 1-16.* There is no MIDI channel 0 (according to the MIDI standard). To transmit in MIDI channel 7, enter: " 7 MIDI.CHANNEL! ". This tells HMSL that all successive MIDI data is sent to channel 7. When MIDI data is sent, channel number is encoded in the command in a 4 bit field. This field will be zero for channel 1, 1 for channel 2, ... and 15 for channel 16.

Associated Error Message: " MIDI.CHANNEL! Channel number out of range! "

MIDI.CLOCK (-- , send one MIDI clock tick)

This used for synchronizing with external sequencers by providing them with a clock "pulse".

MIDI.CONTINUE (-- , continue playing an external sequence)

Used with MIDI.START and MIDI.STOP .

MIDI.CONTROL (control# value -- , Send MIDI Control message)

Sends control bytes over MIDI (see the MIDI specification and the specific synthesizer documentation for how to use this). This is used in HMSL, for example, for implementing commands like MIDI.ALLOFF. It is extremely useful for defining your own custom MIDI instruments in HMSL. Here is an example:

```
: MIDI.ALLOFF $ 7B 00 MIDI.CONTROL ;
```

Here is an example of setting the volume (loudness) for an entire channel. Some synthesizers do not support this.

```
7  32  MIDI.CONTROL  ( set to 1/4 volume )
```

MIDI.END.SYSEX (-- , end a system exclusive message)

Sends a hex F7 byte then turns on running status.

MIDI.LOCAL.OFF (-- disallow keyboard input)

Removes ability to do local keyboard input.

MIDI.LOCAL.ON (-- , allow keyboard input)

MIDI.NOTE.PRESSURE (note pressure -- , set aftertouch pressure)

Sets polyphonic aftertouch pressure for the note. This is not a commonly supported feature on commercial keyboards. The Ensoniq EPS, however, does support polyphonic aftertouch.

MIDI.NOTEOFF (note velocity -- , turn off a note)

This word will turn OFF a note at the given velocity on the current channel. Note and velocity are both restricted to the MIDI range 0 -> 127 (7 bits). The velocity is ignored on most synthesizers for NOTEOFF commands. It is included here because the MIDI specification allows for it. A velocity of 64 is typically used. This word is used in MIDI.LASTOFF.

MIDI.NOTEON (note velocity -- , turn on a note)

This word will turn ON a note at the given velocity on the current channel. MIDI note and velocity are both restricted to the range 0 -> 127 (7 bits). Many synthesizers have a minimum note value of 36 which corresponds to the bottom C. Sending a note on message with a velocity of 0 will turn the note off.

See MIDI.NOTEON.FOR and MIDI.NOTEON.AT in the MIDI Utilities glossary.

MIDI.PITCH.BEND (bend -- , centered at zero)

This add $2000 to BEND and calls MIDI.BEND. Thus if you pass zero, you get NO bend. If you pass a negative number then pitch bends down. Positive numbers bend up. This is more intuitive then MIDI.BEND.

MIDI.PRESET (preset# -- , select a preset)

This command changes the preset patch on the current channel. The number of presets allowed depends on the synthesizer used. The allowable range is 1 to 128. MIDI.PRESET subtracts one from the preset before sending because many synthesizers add one when receiving. Not responsible for synthesizers that only receive program change in OMNI ON, and other such aberrations. This is a purely channel-specific preset (program) change. Also, this may not change what are often called "banks"—you do that on the synthesizer itself or via some control word you write. "Preset" is referred to as "program" in the MIDI spec.

MIDI.PROGRAM (program# -- , same as MIDI.PRESET)

MIDI.PRESSURE (pressure -- , channel based aftertouch)

This is more likely to be implemented on commercial synthesizers than polyphonic aftertouch.

MIDI.SONG.POINTER (position -- , send song position pointer)

Sends F2 common message. Used to set the song pointer in an external sequencer so you can start in the middle.

MIDI.SONG.SELECT (song# -- , send song selection)

Sends F3 common message. Used to select from among several songs in an external sequencer.

MIDI.START (-- , start external slave sequencers)

Sends hex FA MIDI real time message.

MIDI.START.SYSEX (-- , start a system exclusive message)

Turns off running status and sends a hex F0 byte. This is typically followed by a vendor code which you must send and then it's up to you and the company whose workers you are supporting by buying their stuff. Consult the manual for the synthesizer to which you are sending because they are all different. See MIDI.END.SYSEX and MIDI.XMIT.HILO . MIDI.START.SYSEX is very important and useful if you want to write system exclusive code for a specific MIDI device. This can be hairy stuff because many synthesizers are not very well documented in this regard.

SYSEX can also be a useful way to pass information between two computers using HMSL. That is, you can send commands, data, and so on back and forth over MIDI cables by just agreeing on a simple use of the SYSEX protocol.

Related Words: MIDI.XMIT.HILO MIDI.XMIT.LOHI MIDI.END.SYSEX MIDI.XMIT MIDI.FLUSH SHIFT AND OR

MIDI.STOP (-- , stop external slave sequencers)

MIDI Utilities

These words can be useful when working with MIDI.

14->7LO7HI (14_bits -- lo7 hi7)

MIDI data bytes can only use the bottom 7 bits. If the 8th bit is set, then the byte is a command and not data. This word takes a 14 bit data value and breaks it up into two legal 7 bit values. This is used internally by MIDI.BEND and is often used to generate system exclusive messages.

7LO7HI->14 (lo7 hi7 -- 14_bits)

BYTE->HILO (byte -- hi lo , breaks byte into nibbles)

This is useful for sending system exclusive data. A byte is 8 bits. A nibble is 4 bits.

GET.HIGH.BITS (addr count -- highbits)

Extract high bits from string. These are used for packing and unpacking 8 bit data into 7 bit data. Used mostly when sending system exclusive messages. This is part of the MIDI File transmission standard.

HILO->BYTE (hi lo -- byte , packs two nibbles into one byte)

MIDI.CLEAR (--)

Clears any bytes in input buffer by repeatedly calling MIDI.RECV .

MIDI.CVMDIS (cvm_byte -- , display name of command)

Displays that name of the command whose channel voice message byte is on the stack. Used internally by MIDI.SCOPE.

MIDI.CZ.KILL (-- , turn off notes on CZ-101)

Turns off CZ-101 keyboard, restores use of keyboard. Specifically useful for CZ-101/1000; don't quite know what it will do on other synthesizers. (Note: a great deal of early HMSL MIDI words were tested on the CZ-101, since it was the cheapest around at the time. We have left the two specific CZ words in just for nostalgia ...).

MIDI.CZ.KILLALL (-- , kill all channels on CZ-101)

Turns off all channels of CZ-101/1000 keyboard, restores use of all channels when done.

MIDI.KILL (-- , kill all notes on current channel)

Used for devices that do not support MIDI.ALLOFF. Turns off all notes that are playing on the current channel by sending 128 MIDI.NOTEOFF commands.

MIDI.KILLALL (-- , kills all channels)

Turns off all notes, hopefully, on all channels. This word sends Note Off commands for every possible note on all channels. This will work for all synthesizers except ones that cannot handle high data rates. It takes several seconds to execute.

MIDI.LASTOFF (-- , turn off last note in channel)

When you do a MIDI.NOTEON, HMSL keeps track of the last note played for each channel. This word will turn off that note. It is an alternative to MIDI.NOTEOFF. This works best if you are doing monophonic melodies in a channel; otherwise some notes will hang. For example, here's a word that does a glissando on a given MIDI channel:

Here is an example:

```
: MIDI.GLISS
   96 36
   DO
      I 64 ( note=i velocity=64 )
      MIDI.NOTEON
      100 MSEC MIDI.LASTOFF
```

```
      LOOP
;
```

MIDI.MONITOR (--)

Print all bytes received as hex numbers until a key is pressed.

MIDI.NORMALIZE (-- , reset controllers on all channels)

This is handy if you have been messing with controllers and want to put things back to normal. It is defined as:

```
: MIDI.NORMALIZE  ( -- , reset controllers on all channels )
   midi-channel @
   17 1
   DO
      i midi.channel!
      0 midi.pitch.bend  \ zero out pitch bend
      7 127 midi.control  \ full volume
      1 0 midi.control  \ modulation wheel off
      5 0 midi.control  \ portamento time
      64 0 midi.control \ sustain pedal off
      65 0 midi.control \ portamento off
      0 midi.pressure
   LOOP
   midi.channel!
;
```

MIDI.NOTEON.FOR (note velocity ontime --)

This turns a note on at the current virtual time that lasts for ONTIME ticks. Uses Event Buffering. The virtual time is NOT advanced by this word.

MIDI.NOTEON.AT (note velocity time --)

This turns on a note at the time given. You can turn it off using a velocity of zero. Uses Event Buffering.

MIDI.PANIC (-- , send ALLOFF to all 16 channels)

Turn off all notes on synthesizers that respond to the ALLOFF message.

MIDI.RECALL.BEND (-- last-bend , for channel)

MIDI.RECALL.PRESET (-- last-preset , for channel)

Simply calls MIDI.RECALL.PROGRAM .

MIDI.RECALL.PROGRAM (-- last-program , for channel)

MIDI.RECALL.NOTE (-- last-note, for channel)

The last note turned on is stored for each channel. This information is used by MIDI.LASTOFF.

MIDI.RECALL.VEL (-- lastvelocity , velocity of last note)

This is useful for doing crescendi, decrescendi, or any sort of velocity function.

MIDI.SCOPE (-- , dump MIDI input to screen)

Dumps MIDI input until user hits any key. Useful for seeing what kinds of data transmission is occurring.

MIDI.THRU (--)

Passes all bytes received directly to output until a key on the ASCII keyboard is pressed. This allows you to play a synthesizer attached to the output of your computer using a MIDI keyboard attached to the input of your computer without having to change any cables.

PUT.HIGH.BITS (addr count highbits --)

Put high bits back on a string of characters.

MIDI Tests

These words were written while testing and experimenting with MIDI. You may find them useful. Don't forget to set the MIDI channel to some appropriate value for your studio. See MIDI.CHANNEL! .

MIDI.BLAST (note -- , keeps hitting the note)

Keeps playing the note until the user presses a key.

MIDI.CONT (-- , continuous stream for hardware testing)

Plays a single note until the user presses any key. For test purposes only. Will just play one note as quickly as possible. This was written for debugging MIDI serial interfaces by looking at the data with an oscilloscope.

MIDI.ORGAN (-- , Use computer keyboard like an organ.)

This frivolous word will sound a MIDI note corresponding to the ASCII value of the key you press. You can get out by hitting the 'q' key. This is useful for testing your hardware connections and sound system before trying anything tricky. Velocity sensitive computer keyboards are not supported in this release. Note this is purely a test word, but a useful one as it will tell you if your cables are ok, and that the synthesizer is set up properly, etc.. That is, if this doesn't work, check lower level possible problems!

MIDI.SEQOUT (-- , play 5 notes)

A simple sequence which plays five notes.

MIDI.TEST (-- , quick MIDI test.)

Turns one note on each of channels 1, 2, and 3, and gives a message that they will turn off when the user presses any key.

MIDI.TESTPR (N -- , tests presets)

Plays sequence using N preset on current MIDI channel. This is useful when deciding which presets to use.

MIDI.WALK (N -- , random walk)

A simple test piece. Plays a simple "melody" with length N. Can be aborted by pressing any key.

Low-Level MIDI Words

These words will only be needed by users who want to extend the current set of commands. Look at the source code in **H:MIDI** for examples of using these words to implement user level MIDI commands.

MIDI-CHANNEL (-- addr)

Variable which keeps track of current MIDI channel in HMSL. To change the current channel, set this variable to a number from 1 to 16. The channel should usually be set using the word MIDI.CHANNEL! (see below) which checks for valid ranges and does some other housekeeping.

MIDI.COMMON (status -- , send status byte for common message)

Used for implementing commands like MIDI.SONG.SELECT

MIDI.CVM (status-byte -- , send channel voice message)

For internal use. This ORs the status with the channel, then transmits it. The value transmitted is saved in the MIDI-LAST-CVM variable. If the variable MIDI-IF-OPT is true, then transmission will only take place if the new message is different from the last. This is known as "running status" which cuts down the number of bytes transmitted.

MIDI.CVM+1D (data status-byte -- , send cvm + 1 data byte)

Does the following: MIDI.CVM, MIDI.DATA.XMIT, and MIDI.FLUSH. Result is to send the cvm plus one data byte.

MIDI.CVM+2D (data1 data2 status-byte -- , send cvm + 2 data bytes)

Like MIDI.CVM+1D, but sends two data bytes. As an example, a MIDI note off command is defined as a hex byte 8x where x is the channel number minus one, followed by the note and the velocity. For channel=3, note=50, velocity=64, the MIDI byte stream would be (hex) 92,50,64. MIDI.NOTEOFF is defined as follows:

: MIDI.NOTEOFF (note velocity --) 80 MIDI.CVM+2D ;

MIDI.CVM+3D (data1 data2 data3 status-byte -- , cvm + 3 data bytes)

Like MIDI.CVM+1D, but sends three data bytes.

MIDI.ORCH (status -- status_OR_channel)

Fetches value of MIDI-channel variable and ORs it with status. This makes it a valid, complete channel voice message. This is used to construct commands like

MIDI.DATA.XMIT (data -- , send valid data)

Uses MIDI.XMIT to send only valid data. ANDs data with hex 7F.

MIDI.REAL.TIME (byte -- , send real time message)

Used for implementing commands like MIDI.CLOCK .

MIDI.XMIT.HILO (byte -- , transmit byte as two nibbles, hi first)

This is useful in building system exclusive words that dump data to synthesizers.

MIDI.XMIT.LOHI (byte -- , transmit byte as two nibbles, lo first)

This is useful in building system exclusive words that dump data to synthesizers.

Host-Dependent MIDI Words

The above words are all host-independent. They should run on any host (specifically, either the Macintosh or the Amiga) that can support the following host dependent words. The following words provide the lowest level of MIDI data transmission. The source code for these words is completely different on the Macintosh and Amiga versions of HMSL, although their usage is the same. They are the lowest level of the MIDI toolbox and will only be needed if you want to write your own commands, eg. system exclusive commands.

MIDI-RECV-SIZE (-- addr , variable containing buffer size)

MIDI-XMIT-SIZE (-- addr , variable containing buffer size)

When the MIDI system is initialized, these variables are read to find out how big to make the Serial I/O buffers. You can change them then reinitialize the MIDI system to get bigger buffers. The default size is 2048 bytes.

```
6000 MIDI-RECV-SIZE !
MIDI.TERM  MIDI.INIT   ( reallocate buffers )
```

To make this change permanent, run HMSL but do *not* initialize HMSL. Set these two variables to what you want, then use SAVE-FORTH to save the changes. You can then enter HMSL.INIT and run with the bigger buffers.

MIDI.FLUSH (-- , flushes all data to output.)

Bytes sent using MIDI.XMIT are sometimes held in an internal buffer to optimize the serial I/O. If you are using MIDI.XMIT, call MIDI.FLUSH after you have sent the last byte in a command.

MIDI.RECV (-- byte true | false)

If a byte has been received from MIDI input, it is placed on the stack with a true flag. If no data is available a false is left. See MIDI.TIME@

MIDI.RECV? (-- flag , have any bytes been received?)

Returns true if any bytes are in MIDI input buffer.

MIDI.RTC.TIME@ (-- realtime , time when last byte received)

This returns the "real time" that the last MIDI byte was received at the serial port. Call this after MIDI.RECV when you are recording MIDI data and want accurate timing.

MIDI.SER.INIT (-- , initialize low level serial interface)

Called by MIDI.INIT . Internal use only.

MIDI.SER.TERM (-- , terminate low level serial interface)

Called by MIDI.TERM . Internal use only.

MIDI.TIME@ (-- time , time when last byte received)

This returns the "advance time" that the last MIDI byte was received at the serial port.

MIDI.XMIT (byte -- , output a byte over the MIDI line)

This is the basic output word used to build the specific MIDI commands. You may need to call MIDI.FLUSH for actual serial output to occur.

Macine Specific Routines (MIDI Manager, etc.)

For the Macintosh and Amiga, there are a number of very useful and important host-specific MIDI words and utilities that are documented in the individual *Macintosh* or *Amiga HMSL Supplements*. For example, the Macintosh makes use of the *Apple MIDI Manager*, and HMSL supplies some routines for changing the size of its buffer. If you are working extensively with MIDI programming, it will be useful for you to familiarize yourself with the specifics of the machine you're working on.

MIDI Input Parsing

Introduction

If you want to interpret MIDI input from a keyboard or other MIDI source, the easiest way is to use this **MIDI parser**. When you turn the **MIDI.PARSER on**, HMSL looks at the *MIDI input stream*. If there is any data coming in, it determines what type of MIDI command it is, then collects the appropriate number of bytes. Once it has the complete data for an incoming command it passes that data to a word of your choice.

For example, if MIDI.PARSE sees a MIDI Note On command, it will collect two data bytes. It then calls the word whose CFA is stored in MP-ON-VECTOR . The default word is 2DROP .

MIDI Parser User Vectors

The vectors that can be set and the number of bytes passed are:

Variable Name	Number of Bytes	MIDI Code (Hex)
MP-ON-VECTOR	passes 2	90
MP-OFF-VECTOR	passes 2	80
MP-NOTEPR-VECTOR	passes 2	A0
MP-CONTROL-VECTOR	passes 2	B0
MP-PROGRAM-VECTOR	passes 1	C0
MP-PRESSURE-VECTOR	passes 1	D0
MP-BEND-VECTOR	passes 2	E0
MP-SYSTEM-VECTOR	passes 0	Fx

MP-SYSTEM-VECTOR, by default, contains a word which further parses the MIDI System Messages as follows:

MP-SYSEX-VECTOR	passes 1	F0
MP-F1-VECTOR	passes 1	F1
MP-POINTER-VECTOR	passes 2	F2
MP-SELECT-VECTOR	passes 1	F3
MP-F4-VECTOR	passes 0	F4
MP-F5-VECTOR	passes 0	F5
MP-TUNE-VECTOR	passes 0	F6
MP-EOX-VECTOR	passes 0	F7
MP-CLOCK-VECTOR	passes 0	F8

```
MP-F9-VECTOR          passes 0       F9
MP-START-VECTOR       passes 0       FA
MP-CONTINUE-VECTOR    passes 0       FB
MP-STOP-VECTOR        passes 0       FC
MP-FD-VECTOR          passes 0       FD
MP-SENSING-VECTOR     passes 0       FE
MP-RESET-VECTOR       passes 0       FF
```

For a complete description of these codes, see the International MIDI Association MIDI Standard (see bibliography).

MIDI Parser Utilities

MIDI.PARSE (-- , Parse MIDI input if any.)

When the MIDI Parser is on, this word is called repeatedly from the top level of HMSL. If the user wants to use the MIDI Parser, but not HMSL per se, then this word can be called from a BEGIN ...UNTIL loop or similar control structure.

MIDI.PARSE.BYTE (byte --)

Parse one byte of MIDI data received from somewhere. This is called by MIDI.PARSE and also by the MIDI File reader. This is not normally called by the user.

MIDI.PARSER.ON (--)

Turns on MIDI Parser. HMSL will now call MIDI.PARSE.MANY in its loop.

MIDI.PARSER.OFF (--)

Turns MIDI Parser off.

MIDI.PARSE.LOOP (-- , calls MIDI.PARSE in a loop)

Test word for seeing if your vectors work! Hit any key to stop.

MIDI.PARSE.MANY (--)

This word, actually used in the top level of HMSL, will continue to parse as many MIDI messages as are in the buffer. The maximum number of messages it will parse is set in the variable MIDI-PARSE-MAX, default of 4. This word will provide better response to MIDI input than MIDI.PARSE.

MP.CHANNEL@ (-- channel-of-last-MIDI-message)

This is used to distinguish messages from multiple inputs to the MIDI parser.

MP.GET.ADDR# (addr count --)

Returns address of the complete MIDI packet. Can be used with MIDI.PACK. See chapter on Recording and Sequencing.

MP.RESET (-- , reset vectors to their original state)

This should be done before and after using the MIDI Parser to avoid bugs caused by executing the wrong function.

MP-xxx-VECTOR (-- addr , location of parser vector for function xxx)

"xxx" is the name of the MIDI function to be vectored. The actual names can be found in the table above. The MIDI Parser can be instructed to use a custom function by placing its CFA at the address returned. The custom word must consume the appropriate number of bytes as given in the table. For example:

```
VARIABLE INPUT-PRESSURE
: MY.PRESSURE  ( pressure -- , eat one byte )
    INPUT-PRESSURE !
;
'C MY.PRESSURE  MP-PRESSURE-VECTOR !
```

MIDI Parser Examples

Here is an example of a simple MIDI scope done using this parser.

```
: DUMP.NOTE.ON ( note velocity -- , print note played )
   ." ON " swap . . CR
;

MP.RESET     ( make sure vectors are set to default )
'C DUMP.NOTE.ON MP-ON-VECTOR !
MIDI.PARSE.LOOP   ( use low level loop, play keyboard now )
MP.RESET   ( restore old vectors )
```

Here is a fun example that inverts the keyboard and interferes with selecting presets. Enter this in a file:

```
ANEW TASK-NOTE_INVERTER
: INVERT.NOTE  ( note velocity -- inverted_note velocity )
   128 ROT - SWAP
;
: INVERT.NOTEON ( note velocity -- , play inverted )
   INVERT.NOTE MIDI.NOTEON
;
: INVERT.NOTEOFF ( note velocity -- , turn off inverted )
   INVERT.NOTE MIDI.NOTEOFF
;
: RANDOM.PROGRAM ( program  -- , choose nearby program )
\ "Program" is sometimes known as "Preset"
   9 CHOOSE 4 -  ( pick random offset )
   +  ( add random offset to program value )
   1 127 CLIPTO  ( constrain to valid range )
   MIDI.PRESET
;
: INVERT.PLAY ( -- )
   MP.RESET
   'C INVERT.NOTEON MP-ON-VECTOR !
   'C INVERT.NOTEOFF MP-OFF-VECTOR !
   'C RANDOM.PROGRAM MP-PROGRAM-VECTOR !
   MIDI.PARSER.ON ( turns parser on )
   HMSL      ( play and watch notes printed on screen )
   ( after quitting HMSL, set vectors back to default )
   MP.RESET
;
```

Then compile the file and enter:

```
INVERT.PLAY
```

MIDI Parsing Advice

There are a few things that should be kept in mind here. Many synthesizers *do not normally send Note OFF messages*. They often just send a Note ON message with a velocity of zero. If you want something to only occur when a note is pressed then you must check for nonzero velocities.

Also remember that some quantities are converted to a *zero based range* when transmitted. The MIDI Parser does not convert back. MIDI program changes, for example, are transmitted at a value of one less than the specified value. If you select program 7 on your synthesizer, the MIDI Parser will give you a value of 6. This is done so that humans can refer to programs 1 to 128, but the computer can refer to them as 0 to 127, which fits in a 7 bit field. There are more examples of using the MIDI Parser in the "demo_action" and "master_slave" sample programs.

The MIDI Parsing usually works best if you have a separate keyboard controller and sound generator. If you are sending notes back to the same synthesizer, try MIDI.LOCAL.OFF so that you only hear the notes from HMSL and not what you are directly playing.

MIDI *bend messages come in as two bytes*. Convert them to a single value using 7LO7HI->14. Here is an example of a word that will print MIDI bend values.

```
: BEND.PARSER  ( lo hi -- , process bend message )
   7LO7HI->14  ( pack bytes together )
   .HEX  CR  ( just print result for now )
;
'C BEND.PARSER  MP-BEND-VECTOR !
```

Try doing pitch bend with this installed. Notice that when there is NO bend, the value is $2000.

System Exclusive

Due to the current diversity in system exclusive implementations in various MIDI devices, there is at present no standard system exclusive library in HMSL. However, several system exclusive implementations have been written for various devices (Roland DEP-5 and SRV-2000, Yamaha FB-O1 and several others) in HMSL by various composers, and these have proved both easy to write and powerful. In one such piece, a nine-dimensional shape was created which controlled an instrument which drove the parameters of the Roland DEP-5 in real time (like reverb time, chorusing depth and modulation, gate time, and others). For more information on system exclusive techniques in HMSL for use with "performance MIDI" capable devices, please contact the authors. Also look in the SySex directory on the HMSL_User disk for some examples. The HMSL BBS also has many system exclusive examples. See your release notes for information on the HMSL BBS.

Note that the MIDI.PARSER does support all the MIDI Standard system exclusive codes. Also note that by using the words MIDI.START.SYSEX , MIDI.XMIT.HILO , MIDI.XMIT.LOHI , MIDI.XMIT , MIDI.FLUSH and MIDI.END.SYSEX the user may easily write any system exclusive application. In addition, simple voice editors can be written using the control grid objects in conjunction with system exclusive libraries.

SYSEX Example

Here is an example of code that can be used to modify voice parameters in a Yamaha FB-01. The message we must send consists of the following bytes (in hex):

```
F0  43  10+ch  15  40+p#  ll  hh  F7
```

The p# is the parameter number expressed as an offset in the voice data block. The 'll' and 'hh' are the low 7 bits and the high 7 bits of the new value. All numbers sent within a System Exclusive must be between 0 and 127. Anything higher will have the high bit set and be considered a command byte.

```
: FB.START.SYSEX ( -- , send first two bytes, $F0 $43 )
   midi.start.sysex
   43 midi.xmit  ( Yamaha vendor code )
;

: FB.VOICE.DATA! ( value offset -- , change voice data parameter)
   fb.start.sysex
   10 midi.orch midi.xmit  ( send on current channel )
   15 midi.xmit
   40 + midi.data.xmit  ( convert offset to parameter # )
   midi.xmit.lohi  ( send new value as two bytes )
   midi.end.sysex
;

: FB.LFO.SPEED ( rate -- , set voice LFO rate )
   8 fb.voice.data!
;
```

Standard MIDI Files

Standard MIDIFiles allow the *interchange of data* between various MIDI application programs. Using a MIDIFile, you can import musical information from sequencers, etc, and export information to notation software, etc.

MIDIFiles are files that contain an informative header followed by one or more tracks. Tracks consist of a stream of MIDI events with a time stamp. There are also other events, called MetaEvents, for things like lyrics, copyright information, tempo changes, etc.

MIDIFiles come in three Formats: 0, 1 and 2.

Format 0 = One track containing everything.

Format 1 = Multiple parallel tracks.

Format 2 = Multiple tracks with no specific relation between them.

Most Sequencers will save in Format 1.

We strongly recommend looking at the source code for this MIDIFile support to see how it works. You may want to create a modified version of this code to suit your particular needs.

You will probably also want to get a copy of the Standard MIDI File spec from the International MIDI Society.

Loading MIDIFIle support.

To load the MIDIFile package, enter:

```
INCLUDE HT:MIDIFILE
```

MIDI File Glossary

You will notice that some of these words take a filename as <filename> while some take it as $filename. Consider MIDIFILE0{ and $MIDIFILE0{ . There stack diagrams are:

MIDIFILE0{ (<filename> --)

$MIDIFILE0{ ($filename --)

A parameter in parentheses implies that the name follows the word, for example:

```
MIDIFILE0{  myfile    SHEP  }MIDIFILE0
```

If you tried to put this in a colon definition, it would not work! For example:

```
: FOO MIDIFILE0{  myfile    SHEP  }MIDIFILE0 ;  \ Won't work!!!
```

The compiler will not recognize MYFILE. This is because MIDIFILE0{ reads the name of the file from input only when it is executed. Thus this word is legal:

```
: MOO ( <filename> -- )
   MIDIFILE0{  SHEP  }MIDIFILE0
;
MOO myfile  \ MIDIFILE0{ grabs filename now
```

If you want to specify a filename in a colon definition, you must use the dollar sign version of a word. An examples is LOGTO versus $LOGTO and FOPEN versus $FOPEN. This is legal:

```
: LOO " myfile" $MIDIFILE0{  SHEP  }MIDIFILE0 ;  \ legal
```

When you define words that take a filename, you should always have a $ version for this reason. It is then very easy to define a word that takes the filename from input. The definition of MIDIFILE0{, for example, is:

```
: MIDIFILE0{  ( <filename> -- )
   FILEWORD $MIDIFILE0{
;
```

Please note that there are some undocumented but very useful commands in the file **HT:MIDIFILE**. Please read the file.

Reading MIDI Files

The file is parsed using a vectored parser. This means that you define words that are called as the file is read. Your words can then do whatever you want with the data.

To read a MIDI File, we take advantage of the MIDI Parser. Data is read from the file and passed to the MIDI Parser. (Please read the above section in the manual on the MIDI Parser if you are not already familiar with it.) By putting your own functions in the MIDI Parser you can process the file as you wish. After a MIDI File is parsed the contents of the header will be placed in the variables MF-NTRKS, MF-FORMAT and MF-DIVISION.

MF-NTRKS (-- var-addr , number of tracks in file)

MF-FORMAT (-- var-addr , file format = 0 | 1 | 2)

MF-DIVISION (-- ticks/beat)

For a full explanation of this, see the MIDI File spec.

$MF.LOAD.SHAPE (track# $filename -- , load notes into MF-SHAPE)

See MF.LOAD.SHAPE. Use this when you want to pass a filename in a colon definition. See example.

MF.CHECK (<filename> -- , print out the contents of a file)

Try doing this to see what tracks are in a file before experimenting with the other MF words. We recommend studying the definition of this word as an example of how to parse a MIDI FIle.

MF.PARSE.EVENT (-- , MIDI Parse event)

Reads event from file then calls the MIDI Parser to process the event. To process meta events, system exclusive events and escape events, it will call the deferred words MF.PROCESS.META, etc.

MF.PARSE.TRACK (size track# --)

Analyse events in track. Calls MF.PARSE.EVENT. Sets MF-EVENT-TIME as it reads time from file.

MF.PROCESS.ESCAPE (size --)

Deferred word called by MF.PARSE.EVENTS. Set this to whatever you want for processing escape sequences from the file.

MF.PROCESS.META (size MetaID --)

Deferred word called by MF.PARSE.EVENTS. Set this to whatever you want for processing Meta Events.

MF.PROCESS.SYSEX (size --)

Deferred word called by MF.PARSE.EVENTS. Set this to whatever you want for processing system exclusive messages from the file.

MF.PROCESS.TRACK (size track# --)

Deferred word that is called by $MF.DOFILE when each track is encountered. You can use MF.PARSE.TRACK to break the track into events.

MF.LOAD.SHAPE (track# <filename> -- , load notes into MF-SHAPE)

This word loads the note information from the desired track into a shape called MF-SHAPE. The notes will be stored in expanded form with absolute time. If you want to copy the data to another shape use SH.COMPRESS.NOTES which converts the notes to packed form. See the shape chapter for information on SH.COMPRESS.NOTES.

LOAD.ABS.SHAPE (shape <filename> --)

Use MF.LOAD.SHAPE and CLONE: to load your shape.

To play these shapes you will need to set the player to absolute mode. Here is an example:

```
OB.SHAPE SH1
OB.PLAYER PL1
OB.MIDI.INSTRUMENT INS1
```

```
: READ.MF  ( -- , read a midifile into a shape )
   0 " hmf:simple.mf" $mf.load.shape  \ loads as absolute expanded
   mf-shape sh1 sh.compress.notes     \ convert to compressed
;

: PLAY.MF  ( -- , play the shape read from MIDIFile )
   sh1 ins1 build: pl1
   0 put.offset: ins1     \ 0 offset because stored as MIDI notes
   use.absolute.time: pl1 \ absolute not relative time
   3 put.on.dim: pl1      \ use ontimes of compressed notes
   pl1 hmsl.play
;
```

Example of Reading a MIDI File

This example just prints notes as it reads them from the file.

```
: DUMP.NOTE  ( note velocity -- )
   MF-EVENT-TIME @ .  ( print time )
   SWAP . . CR  ( print note vel )
;
: DUMP.TRACK ( size track# -- )
   DUP ."  Track = " . CR
   MF.PARSE.TRACK
;
:  MF.DUMP  ( -- , show notes in file )
   'C DUMP.NOTE MP-ON-VECTOR !
   ( set as many MP vectors as you want )
   'C DUMP.TRACK IS MF.PROCESS.TRACK
   MF.DOFILE
;
MF.DUMP  HMF:TEST.MF
```

Writing MIDI Files

These words support writing to a MIDIFile.

}MIDIFILE0 (-- , end a capture session)

See MIDIFILE0{

}MIDIFILE1 (-- , end a capture session)

See MIDIFILE1{

$CAPTURED>MIDIFILE0 ($filename -- , save captured in file)

Writes MIDI data captured using CAPTURE{ to the specified MIDIFile in format 0.

$CAPTURED>MIDIFILE1 ($filename -- , save captured in file)

Writes MIDI data captured using CAPTURE{ to the specified MIDIFile in format 1.

$MF.BEGIN.FORMAT0 ($filename -- position)

Start writing format0 file. Calls $MF.CREATE , writes header, and begins track. Returns position for MF.END.FORMAT0. Here is how it is defined.

```
: $MF.BEGIN.FORMAT0  ( $name -- pos , begin format0 file )
    $mf.create
```

```
            0 1 ticks/beat @ mf.write.header
            mf.begin.track  ( startpos )
        ;
```

$MIDIFILE0{ ($filename --)

Write following MIDI to a format 0 file:

" myfile" $MIDIFILE0{ 60 64 20 MIDI.NOTEON.FOR }MIDIFILE0

See MIDIFILE0{ and explanation of $ words at the beginning of this section

$MIDIFILE1{   ($filename --)

Similar to $MIDIFILE0{ except it creates a format 1 file.

$SAVE.ABS.SHAPE (shape $filename --)

Identical to SAVE.ABS.SHAPE except filename is passed on stack. These can be used inside a colon definition, where SAVE.ABS.SHAPE cannot be used. For example:

```
: SAVE.SH1  ( -- )
   sh1 " harddisk:mf1" $SAVE.ABS.SHAPE
;
```

$SAVE.REL.SHAPE (shape $filename --)

Identical to SAVE.REL.SHAPE except filename is passed on stack.

MIDIFILE0{   (<filename> --)

Reads the filename from the input stream and calls $MIDIFILE0{.

All subsequent MIDI output will be routed to a format 0 MIDI File and stamped with the current Virtual Time. The data is first collected in Packed MIDI form using CAPTURE{. This ensures that the data is sorted in time order. Then it is written to the file.

Here is an example:.

```
MIDIFILE0{ myfile  SHEP  }MIDIFILE0
```

Play with the shape editor, then quit. This file should be loadable into a sequencer.

To speed up this process, you may want to use the software clock instead of the hardware clock. When we call USE.SELF.TIMER, HMSL will advance the clock itself as it plays the piece. Thus it will run as fast, or as slow, as it can while still keeping everything in sync. Here is how to do that.

```
MF-ECHO ON  ( hear while saving )
INCLUDE HP:XFORMS
USE.SELF.TIMER
MIDIFILE0{ myfile  XF.PLAY  }MIDIFILE0
USE.HARDWARE.TIMER
```

MIDIFILE1{   (<filename> --)

Similar to MIDIFILE0{ except it creates a format 1 file.

MF.BEGIN.FORMAT0 (<filename> -- position)

Start writing format0 file. Calls FILEWORD to get the name then $MF.BEGIN.FORMAT0. Returns position for use by MF.END.FORMAT0 to set track size.

MF.BEGIN.TRACK (-- position , start a new track)

Save position for MF.END.TRACK. Use with Format 1 & 2 files.

MF.END.FORMAT0 (position --)

Close file created by MF.BEGIN.FORMAT0.

MF.BEGIN.TRACK (position --, end a track)

Use position from MF.BEGIN.TRACK.

MF.WRITE.EVENT (addr count time -- , to open file)

MF.WRITE.HEADER (format ntrks division --)

MF.WRITE.META (addr count event-type --)

Write meta event of COUNT bytes to the current file. See MF.WRITE.SEQ# for example. Uses MF-EVENT-TIME as time.

MF.WRITE.NOTEOFF (time note velocity -- , to open file)

MF.WRITE.NOTEON (time note velocity -- , to open file)

MF.WRITE.SEQ# (sequence# --)

Write sequence number to open file. Defined as:

```
:  MF.WRITE.SEQ#  ( SEQ# -- )
   MF-EVENT-PAD W!  ( store as two bytes )
   MF-EVENT-PAD  2  0  MF.WRITE.META
;
```

MF.WRITE.TEMPO (microseconds/beat --)

Write MetaEvent $51 to store tempo.

MF.WRITE.TIME (abstime -- , write as virtual length number)

Convert time since beginning of track to time since last event, then write to file as VLN.

MF.WRITE.TIMESIG (numer denom clocks 32nds --)

Write a metaevent $58 to the file to store a time signature. The parameters are:

numer = numerator of time signature, eg. **7** / 4

denom = denominator of time signature, eg. 7 / **4**

clocks = MIDI CLocks per metronome clock

32nds = number of 32nd notes per 24 MIDI Clocks

SAVE.ABS.SHAPE (shape <filename> --)

This saves a shape with absolute time format to a MIDIFile in format0. This assumes each note has a separate On and Off event and that times are ABSOLUTE, ie. ticks since start of player.

See: $SAVE.ABS.SHAPE

SAVE.REL.SHAPE (shape <filename> --)

This saves a shape with relative time format to a MIDIFile in format0. This assumes each note has one event and that times are RELATIVE, ie. ticks until next note. The ON time of each note is assumed to be 1/2 of the duration.

Transferring MIDI files between other programs and HMSL.

The MIDI Files created by HMSL can be read by other programs that follow the MIDI File Specification. These include most sequencers and notation programs. The most important issue in this process is timing. How do you generate notes using HMSL that will be interpreted as exact quarter notes, half notes,, etc.? If you are not careful, you will end up with very strange dotted triplets plus 64th note rests and other strange creatures. This is fine if that's what you intended but if you want whole notes, you should get whole notes.

The secret to this process is a field in the MIDI File header called **division** which defines the **number of ticks per beat**. When a program reads a MIDI File, it will save that division information. When it then sees a note-**on** followed by a note-**off**, it will compare the duration of the note to the ticks per beat and determine whether it is a quarter note or whatever.

The HMSL words that create MIDI Files use a variable called **TICKS/BEAT.** When they write a header, they use the value in this variable for the division parameter in the header. Thus by setting this variable and then generating a MIDI File whose notes are ratios of this number, you can generate exact note durations suitable for staff notation. Many composers have thus used HMSL to generate scores for human performers.

To avoid getting lots of undesired rests in the score, generate the notes as **legato**. In other words, the ON times should equal the duration. This can be accomplished, for example, by setting the duty.cycle in a player to 1 1. See PUT.DUTY.CYCLE:. In the Score Entry System you would enter the word LEGATO.

Another factor to consider is the tempo. By adjusting the real time clock rate, you can match tempi from other programs. The best way to determine what values to use is to create MIDI File using the other program and then examine it. Enter a few notes into your program then write out a MIDI File. Then use MF.CHECK to look at the division in the header using HMSL's MF.CHECK.

Let's suppose the tempo in the other program is defined as 120.00 beats per minute. Let's define the division as D in ticks/beat.

You should then set the real time clock to:

```
R = ticks/second = D * Tempo / 60
```

For a tempo of 120 and a D of 100 ticks/beat you would enter:

```
100 120 * 60 / RTC.RATE!
```

Example of Writing a MIDI File

Here is a sample sequence of events for writing to a file. This will write to the logical volume HMF: which is the MidiFiles directory on the HMSL_User disk. Feel free to write to somewhere else on disk.

```
: MAKE.MF ( -- )
   100 TICKS/BEAT !  \ define a quarter note
   " HMF:TEST.MF" $MF.BEGIN.FORMAT0
   MF-START-POS !  ( save starting position )
    0 50 64 MF.WRITE.NOTEON  ( write note )
   100 50 0 MF.WRITE.NOTEOFF  ( finish it )
   200 55 80 MF.WRITE.NOTEON  ( write note )
   400 55 0 MF.WRITE.NOTEOFF  ( finish it )
   MF-START-POS @ MF.END.FORMAT0
;
```

Now check it out. Enter interactively:

```
MAKE.MF  ( should write to disk )
MF.CHECK  HMF:TEST.MF
```

It should have displayed the file header and the note events.

If you read this file into commercial notation program, it should display as:

a quarter note at time 0

a quarter note rest at time 100

a half note at time 200

Example of writing random quarter and eighth notes in MIDI File

Here is a quick example that generates random quarter and eighth notes. Enter these in a file.

```
: RAND.NOTE  ( -- )
   12 CHOOSE \ one octave range
   60 +      \ start at Middle C
   50 30 CHOOSE +  \ random velocity
   MIDI.NOTEON  \ play note
\
   TICKS/BEAT @  \ use global definition of quarter note
   2 CHOOSE      \ divide beat by 2 roughly 1 out of every 2 notes
   IF 2/
   THEN
   VTIME+!  \ advance virtual time
\
```

```
      MIDI.LASTOFF \ turn off the note
   ;
   : RAND>FILE  ( -- , create MIDI FIle with random notes )
      120 TICKS/BEAT !  \ define quarter note
      " harddisk:myfile.mf" $MIDIFILE0{  \ use any filename you want
      RNOW  \ set virtual time to now
      32 0
      DO
         RAND.NOTE
      LOOP
      }MIDIFILE0
   ;
```

Compile the above, then enter at the keyboard:

```
RAND.FILE
MF.CHECK  harddisk:myfile.mf
```