MIDI documentation

Pete Goodliffe

21st October 1999

Contents

List of Tables

Chapter 1

Introduction to and description of MIDI

1.1 Introduction

MIDI is the acronym for Musical Instrument Digital Interface. MIDI enables synthesizers, sequencers, home computers, rhythm machines, etc to be interconnected through a standard interface.

Each MIDI equipped instrument usually contains a receiver and a transmitter. Some instruments may contain only a receiver or transmitter. The receiver receives messages in MIDI format and executes MIDI commands. It consists of an optoisolator, Universal Asynchronous Receiver/Transmitter (UART), and other hardware needed to perform the intended functions. The transmitter originates messages in MIDI format, and transmits them by way of a UART and line driver.

The MIDI standard hardware and data format are defined in this specification.

1.2 Conventions

Status and Data bytes given these Tables are given in binary, with most significant byte first.

Numbers prefixed by an "&" are in hexadecimal.

All other numbers are in decimal.

1.3 Hardware

The interface operates at 31.25 (+/- 1%) Kbaud, asynchronous, with a start bit, 8 data bits (D0 to D7), and a stop bit. This makes a total of 10 bits for a period of 320 microseconds per serial byte.

Circuit 5 mA current loop type. Logical 0 is current ON. One output shall drive one and only one input. The receiver shall be opto-isolated and require less than 5 mA to turn on. Sharp PC-900 and HP 6N138 optoisolators have been found acceptable. Other high-speed optoisolators may be satisfactory. Rise and fall times should be less than 2 microseconds.

Connectors DIN 5 pin (180 degree) female panel mount receptacle. An example is the SWITCHCRAFT 57 GB5F. The connectors shall be labelled "MIDI IN" and "MIDI OUT". Note that pins 1 and 3 are not used, and should be left unconnected in the receiver and transmitter.

1.3.1 Notes

1. Optoisolator is Sharp PC-900. (HP 6N138 or other optoisolator can be used with appropriate changes)
2. Gates "A" are IC or transistor.
3. Resistors are 5%

Cables shall have a maximum length of fifty feet (15 meters), and shall be terminated on each end by a corresponding 5-pin DIN male plug, such as the SWITCHCRAFT 05GM5M. The cable shall be shielded twisted pair, with the shield connected to pin 2 at both ends.

A "MIDI THRU" output may be provided if needed, which provides a direct copy of data coming in MIDI IN. For very long chain lengths (more than three instruments), higher-speed optoisolators must be used to avoid additive rise/fall time errors which affect pulse width duty cycle.

1.4 Data format

All MIDI communication is acheived through multi-byte "messages" consisting of one **Status byte** followed by one or two **Data bytes**, except Real-Time and Exclusive messages (see below).

1.5 Message types

Messages are divided into two main categories: Channel and System.

1.5.1 Channel

Channel messages contain a four-bit number in the Status byte which address the message specifically to one of sixteen channels. These messages are thereby intended for any units in a system whose channel number matches the channel number encoded into the Status byte.

There are two types of Channel messages: Voice and Mode.

Voice To control the instruments voices, Voice messages are sent over the Voice Channels.

Mode To define the instruments response to Voice messages, Mode messages are sent over the instuments Basic Channel.

1.5.2 System

System messages are not encoded with channel numbers.

There are three types of System messages: Common, Real-Time, and Exclusive.

Common Common messages are intended for all units in a system.

Real-Time Real-Time messages are intended for all units in a system. They contain Status bytes only - no Data bytes. Real-Time messages may be sent at any time - even between bytes of a message which has a different status. In such cases the Real-Time message is either ignored or acted upon, after which the receiving process resumes under the previous status.

Exclusive Exclusive messages can contain any number of Data bytes, and are terminated by an End of Exclusive (EOX) or any other Status byte. These messages include a Manufacturers Identification (ID) code. If the receiver does not recognize the ID code, it should ignore the ensuing data. So that other users can fully access MIDI instruments, manufacturers should publish the format of data following their ID code. Only the manufacturer can update the format following their ID.

1.6 Data types

1.6.1 Status Bytes

Status bytes are eight-bit binary numbers in which the Most Significant Bit (MSB) is set (binary 1). Status bytes serve to identify the message type, that is, the purpose of the Data bytes which follow the Status byte.

Except for Real-Time messages, new Status bytes will always command the receiver to adopt their status, even if the new Status is received before the last message was completed.

1.6.2 Running Status

For Voice and Mode messages only, when a Status byte is received and processed, the receiver will remain in that status until a different Status byte is received. Therefore, if the same Status byte would be repeated, it may (optionally) be omitted so that only the correct number of Data bytes need be sent. Under Running Status, then, a complete message need only consist of specified Data bytes sent in the specified order.

The Running Status feature is especially useful for communicating long strings of Note On/Off messages, where "Note On with Velocity of 0" is used for Note Off. (A separate Note Off Status byte is also available.)

Running Status will be stopped when any other Status byte intervenes, except that Real-Time messges will only interrupt the Running Status temporarily.

1.6.3 Unimplemented Status

Any status bytes received for functions which the receiver has not implemented should be ignored, and subsequent data bytes ignored.

1.6.4 Undefined Status

Undefined Status bytes must not be used. Care should be taken to prevent illegal messages from being sent during power-up or power-down. If undefined Status bytes are received, they should be ignored, as should subsequent Data bytes.

1.6.5 Data Bytes

Following the Status byte, there are (except for Real-Time messages) one or two Data bytes which carry the content of the message. Data bytes are eight-bit binary numbers in which the MSB is reset (binary 0). The number and range of Data bytes which must follow each Status byte are specified in the tables which follow. For each Status byte the correct number of Data bytes must always be sent. Inside the receiver, action on the message should wait until all Data bytes required under the current status are received. Receivers should ignore Data bytes which have not been properly preceeded by a valid Status byte (with the exception of "Running Status" above).

1.7 Channel modes

Synthesizers contain sound generation elements called voices. Voice assignment is the algorithmic process of routing Note On/Off data from the keyboard to the voices so that the musical notes are correctly played with accurate timing.

When MIDI is implemented, the relationship between the sixteen available MIDI channels and the synthesizers voice assignment must be defined. Several Mode messages are available for this purpose (see Table 3). They are Omni (On/Off), Poly, and Mono. Poly and Mono are mutually exclusive, i.e., Poly Select disables Mono, and vice versa. Omni, when on, enables the receiver to receive Voice messages in all voice Channels without discrimination. When Omni is off, the receiver will accept Voice messages from only the selected Voice Channel(s). Mono, when on, restricts the assignment of Voices to just one voice per Voice Channel (Monophonic). When Mono is off (= Poly On), any number of voices may be allocated by the Receivers normal voice assignment algorithm (Polyphonic.)

For a receiver assigned to Basic Channel N, the four possible modes arising from the two Mode messages are:

Mode	**Omni**	**Poly/Mono**	**Description**
1	On	Poly	Voice messages are received from all Voice channels and assigned to voices polyphonically.
2	On	Mono	Voice messages are received from all Voice Channels, and control only one voice, monophonically.
3	Off	Poly	Voice messages are received in Voice channel N only, and are assigned to voices polyphonically.
4	Off	Mono	Voice messages are received in Voice channels N thru N+M-1, and assigned monophonically to voices 1 thru M, respectively. The number of voices M is specified by the third byte of the Mono Mode Message.

Four modes are applied to transmitters (also assigned to Basic Channel N). Transmitters with no

channel selection capability will normally transmit on Basic Channel 1 (N=0).

Mode	Omni	Poly/Mono	Description
1	On	Poly	All voice messages are transmitted in Channel N.
2	On	Mono	Voice messages for one voice are sent in Channel N.
3	Off	Poly	Voice messages for all voices are sent in Channel N.
4	Off	Mono	Voice messages for voices 1 thru M are transmitted in Voice Channels N thru N+M-1, respectively. (Single voice per channel).

A MIDI receiver or transmitter can operate under one and only one mode at a time. Usually the receiver and transmitter will be in the same mode. If a mode cannot be honored by the receiver, it may ignore the message (and any subsequent data bytes), or it may switch to an alternate mode (usually Mode 1, Omni On/Poly).

Mode messages will be recognized by a receiver only when sent in the Basic Channel to which the receiver has been assigned, regardless of the current mode. Voice messages may be received in the Basic Channel and in other channels (which are all called Voice Channels), which are related specifically to the Basic channel by the rules above, depending on which mode has been selected.

A MIDI receiver may be assigned to one or more Basic Channels by default or by user control. For example, an eight-voice synthesizer might be assigned to Basic Channel 1 on power-up. The user could then switch the instrument to be configured as two four-voice synthesizers, each assigned to its own Basic Channel. Separate Mode messages would then be sent to each four-voice synthesizer, just as if they were physically separate instruments.

1.7.1 Power-up default conditions

On power-up all instruments should default to Mode #1. Except for Note On/Off Status, all Voice messages should be disabled. Spurious or undefined transmissions must be suppressed.

Chapter 2

Summary of Status Bytes

Status byte		# of data bytes	Description
Channel voice messages			
1000nnnn	(&8x)	2	Note Off event
1001nnnn	(&9x)	2	Note On event (velocity=0: Note Off)
1010nnnn	(&ax)	2	Polyphonic key pressure / after touch
1011nnnn	(&bx)	2	Control change
1100nnnn	(&cx)	1	Program change
1101nnnn	(&dx)	1	Channel pressure / after touch
1110nnnn	(&ex)	2	Pitch bend change
Channel mode messages			
1011nnnn	(&bx)	2	Selects Channel Mode
System Messages			
11110000	(&f0)	*****	System Exclusive
11110sss	(&fx)	0 to 2	System Common
11111ttt	(&fx)	0	System Real Time

Table 2.1: Summary of Status Bytes

2.1 Notes

Symbol	Description
nnnn	N-1, where N = Channel #, i.e. 0000 is Channel 1. 0001 is Channel 2. .... 1111 is Channel 16.
****	0iiiiiii, data, ..., EOX
iiiiiii	dentification
sss	to 7
ttt	to 7

Chapter 3

Channel Voice Messages

Status byte		Data bytes		Description
1000nnnn	(&8x)	0kkkkkkk	0vvvvvvv	Note Off (see notes 1-4) vvvvvvv: note off velocity
1001nnnn	(&9x)	0kkkkkkk	0vvvvvvv	Note On (see notes 1-4) vvvvvvv ¿ 0: velocity vvvvvvv = 0: note off
1010nnnn	(&ax)	0kkkkkkk	0vvvvvvv	Polyphonic Key Pressure (After-Touch) vvvvvvv: pressure value
1011nnnn	(&bx)	0ccccccc	0vvvvvvv	Control Change ccccccc: control # 0-121 (see notes 5-8 and Table 8) vvvvvvv: control value ccccccc = 122 thru127: Reserved (see Table 3)
1100nnnn	(&cx)	0ppppppp		Program Change ppppppp: program number 0-127
1101nnnn	(&dx)	0vvvvvvv		Channel Pressure (After-Touch) vvvvvvv: pressure value
1110nnnn	(&ex)	0vvvvvvv	0ppppppp	Pitch Bend Change (see note 10) vvvvvvv: LSB ppppppp: MSB

Table 3.1: Channel Voice Messages

3.1 Notes

1. nnnn: Voice Channel # (1-16, coded as defined in Table 2.1 notes)
2. kkkkkkk: note # (0 - 127) kkkkkkk = 60: Middle C of keyboard

3. vvvvvvv: key velocity. A logarithmic scale would be advisable.

   vvvvvvv = 64: in case of no velocity sensors
   vvvvvvv = 0: Note Off, with velocity = 64

4. Any Note On message sent should be balanced by sending a Note Off message for that note in that channel at some later time.

5. ccccccc: control number

ccccccc	**Description**
0	Continuous Controller 0 MSB
1	Continuous Controller 1 MSB
2	Continuous Controller 2 MSB
3	Continuous Controller 3 MSB
4-31	Continuous Controllers 4-31 MSB
32	Continuous Controller 0 LSB
33	Continuous Controller 1 LSB
34	Continuous Controller 2 LSB
35	Continuous Controller 3 LSB
36-63	Continuous Controllers 4-31 LSB
64-95	Switches (On/Off)
96-121	Undefined
122-127	Reserved for Channel Mode messages (see Table 3).

   See Table 9.1 for a full listing of defined controllers.

6. All controllers are specifically defined by agreement of the MIDI Manufacturers Association (MMA) and the Japan MIDI Standards Committee (JMSC). Manufacturers can request throught the MMA or JMSC that logical controllers be assigned to physical ones as necessary. The controller allocation table must be provided in the user's operation manual.

7. Continuous controllers are divided into Most Significant and Least Significant Bytes. If only seven bits of resolution are needed for any particular controllers, only the MSB is sent. It is not necessary to send the LSB. If more resolution is needed, then both are sent, first the MSB, then the LSB. If only the LSB has changed in value, the LSB may be sent without resending the MSB.

8. vvvvvvv: control value (MSB)
   for controllers 0-127, 0: min, 127: max
   for switches 0: off, 127: on, 1-126: ignored

9. Any messages (e.g. Note On), which are sent successively under the same status, can be sent without a Status byte until a different Status byte is needed.

10. Sensitivity of the pitch bender is selected in the receiver. Center position value (no pitch change) is &2000, which would be transmitted &en &00 &40.

Chapter 4

Channel Mode Messages

Status byte	Data bytes	Description
1011nnnn (&bx)	0ccccccc 0vvvvvvv	Mode Messages ccccccc = 122: Local Control vvvvvvv = 0: Local Control Off vvvvvvv = 127: Local Cntrl On
		ccccccc = 123: All Notes Off vvvvvvv = 0
		ccccccc = 124: Omni Mode Off (All Notes Off) vvvvvvv = 0
		ccccccc = 125: Omni Mode On (All Notes Off) vvvvvvv = 0
		ccccccc = 126: Mono Mode On (Poly Mode Off) (All Notes Off) vvvvvvv = M, where M is the number of channels. vvvvvvv = 0, the number of channels equals the number of voices in the receiver.
		ccccccc = 127: Poly Mode On (Mono Mode Off) vvvvvvv = 0 (All Notes Off)

Table 4.1: Channel Mode Messages

4.1 Notes

1. nnnn: Basic Channel # (1-16, coded as defined in Table 2.1)
2. Messages 123 through 127 function as All Notes Off messages. They will turn off all voices

controlled by the assigned Basic Channel. Except for message 123, All Notes Off, they should not be sent periodically, but only for a specific purpose. In no case should they be used in lieu of Note Off commands to turn off notes which have been previously turned on. Therefore any All Notes Off command (123-127) may be ignored by receiver with no possibility of notes staying on, since any Note On command must have a corresonding specific Note Off command.

3. Control Change #122, Local Control, is optionally used to interrupt the internal control path between the keyboard, for example, and the sound-generating circuitry. If 0 (Local Off mesage) is received, the path is disconnected: the keyboard data goes only to MIDI and the sound-generating circuitry is controlled only by incoming MIDI data. If a &7f (Local On message) is received, normal operation is restored.

4. The third byte of "Mono" specifies the number of channels in which Monophonic Voice messages are to be sent. This number, "M", is a number between 1 and 16. The channel(s) being used, then, will be the current Basic Channel (= N) thru N+M-1 up to a maximum of 16. If M=0, this is a special case directing the receiver to assign all its voices, one per channel, from the Basic Channel N through 16.

Chapter 5

System Common Messages

Status byte		Data bytes		Description
11110001	(&f1)			Undefined
11110010	(&f2)	0lllllll	0hhhhhhh	Song Position Pointer lllllll: (Least significant) hhhhhhh: (Most significant)
11110011	(&f3)	0sssssss		Song Select sssssss: Song #
11110100	(&f4)			Undefined
11110101	(&f5)			Undefined
11110110	(&f6)			Tune Request
11110111	(&f7)			EOX: "End of System Exclusive" flag

Table 5.1: System Common Messages

5.1 Notes

1. Song Position Pointer: Is an internal register which holds the number of MIDI beats (1 beat = 6 MIDI clocks) since the start of the song. Normally it is set to 0 when the start switch is pressed, which starts sequence playback. It then increments with every sixth MIDI clock receipt, until stop is pressed. If continue is pressed, it continues to increment. It can be arbitrarily preset (to a resolution of 1 beat) by the Song Position Pointer message.

2. Song Select: Specifies which song or sequence is to be played upon receipt of a Start (Real-Time) message.

3. Tune Request: Used with analog synthesizers to request them to tune their oscillators.

4. EOX: Used as a flag to indicate the end of a System Exclusive transmission (see Table 6).

Chapter 6

System Real Time Messages

Status byte		Data bytes	Description
11111000	(&f8)		Timing Clock
11111001	(&f9)		Undefined
11111010	(&fa)		Start
11111011	(&fb)		Continue
11111100	(&fc)		Stop
11111101	(&fd)		Undefined
11111110	(&fe)		Active Sensing
11111111	(&ff)		System Reset

Table 6.1: System Real Time Messages

6.1 Notes

1. The System Real Time messages are for synchronizing all of the system in real time.
2. The System Real Time messages can be sent at any time. Any messages which consist of two or more bytes may be split to insert Real Time messages.
3. Timing clock (&f8) The system is synchronized with this clock, which is sent at a rate of 24 clocks/quarter note.
4. Start (from the beginning of song) (&fa) This byte is immediately sent when the play switch on the master (e.g. sequencer or rhythm unit) is pressed.
5. Continue (&fb) This is sent when the continue switch is hit. A sequence will continue at the time of the next clock.
6. Stop (&fc) This byte is immediately sent when the stop switch is hit. It will stop the sequence.
7. Active Sensing (&fe) Use of this message is optional, for either receivers or transmitters. This is a "dummy" Status byte that is sent every 300 ms (max), whenever there is no other activity on MIDI. The receiver will operate normally if it never receives &fe. Otherwise, if

&fe is ever received, the receiver will expect to receive &fe or a transmission of any type every 300 ms (max). If a period of 300 ms passes with no activity, the receiver will turn off the voices and return to normal operation.

8. System Reset (&ff) This message initializes all of the system to the condition of just having turned on power. The system Reset message should be used sparingly, preferably under manual command only. In particular, it should not be sent automatically on power up.

Chapter 7

System Exclusive Messages

Status byte	Data bytes	Description
11110000 (&f0)		Bulk dump etc.
	0iiiiiii	iiiiiii: identification
	...	
	(0*******)	
	...	Any number of bytes may be sent here, for any purpose, as long as they all have a zero in the most significant bit.
	(0*******)	
	...	
	11110111	EOX: "End of System Exclusive"

Table 7.1: System Exclusive Messages

7.1 Notes

1. iiiiiii: identification ID (0-127)
2. All bytes between the System Exclusive Status byte and EOX or the next Status byte must have zeroes in the MSB.
3. The ID number can be obtained from the MMA or JMSC.
4. In no case should other Status or Data bytes (except Real-Time) be interleaved with System Exclusive, regardless of whether or not the ID code is recognized.
5. EOX or any other Status byte, except Real-Time, will terminate a System Exclusive message, and should be sent immediately at its conclusion.

Chapter 8

MIDI Note Numbers

Octave	Note numbers											
	C	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Table 8.1: MIDI note numbers

8.1 Notes

The MIDI specification only defines note number 60 as “Middle C”. Thus there is no strict definition of octave numbers, and some range note numbers over -1 to 9, and some 0 to 10.

Chapter 9

Control and Mode Messages

This table lists the controllers currently defined. See Table 2 for a description the control change command that uses these controllers. The first byte to be sent is the control change status byte, the second and third data bytes to follow that are shown below.

The first 32 controllers (0 to 31) are coarse adjustments for various parameters. The controllers from 32 to 63 are their respective fine adjustments (ie, add 32 to the coarse adjust number to get the fine adjust number, except for the General Purpose Sliders which have no fine adjust equivalents). For example, the coarse adjustment for Channel Volume is controller number 7. The fine adjustment for Channel Volume is controller number 39 (7+32). Many devices use only the coarse adjustments, and ignore the fine adjustments.

This specification was updated in 1995 by the Midi Manufacturers Association, and in this update, some of the ranges described in Table 1 note 5 have been superseded.

2nd data byte value				
binary	**hex**	**dec**	**Function**	**3rd data byte**
00000000	&00	0	Bank select MSB	0-127
00000001	&01	1	Modulation wheel MSB	0-127
00000010	&02	2	Breath control MSB	0-127
00000011	&03	3	Undefined MSB	0-127
00000100	&04	4	Foot controller MSB	0-127
00000101	&05	5	Portamento time MSB	0-127
00000110	&06	6	Data Entry MSB	0-127
00000111	&07	7	Channel volume MSB (formerly Main volume)	0-127
00001000	&08	8	Ballance MSB	0-127
00001001	&09	9	Undefined	0-127
00001010	&0A	10	Pan MSB	0-127
00001011	&0B	11	Expression Controller MSB	0-127
00001100	&0C	12	Effect Control 1 MSB	0-127
00001101	&0D	13	Effect Control 2 MSB	0-127
00001110	&0E	14	Undefined	0-127
00001111	&0F	15	Undefined	0-127
00010000	&10	16	General Purpose Controller #1	0-127

continued overleaf

2nd data byte value				
binary	**hex**	**dec**	**Function**	**3rd data byte**
00010001	&11	17	General Purpose Controller #2	0-127
00010010	&12	18	General Purpose Controller #3	0-127
00010011	&13	19	General Purpose Controller #4	0-127
00010100	&14	20	Undefined	0-127
00010101	&15	21	Undefined	0-127
00010110	&16	22	Undefined	0-127
00010111	&17	23	Undefined	0-127
00011000	&18	24	Undefined	0-127
00011001	&19	25	Undefined	0-127
00011010	&1A	26	Undefined	0-127
00011011	&1B	27	Undefined	0-127
00011100	&1C	28	Undefined	0-127
00011101	&1D	29	Undefined	0-127
00011110	&1E	30	Undefined	0-127
00011111	&1F	31	Undefined	0-127
00100000	&20	32	Bank select LSB	0-127
00100001	&21	33	Modulation wheel LSB	0-127
00100010	&22	34	Breath control LSB	0-127
00100011	&23	35	Undefined	0-127
00100100	&24	36	Foot controller LSB	0-127
00100101	&25	37	Portamento time LSB	0-127
00100110	&26	38	Data entry LSB	0-127
00100111	&27	39	Channel volume LSB (formerly Main volume)	0-127
00101000	&28	40	Ballance LSB	0-127
00101001	&29	41	Undefined	0-127
00101010	&2A	42	Pan	0-127
00101011	&2B	43	Expression Controller LSB	0-127
00101100	&2C	44	Effect Control 1 LSB	0-127
00101101	&2D	45	Effect Control 2 LSB	0-127
00101110	&2E	46	Undefined	0-127
00101111	&2F	47	Undefined	0-127
00110000	&30	48	General Purpose Controller #1 LSB	0-127
00110001	&31	49	General Purpose Controller #2 LSB	0-127
00110010	&32	50	General Purpose Controller #3 LSB	0-127
00110011	&33	51	General Purpose Controller #4 LSB	0-127
00110100	&34	52	Undefined	0-127
00110101	&35	53	Undefined	0-127
00110110	&36	54	Undefined	0-127
00110111	&37	55	Undefined	0-127
00111000	&38	56	Undefined	0-127
00111001	&39	57	Undefined	0-127
00111010	&3A	58	Undefined	0-127
00111011	&3B	59	Undefined	0-127
00111100	&3C	60	Undefined	0-127
00111101	&3D	61	Undefined	0-127

continued overleaf

2nd data byte value				
binary	**hex**	**dec**	**Function**	**3rd data byte**
00111110	&3E	62	Undefined	0-127
00111111	&3F	63	Undefined	0-127
01000000	&40	64	Damper pedal on/off (Sustain)	¡= 63 off, ¿ 63 on
01000001	&41	65	Portamento on/off	¡= 63 off, ¿ 63 on
01000010	&42	66	Sustenuto on/off	¡= 63 off, ¿ 63 on
01000011	&43	67	Soft pedal on/off	¡= 63 off, ¿ 63 on
01000100	&44	68	Legato Footswitch	¡= 63 off, ¿ 63 on
01000101	&45	69	Hold 2	¡= 63 off, ¿ 63 on
01000110	&46	70	Sound Controller 1 (Sound Variation) LSB	0-127
01000111	&47	71	Sound Controller 2 (Timbre) LSB	0-127
01001000	&48	72	Sound Controller 3 (Release Time) LSB	0-127
01001001	&49	73	Sound Controller 4 (Attack Time) LSB	0-127
01001010	&4A	74	Sound Controller 5 (Brightness) LSB	0-127
01001011	&4B	75	Sound Controller 6 LSB	0-127
01001100	&4C	76	Sound Controller 7 LSB	0-127
01001101	&4D	77	Sound Controller 8 LSB	0-127
01001110	&4E	78	Sound Controller 9 LSB	0-127
01001111	&4F	79	Sound Controller 10 LSB	0-127
01010000	&50	80	General Purpose Controller #5 LSB	0-127
01010001	&51	81	General Purpose Controller #6 LSB	0-127
01010010	&52	82	General Purpose Controller #7 LSB	0-127
01010011	&53	83	General Purpose Controller #8 LSB	0-127
01010100	&54	84	Portamento Control Source No	0-127
01010101	&55	85	Undefined	0-127
01010110	&56	86	Undefined	0-127
01010111	&57	87	Undefined	0-127
01011000	&58	88	Undefined	0-127
01011001	&59	89	Undefined	0-127
01011010	&5A	90	Undefined	0-127
01011011	&5B	91	Effects 1 Depth LSB	0-127
01011100	&5C	92	Effects 2 Depth LSB	0-127
01011101	&5D	93	Effects 3 Depth LSB	0-127
01011110	&5E	94	Effects 4 Depth LSB	0-127
01011111	&5F	95	Effects 5 Depth LSB	0-127
01100000	&60	96	Data entry +1	N/A
01100001	&61	97	Data entry -1	N/A
01100010	&62	98	Non-Registered Parameter Number LSB	0-127
01100011	&63	99	Non-Registered Parameter Number MSB	0-127
01100100	&64	100	Registered Parameter Number LSB	0-127
01100101	&65	101	Registered Parameter Number MSB	0-127
01100110	&66	102	Undefined	?
01100111	&67	103	Undefined	?
01100111	&67	103	Undefined	?
01101000	&68	104	Undefined	?

continued overleaf

2nd data byte value				
binary	**hex**	**dec**	**Function**	**3rd data byte**
01101001	&69	105	Undefined	?
01101010	&6A	106	Undefined	?
01101011	&6B	107	Undefined	?
01101100	&6C	108	Undefined	?
01101101	&6D	109	Undefined	?
01101110	&6E	110	Undefined	?
01101111	&6F	111	Undefined	?
01110000	&70	112	Undefined	?
01110001	&71	113	Undefined	?
01110010	&72	114	Undefined	?
01110011	&73	115	Undefined	?
01110100	&74	116	Undefined	?
01110101	&75	117	Undefined	?
01110110	&76	118	Undefined	?
01110111	&77	119	Undefined	?
01111000	&78	120	All Sound Off	0
01111001	&79	121	Reset All Controllers	0
01111010	&7A	122	Local Control on/off	0=off 127=on
01111011	&7B	123	All Notes Off	0
01111100	&7C	124	Omni mode off (includes all notes off)	0
01111101	&7D	125	Omni mode on (includes all notes off)	0
01111110	&7E	126	Poly mode on/off (includes all notes off)	**
01111111	&7F	127	Poly mode on (incl mono=off&all notes off)	0

Table 9.1: MIDI Controller Messages

9.1 Notes

**: This equals the number of channels, or zero if the number of channels equals the number of voices in the reciever

Chapter 10

General MIDI and Roland's GS Standard

10.1 Brief Overview of Proposed General MIDI Level 1 Spec

The heart of General MIDI (GM) is the Instrument Patch Map, shown in Table 10.1. This is a list of 128 sounds, with corresponding MIDI program numbers. Most of these are imitative sounds, though the list includes synth sounds, ethnic instruments and a handful of sound effects.

The sounds fall roughtly into sixteen families of eight variations each. Grouping sounds makes it easy to re-orchestrate a piece using similar sounds. The Instrument Map isn't the final word on musical instruments of the world, but its pretty complete.

General MIDI also includes a Percusssion Key Map, show in Table 10.2. This mapping derives from the Roland/Sequential mapping used on early drum machines. As with the Instrument Map, it doesn't cover every percussive instrument in the world, but it's more than adequate as a basic set.

To avoid concerns with channels, GM restricts percussion to MIDI Channel 10. Theoretically, the lower nine channels are for the instruments, but the GM spec states that a sound module must respond to all sixteen MIDI channels, with dynamic voice allocation and a minimum of 24 voices.

General MIDI doesn't mention sound quality of synthesis methods. Discussions are under way on standardizing sound parameters such as playable range and envelope times. This will ensure that an arrangement that relies on phrasing and balance can play back on a variety of modules.

Other requirements for a GM sound module include response to velocity, mod wheel, aftertouch, sustain and expression pedal, main volume and pan, and the All Notes Off and Reset All Controllers messages. The module also must respond to both Pitch Bend and Pitch Bend Sensitivity (a MIDI registered parameter). The default pitch bend range is +-2 semitones.

Middle C (C3) corresponds to MIDI key 60, and master tuning must be adjustable. Finally, the MIDI Manufacturers Association (MMA) created a new Universal System Exclusive message to turn General MIDI on and off (for devices that might have 'consumer' and 'programmable' settings). Table 10.3 summarizes these requirements.

General MIDI has room for future expansion, including additional drum and instrument assignments and more required controllers. Also under discussion is an "authorizing document" that would standardize things such as channel assignments (e.g., lead on 1, bass on 2, etc.) and setup information in a MIDI file.

10.2 Roland's GS Standard

When Warner New Media first proposed a General MIDI standard, most MMA members gave it little thought. As discussions proceeded, Roland listened and developed a sound module to meet the proposed specification. At the same NAMM show where the MMA ratified General MIDI Level 1, Roland showed their Sound Brush and Sound Canvas, a Standard MIDI File player and GM-compatible sound module.

Some companies feel that General MIDI doesn't go far enough, so Roland created a superset of General MIDI Level 1, which they call GS Standard. It obeys all the protocols and sound maps of General MIDI and adds many extra controllers and sounds. Some of the controllers use Unregistered Parameter Numbers to give macro control over synth parameters such as envelope attack and decay rates.

The new MIDI Bank Select message provides access to extra sounds (including variations on the stock sounds and a re-creation of the MT-32 factory patches). The programs in each bank align with the original 128 in General MIDI's Instrument Patch Map, with eight banks housing related families. The GS Standard includes a 'fall back' system. If the Sound Canvas receives a request for a bank/program number combination that does not exist, it will reassign it to the master instrument in that family. A set of Roland System Exclusive messages allows reconfiguration and customization of the sound module.

This means that a Roland GS Standard sound module will correctly play back any song designed for General MIDI. In addition, if the song's creator wants to create some extra nuance, they can include the GS Standard extensions in their sequence. None of these extensions are so radical as to make the song unplayable on a normal GM sound module. After all, compatibility is what MIDI - and especially General MIDI - is all about.

10.3 General MIDI Instrument Patch Map

The specification groups sounds into sixteen families, with 8 instruments in each family.

Prog#	**Instrument**	**Prog#**	**Instrument**
(1-8: Piano)		**(9-16 Percussion)**	
1	Acoustic Grand	9	Celesta
2	Bright Acoustic	10	Glockenspiel
3	Electric Grand	11	Music Box
4	Honky-Tonk	12	Vibraphone
5	Electric Piano 1	13	Marimba
6	Electric Piano 2	14	Xylophone
7	Harpsichord	15	Tubular Bells
8	Clav	16	Dulcimer

continued overleaf

Prog#	Instrument	Prog#	Instrument
(17-24: Organ)		**(25-32: Guitar)**	
17	Drawbar Organ	25	Acoustic Guitar(nylon)
18	Percussive Organ	26	Acoustic Guitar(steel)
19	Rock Organ	27	Electric Guitar(jazz)
20	Church Organ	28	Electric Guitar(clean)
21	Reed Organ	29	Electric Guitar(muted)
22	Accoridan	30	Overdriven Guitar
23	Harmonica	31	Distortion Guitar
24	Tango Accordian	32	Guitar Harmonics
(33-40: Bass)		**(41-48: Strings)**	
33	Acoustic Bass	41	Violin
34	Electric Bass(finger)	42	Viola
35	Electric Bass(pick)	43	Cello
36	Fretless Bass	44	Contrabass
37	Slap Bass 1	45	Tremolo Strings
38	Slap Bass 2	46	Pizzicato Strings
39	Synth Bass 1	47	Orchestral Strings
40	Synth Bass 2	48	Timpani
(49-56: Ensemble)		**(57-64: Brass)**	
49	String Ensemble 1	57	Trumpet
50	String Ensemble 2	58	Trombone
51	SynthStrings 1	59	Tuba
52	SynthStrings 2	60	Muted Trumpet
53	Choir Aahs	61	French Horn
54	Voice Oohs	62	Brass Section
55	Synth Voice	63	SynthBrass 1
56	Orchestra Hit	64	SynthBrass 2
(65-72: Reed)		**(73-80: Pipe)**	
65	Soprano Sax	73	Piccolo
66	Alto Sax	74	Flute
67	Tenor Sax	75	Recorder
68	Baritone Sax	76	Pan Flute
69	Oboe	77	Blown Bottle
70	English Horn	78	Skakuhachi
71	Bassoon	79	Whistle
72	Clarinet	80	Ocarina
(81-88: Synth Lead)		**(89-96: Synth Pad)**	
81	Lead 1 (square)	89	Pad 1 (new age)
82	Lead 2 (sawtooth)	90	Pad 2 (warm)
83	Lead 3 (calliope)	91	Pad 3 (polysynth)
84	Lead 4 (chiff)	92	Pad 4 (choir)
85	Lead 5 (charang)	93	Pad 5 (bowed)
86	Lead 6 (voice)	94	Pad 6 (metallic)
87	Lead 7 (fifths)	95	Pad 7 (halo)

continued overleaf

Prog#	Instrument	Prog#	Instrument
88	Lead 8 (bass+lead)	96	Pad 8 (sweep)
(97-104: Synth Effects)		**(105-112: Ethnic)**	
97	FX 1 (rain)	105	Sitar
98	FX 2 (soundtrack)	106	Banjo
99	FX 3 (crystal)	107	Shamisen
100	FX 4 (atmosphere)	108	Koto
101	FX 5 (brightness)	109	Kalimba
102	FX 6 (goblins)	110	Bagpipe
103	FX 7 (echoes)	111	Fiddle
104	FX 8 (sci-fi)	112	Shanai
(113-120: Percussive)		**(121-128: Sound Effects)**	
113	Tinkle Bell	121	Guitar Fret Noise
114	Agogo	122	Breath Noise
115	Steel Drums	123	Seashore
116	Woodblock	124	Bird Tweet
117	Taiko Drum	125	Telephone Ring
118	Melodic Tom	126	Helicopter
119	Synth Drum	127	Applause
120	Reverse Cymbal	128	Gunshot

Table 10.1: General MIDI instrument definitions

10.4 General MIDI Percussion Key Map

General MIDI assigns drum sounds to note numbers. MIDI Channel 10 is reserved for percussion.

MIDI Key	Drum Sound	MIDI Key	Drum Sound
35	Acoustic Bass Drum	59	Ride Cymbal 2
36	Bass Drum 1	60	Hi Bongo
37	Side Stick	61	Low Bongo
38	Acoustic Snare	62	Mute Hi Conga
39	Hand Clap	63	Open Hi Conga
40	Electric Snare	64	Low Conga
41	Low Floor Tom	65	High Timbale
42	Closed Hi-Hat	66	Low Timbale
43	High Floor Tom	67	High Agogo
44	Pedal Hi-Hat	68	Low Agogo
45	Low Tom	69	Cabasa
46	Open Hi-Hat	70	Maracas
47	Low-Mid Tom	71	Short Whistle
48	Hi-Mid Tom	72	Long Whistle
49	Crash Cymbal 1	73	Short Guiro
50	High Tom	74	Long Guiro
51	Ride Cymbal 1	75	Claves
52	Chinese Cymbal	76	Hi Wood Block

continued overleaf

MIDI Key	Drum Sound	MIDI Key	Drum Sound
53	Ride Bell	77	Low Wood Block
54	Tambourine	78	Mute Cuica
55	Splash Cymbal	79	Open Cuica
56	Cowbell	80	Mute Triangle
57	Crash Cymbal 2	81	Open Triangle
58	Vibraslap		

Table 10.2: General MIDI Percussion Map

10.5 General MIDI minimum sound module specs

10.5.1 Voices

A minimum of either 24 fully dynamically allocated voices available simultaneously for both melodic and percussive sounds or 16 dynamically allocated voices for melody plus eight for percussion.

10.5.2 Channels

General MIDI mode supports all sixteen MIDI channels. Each channel can play a variable number of voices (polyphony). Each channel can play a different instrument (timbre). Keybased Percussion is always on Channel10.

10.5.3 Instruments

A minimum of sixteen different timbres playing various instrument sounds. A minimum of 128 preset for Intruments (MIDI program numbers).

10.5.4 Note on/Note off

Octave Registration: Middle C(C3) = MIDI key 60. All Voices including percussion respond to velocity.

10.5.5 Controllers

Controller #	**Description**
1	Modulation
7	Main Volume
10	Pan
11	Expression
64	Sustain
121	Reset All Controllers
123	All Notes Off

Registered Parameter #	**Description**
0	Pitch Bend Sensitivity
1	Fine Tuning
2	Coarse Tuning

10.5.6 Additional Channel Messages

- Channel Pressure (Aftertouch)
- Pitch Bend

10.5.7 Power-Up Defaults

- Pitch Bend Amount = 0
- Pitch Bend Sensitivity = +-2 semitones
- Volume = 90
- All Other Controllers = reset

Table 10.3: GM Minimum specifications

Chapter 11

MIDI Note Frequencies

The following table gives the frequency in Hertz for each MIDI note. Note that due to the exponential nature of musical pitch, notes in the higher octaves are easier to discriminate (because they are further apart). Also, the highest four notes are unspecifiable.

Middle C is C5.

Note name	**MIDI Key**	**MIDI Freq(Hz)**	**Freq(%)**	**Note name**	**MIDI Key**	**MIDI Freq(Hz)**	**Freq(%)**
C0	12	8.2	0.08	C#/Db0	16	8.7	0.09
D0	14	9.2	0.09	D#/Eb0	17	9.7	0.10
E0	16	10.3	0.10	F0	18	10.9	0.11
F#/Gb0	18	11.6	0.12	G0	19	12.2	0.12
G#/Ab0	20	13.0	0.13	A0	21	13.8	0.14
A#/Bb0	22	14.6	0.15	B0	23	15.4	0.15
C1	24	16.4	0.16	C#/Db1	25	17.3	0.17
D1	26	18.4	0.18	D#/Eb1	27	19.4	0.19
E1	28	20.6	0.21	F1	29	21.8	0.22
F#/Gb1	30	23.1	0.23	G1	31	24.5	0.24
G#/Ab1	32	26.0	0.26	A1	33	27.5	0.28
A#/Bb1	34	29.1	0.29	B1	35	30.9	0.31
C2	36	32.7	0.33	C#/Db2	37	34.6	0.35
D2	38	36.7	0.37	D#/Eb2	39	38.9	0.39
E2	40	41.2	0.41	F2	41	43.7	0.44
F#/Gb2	42	46.2	0.46	G2	43	49.0	0.49
G#/Ab2	44	51.9	0.52	A2	45	55.0	0.55
A#/Bb2	46	58.3	0.58	B2	47	61.7	0.62
C3	48	65.4	0.65	C#/Db3	49	69.3	0.69
D3	50	73.4	0.73	D#/Eb3	51	77.8	0.78
E3	52	82.4	0.82	F3	53	87.3	0.87
F#/Gb3	54	92.5	0.92	G3	55	98.0	0.98
G#/Ab3	56	103.8	1.04	A3	57	110.0	1.10

continued overleaf

Note name	MIDI Key	MIDI Freq(Hz)	Freq(%)	Note name	MIDI Key	MIDI Freq(Hz)	Freq(%)
A#/Bb3	58	116.5	1.17	B3	59	123.5	1.23
C4	60	130.8	1.31	C#/Db4	61	138.6	1.39
D4	62	146.8	1.47	D#/Eb4	63	155.6	1.56
E4	64	164.8	1.65	F4	65	174.6	1.75
F#/Gb4	66	185.0	1.85	G4	67	196.0	1.96
G#/Ab4	68	207.7	2.08	A4	69	220.0	2.20
A#/Bb4	70	233.1	2.33	B4	71	246.9	2.47
C5	72	261.6	2.62	C#/Db5	73	277.2	2.77
D5	74	293.7	2.94	D#/Eb5	75	311.1	3.11
E5	76	329.6	3.30	F5	77	349.2	3.49
F#/Gb5	78	370.0	3.70	G5	79	392.0	3.92
G#/Ab5	80	415.3	4.15	A5	81	440.0	4.40
A#/Bb5	82	466.2	4.66	B5	83	493.9	4.94
C6	84	523.3	5.23	C#/Db6	85	554.4	5.54
D6	86	587.3	5.87	D#/Eb6	87	622.3	6.22
E6	88	659.3	6.59	F6	89	698.5	6.98
F#/Gb6	90	740.0	7.40	G6	91	784.0	7.84
G#/Ab6	92	830.6	8.31	A6	93	880.0	8.80
A#/Bb6	94	932.3	9.32	B6	95	987.8	9.88
C7	96	1046.5	10.47	C#/Db7	97	1108.7	11.09
D7	98	1174.7	11.75	D#/Eb7	99	1244.5	12.45
E7	100	1318.5	13.19	F7	101	1396.9	13.97
F#/Gb7	102	1480.0	14.80	G7	103	1568.0	15.68
G#/Ab7	104	1661.2	16.61	A7	105	1760.0	17.60
A#/Bb7	106	1864.7	18.65	B7	107	1975.5	19.76
C8	108	2093.0	20.93	C#/Db8	109	2217.5	22.17
D8	110	2349.3	23.49	D#/Eb8	111	2489.0	24.89
E8	112	2637.0	26.37	F8	113	2793.8	27.94
F#/Gb8	114	2960.0	29.60	G8	115	3136.0	31.36
G#/Ab8	116	3322.4	33.22	A8	117	3520.0	35.20
A#/Bb8	118	3729.3	37.29	B8	119	3951.1	39.51
C9	120	4186.0	41.86	C#/Db9	121	4434.9	44.35
D9	122	4698.6	46.99	D#/Eb9	123	4978.0	49.78
E9	124	5274.0	52.74	F9	125	5587.7	55.88
F#/Gb9	126	5919.9	59.20	G9	127	6271.9	62.72
G#/Ab9	128	6644.9	66.45	A9	x	7040.0	70.40
A#/Bb9		7458.6	74.59	B9	x	7902.1	79.02
C10	x	8372.0	83.72	C#/Db10	x	8869.8	88.70
D10	x	9397.3	93.97	D#/Eb10	x	9956.1	99.56
E10	x	10548.1	105.48	F10	x	11175.3	111.75

continued overleaf

Note name	MIDI Key	MIDI Freq(Hz)	Freq(%)	Note name	MIDI Key	MIDI Freq(Hz)	Freq(%)
F#/Gb10	x	11839.8	118.40	G10	x	12543.9	125.44

Table 11.1: MIDI Note frequencies

11.0.8 Frequency to MIDI note formula

How to calculate the frequency or tempo change required to shift a sample by a musical interval measured in cents.

If you have a sample recorded at 120 bpm (beats per minute), and want to shift the pitch up by a perfect fifth (seven semitones, i.e. 700 cents) what rate should it be played back at to achieve this?

The function required to calculate this is:

$$f2 = f1 \times 2^{\frac{C}{1200}}$$

where:

- f1 = original frequency or tempo
- f2 = final frequency or tempo
- C = number of cents shift required

So for this example,

$$f2 = 120 \times 2^{\frac{700}{1200}} = \underline{179.80 \text{ beats per minute}}$$

How to calculate the change in musical pitch, measured in cents from a change in sample frequency playback

If you have a sample recorded at 44.1 kHz and play it back at 32kHz, what will the change in pitch be?

The function required to calculate this is:

$$C = \frac{1200 \times \log \frac{f2}{f1}}{\log 2}$$

where:

- f1 = original frequency or tempo
- f2 = final frequency or tempo
- C = number of cents shift required

So for this example,

$$C = \frac{1200 \times \log \frac{32000}{44100}}{0.3010} = \underline{-555.35 \text{ cents}}, \text{ i.e. a shift down of about five and a half semitones.}$$

Mathematical explanation of the equations:

We know that for a change in pitch of one octave, i.e. 12 semitones, that frequency is doubled, so for an equally tempered scale, the ratio of frequencies for each adjacent semitone is $2^{\frac{1}{12}}$ (the 12th root of 2) which is approximately 1.0595. We also know that a semitone is represented by 100 cents. Let us call this ratio r. So, if $\frac{f1}{f2} = r$, then $f1$ is greater than $f2$ by 100 cents.

From above, we know that $r^{12} = 2$, and $r^1 = r$ so for an arbitrary number of cents, C

$$\text{let } n = \frac{C}{100} \tag{11.1}$$

$$\text{i.e. } r^n = \frac{f2}{f1} \tag{11.2}$$

$$\tag{11.3}$$

Taking logs,

$$n \times \log r = \log \frac{f2}{f1} \tag{11.4}$$

$$n = \frac{\log f2f1}{\log r} \tag{11.5}$$

$$n = \frac{\log f2f1}{\log(2^{\frac{1}{12}})} \tag{11.6}$$

$$\tag{11.7}$$

substituting C back in,

$$C = 100 \times \frac{\log f2f1}{(\frac{1}{12} * \log 2)} \tag{11.8}$$

$$C = 1200 \times \frac{\log f2f1}{\log 2} \tag{11.9}$$

$$\tag{11.10}$$

Rearranging,

$$\frac{C * \log 2}{1200} = \log f2f1 \tag{11.11}$$

$$\tag{11.12}$$

Taking antilogs,

$$\frac{f2}{f1} = 10^{\frac{(C * \log 2)}{1200}} \tag{11.13}$$

$$\underline{\frac{f2}{f1} = 2^{\frac{C}{1200}}} \tag{11.14}$$

11.1 Glossary

What is a 'Cent'? A Cent is a term used to describe a fraction of a musical interval. By definition, on an equally tempered scale there are 100 cents in a semitone. This means that in a scale from for example C3 to C4 there are 1200 cents.

What is frequency? It is the frequency of a musical tone that determines its pitch. Frequency is usually measured in Hertz (Hz), but when describing a tempo is often in beats per minute. A4 is defined to be 440Hz (440 cycles or vibrations per second). Most sounds are complex and comprise a fundamental, plus a number of harmonics. It is the fundamental that we are interested in. Other harmonics contribute to the individual sound of the instrument.

What is an octave? An octave is the musical interval between two notes, where one note is exactly twice the frequency of the other. An octave is a musical interval of twelve semitones on an equally tempered scale.

What is an equally tempered scale? The scale on keyboard instruments since Bach's time has been approximated into equal mathematical divisions. Before that time, if an instrument was tuned to one key, it would be unsuited to playing in other scales unless it was retuned. Bach promoted the retuning of instruments with a set of pieces for the Well Tempered Clavier that used a whole range of different keys. Natural acoustic scales (which vary slightly from 100 cents between each semitone) still exist in ethnic and folk music where fretless instruments and the human voice are used. Some synthesizers also permit retuning to other scale systems.

Chapter 12

Standard MIDI Files

12.1 Abstract

Standard MIDI Files 0.06 March 1, 1988
This document is Dave Oppenheim's current version of the MIDI file specification, as sent to those who have participated in its development. The consensus seems to be to submit this to the MIDI Manufacturers' Association as version 1.0. I apologize for any loss of clarity that might have occurred in the conversion from a Microsoft Word document to this pure text file. I have removed some of the discussion about recent changes to the specification in order to keep the file size reasonable.–Doug Wyatt

12.2 Introduction

This describes a proposed standard MIDI file format. 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, nibbleized, 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.

12.3 Sequences, Tracks, Chunks: File Block Structure

Sequence 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. On the Macintosh, data is passed either in the data fork of a file, or on the Clipboard. (The file type on the Macintosh for a file in this format will be "Midi".) On any other computer, the data is simply the contents of the file. This structure allows future chunk types to be designed which may 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.

This proposal defines two types of chunks: a header chunk and a track chunk. 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>
 ...
```

12.3.1 Track Data Format (MTrk chunk type)

The MTrk chunk type is where actual song data is stored. It is simply a stream of MIDI events (and non-MIDI events), preceded by delta-time values.

Some numbers in MTrk chunks are represented in a form called a 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:

```
Number (hex)    Representation (hex)
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 representation must fit in 32 bits in a routine to write variable-length numbers. Theoretically, larger numbers are possible, but 2 x 108 96ths of a beat at a fast tempo of 500 beats per minute is four days, long enough for any delta-time!

Here is the syntax of an MTrk chunk:

```
    <track data> = <MTrk event>+

    <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 may be omitted after the first byte. The first event in a file must specify status. Delta-time is not considered an event itself: it is an integral part of the specification. Notice that running status occurs across delta-times.

<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 sysex events, running status is not allowed. 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. New for 0.06: programs must never ignore the length of a meta-event which they do recognize, and they shouldn't be surprised if it's bigger than they 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 a meta-event.

<sysex event> is used to specify a MIDI system exclusive message, or as an "escape" to specify any arbitrary 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 at 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. To be able to handle situations like these, two forms of ¡sysex event¿ are provided:

```
    F0 <length> <bytes to be transmitted after F0>
    F7 <length> <all bytes to be transmitted>
```

In both cases, <length> is stored as a variable-length quantity. It is equal to the number of bytes following it, not including itself or the message type (F0 or F7), but all the bytes which follow, including any F7 at the end which is intended to be transmitted. The first form, with the F0 code, is used for syntactically complete system exclusive messages, or the first packet an a series Q that

is, messages in which the F0 should be transmitted. The second form is used for the remainder of the packets within a syntactic sysex message, which do not begin with F0. Of course, the F7 is not considered part of the system exclusive message. Of course, just as in MIDI, running status is not allowed, in this case because the length is stored as a variable-length quantity which may or may not start with bit 7 set.

(New to 0.06) 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. This principle is repeated and illustrated in the paragraphs below.

The vast majority of system exclusive messages will just use the F0 format. 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. As mentioned above, 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.

For special situations when a single system exclusive message is split up, with parts of it being transmitted at different times, such as in a Casio CZ patch transfer, or the FB-01's "system exclusive mode", the F7 form of sysex event is used for each packet except the first. None of the packets would end with an F7 except the last one, which must end with an F7. There also must not be any transmittable MIDI events in- between the packets of a multi-packet system exclusive message. Here is an example: 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
```

The F7 event may also be used as an "escape" to transmit any bytes whatsoever, including real-time bytes, song pointer, or MIDI Time Code, which are not permitted normally in this specification. No effort should be made to interpret the bytes used in this way. Since a system exclusive message is not being transmitted, it is not necessary or appropriate to end the F7 event with an F7 in this case.

12.4 Header Chunk

The header chunk at the beginning of the file specifies some basic information about the data in the file. The data section contains three 16-bit words, stored high byte first (of course). Here's the syntax of the complete 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 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 simultaneous tracks (or MIDI outputs) of a sequence
2 the file contains one or more sequentially independent single-track patterns

The next word, ntrks, is the number of track chunks in the file. The third word, division, is the division of a quarter-note represented by the delta-times in the file. (If division is negative, it represents the division of a second represented by the delta-times in the file, so that the track can represent events occurring in actual time instead of metrical time. It is represented in the following way: the upper byte is one of the four values -24, -25, -29, or -30, corresponding to the four standard SMPTE and MIDI time code formats, and represents the number of frames per second. 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 system allows exact specification of time-code-based tracks, but also allows millisecond-based tracks by specifying 25 frames/sec and a resolution of 40 units per frame.)

Format 0, that is, one multi-channel track, is the most interchangeable representation of data. One application of MIDI files is a simple single-track player in a program which needs to make synthesizers make sounds, but which is primarily concerned 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.

Programs which support several simultaneous tracks should be able to save and read data in format 1, a vertically one-dimensional form, that is, as a collection of tracks. Programs which support several independent patterns should be able to save and read data in format 2, a horizontally one-dimensional form. Providing these minimum capabilities will ensure maximum interchangeability.

MIDI files can express tempo and time signature, and they have been chosen to do so for transferring tempo maps from one device to another. 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 (starting in 0.04) be stored as the first track. It is polite to a tempo map reader to offer 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 in the first track. In format 2, each of the temporally independent 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 reading an unfamiliar format ID should return an error to the user rather than trying to read further.

12.5 Meta-Events

A few meta-events are defined herein. It is not required for every program to support every meta-event. Meta-events initially defined include:

FF 00 02 ssss *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. The number in this track corresponds to the sequence number in the new Cue message discussed at the summer 1987 MMA meeting. 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' locations 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.

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 character 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.

(New for 0.06). Meta event types 01 through 0F are reserved for various types of text events, each of which meets the specification of text events(above) but is used for a different purpose:

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 first 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 separate 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 rehearsal 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.)

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 it has an exact length, which is necessary for tracks which are looped or concatenated.

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 absolutely 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 Q 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 transferred to another device.

FF 54 05 hr mn se fr ff *SMPTE Offset specification)* 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* 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 negative 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 the 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), 32 MIDI clocks per dotted-quarter (24 hex!), and eight notated 32nd-notes per MIDI quarter note.

FF 59 02 sf mi *Key Signature* The values of sf and mi are used as follows:
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. However, as this is an interchange format, growth of the spec proper is preferred to use of this event type. This type of event 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.

12.6 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 separated 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  Event Code  Other Bytes       Comment
(decimal)   (hex)       (decimal)

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 5 sec per
                                          quarter-note
0           C0          5                 Ch. 1, Program Change 5
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:

```
4D 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 that 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   end of track
```

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
```

12.7 MIDI Transmission of MIDI Files

Since it is inconvenient to exchange disks between different computers, and since many computers which will use this format will have a MIDI interface anyway, MIDI seems like a perfect way to send these files from one computer to another. And, while we're going through all the trouble to make a way of sending MIDI Files, it would be nice if they could send any files (like sampled sound files, text files, etc.)

Goals

The transmission protocol for MIDI files should be reasonably efficient, should support fast transmission for computers which are capable of it, and slower transmission for less powerful ones. It should not be impossible to convert a MIDI File to or from an arbitrary internal representation on the fly as it is transmitted, but, as long as it is not too difficult, it is very desirable to use a generic method so that any file type could be accommodated.

To make the protocol efficient, the MIDI transmission of these files will take groups of seven 8-bit bytes and transmit them as eight 7-bit MIDI data bytes. This is certainly in the spirit of the rest of this format (keep it small, because it's not that hard to do). To accommodate a wide range of transmission speeds, files will be transmitted in packets with acknowledge – this allows data to be stored to disk as it is received. If the sender does not receive a response from a reader in a certain amount of time, it can assume an open-loop situation, and then just continue.

The last edition of MIDI Files contained a specialized protocol for sending just MIDI Files. To meet a deadline, unfortunately I don't have time right now to propose a new generalized protocol. This will be done within the next couple of months. I would welcome any proposals anyone else has, and would direct your attention to the proposal from Ralph Muha of Kurzweil, available in a recent MMA bulletin, and also directly from him.

Chapter 13

The MIDI File Format — an alternative version

This alternative guide was found somewhere on the internet, and describes MIDI files as described in the previous chapter. However, the style of writing is different, and it contains a little more experience of dealing with MIDI files, plus some extra meta events.

13.1 Introduction

The *Standard MIDI File* (SMF) is a file format specifically designed to store the data that a sequencer records and plays (whether that sequencer be software or hardware based).

This format stores the standard MIDI messages (ie, status bytes with appropriate data bytes) plus a time-stamp for each message (ie, a series of bytes that represent how many clock pulses to wait before "playing" the event). The format allows saving information about tempo, pulses per quarter note resolution (or resolution expressed in divisions per second, ie SMPTE setting), time and key signatures, and names of tracks and patterns. It can store multiple patterns and tracks so that any application can preserve these structures when loading the file.

NOTE: A *track* usually is analogous to one musical part, such as a Trumpet part. A *pattern* would be analogous to all of the musical parts (ie, Trumpet, Drums, Piano, etc) for a song, or excerpt of a song.

The format was designed to be generic so that any sequencer could read or write such a file without losing the most important data, and flexible enough for a particular application to store its own proprietary, "extra" data in such a way that another application won't be confused when loading the file and can safely ignore this extra stuff that it doesn't need. Think of the MIDI file format as a musical version of an ASCII text file (except that the MIDI file contains binary data too), and the various sequencer programs as text editors all capable of reading that file. But, unlike ASCII, MIDI file format saves data in *chunks* (ie, groups of bytes preceded by an ID and size) which can be parsed, loaded, skipped, etc. Therefore, it can be easily extended to include a program's proprietary info. For example, maybe a program wants to save a "flag byte" that indicates whether the user has turned on an audible metronome click. The program can put this flag byte into a MIDI

file in such a way that another application can skip this byte without having to understand what that byte is for. In the future, the MIDI file format can also be extended to include new "official" chunks that all sequencer programs may elect to load and use. This can be done without making old data files obsolete (ie, the format is designed to be extensible in a backwardly compatible way).

13.2 What's a Chunk?

Data is always saved within a *chunk*. There can be many chunks inside of a MIDI file. Each chunk can be a different size (ie, where size refers to how many bytes are contained in the chunk). The data bytes in a chunk are related in some way. A chunk is simply a group of related bytes.

Each chunk begins with a 4 character (ie, 4 ascii bytes) *ID* which tells what "type" of chunk this is. The next 4 bytes (all bytes are comprised of 8 bits) form a 32-bit length (ie, size) of the chunk. **All chunks must begin with these two fields** (ie, 8 bytes), which are referred to as the *chunk header*.

NOTE: The *Length* does not include the 8 byte chunk header. It simply tells you how many bytes of data are in the chunk *following this header*.

Here's an example header (with bytes expressed in hex):

```
4D 54 68 64 00 00 00 06
```

Note that the first 4 bytes make up the ascii ID of **MThd** (ie, the first four bytes are the ascii values for 'M', 'T', 'h', and 'd'). The next 4 bytes tell us that there should be 6 more data bytes in the chunk (and after that we should find the next chunk header or the end of the file).

In fact, all MIDI files begin with this *MThd header* (and that's how you know that it's a MIDI file).

NOTE: The 4 bytes that make up the *Length* are stored in Motorola 68000 byte order, not Intel reverse byte order (ie, the 06 is the fourth byte instead of the first of the four). All multiple byte fields in a MIDI file follow this standard, often called "Big Endian" form.

13.3 MThd Chunk

The MThd header has an ID of **MThd**, and a Length of **6**.

Let's examine the 6 data bytes (which follow the above, 8 byte header) in an MThd chunk.

The first two data bytes tell the *Format* (which I prefer to call *Type*). There are actually 3 different types (ie, formats) of MIDI files. A type of 0 means that the file contains one single track containing midi data on possibly all 16 midi channels. If your sequencer sorts/stores all of its midi data in one single block of memory with the data in the order that it's "played", then it should read/write this type. A type of 1 means that the file contains one or more simultaneous (ie, all start from an assumed time of 0) tracks, perhaps each on a single midi channel. Together, all of these tracks are considered one sequence or pattern. If your sequencer separates its midi data (i.e. tracks) into different blocks of memory but plays them back simultaneously (ie, as one "pattern"), it will read/write this type. A type of 2 means that the file contains one or more sequentially independant single-track patterns. f your sequencer separates its midi data into different blocks of memory, but

plays only one block at a time (ie, each block is considered a different "excerpt" or "song"), then it will read/write this type.

The next 2 bytes tell how many tracks are stored in the file, *NumTracks*. Of course, for format type 0, this is always 1. For the other 2 types, there can be numerous tracks.

The last two bytes indicate how many pulses (i.e. clocks) per quarter note resolution the time-stamps are based upon, *Division*. For example, if your sequencer has 96 ppqn, this field would be (in hex):

```
00 60
```

Alternately, if the first byte of Division is negative, then this represents the division of a second that the time-stamps are based upon. The first byte will be -24, -25, -29, or -30, corresponding to the 4 SMPTE standards representing frames per second. The second byte (a positive number) is the resolution within a frame (ie, subframe). Typical values may be 4 (MIDI Time Code), 8, 10, 80 (SMPTE bit resolution), or 100.

You can specify millisecond-based timing by the data bytes of -25 and 40 subframes.

Here's an example of a complete MThd chunk (with header):

```
4D 54 68 64     MThd ID
00 00 00 06     Length of the MThd chunk is always 6.
00 01           The Format type is 1.
00 02           There are 2 MTrk chunks in this file.
E7 28           Each increment of delta-time represents a millisecond.
```

13.4 MTrk Chunk

After the MThd chunk, you should find an *MTrk chunk*, as this is the only other currently defined chunk. (If you find some other chunk ID, it must be proprietary to some other program, so skip it by ignoring the following data bytes indicated by the chunk's Length).

An MTrk chunk contains all of the midi data (with timing bytes), plus optional non-midi data for **one track**. Obviously, you should encounter as many MTrk chunks in the file as the MThd chunk's NumTracks field indicated.

The MTrk header begins with the ID of *MTrk*, followed by the Length (ie, number of data bytes to read for this track). The Length will likely be different for each track. (After all, a track containing the violin part for a Bach concerto will likely contain more data than a track containing a simple 2 bar drum beat).

13.5 Variable Length Quantities — Event's Time

Think of a track in the MIDI file in the same way that you normally think of a track in a sequencer. A sequencer track contains a series of *events*. For example, the first event in the track may be to sound a middle C note. The second event may be to sound the E above middle C. These two events may both happen at the same time. The third event may be to release the middle C note.

This event may happen a few musical beats after the first two events (ie, the middle C note is held down for a few musical beats). Each event has a "time" when it must occur, and the events are arranged within a "chunk" of memory in the order that they occur.

In a MIDI file, an event's "time" precedes the data bytes that make up that event itself. In other words, the bytes that make up the event's time-stamp come first. A given event's time-stamp is referenced from the previous event. For example, if the first event occurs 4 clocks after the start of play, then its "delta-time" is 04. If the next event occurs simultaneously with that first event, its time is 00. So, a delta-time is the duration (in clocks) between an event and the preceding event.

NOTE: Since all tracks start with an assumed time of 0, the first event's delta-time is referenced from 0.

A delta-time is stored as a series of bytes which is called a *variable length quantity.* Only the first 7 bits of each byte is significant (right-justified; sort of like an ASCII byte). So, if you have a 32-bit delta-time, you have to unpack it into a series of 7-bit bytes (ie, as if you were going to transmit it over midi in a SYSEX message). Of course, you will have a variable number of bytes depending upon your delta-time. To indicate which is the last byte of the series, you leave bit #7 clear. In all of the preceding bytes, you set bit #7. So, if a delta-time is between 0-127, it can be represented as one byte. The largest delta-time allowed is 0FFFFFFF, which translates to 4 bytes variable length. Here are examples of delta-times as 32-bit values, and the variable length quantities that they translate to:

```
NUMBER        VARIABLE QUANTITY
 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
```

Here's some C routines to read and write variable length quantities such as delta-times. With **WriteVarLen()**, you pass a 32-bit value (ie, unsigned long) and it spits out the correct series of bytes to a file. **ReadVarLen()** reads a series of bytes from a file until it reaches the last byte of a variable length quantity, and returns a 32-bit value.

```
void WriteVarLen(register unsigned long value)
{
   register unsigned long buffer;
   buffer = value &amp. 0x7F;

   while ( (value >>= 7) )
   {
     buffer <<= 8;
     buffer |= ((value &amp. 0x7F) | 0x80);
```

```
   }

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


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

    if ( (value = getc(infile)) &amp. 0x80 )
    {
       value &amp.= 0x7F;
       do
       {
          value - (value << 7) + ((c = getc(infile)) &amp. 0x7F);
       } while (c &amp. 0x80);
    }

    return(value);
}
```

NOTE: The concept of variable length quantities (ie, breaking up a large value into a series of bytes) is used with other fields in a MIDI file besides delta-times, as you'll see later.

13.6 Events

The first (1 to 4) byte(s) in an MTrk will be the first event's delta-time as a variable length quantity. The next data byte is actually the first byte of that event itself. I'll refer to this as the event's *Status*. For MIDI events, this will be the actual MIDI Status byte (or the first midi data byte if running status). For example, if the byte is hex 90, then this event is a *Note-On* upon midi channel 0. If for example, the byte was hex 23, you'd have to recall the previous event's status (ie, midi running status). Obviously, the first MIDI event in the MTrk **must** have a status byte. After a midi status byte comes its 1 or 2 data bytes (depending upon the status - some MIDI messages only have 1 subsequent data byte). After that you'll find the next event's delta time (as a variable quantity) and start the process of reading that next event.

SYSEX (system exclusive) events (status = F0) are a special case because a SYSEX event can be any length. After the F0 status (which is always stored – no running status here), you'll find yet another series of variable length bytes. Combine them with ReadVarLen() and you'll come up

with a 32-bit value that tells you how many more bytes follow which make up this SYSEX event. This length doesn't include the F0 status.

For example, consider the following SYSEX MIDI message:

```
F0 7F 7F 04 01 7F 7F F7
```

This would be stored in a MIDI file as the following series of bytes (minus the delta-time bytes which would precede it):

```
F0 06 7F 04 01 7F 7F F7
```

Some midi units send a system exclusive message as a series of small "packets" (with a time delay inbetween transmission of each packet). The first packet begins with an F0, but it doesn't end with an F7. The subsequent packets don't start with an F0 nor end with F7. The last packet doesn't start with an F0, but does end with the F7. So, between the first packet's opening F0 and the last packet's closing F7, there's 1 SYSEX message there. (Note: only extremely poor designs, such as the crap marketed by Casio exhibit these aberrations). Of course, since a delay is needed inbetween each packet, you need to store each packet as a separate event with its own time in the MTrk. Also, you need some way of knowing which events shouldn't begin with an F0 (ie, all of them except the first packet). So, the MIDI file spec redefines a midi status of F7 (normally used as an end mark for SYSEX packets) as a way to indicate an event that doesn't begin with F0. If such an event follows an F0 event, then it's assumed that the F7 event is the second "packet" of a series. In this context, it's referred to as a SYSEX CONTINUATION event. Just like the F0 type of event, it has a variable length followed by data bytes. On the other hand, the F7 event could be used to store MIDI REALTIME or MIDI COMMON messages. In this case, after the variable length bytes, you should expect to find a MIDI Status byte of F1, F2, F3, F6, F8, FA, FB, FC, or FE. (Note that you wouldn't find any such bytes inside of a SYSEX CONTINUATION event). When used in this manner, the F7 event is referred to as an ESCAPED event.

A status of FF is reserved to indicate a special non-MIDI event. (Note that FF is used in MIDI to mean "reset", so it wouldn't be all that useful to store in a data file. Therefore, the MIDI file spec arbitrarily redefines the use of this status). After the FF status byte is another byte that tells you what *Type* of non-MIDI event it is. It's sort of like a second status byte. Then after this byte is another byte(s – a variable length quantity again) that tells how many more data bytes follow in this event (ie, its Length). This Length doesn't include the FF, Type byte, nor the Length byte. These special, non-MIDI events are called *Meta-Events*, and most are optional unless otherwise noted. What follows are some defined Meta-Events (including the FF Status and Length). Note that unless otherwise mentioned, more than one of these events can be placed in an Mtrk (even the same Meta-Event) at any delta-time. (Just like all midi events, Meta-Events have a delta-time from the previous event regardless of what type of event that may be. So, you can freely intermix MIDI and Meta events).

13.6.1 Sequence Number

```
FF 00 02 ss ss
```

This optional event which must occur at the beginning of a MTrk (ie, before any non-zero delta-times and before any midi events) specifies the number of a sequence. The two data bytes **ss ss**, are that number which corresponds to the *MIDI Cue* message. In a format 2 MIDI file, this number

identifies each "pattern" (ie, Mtrk) so that a "song" sequence can use the MIDI Cue message to refer to patterns. If the bf ss ss numbers are omitted (ie, Length byte = 0 instead of 2), then the MTrk's location in the file is used (ie, the first MTrk chunk is the first pattern). In format 0 or 1, which contain only one "pattern" (even though format 1 contains several MTrks), this event is placed in only the first MTrk. So, a group of format 1 files with different sequence numbers can comprise a "song collection".

There can be only one of these events per MTrk chunk in a Format 2. There can be only one of these events in a Format 0 or 1, and it must be in the first MTrk.

13.6.2 Text

```
FF 01 len text
```

Any amount of text (amount of bytes = **len**) for any purpose. It's best to put this event at the beginning of an MTrk. Although this text could be used for any purpose, there are other text-based Meta-Events for such things as orchestration, lyrics, track name, etc. This event is primarily used to add "comments" to a MIDI file which a program would be expected to ignore when loading that file.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.3 Copyright

```
FF 02 len text
```

A copyright message (ie, text). It's best to put this event at the beginning of an MTrk.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.4 Sequence/Track Name

```
FF 03 len text
```

The name of the sequence or track (ie, text). It's best to put this event at the beginning of an MTrk.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.5 Instrument

```
FF 04 len text
```

The name of the instrument that the track plays (ie, text). This might be different than the Sequence/Track Name. For example, maybe the name of your sequence (ie, Mtrk) is "Butterfly", but since the track is played on a piano, you might also include an Instrument Name of "Piano".

It's best to put one (or more) of this event at the beginning of an MTrk to provide the user with identification of what instrument(s) is playing the track. Usually, the instruments (ie, patches, tones, banks, etc) are setup on the audio devices via *MIDI Program Change* events within the

MTrk, particularly in MIDI files that are intended for *General MIDI Sound Modules*. So, this event exists merely to provide the user with visual feedback of the instrumentation for a track.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.6 Lyric

```
FF 05 len text
```

A song lyric (ie, text) which occurs on a given beat. A single Lyric MetaEvent should contain only one syllable.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.7 Marker

```
FF 06 len text
```

A marker (ie, text) which occurs on a given beat. Marker events might be used to denote a loop start and loop end (ie, where the sequence loops back to a previous event).

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.8 Cue Point

```
FF 07 len text
```

A cue point (ie, text) which occurs on a given beat. A Cue Point might be used to denote where a WAVE (ie, sampled sound) file starts playing, where the **text** would be the WAVE's filename.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.6.9 MIDI Port

```
FF 21 01 pp
```

This optional event which normally occurs at the beginning of an MTrk (ie, before any non-zero delta-times and before any midi events) specifies out of which MIDI Port (ie, buss) the MIDI events in the MTrk go. The data byte **pp**, is the port number, where 0 would be the first MIDI buss in the system.

The MIDI spec has a limit of 16 MIDI channels per MIDI input/output (ie, port, buss, jack, or whatever terminology you use to describe the hardware for a single MIDI input/output). The MIDI channel number for a given event is encoded into the lowest 4 bits of the event's Status byte. Therefore, the channel number is always 0 to 15. Many MIDI interfaces have multiple MIDI input/output busses in order to work around limitations in the MIDI bandwidth (ie, allow the MIDI data to be sent/received more efficiently to/from several external modules), and to give the musician more than 16 MIDI Channels. Also, some sequencers support more than one MIDI interface used for simultaneous input/output. Unfortunately, there is no way to encode more than 16 MIDI channels into a MIDI status byte, so a method was needed to identify events that would be output on, for example, channel 1 of the second MIDI port versus channel 1 of the first MIDI

port. This MetaEvent allows a sequencer to identify which MTrk events get sent out of which MIDI port. The MIDI events following a MIDI Port MetaEvent get sent out that specified port.

13.6.10 End of Track

`FF 2F 00`

This event is NOT optional. It must be the last event in every MTrk. It's used as a definitive marking of the end of an MTrk. Only 1 per MTrk.

13.6.11 Tempo

`FF 51 03 tt tt tt`

Indicates a tempo change. The 3 data bytes of **tt tt tt** are the tempo in microseconds per MIDI quarter note. Another way of expressing "microsecs per quarter note" is "24ths of a microsecs per MIDI clock" since there are 24 midi clocks in each quarter note. Representing tempos as time per beat instead of beat per time allows exact, long-term synch with time-based protocols like SMPTE. To convert this value to beats per minute:

`TempoBPM = 60,000,000/(tt tt tt)`

For example, a tempo of 120 BPM = `07 A1 20`.

13.6.12 SMPTE Offset

`FF 54 05 hr mn se fr ff`

Designates the SMPTE start time (hours, minutes, secs, frames, subframes) of the Mtrk. It should be at the start of the MTrk. The hour should not be encoded with the SMPTE format as it is in *MIDI Time Code*. In a format 1 file, the SMPTE OFFSET must be stored with the tempo map (ie, the first MTrk), and has no meaning in any other MTrk. The **ff** field contains fractional frames in 100ths of a frame, even in SMPTE based MTrks which specify a different frame subdivision for delta-times (ie, different from the subframe setting in the MThd).

13.6.13 Time Signature

`FF 58 04 nn dd cc bb`

Time signature is expressed as 4 numbers. **nn** and **dd** represent the "numerator" and "denominator" of the signature as notated on sheet music. The denominator is a negative power of 2: 2 = quarter note, 3 = eighth, etc. The **cc** 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 event allows a program to relate what MIDI thinks of as a quarter, to something entirely different. For example, 6/8 time with a metronome click every 3 eighth notes and 24 clocks per quarter note would be the following event:

`FF 58 04 06 03 24 08`

13.6.14 Key Signature

```
FF 59 02 sf mi
```

sf = -7 for 7 flats, -1 for 1 flat, etc, 0 for key of c, 1 for 1 sharp, etc.

mi = 0 for major, 1 for minor

13.6.15 Proprietary Event

```
FF 7F len data...
```

This can be used by a program to store proprietary data. The first byte(s) should be a unique ID of some sort so that a program can identity whether the event belongs to it, or to some other program. A 4 character (ie, ascii) ID is recommended for such.

Note that **len** could be a series of bytes since it is expressed as a variable length quantity.

13.7 Errata

In a format 0 file, the tempo and time signature changes are scattered throughout the one MTrk. In format 1, the very first MTrk should consist of just the tempo and time signature events so that it could be read by some device capable of generating a "tempo map". In format 2, each MTrk should begin with at least one initial tempo and time signature event.

NOTE: If there are no tempo and time signature events in a MIDI file, assume 120 BPM and 4/4.

13.8 RMID Files

The method of saving data in chunks (ie, where the data is preceded by an 8 byte header consisting of a 4 char ID and a 32-bit size field) is the basis for Interchange File Format. You should now read the article About Interchange File Format (see Chapter 14 on page 57) for background information.

As mentioned, MIDI File format is a "broken" IFF. It lacks a file header at the start of the file. One bad thing about this is that a standard IFF parsing routine will choke on a MIDI file (because it will expect the first 12 bytes to be the group ID, filesize, and type ID fields). In order to fix the MIDI File format so that it strictly adheres to IFF, Microsoft simply made up a 12-byte header that is prepended to MIDI files, and thereby came up with the RMID format. An RMID file begins with the group ID (4 ascii chars) of 'R', 'I', 'F', 'F', followed by the 32-bit filesize field, and then the type ID of 'R', 'M', 'I', 'D'. Then, the chunks of a MIDI file follow (ie, the MThd and MTrk chunks). If you chop off the first 12 bytes of an RMID file, then you end up with a standard MIDI file.

Note that chunks within a MIDI file are **not** padded out (with an extra 0 byte) to an even number of bytes. I don't know as if the RMID format corrects this aberration of the MIDI file format too.

Chapter 14

About Interchange File Format

Electronic Arts is a company that deserves credit for helping make life easier for both programmers and end users. By establishing **Interchange Format Files** (ie, IFF) and releasing the documentation for such, as well as C source code for reading and writing IFF type of files, Electronic Arts has helped make it easier for programmers to develop "backward compatible" and "extensible" file formats. IFF also helps developers write programs that easily read data files created each others' IFF compliant software, even if there is no business relationship between the developers. In a nutshell, IFF helps minimize problems such as new versions of a particular program having trouble reading data files produced by older versions, or needing a new file format everytime a new version needs to store additional information. It also encourages standardized file formats that aren't tied to a particular product. All of this is good for endusers because it means that their valuable data isn't locked into some proprietary standard that can't be used with a wide variety of hardware and software. Above all else, endusers don't want their work to be held hostage by a single, corporate entity over whom the enduser has no direct control, but that's exactly what happens whenever an enduser saves his data using a program that produces a proprietary, unpublished file format. IFF helps to break this needlessly proprietary stranglehold that developers have exerted upon endusers' works.

An IFF file is a set of data that is in a form that many, unrelated programs can read. An IFF file should not have anything in it that was intended specifically for just one, particular program. If a program must save some "personal" (ie, proprietary) data in an IFF file, it must be saved in a manner which allows another program to "skip over" this data. There are several different types of IFF files. ILBM and GIFF files store picture data. SMUS files store musical scores. WAVE and AIFF files store sampled sounds. Each of these files must start with an ID which indicates that it is indeed an IFF file, followed by an ID that indicates which type of file. So what is an ID? An ID is four, printable ascii characters (ie, 8-bit bytes). If you use a file viewer (capable of displaying each byte as an ascii character) to look at an IFF file, you will notice that every so often you will see 4 "readable" characters in a row. These 4 characters are an ID. Every IFF file must start with one of the following 3 IDs. (I've enclosed each ID in single quotes).

```
'FORM'  'LIST'  'CAT '
```

If the first 4 chars (bytes) in a file are not one of these, then it is not an IFF file. These IDs are referred to as **group ID**s in EA literature because each is like a "master ID" after which there may follow more IDs (ie, chunks) that are grouped under that master ID.

Note that the last character in the 'CAT ' ID is a blank space (ie, ascii 32).

After this group ID, there is an UNSIGNED LONG (ie, 32-bit binary value) that indicates how many bytes are in the entire file. This count does not include the 4 byte group ID, nor this ULONG. This ULONG is useful if you wish to load the rest of the file into memory to examine it. After this ULONG, there is an ID that indicates which type of IFF file this is. As mentioned earlier, "ILBM", "WAVE", and "AIFF" are 3 types of IFF files. There are many more, and programmers are always inventing new types for lack of better things to do. Here is the beginning of a typical ILBM file.

```
'FORM' <- OK. This really is an IFF file because it has one
          of the 3 defined group IDs.
13000  <- There are 13000 more bytes after this ULONG
'ILBM' <- It is an ILBM (picture) file
```

All IFF files start with something similiar to the above, 12 byte "header", except that instead of 'FORM', the group ID can be 'LIST' or 'CAT '. Of course, the ULONG size and file type ID may be different in various files, but nevertheless, a 12 byte header always appears at the beginning of an IFF file. For example, here's an example AIFF header:

```
'FORM' <- OK. This really is an IFF file because it has one
          of the 3 defined group IDs.
4000   <- There are 4000 more bytes after this ULONG
'AIFF' <- It is an AIFF (digital audio) file
```

What you find after the header depends on which type it is (ie, From here on, an ILBM will be different than an AIFF).

One thing that all IFF files do have in common after the group ID, byte count, and type ID, is that data is organized into chunks. OK, more jargon. What's a chunk? A chunk consists of an ID, a ULONG that tells how many bytes of data are in the chunk, and then all those data bytes. For example, here is a CMAP chunk (which would be found in an ILBM file).

```
'CMAP'       <- This is the 4 byte chunk ID
6            <- This tells how many data bytes are in the chunk chunkSize).
0,0,0,1,1,4  <- Here are the 6 data bytes
```

Notice that the chunk size doesn't include the 4 byte ID or the ULONG for the chunk Size.

So, all IFF files are made up of several chunks (ie, groups of data). Each group of data starts with a convenient ID (so that a program can ascertain what kind of data is in the chunk) and a ULONG size (so that a program can ascertain how many bytes of data are in the chunk). There are a few other details to note. A chunk cannot have an odd number of data bytes (such as 3). If necessary, an extra zero byte must be written to make an even number of data bytes. The chunk Size doesn't include this extra byte. So for example, if you want to write 3 bytes in a CMAP chunk, it would look like this:

```
'CMAP'
3          <- Note that chunk Size is 3
0,1,33,0   <- Note that there is an extra zero byte
```

The reason for this extra "pad byte" for odd-sized chunks has to do with Motorola's 68000 CPU requiring that LONGs be aligned to even memory addresses. IFF files were first used on 68000 based computers, and padding out odd-sized chunks made it easier to load and parse an IFF file on such a computer (ie, if you load the entire file into a single block of RAM starting upon an even address, all of chunk IDs and Sizes will conveniently fall upon even memory addresses).

In the preceding example, the group ID was 'FORM'. There are 2 other group IDs as well. A 'CAT ' is a collection of many different FORMs all stuck together consecutively in 1 IFF file. For example, if you had an animation with 6 sound effects, you might save the animation frames in an ANIM FORM, and you might save the sound effects in several AIFF FORMs (one per sound effect). You could save the animation and sound in 7 separate files. The ANIM file would start this way:

```
FORM
120000  <- Whatever the size happens to be (this is expressed in 32 bits)
ANIM
```

Each AIFF file would start this way:

```
FORM
8000    <- whatever size
AIFF
```

If the user wanted to copy the data to another disk, he would have to copy 7 files. On the other hand, you could save all the data in one CAT file.

```
CAT
4+120008+8008+2028+...  <- The total size of the ANIM and the 6 AIFF files
'      '                <- Type of CAT. 4 spaces for the type ID means "a
                           grab bag" of IFF FORMs are going to be inside of
                           this CAT. If it just so happened that all of the
                           enclosed FORMs were 1 type, such as ILBM, then
                           this type ID would be 'ILBM'.
FORM
120000
ANIM
...all the chunks in the ANIM file placed here (note: ANIMs have imbedded
   ILBM FORMs. The guy who devised the ANIM type of IFF file broke the
   rules by mistake, and nobody caught his error until it was too late).

FORM
8000
AIFF
...all the chunks in the first sound effect here

FORM
2020
AIFF
...all the chunks in the second sound effect here

...etc. for the other 4 sound effects
```

To further complicate matters, there are LISTs. LISTs are a lot like CATs except that there is an optional, additional group ID associated with LISTs. That ID is a PROP. LISTs can have imbedded PROPS just like an ILBM can have an imbedded CMAP chunk. A PROP header looks very much like a FORM header in that you must follow it with a type ID. For example, here is an ILBM PROP with a CMAP in it.

```
PROP        <- Here's a PROP
4+14        <- Here's how many bytes follow in the PROP
ILBM        <- It's an ILBM PROP
CMAP        <- Here's a CMAP chunk inside of this ILBM PROP
6           <- There are 6 bytes following in this CMAP chunk
0,0,0,1,1,4
```

LISTs are meant to encompass similiar FORMs (i.e. several AIFF files stuck together). Often, when you have similiar FORMs stuck together, some of the chunks in the individual FORMs are the same. For example, assume that we have 2 AIFF sound effects. AIFF FORMs can have a NAME chunk which contains the ascii string that is the name of the sound effect. Also assume that both sounds are called "car crash". With a CAT, we end up having to identical NAME chunks in each AIFF FORM like so:

```
CAT           <- We put the 2 files into 1 CAT
4+1008+508
AIFF          <- It's an CAT of several AIFF FORMs

FORM          <- here's the start of the first sound effect file
1000
AIFF

...some chunks

NAME          <- here's the name chunk for the 1st sound effect
9
'car crash',0

...more chunks

FORM          <- here's the start of the second sound effect file
500
AIFF

...some chunks

NAME          <- here's the name chunk for the 2nd sound effect.
9                Look familiar?
'car crash',0

...more chunks
```

With a LIST, we can have PROPs. A PROP is group ID that allows us to place chunks that pertain to all the FORMs in the LIST. So, we can rip out the NAME chunks inside both AIFF

FORMs and replace it with one NAME chunk inside of a PROP.

```
LIST         <- Notice that we use a LIST instead of a CAT
4+30+990+490+...
AIFF

PROP         <- Here's where we put chunks intended for ALL the
22              subsequent FORMS; inside a PROP.
AIFF         <- type of PROP
NAME         <- here's the name chunk inside of the PROP
9
'car crash',0


FORM         <- here's the start of the first sound effect file
982          <- size is 18 bytes less because no NAME chunk here
AIFF

...some chunks, but no NAME chunk

FORM         <- here's the start of the second sound effect file
482
AIFF

...some chunks, but no NAME for this guy either
```

Notice that the PROP group ID is followed by a type ID (in this case AIFF). This means that the PROP only affects any AIFF FORMs. If you were to sneak in an SMUS FORM at the end, the NAME chunk would not apply to it. Also, if you included a NAME chunk in one of the AIFF FORMs, it would override the PROP. For example, assume that you have a LIST containing 10 AIFF FORMs. All but 1 of them is named "Snare Hit". You can store a NAME chunk in a PROP AIFF for "Snare Hit". Then, in the one AIFF FORM whose name is not "Snare Hit", you can include a NAME chunk to override the NAME chunk in the PROP.

It should be noted that you can take several LISTs and squash them together inside of a CAT or another LIST. Personally, I have never seen a data file with this level of nesting, and doubt that it would be of much use.

In the above examples, psuedo code was used to represent the headers. Let's look at how a hex file viewer might display the actual contents of an IFF file (in hex bytes). First, an IFF header for a FORM AIFF, psuedo code.

```
FORM
4096
AIFF
```

Now here's a view of the actual data file.

```
46 4F 52 4D      <- FORM
00 00 10 00      <- hex 00001000, or 4096 decimal
41 49 46 46      <- AIFF
```

Note that the ULONG byte count is stored in Big Endian order (ie, the Most Significant Byte is first, and the Least Significant Byte is last). This is how the Motorola 680x0 stores long values in memory (ie, the opposite order of Intel 80x86). IFF files use Big Endian order for all 16-bit (ie, SHORT) and 32-bit (ie, LONG) values.

Microsoft, never one to NOT take someone else's ideas, alter them slightly, and then make money off of those ideas (keeping all of the profit in the MS vaults), decided that IFF was a good idea, but since MS is closely tethered to Intel CPUs, a version of IFF was needed which stored LONG or SHORT values in Little Endian order. So, MS decided to create some new group IDs. MS took the FORM ID and created a Little Endian version of it known as RIFF. For example, the WAVE file format has a RIFF group ID. All of the SHORT and LONG values in the file are stored in Little Endian order. Let's take a look at an example header for a WAVE file. Assume that there are 258 bytes of data after the byte count.

```
52 49 46 46      <- RIFF
02 01 00 00      <- hex 00000102, or 258 decimal
57 41 56 45      <- WAVE
```

Note that the ULONG byte count is stored in Little Endian order (ie, the Least Significant Byte is first, and the Most Significant Byte is last). Good old backwards-thinking Intel.

Now, there's some real justification for creating a RIFF group ID, if you're working with an Intel CPU. But Microsoft couldn't stop there. True to their "not made here, so if we're going to accept it, we have to inflict our brutish, unneeded brand upon it" mentality, Microsoft created another group ID called RIFX. What's an RIFX file? It's simply a FORM with RIFX replacing the FORM ID. Ah yes, yet another example of useful Microsoft work. So, if you want to turn a FORM AIFF into a RIFX AIFF, you just change the first 4 bytes to RIFX. Needless to say, nobody has ever used the RIFX group ID, and it will undoubtably suffer a justifiably ignoble disappearance, much like MS Bob.

Just like everyone else, programmers make mistakes. As mentioned before, the Amiga's ANIM file format was a mistake. It puts FORM headers inside of a FORM group ID. That's not supposed to happen. You can put FORM headers inside of a CAT or LIST, but not another FORM. A mistake was also made with the MIDI file format. The programmer who devised it didn't put a proper IFF header on the file. It should be:

```
FORM   <- group ID. Indicates an IFF file that contains one type of data
3000   <- whatever size the file happens to be
MIDI   <- type of data. What follows will be chunks as defined by the
          MIDI type of IFF file.
```

But the programmer omitted the FORM group ID, and simply put the MThd chunk first. So, a MIDI file starts as so:

```
MThd   <- Chunk ID
6      <- size of MThd chunk
```

Another deviation from the standard occurs with padding out odd-sized chunks with an extra byte. Some programmers didn't bother doing this when devising new IFF type files, and occasionally, one will come across some specification for a new IFF type that allows odd-sized chunks.

Unfortunately, these programmers released their work based upon these aberrations before getting that work reviewed by other programmers who might have offered good reasons why the aberrations should be corrected. It makes it that much harder for software to read and write files if it has to deal with aberrations of the IFF standard. There's no reason for that, particularly when a strict adherence to the standard sacrifices almost nothing in the way of quality and efficiency over an aberration. But try to tell that to a paranoid programmer who thinks that if he shows anyone what he's doing before his product is shrink-wrapped, someone will steal his soul... well, IFF does give the computer industry a means for resolving needless hassles with data file formats, and it has worked very successfully in a number of instances, although occasionally people don't always use the standard wisely, or don't quite grasp EA's altruistic notion that there is no good reason why a file format should ever be proprietary or unpublished. (I urge consumers to avoid products where that is the case).

Chapter 15

MIDI Time Code and Cueing

Detailed Specification

(Supplement to MIDI 1.0)

12 February 1987

15.1 Justification For MIDI Time Code and Cueing

The merit of implementing the MIDI Time Code proposal within the current MIDI specification is as follows:

SMPTE has become the de facto timing reference standard in the professional audio world and in almost the entire video world. SMPTE is also seeing more and more use in the semi-professional audio area. We hope to combine this universal timing reference, SMPTE, with the de facto standard for controlling musical equipment, MIDI.

Encoding SMPTE over MIDI allows a person to work with one timing reference throughout the entire system. For example, studio engineers are more familiar with the idea of telling a multitrack recorder to punch in and out of record mode at specific SMPTE times, as opposed to a specific beat in a specific bar. To force a musician or studio engineer to convert back and forth between a SMPTE time and a specific bar number is tedious and should not be necessary (one would have to take into account tempo and tempo changes, etc.).

Also, some operations are referenced only as SMPTE times, as opposed to beats in a bar. For example, creating audio and sound effects for video requires that certain sounds and sequences be played at specific SMPTE times. There is no other easy way to do this with Song Position Pointers, etc., and even if there was, it would be an unnatural way for a video or recording engineer to work.

MIDI Time Code is an absolute timing reference, whereas MIDI Clock and Song Position Pointer are relative timing references. In virtually all audio for film/video work, SMPTE is already being used as the main time base, and any musical passages which need to be recorded are usually done by getting a MIDI-based sequencer to start at a pre-determined SMPTE time code. In most cases, though, it is SMPTE which is the Master timing reference being used. In order for MIDI-based devices to operate on an absolute time code which is independent of tempo, MIDI Time Code must

be used. Existing devices merely translate SMPTE into MIDI Clocks and Song Position Pointers based upon a given tempo. This is not absolute time, but relative time, and all of the SMPTE cue points will change if the tempo changes. The majority of sound effects work for film and video does not involve musical passages with tempos, rather it involves individual sound effect "events" which must occur at specific, absolute times, not relative to any "tempo".

15.2 MIDI Time Code System Components

15.2.1 SMPTE to MTC Converter

This box would either convert longitudinal (audio-type) or vertical (video-type) SMPTE time code from a master timing device into MTC. The function could be integrated into video tape recorders (VTRs) or syncronization units that control audio tape recorders (ATRs). Alternately, a stand-alone box would do the conversion, or simply generate MTC directly. Note that conversion from MTC to SMPTE time code is not envisioned, as it is of little practical value.

15.2.2 Cue List Manager

This would be a device or computer program that would maintain a cue list of desired events, and send the list to the slaves. For performance, the manager might pass the Time Code from the SMPTE-MTC converter through to the slaves, or, in a stand-alone system it might generate Time Code itself. This "central controller" would presumably also contain all library functions for downloading sound programs, samples, sequences, patterns, and so on, to the slaves. A Cue List Manager would pre-load intelligent MTC peripherals (see below) with this data.

15.2.3 MTC Sequencer

To control existing equipment or any device which does not recognize MTC in an MTC system, this device would be needed. It would receive the cue list from the manager, and convert the cues into normal MIDI commands. At the specified SMPTE times, the sequencer would then send the MIDI commands to the specific devices. For example, for existing MIDI equipment it might provide MIDI messages such as Note On, Note Off, Song Select, Start, Stop, Program Changes, etc. Non-MIDI equipment (such as CD players, mixing consoles, lighting, sound effects cartridge units and ATRs) may also be controlled if such a device had relay controls.

15.2.4 Intelligent MTC Peripheral

In this category belong devices capable of receiving an MTC Cue List from the manager, and triggering themselves appropriately when the correct Time Code (SMPTE or MIDI) has been received. Above this minimum, the device might be able to change its programming in response to the Cue List, or prepare itself for ensuing events.

For example, an intelligent MTC-equipped analog multitrack tape machine might read in a list of punch in/punch out cues from the Cue List Manager, and then alter then to internally compensate for its bias current rise and fall times. A sampling-based sound effects device might preload samples

from its own disk drive into a RAM buffer, in anticipation of needing them for cues later on in the cue list.

It should be mentioned that while these functions are separately described, actual devices may incorporate a mixture of these functions, suited to specific applications in their market.

15.3 A MIDI Time Code System

The MIDI Time Code format contains two parts: Time Code and Set Up. Time Code is relatively straightforward: hours, minutes, seconds and frame numbers (approximately 1/30 of a second) are encoded and distributed throughout the MIDI system so that all the units know exactly what time it is.

Set Up, however, is where MTC gains its power. It is a format for informing MIDI devices of events to be performed at specific times. Ultimately, this aspect of MTC will lead to the creation of an entirely new class of production equipment. Before getting into the nuts and bolts of the spec itself, let's talk about some of the uses and features of forthcoming devices that have been envisioned.

Set Up begins with the concept of a cue list. In video editing, for example, it is customary to transfer the video master source tapes, which may be on expensive, two-inch recorders, to less-expensive recorders. The editing team then works over this copy, making a list of all the segments that they want to piece together as they are defined by their SMPTE times.

For example, the first scene starts at time A and ends at time B, the next scene starts at time C and ends at time D. A third scene may even lie between the first two. When done, they feed this cue list time information into the editing system of the master recorder(s) or just give the cue list to an editor who does the work manually. The editing system or editor then locates the desired segments and assembles them in the proper sequence.

Now suppose that instead of one or two video recorders, we have twenty devices that will play a part in our audio/video or film production: special effects generators for fades and superimpositions, additional decks with background scenery, live cameras, MIDI sequencers, drum machines, synthesizers, samplers, DDLs, soundtrack decks, CDs, effects devices, and so on. As it stands now, each of these devices must be handled more or less separately, with painstaking and time-consuming assembly editing or multitrack overdubs. And when a change in the program occurs (which always happens), anywhere from just a few items to the whole system may need to be reprogrammed by hand.

This is where MIDI Time Code comes in. It can potentially control all of these individual production elements so that they function together from a single cue list. The master controller which would handle this function is described as a Cue List Manager. On such a console, you would list what you want each device to do, and when to do it. The manager would then send the cue list to the various machines via the MTC Set Up protocol. Each unit would then react as programmed when the designated MIDI Time Code (or conventional SMPTE Time Code) appears. Changes? No problem. Simply edit the cue list using simple word-processing functions, then run the tape again.

MTC thus integrates into a manageable system all of the diverse tools at our disposal. It would drastically reduce the time, money and frustration needed to produce a film or video.

Having covered the basic aspects of a MIDI Time Code system, as well as examples of how an

overall system might function, we will now take a look at the actual MIDI specification itself.

15.4 MIDI Time Code

For device synchronization, MIDI Time Code uses two basic types of messages, described as Quarter Frame and Full. There is also a third, optional message for encoding SMPTE user bits.

15.4.1 Quarter Frame Messages

Quarter Frame messages are used only while the system is running. They are rather like the PPQN or MIDI clocks to which we are accustomed. But there are several important ways in which Quarter Frame messages differ from the other systems.

As their name implies, they have fine resolution. If we assume 30 frames per second, there will be 120 Quarter Frame messages per second. This corresponds to a maximum latency of 8.3 milliseconds (at 30 frames per second), with accuracy greater than this possible within the specific device (which may interpolate inbetween quarter frames to "bit" resolution). Quarter Frame messages serve a dual purpose: besides providing the basic timing pulse for the system, each message contains a unique nibble (four bits) defining a digit of a specific field of the current SMPTE time.

Quarter frames messages should be thought of as groups of eight messages. One of these groups encodes the SMPTE time in hours, minutes, seconds, and frames. Since it takes eight quarter frames for a complete time code message, the complete SMPTE time is updated every two frames. Each quarter frame message contains two bytes. The first byte is F1, the Quarter Frame System Common byte. The second byte contains a nibble that represents the message number (0 through 7), and a nibble for one of the digits of a time field (hours, minutes, seconds or frames).

15.4.2 Quarter Frame Messages (2 bytes):

```
F1  <message>


     F1 = Currently unused and undefined System Common status byte

     <message> = 0nnn dddd


          dddd = 4 bits of binary data for this Message Type

          nnn =  Message Type:

               0 = Frame count LS nibble

               1 = Frame count MS nibble
```

```
2 = Seconds count LS nibble

3 = Seconds count MS nibble

4 = Minutes count LS nibble

5 = Minutes count MS nibble

6 = Hours count LS nibble

7 = Hours count MS nibble and SMPTE Type
```

After both the MS nibble and the LS nibble of the above counts are assembled, their bit fields are assigned as follows:

```
FRAME COUNT:  xxx yyyyy


              xxx = undefined and reserved for future use.  Transmitter

                   must set these bits to 0 and receiver should ignore!

              yyyyy = Frame number (0-29)


SECONDS COUNT:  xx yyyyyy


              xx = undefined and reserved for future use.  Transmitter

                    must set these bits to 0 and receiver should ignore!

              yyyyyy = Seconds Count  (0-59)


MINUTES COUNT:  xx yyyyyy


              xx = undefined and reserved for future use.  Transmitter

                    must set these bits to 0 and receiver should ignore!

              yyyyyy = Minutes Count  (0-59)
```

```
HOURS COUNT:  x yy zzzzz


              x = undefined and reserved for future use.  Transmitter

                    must set this bit to 0 and receiver should ignore!


              yy = Time Code Type:

                   0 = 24 Frames/Second

                   1 = 25 Frames/Second

                   2 = 30 Frames/Second (Drop-Frame)

                   3 = 30 Frames/Second (Non-Drop)


              zzzzz = Hours Count  (0-23)
```

15.4.3 Quarter Frame Message Implementation

When time code is running in the forward direction, the device producing the MIDI Time Code will send Quarter Frame messages at quarter frame intervals in the following order:

```
F1 0X

F1 1X

F1 2X

F1 3X

F1 4X

F1 5X

F1 6X

F1 7X
```

after which the sequence repeats itself, at a rate of one complete 8-message sequence every 2 frames (8 quarter frames). When time code is running in reverse, the quarter frame messages are sent in reverse order, starting with F1 7X and ending with F1 0X. Again, at least 8 quarter frame messages must be sent. The arrival of the F1 0X and F1 4X messages always denote frame boundaries.

Since 8 quarter frame messages are required to definitely establish the actual SMPTE time, timing lock cannot be achieved until the reader has read a full sequence of 8 messages, from first message to last. This will take from 2 to 4 frames to do, depending on when the reader comes on line.

During fast forward, rewind or shuttle modes, the time code generator should stop sending quarter frame messages, and just send a Full Message once the final destination has been reached. The generator can then pause for any devices to shuttle to that point, and resume by sending quarter frame messages when play mode is resumed. Time is considered to be "running" upon receipt of the first quarter frame message after a Full Message.

Do not send quarter frame messages continuously in a shuttle mode at high speed, since this unnecessarily clogs the MIDI data lines. If you must periodically update a device's time code during a long shuttle, then send a Full Message every so often.

The quarter frame message F1 0X (Frame Count LS nibble) must be sent on a frame boundary. The frame number indicated by the frame count is the number of the frame which starts on that boundary. This follows the same convention as normal SMPTE longitudinal time code, where bit 00 of the 80-bit message arrives at the precise time that the frame it represents is actually starting. The SMPTE time will be incremented by 2 frames for each 8-message sequence, since an 8-message sequence will take 2 frames to send.

Another way to look at it is: When the last quarter frame message (F1 7X) arrives and the time can be fully assembled, the information is now actually 2 frames old. A receiver of this time must keep an internal offset of +2 frames for displaying. This may seem unusual, but it is the way normal SMPTE is received and also makes backing up (running time code backwards) less confusing - when receiving the 8 quarter frame messages backwards, the F1 0X message still falls on the boundary of the frame it represents.

Each quarter frame message number (0-¿7) indicates which of the 8 quarter frames of the 2-frame sequence we are on. For example, message 0 (F1 0X) indicates quarter frame 0 of frame #1 in the sequence, and message 4 (F1 4X) indicates quarter frame 1 of frame #2 in the sequence. If a reader receives these message numbers in descending sequence, then it knows that time code is being sent in the reverse direction. Also, a reader can come on line at any time and know exactly where it is in relation to the 2-frame sequence, down to a quarter frame accuracy.

It is the responsibility of the time code reader to insure that MTC is being properly interpreted. This requires waiting a sufficient amount of time in order to achieve time code lock, and maintaining that lock until synchronization is dropped. Although each passing quarter frame message could be interpreted as a relative quarter frame count, the time code reader should always verify the actual complete time code after every 8-message sequence (2 frames) in order to guarantee a proper lock.

For example, let's assume the time is 01:37:52:16 (30 frames per second, non-drop). Since the time is sent from least to most significant digit, the first two Quarter Frame messages will contain the data 16 (frames), the second two will contain the data 52 (seconds), the third two will represent 37 (minutes), and the final two encode the 1 (hours and SMPTE Type). The Quarter Frame Messages description defines how the binary data for each time field is spread across two nibbles. This scheme (as opposed to simple BCD) leaves some extra bits for encoding the SMPTE type (and for future use).

Now, let's convert our example time of 01:37:52:16 into Quarter Frame format, putting in the correct hexadecimal conversions:

```
F1 00

F1 11     10H = 16 decimal


F1 24

F1 33     34H = 52 decimal


F1 45

F1 52     25H = 37 decimal


F1 61

F1 76     01H = 01 decimal (SMPTE Type is 30 frames/non-drop)
```

(note: the value transmitted is "6" because the SMPTE Type (11 binary) is encoded in bits 5 and 6)

For SMPTE Types of 24, 30 drop frame, and 30 non-drop frame, the frame number will always be even. For SMPTE Type of 25, the frame number may be even or odd, depending on which frame number the 8-message sequence had started. In this case, you can see where the MIDI Time Code frame number would alternate between even and odd every second.

MIDI Time Code will take a very small percentage of the MIDI bandwidth. The fastest SMPTE time rate is 30 frames per second. The specification is to send 4 messages per frame - in other words, a 2-byte message (640 microseconds) every 8.333 milliseconds. This takes 7.68 % of the MIDI bandwidth - a reasonably small amount. Also, in the typical MIDI Time Code systems we have imagined, it would be rare that normal MIDI and MIDI Time Code would share the same MIDI bus at the same time.

15.4.4 Full Message

Quarter Frame messages handle the basic running work of the system. But they are not suitable for use when equipment needs to be fast-forwarded or rewound, located or cued to a specific time, as sending them continuously at accelerated speeds would unnecessarily clog up or outrun the MIDI data lines. For these cases, Full Messages are used, which encode the complete time into a single message. After sending a Full Message, the time code generator can pause for any mechanical devices to shuttle (or "autolocate") to that point, and then resume running by sending quarter frame messages.

Full Message - (10 bytes)

```
F0 7F <chan> 01 <sub-ID 2> hr mn sc fr F7
```

```
F0 7F = Real Time Universal System Exclusive Header

<chan> = 7F (message intended for entire system)

01 = <sub-ID 1>, 'MIDI Time Code'

<sub-ID 2> = 01, Full Time Code Message

hr = hours and type: 0 yy zzzzz

     yy = type:

          00 = 24 Frames/Second

          01 = 25 Frames/Second

          10 = 30 Frames/Second (drop frame)

          11 = 30 Frames/Second (non-drop frame)

     zzzzz = Hours (00->23)

mn = Minutes (00->59)

sc = Seconds (00->59)

fr = Frames (00->29)

F7 = EOX
```

Time is considered to be "running" upon receipt of the first Quarter Frame message after a Full Message.

15.4.5 User Bits

"User Bits" are 32 bits provided by SMPTE for special functions which vary with the application, and which can be programmed only from equipment especially designed for this purpose. Up to four characters or eight digits can be written. Examples of use are adding a date code or reel number to the tape. The User Bits tend not to change throughout a run of time code.

User Bits Message - (15 bytes)

```
F0 7F <chan> 01 <sub-ID 2> u1 u2 u3 u4 u5 u6 u7 u8 u9 F7
```

```
F0 7F = Real Time Universal System Exclusive Header

<chan> = 7F (message intended for entire system)

01 = <sub-ID 1>, MIDI TIme Code

<sub-id 2> = 02, User Bits Message

u1 = 0000aaaa

u2 = 0000bbbb

u3 = 0000cccc

u4 = 0000dddd

u5 = 0000eeee

u6 = 0000ffff

u7 = 0000gggg

u8 = 0000hhhh

u9 = 000000ii

F7 = EOX
```

These nibble fields decode in an 8-bit format: aaaabbbb ccccdddd eeeeffff gggghhhh ii. It forms 4 8-bit characters, and a 2 bit Format Code. u1 through u8 correspond to SMPTE Binary Groups 1 through 8. u9 are the two Binary Group Flag Bits, as defined by SMPTE.

This message can be sent whenever the User Bits values must be transferred to any devices down the line. Note that the User Bits Message may be sent by the MIDI Time Code Converter at any time. It is not sensitive to any mode.

15.5 MIDI Cueing

MIDI Cueing uses Set-Up Messages to address individual units in a system. (A "unit" can be be a multitrack tape deck, a VTR, a special effects generator, MIDI sequencer, etc.)

Of 128 possible event types, 19 are currently defined.

15.5.1 Set-Up Messages (13 bytes plus any additional information):

```
F0 7E <chan> 04 <sub-ID 2> hr mn sc fr ff sl sm <add. info> F7


     F0 7E = Non-Real Time Universal System Exclusive Header

     <chan> = Channel number
     04 = <sub-ID 1>, MIDI Time Code

     <sub-ID 2> = Set-Up Type
          00 = Special
          01 = Punch In points
          02 = Punch Out points
          03 = Delete Punch In point
          04 = Delete Punch Out point
          05 = Event Start points
          06 = Event Stop points
          07 = Event Start points with additional info.
          08 = Event Stop points with additional info.
          09 = Delete Event Start point
          0A = Delete Event Stop point
          0B = Cue points
          0C = Cue points with additional info
          0D = Delete Cue point
          0E = Event Name in additional info

     hr = hours and type: 0 yy zzzzz

          yy = type:
               00 = 24 Frames/Second
               01 = 25 Frames/Second
               10 = 30 Frames/Second drop frame
               11 = 30 Frames/Second non-drop frame
          zzzzz = Hours (00-23)

     mn = Minutes (00-59)

     sc = Seconds (00-59)

     fr = Frames (00-29)

     ff = Fractional Frames (00-99)

     sl, sm = Event Number (LSB first)

     <add. info.>
```

F7 = EOX

Description of Set-Up Types:

00 Special refers to the set-up information that affects a unit globally (as opposed to individual tracks, sounds, programs, sequences, etc.). In this case, the Special Type takes the place of the Event Number. Five are defined. Note that types 01 00 through 04 00 ignore the event time field.

00 00 Time Code Offset refers to a relative Time Code offset for each unit. For example, a piece of video and a piece of music that are supposed to go together may be created at different times, and more than likely have different absolute time code positions - therefore, one must be offset from the other so that they will match up. Just like there is one master time code for an entire system, each unit only needs one offset value per unit.

01 00 Enable Event List means for a unit to enable execution of events in its list if the appropriate MTC or SMPTE time occurs.

02 00 Disable Event List means for a unit to disable execution of its event list but not to erase it. This facilitates an MTC Event Manager in muting particular devices in order to concentrate on others in a complex system where many events occur simultaneously.

03 00 Clear Event List means for a unit to erase its entire event list.

04 00 System Stop refers to a time when the unit may shut down. This serves as a protection against Event Starts without matching Event Stops, tape machines running past the end of the reel, and so on.

05 00 Event List Request is sent by a master to an MTC peripheral. If the device ID (Channel Number) matches that of the peripheral, the peripheral responds by transmitting its entire cue list as a sequence of Set Up Messages, starting from the SMPTE time indicated in the Event List Request message.

01/02 Punch In and Punch Out refer to the enabling and disabling of record mode on a unit. The Event Number refers to the track to be recorded. Multiple punch in/punch out points (and any of the other event types below) may be specified by sending multiple Set-Up messages with different times.

03/04 Delete Punch In or Out deletes the matching point (time and event number) from the Cue List.

05/06 Event Start and Stop refer to the running or playback of an event, and imply that a large sequence of events or a continuous event is to be started or stopped. The event number refers to which event on the targeted slave is to be played. A single event (ie. playback of a specific sample, a fader movement on an automated console, etc.) may occur several times throughout a given list of cues. These events will be represented by the same event number, with different Start and Stop times.

07/08 Event Start and Stop with Additional Information refer to an event (as above) with additional parameters transmitted in the Set Up message between the Time and EOX. The additional parameters may take the form of an effects unit's internal parameters, the volume level of a sound effect, etc. See below for a description of additional information.

09/0A Delete Event Start/Stop means to delete the matching (event number and time) event (with or without additional information) from the Cue List.

0B Cue Point refers to individual event occurences, such as marking "hit" points for sound effects, reference points for editing, and so on. Each Cue number may be assigned to a specific reaction, such as a specific one-shot sound event (as opposed to a continuous event, which is handled by Start/Stop). A single cue may occur several times throughout a given list of cues. These events will be represented by the same event number, with different Start and Stop times.

0C Cue Point with Additional Information is exactly like Event Start/Stop with Additional Information, except that the event represents a Cue Point rather than a Start/Stop Point.

0D Delete Cue Point means to Delete the matching (event number and time) Cue Event with or without additional information from the Cue List.

0E Event Name in Additional Information. This merely assigns a name to a given event number. It is for human logging purposes. See Additional Information description.

15.5.2 Event Time

This is the SMPTE/MIDI Time Code time at which the given event is supposed to occur. Actual time is in 1/100th frame resoultion, for those units capable of handling bits or some other form of sub-frame resolution, and should otherwise be self-explanatory.

15.5.3 Event Number

This is a fourteen-bit value, enabling 16,384 of each of the above types to be individually addressed. "sl" is the 7 LS bits, and "sm" is the 7 MS bits.

15.5.4 Additional Information description

Additional information consists of a nibblized MIDI data stream, LS nibble first. The exception is Set-Up Type OE, where the additional information is nibblized ASCII, LS nibble first. An ASCII newline is accomplished by sending CR and LF in the ASCII. CR alone functions solely as a carriage return, and LF alone functions solely as a Line-Feed.

For example, a MIDI Note On message such as 91 46 7F would be nibblized and sent as 01 09 06 04 0F 07. In this way, any device can decode any message regardless of who it was intended for. Device-specific messages should be sent as nibblized MIDI System Exclusive messages.

15.6 Potential Problems

There is a possible problem with MIDI merger boxes improperly handling the F1 message, since they do not currently know how many bytes are following. However, in typical MIDI Time Code systems, we do not anticipate applications where the MIDI Time Code must be merged with other MIDI signals occuring at the same time.

Please note that there is plenty of room for additional set-up types, etc., to cover unanticipated situations and configurations.

It is recommended that each MTC peripheral power up with its MIDI Manufacturer's System Exclusive ID number as its default channel/device ID. Obviously, it would be preferable to allow the user to change this number from the device's front panel, so that several peripherals from the same manufacturer may have unique IDs within the same MTC system.

15.7 Signal Path Summary

Data sent between the Master Time Code Source (which may be, for example, a Multitrack Tape Deck with a SMPTE Synchronizer) and the MIDI Time Code Converter is always SMPTE Time Code.

Data sent from the MIDI Time Code Converter to the Master Control/Cue Sheet (note that this may be a MTC-equipped tape deck or mixing console as well as a cue-sheet) is always MIDI Time Code. The specific MIDI Time Code messages which are used depend on the current operating mode, as explained below:

Play Mode: The Master Time Code Source (tape deck) is in normal PLAY MODE at normal or vari-speed rates. The MIDI Time Code Converter is transmitting Quarter Frame ("F1") messages to the Master Control/Cue Sheet. The frame messages are in ASCENDING order, starting with "F1 0X" and ending with "F1 7X". If the tape machine is capable of play mode in REVERSE, then the frame messages will be transmitted in REVERSE sequence, starting with "F1 7X" and ending with "F1 0X".

Cue Mode: The Master Time Code Source is being "rocked", or "cued" by hand. The tape is still contacting the playback head so that the listener can cue, or preview the contents of the tape slowly. The MIDI Time Code Converter is transmitting FRAME ("F1") messages to the Master Control/Cue Sheet. If the tape is being played in the FORWARD direction, the frame messages are sent in ASCENDING order, starting with "F1 0X" and ending with "F1 7X". If the tape machine is played in the REVERSE direction, then the frame messages will be transmitted in REVERSE sequence, starting with "F1 7X" and ending with "F1 0X".

Because the tape is being moved by hand in Cue Mode, the tape direction can change quickly and often. The order of the Frame Message sequence must change along with the tape direction.

Fast-Forward/Rewind Mode: In this mode, the tape is in a high-speed wind or rewind, and is not touching the playback head. No "cueing" of the taped material is going on. Since this is a "search" mode, synchronization of the Master Control/Cue Sheet is not as important as in the Play or Cue Mode. Thus, in this mode, the MIDI Time Code Converter only needs to send a "Full Message" every so often to the Cue Sheet. This acts as a rough indicator of the Master's position. The SMPTE time indicated by the "Full Message" actually takes effect upon the reception of the next "F1" quarter frame message (when "Play Mode" has resumed).

Shuttle Mode: This is just another expression for "Fast-Forward/Rewind Mode".

15.8 References and Credits

SMPTE 12M (ANSI V98.12M-1981).

MIDI Time Code specification created by Chris Meyer and Evan Brooks of Digidesign. Thanks to Stanley Jungleib of Sequential for additional text. Also many thanks to all of the MMA and JMSC members for their suggestions and contributions to the spec.