INTERNET-DRAFT
June 27, 2003
Expires: December 27, 2003

J. Lazzaro
J. Wawrzynek
UC Berkeley

RTP Payload Format for MIDI

<draft-ietf-avt-mwpp-midi-rtp-08.txt>

Status of this Memo

Copyright Notice

Abstract

This memo describes an RTP payload format for the MIDI command language. The format encodes all commands that may legally appear on a MIDI 1.0 DIN cable. The format is suitable for interactive applications (such as the remote operation of musical instruments) and content-delivery applications (such as file streaming). The format may be used over unicast and multicast UDP as well as TCP, and defines tools for graceful recovery from packet loss. Stream behavior, including the MIDI rendering method, may be customized during session setup. The format also serves as a mode for the mpeg4-generic format, to support the MPEG 4 Audio Object Types for General MIDI, Downloadable Sounds Level 2, and Structured Audio.

Table of Contents

1. Introduction 4
1.1 Terminology 5
2. Packet Format. 5
2.1 RTP Header 6
2.2 MIDI Payload 7
3. MIDI Command Section 8
3.1 Timestamps 10
3.2 Command Coding 11
4. The Recovery Journal System 15
5. Recovery Journal Format 17
6. Session Description Protocol 20
6.1 Session Descriptions for Native Streams 20
6.2 Session Description for mpeg4-generic Streams 21
6.3 Session Configuration Tools 22
7. Extensibility 23
8. Congestion Control 24
A. The Recovery Journal Channel Chapters 25
A.1 Recovery Journal Definitions 25
A.2 Chapter P: MIDI Program Change 27
A.3 Chapter W: MIDI Pitch Wheel 28
A.4 Chapter N: MIDI NoteOff and NoteOn 29
A.4.1 Header Structure 30
A.4.2 Note Structures 31
A.5 Chapter A: MIDI Poly Aftertouch 31
A.6 Chapter T: MIDI Channel Aftertouch 32
A.7 Chapter C: MIDI Control Change 33
A.7.1 Log Inclusion Rules 33
A.7.2 Log Coding Rules 34
A.7.3 Portamento Control 37
A.7.4 The Parameter System 37
A.8 Chapter E: MIDI Reset All Controllers 38
A.9 Chapter M: MIDI Parameter System 40
A.9.1 Log Inclusion Rules 40
A.9.2 Log Coding Rules 41
A.9.2.1 COARSE and FINE Fields 42
A.9.2.2 The BUTTON Field 42
A.9.2.3 A-COARSE, A-FINE, and A-BUTTON 43
A.9.2.4 COUNT and TCOUNT 45
B. The Recovery Journal System Chapters 45
B.1 System Chapter D: Simple System Commands 45
B.1.1 Undefined System Commands 47
B.2 System Chapter V: Active Sense Command 49
B.3 System Chapter Q: Sequencer State Commands 49
B.3.1 Non-compliant Sequencers 51
B.4 System Chapter F: MIDI Time Code 51

B.4.1 Partial Frames 53
B.5 System Chapter X: System Exclusive 54
B.5.1 Chapter Format 55
B.5.2 Coding Tools 56
C. SDP Session Configuration Tools 57
C.1 The Journalling System 58
C.1.1 The j_sec Parameter 58
C.1.2 The j_update Parameter 60
C.1.2.1 The anchor Sending Policy 60
C.1.2.2 The closed-loop Sending Policy 60
C.1.2.3 The open-loop Sending Policy 63
C.1.3 Chapter Inclusion Parameters 65
C.2 Command Execution Semantics 68
C.2.1 The async Algorithm 69
C.2.2 The buffer Algorithm 70
C.3 Timing Tools 71
C.3.1 ptime and maxptime 71
C.3.2 The guardtime Parameter 72
C.3.3 MIDI Time Code Issues 72
C.4 Multiple Streams 73
C.4.1 The musicport Parameter 73
C.4.2 The zerosync Parameter 75
C.5 MIDI Rendering 78
C.5.1 The rinit Parameter 78
C.5.2 Encoding rinit Data Objects 79
C.5.3 MIDI Channel Mapping 80
C.5.3.1 smf_info 80
C.5.3.2 smf_inline, smf_url, smf_cid 81
C.5.3.3 chanmask 81
C.5.4 The audio/asc MIME Type 82
D. Parameter Syntax Definitions 83
E. A MIDI Overview for Networking Specialists 87
E.1 Commands Types 88
E.2 Running Status 89
E.3 Command Timing 89
F. Acknowledgements 89
G. Security Considerations 90
H. IANA Considerations 90
H.1 rtp-midi MIME Registration 90
H.2 mpeg4-generic MIME Registration 93
H.3 asc MIME Registration 96
I. References 97
I.1 Normative References 97
I.2 Informative References 98
J. Author Addresses 99
K. Intellectual Property Rights Statement 99
L. Full Copyright Statement 100
M. Change Log for <draft-ietf-avt-mwpp-midi-rtp-08.txt> 101

1. Introduction

The Internet Engineering Task Force (IETF) has developed a set of focused tools for multimedia networking ([2] [6] [14] [16]). These tools can be combined in different ways to support a variety of real-time applications over Internet Protocol (IP) networks.

For example, a telephony application might use the Session Initiation Protocol (SIP, [14]) to set up a phone call. Call setup would include negotiations to agree on a common audio codec [15]. Negotiations would use the Session Description Protocol (SDP, [6]) to describe candidate codecs.

After a call is set up, audio data would flow between the parties using the Real Time Protocol (RTP, [2]) under the Audio/Visual Profile (AVP, [3]). The tools used in this telephony example (SIP, SDP, RTP/AVP) might be combined in a different way to support a content streaming application, perhaps in conjunction with other tools (such as the Real Time Streaming Protocol (RTSP, [16])).

The MIDI command language [1] is widely used in musical applications that are analogous to the examples described above. On stage and in the recording studio, MIDI is used for the interactive remote control of musical instruments, an application similar in spirit to telephony. On web pages, Standard MIDI Files (SMFs, [1]) rendered using the General MIDI standard [1] provide a low-bandwidth substitute for audio streaming.

This memo is motivated by a simple premise: if MIDI performances could be sent as RTP streams that are managed by IETF session tools, a hybridization of the MIDI and IETF application domains may occur.

For example, interoperable MIDI networking may foster network music performance applications, in which a group of musicians, located at different physical locations, interact over a network to perform as they would if located in the same room [12]. As another example, the streaming community may begin to use MIDI for low-bitrate audio coding, perhaps in conjunction with normative sound synthesis methods [5]. As another example, manufacturers of professional audio equipment and electronic musical instruments may consider adopting the IETF multimedia stack (IP, SIP, RTP) as the networking layer for a MIDI control plane.

To provide a foundation for RTP MIDI applications, this memo extends two of the IETF tools (RTP and SDP) to support MIDI. Sections 2-5 and Appendices A-B extend RTP/AVP by adding a MIDI payload format. Section **6 and Appendices C-D extend SDP by adding session configuration tools to** customize stream behavior (including the MIDI rendering method) during session setup.

Some applications may require MIDI media delivery at a certain service quality level (latency, jitter, packet loss, etc). RTP itself does not provide service guarantees. However, applications may use lower-layer network protocols to configure the quality of the transport services that RTP uses. These protocols may act to reserve network resources for RTP flows [19], or may simply direct RTP traffic onto a dedicated "media network" in a local installation. Note that RTP and the MIDI payload format do provide tools that applications may use to achieve the best possible real-time performance at a given service level.

This memo normatively defines the syntax and semantics of the MIDI payload format. However, this memo does not define algorithms for sending and receiving packets. An ancillary document [18] provides informative guidance on algorithms. Supplemental information may be found in related conference publications [12] [13].

Throughout this memo, the phrase "native stream" refers to a stream that uses the rtp-midi MIME type. The phrase "mpeg4-generic stream" refers to a stream that uses the mpeg4-generic MIME type (in mode rtp-midi) to operate in an MPEG 4 environment [4]. Section 6 describes this distinction in detail.

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [11].

2. Packet Format.

In this section, we introduce the format of RTP MIDI packets. The description includes some background information on RTP/AVP, for the benefit of MIDI implementors new to IETF tools. Implementors should consult [2,3] for an authoritative description of RTP/AVP.

This memo assumes the reader is familiar with MIDI syntax and semantics. Appendix E provides a MIDI overview, at a level of detail sufficient to understand most of this memo. Implementors should consult [1] for an authoritative description of MIDI.

The MIDI payload format maps a MIDI command stream (16 voice channels + systems) onto an RTP stream. An RTP media stream is a sequence of logical packets that share a common format. Each packet consists of two parts: the RTP header and the MIDI payload. Figure 1 shows this format (vertical space delineates the header and payload).

We describe RTP packets as "logical" packets to highlight the fact that

RTP itself is not a network-layer protocol. Instead, RTP packets are mapped onto network protocols (such as unicast UDP, multicast UDP, or TCP) by an application [17].

2.1 RTP Header

[2] provides a complete description of the RTP header fields. In this section, we clarify the role of a few RTP header fields for MIDI applications. All fields are coded in network byte order (big-endian).

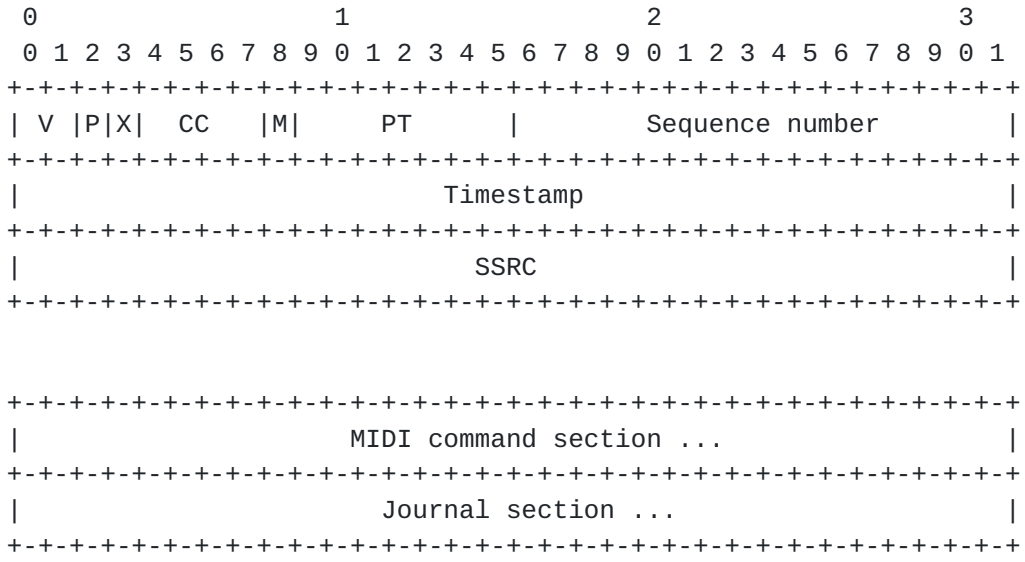

Figure 1 -- Packet format

The behavior of the 1-bit M field depends on the MIME type of the stream. For native streams, the M bit MUST be set to 1 if the MIDI command section codes one or more MIDI commands, and MUST be set to 0 otherwise. For mpeg4-generic streams, the M bit MUST be set to 1 for all packets (to conform to [4]).

The 16-bit sequence number field is initialized to a randomly chosen value, and is incremented by one (modulo 2^16) for each packet sent in the stream. A related quantity, the 32-bit extended packet sequence number, may be computed by tracking rollovers of the 16-bit sequence number. Note that different receivers of the same stream may compute different extended packet sequence numbers, depending on when the receiver joined the session.

The 32-bit timestamp field sets the base timestamp value for the packet. The payload codes MIDI command timing relative to this value. The

timestamp units are set during session configuration by the srate rtpmap parameter (Sections 6.1-2). For example, if srate has a value of 44100 Hz, two packets whose base timestamp values differ by 2 seconds have RTP timestamp fields that differ by 88200. By default the timestamp field is initialized to a randomly chosen value (see Appendix C.4.2 for an exception).

RTP timestamps do not necessarily increment at a fixed rate, because packets are not necessarily sent at a fixed rate. The degree of packet transmission regularity reflects the underlying application dynamics. Interactive applications may vary the packet sending rate to track the gestural rate of a human performer, whereas content-streaming applications may send packets at a fixed rate.

Therefore, the timestamps for two sequential RTP packets may be identical, or the second packet may have a timestamp arbitrarily larger than the first packet (modulo 2^32). Section 3 places additional restrictions on the RTP timestamps for two sequential RTP packets, as does the guardtime fmtp parameter (Appendix C.3.2).

The media time coded by a packet is computed by subtracting the last command timestamp in the MIDI command section from the RTP timestamp (modulo 2^32). If the MIDI list of the MIDI command section of a packet is empty, the media time coded by the packet is 0 ms. Appendix C.3.1 discusses media time issues in detail.

2.2 MIDI Payload

The payload (Figure 1) MUST begin with the MIDI command section. The MIDI command section codes a (possibly empty) list of timestamped MIDI commands, and provides the essential service of the payload format.

The payload MAY also contain a journal section. The journal section provides resiliency by coding the recent history of the stream. A flag in the MIDI command section codes the presence of a journal section in the payload.

Section 3 defines the MIDI command section. Sections 4-5 and Appendices A-B define the recovery journal, the default format for the journal section. Here, we describe how these payload sections operate in a stream.

The journalling method for a stream is set at the start of a session and MUST NOT be changed thereafter. A stream may be set to use the recovery journal, to use an alternative journal format (none are defined in this memo), or to not use a journal.

The default journalling method of a stream is inferred from its

transport type. Streams that use unreliable transport (such as UDP) default to using the recovery journal. Streams that use reliable transport (such as TCP) default to not using a journal. Appendix C.1.1 defines session configuration tools for overriding these defaults.

If a stream uses the recovery journal, every payload in the stream MUST include a journal section. If a stream does not use journalling, a journal section MUST NOT appear in a stream payload. If a stream uses an alternative journal format, the specification for the journal format defines an inclusion policy.

If a stream sent over reliable transport does not use journalling, the sender MUST transmit an RTP packet stream with consecutive sequence numbers (modulo 2^16). If a stream sent over reliable transport uses the recovery journal, the sender MAY transmit an RTP stream with missing or out-of-order packets.

The payload of a stream encodes data for a single MIDI command name space (16 voice channels + systems). Applications may use several streams in a session. Session configuration tools for multi-stream sessions are defined in Appendix C.4.

In some applications, a receiver renders MIDI commands into audio (or into control actions, such as the rewind of a tape deck or the dimming of stage lights). In other applications, a receiver presents a MIDI stream to software programs via an Application Programmer Interface (API). Appendix C.5 defines session configuration tools to specify what receivers should do with a MIDI command stream.

If a stream is sent over UDP transport, the Maximum Transmission Unit (MTU) of the underlying network limits the practical size of the payload section (for example, an Ethernet MTU is 1500 octets). Note that MTU size restrictions do not apply to RTP packets sent over TCP streams. The session configuration tools defined in Appendix C.4 may be used to split a dense MIDI name space into several UDP streams, so that the payload fits comfortably into an MTU.

3. MIDI Command Section

Figure 2 shows the format of the MIDI command section.

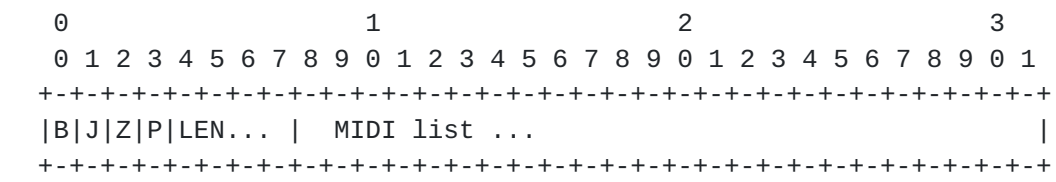

Figure 2 -- MIDI command section

The MIDI command section begins with a variable-length header.

The header field LEN codes the number of octets in the MIDI list that follows the header. If the header flag B is 0, the header is one octet long, and LEN is a 4-bit field, supporting a maximum MIDI list length of **15 octets.** If B is 1, the header is two octets long, and LEN is a 12-bit field, supporting a maximum MIDI list length of 4095 octets. A LEN value of 0 is legal, and codes an empty MIDI list

If the J header bit is set to 1, a journal section MUST appear after MIDI command section in the payload. If the J header bit is set to 0, the payload MUST NOT contain a journal section.

If the LEN header field is nonzero, the MIDI list has the structure shown in Figure 3.

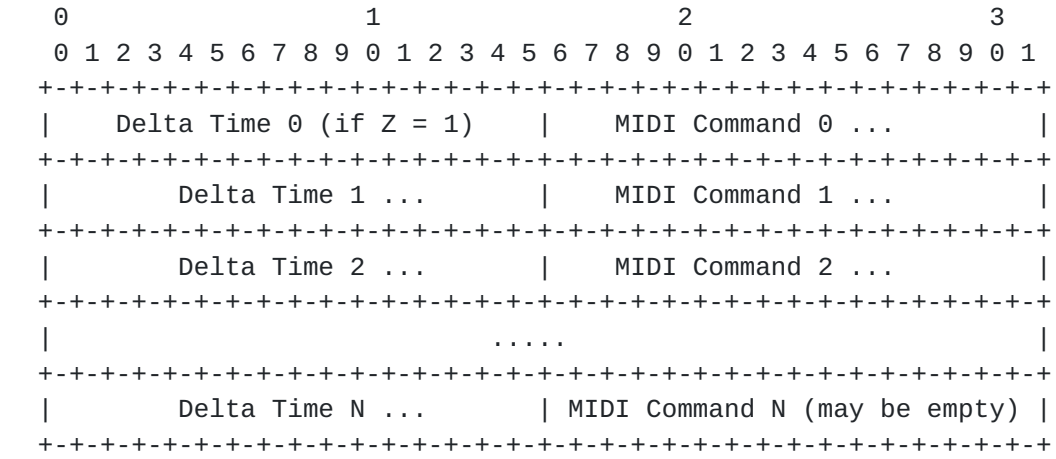

Figure 3 -- MIDI list structure

If the header flag Z is 1, the MIDI list begins with a complete MIDI command (MIDI Command 0) preceded by a delta time (Delta Time 0). If Z is 0, the Delta Time 0 field is not present in the MIDI list, and MIDI Command 0 has an implicit delta time of 0. The MIDI list structure may also optionally encode a list of N additional complete MIDI commands.

Each additional command MUST be preceded by a delta time.

The final MIDI Command field in the MIDI list MAY be empty. Senders may use this feature to precisely set the media time of a packet.

3.1 Timestamps

The RTP MIDI delta time syntax is a modified form of the MIDI File delta time syntax [1]. RTP MIDI delta times use 1-4 octet fields to encode 32-bit unsigned integers. Figure 4 shows the encoded and decoded forms of delta times. Note that delta time values may be legally encoded in multiple formats; for example, there are four legal ways to encode the zero delta time (0x00, 0x8000, 0x800000, 0x80000000).

RTP MIDI uses delta times to encode a timestamp for each MIDI command. The timestamp for MIDI Command K is the summation (modulo 2^32) of the RTP timestamp and decoded delta times 0 through K. This cumulative coding technique, borrowed from MIDI File delta time coding, is efficient because it reduces the number of multi-octet delta times.

```
One-Octet Delta Time:

   Encoded form: 0ddddddd
   Decoded form: 00000000 00000000 00000000 0ddddddd

Two-Octet Delta Time:

   Encoded form: 1ccccccc 0ddddddd
   Decoded form: 00000000 00000000 00cccccc cddddddd

Three-Octet Delta Time:

   Encoded form: 1bbbbbbb 1ccccccc 0ddddddd
   Decoded form: 00000000 000bbbbb bbcccccc cddddddd

Four-Octet Delta Time:

   Encoded form: 1aaaaaaa 1bbbbbbb 1ccccccc 0ddddddd
   Decoded form: 0000aaaa aaabbbbb bbcccccc cddddddd
```

Figure 4 -- Decoding delta time formats

All command timestamps in a packet MUST be less than or equal to the RTP timestamp of the next packet in the stream (modulo 2^32).

By default, a command timestamp indicates the execution time for the command. The difference between two timestamps indicates the time delay between the execution of the commands. This difference may be zero, coding simultaneous execution.

This default interpretation of timestamp semantics is a good choice to use for transcoding a Standard MIDI File (SMF) into an RTP MIDI stream. To code an SMF that uses metric time markers, use the tempo map (encoded as SMF meta-events) to convert metric units into seconds-based RTP timestamp units.

MIDI command sources that use implicit command timing, such as MIDI 1.0 DIN cables, must be annotated with timestamps as part of the RTP transcoding process. Appendix C.2 describes session configuration tools for transcoding MIDI sources of this type.

3.2 Command Coding

Each non-empty MIDI Command field in the MIDI list codes one of the MIDI command types that may legally appear on a MIDI 1.0 DIN cable. Note that SMF meta-events do not fit this definition and MUST NOT appear in the MIDI list. As a rule, each MIDI Command field codes a complete command, in the binary command format defined in [1]. In the remainder of this section, we describe exceptions to this rule.

The first MIDI channel command in the MIDI list MUST include a status octet. Running status coding, as defined in [1], MAY be used for all subsequent MIDI channel commands in the list. If the status octet of the first MIDI channel command in the list does not appear in the source data stream, the P (phantom) header bit MUST be set to 1. In all other cases, the P bit MUST be set to 0.

As in [1], System Common and System Exclusive messages (0xF0 ... 0xF7) cancel running status state, but System Real-time messages (0xF8 ... 0xFF) do not effect running status state. As receivers MUST be able to decode running status, sender implementors should feel free to use running status to improve bandwidth efficiency. However, senders SHOULD NOT introduce timing jitter into an existing MIDI command stream through an inappropriate use or removal of running status coding.

On a MIDI 1.0 DIN cable [1], a System Real-time command may be embedded inside of another "host" MIDI command. This syntactic construction is not supported in the payload format: a MIDI Command field in the MIDI list codes exactly one complete MIDI command.

To encode an embedded System Real-time command, senders MUST extract the command from its host, and code it in the MIDI list as a separate command. The host command and System Real-time command SHOULD appear in

the same MIDI list. The delta time of the System Real-time command SHOULD result in a command timestamp that encodes the System Real-time command placement in its original embedded position.

Two methods are provided for encoding MIDI System Exclusive (SysEx) commands in the MIDI list. A SysEx command may be encoded in a MIDI Command field verbatim: a 0xF0 octet, followed by an arbitrary number of data octets, followed by a 0xF7 octet.

Alternatively, a SysEx command may be encoded as multiple segments. The command is divided into two or more SysEx command segments; each segment is encoded in its own MIDI Command field in the MIDI list.

The payload format supports segmentation in order to encode SysEx commands that encode information in the temporal pattern of data octets. By encoding these commands as a series of segments, each data octet may be associated with a distinct delta time. Segmentation also supports the coding of large SysEx commands across several packets.

To segment a SysEx command, first partition its data octet list into two or more sublists. Each sublist MUST contain at least one data octet. To complete the segmentation, add status octets to the head and tail of each sublist, as detailed in Figure 5. Figure 6 shows example segmentations of a SysEx command.

The relative ordering of SysEx command segments in a MIDI list must match the relative ordering of the sublists in the original SysEx command. Only System Real-time MIDI commands may appear between SysEx command segments. If the command segments of a SysEx command are placed in the MIDI lists of two or more RTP packets, the segment ordering rules apply to the concatenation of all affected MIDI lists.

Sublist Position	Head Status Octet	Tail Status Octet
first	0xF0	0xF0
middle	0xF7	0xF0
last	0xF7	0xF7

Figure 5 -- Command segmentation status octets

SysEx commands carried on a MIDI 1.0 DIN cable may use the "dropped 0xF7" construction [1]. In this coding method, the 0xF7 octet is dropped from the end of the SysEx command, and the status octet of the next MIDI command acts both to terminate the SysEx command and start the next command. To encode this construction in the payload format, follow these steps:

- Determine the appropriate delta times for the SysEx command and the command that follows the SysEx command.
- Insert the "dropped" 0xF7 octet at the end of the SysEx command, to form the standard SysEx syntax.
- Code both commands into the MIDI list using the rules above.
- Replace the 0xF7 octet that terminates the verbatim SysEx encoding or the last segment of the segmented SysEx encoding with a 0xF5 octet. This substitution informs the receiver of the original dropped 0xF7 coding.

[1] reserves the System Common opcodes 0xF4 and 0xF5 and the System Real-time opcodes 0xF9 and 0xFD for future use. We refer to these opcodes as undefined opcodes. By default, undefined opcodes MUST NOT appear in a MIDI Command field in the MIDI list.

During session configuration, a stream may be customized to allow transport of the undefined opcodes (Appendix C.1.3). In this case, commands that use the undefined System Common opcodes MUST be terminated with a 0xF7 octet and coded using the System Exclusive verbatim rule. Commands that use the undefined System Real-time opcodes MUST be coded using the System Real-time rules.

Original SysEx command:

0xF0 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0xF7

A two-segment segmentation:

0xF0 0x01 0x02 0x03 0x04 0xF0

0xF7 0x05 0x06 0x07 0x08 0xF7

A different two-segment segmentation:

0xF0 0x01 0xF0

0xF7 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0xF7

A three-segment segmentation:

0xF0 0x01 0x02 0xF0

0xF7 0x03 0x04 0xF0

0xF7 0x05 0x06 0x07 0x08 0xF7

The segmentation with the largest number of segments:

0xF0 0x01 0xF0

0xF7 0x02 0xF0

0xF7 0x03 0xF0

0xF7 0x04 0xF0

0xF7 0x05 0xF0

0xF7 0x06 0xF0

0xF7 0x07 0xF0

0xF7 0x08 0xF7

Figure 6 -- Example segmentations

4. The Recovery Journal System

The recovery journal is the default resiliency tool for unreliable transport. In this section, we normatively define the roles that senders and receivers play in the recovery journal system.

MIDI is a fragile code. A single lost command in a MIDI command stream may produce an artifact in the rendered performance. We normatively classify rendering artifacts into two categories:

- Transient artifacts. Transient artifacts produce immediate but short-term glitches in the performance. For example, a lost NoteOn (0x9) command produces a transient artifact: one note fails to play, but the artifact does not extend beyond the end of that note.

- Indefinite artifacts. Indefinite artifacts produce long-lasting errors in the rendered performance. For example, a lost NoteOff (0x8) command may produce an indefinite artifact: the note that should have been ended by the lost NoteOff command may sustain indefinitely. As a second example, the loss of a Control Change (0xB) command for controller number 7 (Channel Volume) may produce an indefinite artifact: after the loss, all notes on the channel may play too softly or too loudly.

The purpose of the recovery journal system is to satisfy the recovery journal mandate: the MIDI performance rendered from an RTP MIDI stream sent over unreliable transport MUST NOT contain indefinite artifacts.

The recovery journal system does not use packet retransmission to satisfy this mandate. Instead, each packet includes a special section, called the recovery journal.

The recovery journal codes the history of the stream, back to an earlier packet called the checkpoint packet. The range of coverage for the journal is called the checkpoint history. The recovery journal codes the information necessary to recover from the loss of an arbitrary number of packets in the checkpoint history. Appendix A.1 normatively defines the checkpoint packet and the checkpoint history.

When a receiver detects a packet loss, it compares its own knowledge about the history of the stream with the history information coded in the recovery journal of the packet that ends the loss event. By noting the differences in these two versions of the past, a receiver is able to transform all indefinite artifacts in the rendered performance into transient artifacts, by executing MIDI commands to repair the stream.

We now state the normative role for senders in the recovery journal

system.

Senders prepare a recovery journal for every packet in the stream. In doing so, senders choose the checkpoint packet identity for the journal. Senders make this choice by applying a sending policy. Appendix C.1.2 normatively defines three sending policies: "closed-loop", "open-loop", and "anchor".

By default, senders MUST use the closed-loop sending policy. If the session description overrides this default policy, by using the fmtp parameter j_update defined in Appendix C.1.2, senders MUST use the specified policy.

After choosing the checkpoint packet identity for a packet, the sender creates the recovery journal. By default, this journal MUST conform to the normative semantics in Section 5 and Appendices A-B in this memo. In Appendix C.1.3, we define fmtp parameters that modify the normative semantics for recovery journals. If the session description uses these parameters, the journal created by the sender MUST conform to the modified semantics.

Next, we state the normative role for receivers in the recovery journal system.

A receiver MUST detect each RTP sequence number break in a stream. If the sequence number break is due to a packet loss event (as defined in [2]) the receiver MUST repair all indefinite artifacts in the rendered MIDI performance caused by the loss. If the sequence number break is due to an out-of-order packet (as defined in [2]) the receiver MUST NOT take actions that introduce indefinite artifacts (ignoring the out-of-order packet is a safe option).

Receivers take special precautions when entering or exiting a session. A receiver MUST process the first received packet in a stream as if it were a packet that ends a loss event. Upon exiting a session, a receiver MUST ensure that the rendered MIDI performance does not end with indefinite artifacts.

Receivers are under no obligation to perform indefinite artifact repairs at the moment a packet arrives. A receiver that uses a playout buffer may choose to wait until the moment of rendering before processing the recovery journal, as the "lost" packet may be a late packet that arrives in time to use.

Next, we state the normative role for the creator of the session description in the recovery journal system. Depending on the application, the sender, the receivers, and other parties may take part in creating or approving the session description.

A session description that specifies the default closed-loop sending policy and the default recovery journal semantics satisfies the recovery journal mandate. However, these default behaviors may not be appropriate for all sessions. If the creators of a session description use the parameters defined in Appendix C.1 to override these defaults, the creators MUST ensure that the parameters define a system that satisfy the recovery journal mandate.

Finally, we note that this memo does not specify sender or receiver recovery journal algorithms. Implementations are free to use any algorithm that conforms to the requirements in this section. The non-normative [18] discusses sender and receiver algorithm design.

5. Recovery Journal Format

This section introduces the structure of the recovery journal, and defines the bitfields of recovery journal headers. Appendices A-B complete the bitfield definition of the recovery journal. The recovery journal has a three-level structure:

- Top-level header.
- Channel and system journal headers. Encodes recovery information for a single voice channel (channel journal) or for all systems commands (system journal).
- Chapters. Describes recovery information for a single MIDI command type.

Figure 7 shows the top-level structure of the recovery journal. A recovery journals consists of a 3-octet header, optionally followed by a system journal and a list of channel journals. These elements appear in the order shown in Figure 7.

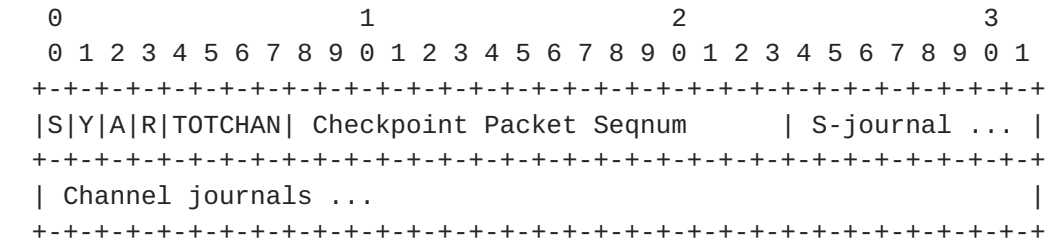

Figure 7 -- Top-level recovery journal format

If the Y bit is set to 1, a system journal follows the recovery journal

header. If the A bit is set to 1, the recovery journal ends with a list of (TOTCHAN + 1) channel journals. If A and Y are both zero, the recovery journal only contains the 3-octet header, and is considered to be an "empty" journal.

A MIDI channel may be represented by (at most) one channel journal in a recovery journal. Channel journals MUST appear in the recovery journal in ascending channel-number order.

The S (single-packet loss) bit appears in most recovery journal structures. The S bit helps receivers efficiently parse the recovery journal in the common case of the loss of a single packet. Appendix A.1 defines S bit semantics.

The R bit is reserved. The semantics for all R fields are uniform throughout the recovery journal, and are defined in Appendix A.1.

The 16-bit Checkpoint Packet Seqnum field codes the sequence number of the checkpoint packet for this journal. The choice of the checkpoint packet sets the depth of the checkpoint history for the journal (defined in Appendix A.1).

Receivers may use the Checkpoint Packet Seqnum field of the packet that ends a loss event to verify that the journal checkpoint history covers the entire loss event. The checkpoint history covers the loss event if the Checkpoint Packet Seqnum field is less than or equal to the highest RTP sequence number previously received on the stream (modulo 2^16).

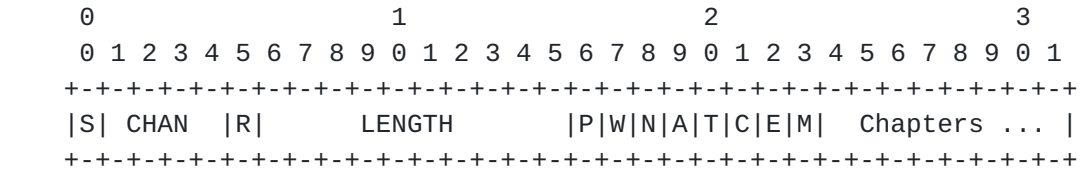

Figure 8 -- Channel journal format

Figure 8 shows the structure of a channel journal: a 3-octet header, followed by a list of leaf elements called channel chapters. A channel journal encodes information about MIDI commands on the MIDI channel coded by the 4-bit CHAN header field.

The 10-bit LENGTH field codes the length of the channel journal. The semantics for LENGTH fields are uniform throughout the recovery journal, and are defined in Appendix A.1.

The third octet of the channel journal header is the Table of Contents

(TOC) of the channel journal. The TOC is a set of bits that encode the presence of a chapter in the journal. Each chapter contains information about a certain class of MIDI channel command:

- Chapter P: MIDI Program Change (0xC)
- Chapter W: MIDI Pitch Wheel (0xE)
- Chapter N: MIDI NoteOff (0x8), NoteOn (0x9)
- Chapter A: MIDI Poly Aftertouch (0xA)
- Chapter T: MIDI Channel Aftertouch (0xD)
- Chapter C: MIDI Control Change (0xB)
- Chapter E: MIDI Reset All Controllers (part of 0xB)
- Chapter M: MIDI Parameter System (part of 0xB)

Chapters appear in a list following the header, in order of their appearance in the TOC. Appendices A.2-9 describe the bitfield format for each chapter, and define the conditions under which a chapter type MUST appear in the recovery journal. If any chapter types are required for a channel, an associated channel journal MUST appear in the recovery journal.

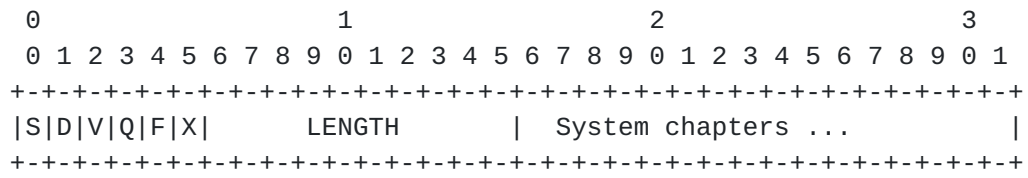

Figure 9 -- System journal format

Figure 9 shows the structure of the system journal: a 2-octet header, followed by a list of system chapters. System chapters code information about a specific class of MIDI Systems command:

- Chapter D: Song Select (0xF3), Tune Request (0xF6), Reset (0xFF), undefined System commands (0xF4, 0xF5, 0xF9, 0xFD)
- Chapter V: Active Sense (0xFE)
- Chapter Q: Sequencer State (0xF2, 0xF8, 0xF9, 0xFA, 0xFB, 0xFC)
- Chapter F: MTC Tape Position (0xF1, 0xF0 0x7F 0xcc 0x01 0x01)
- Chapter X: System Exclusive (all other 0xF0)

If header bits D, V, Q, or F are set to 1, one chapter for each set bit appears in the system chapter list. The chapter list ordering follows the ordering of the header bits. If header bit X is set to 1, one or more Chapter X bitfields appear at the end of the chapter list.

Appendix B describes the bitfield format for the system chapters, and define the conditions under which a chapter type MUST appear in the

recovery journal. If any system chapter type is required to appear in the recovery journal, the system journal MUST appear in the recovery journal.

6. Session Description Protocol

RTP does not perform session management. Instead, RTP is designed to work together with tools that perform session management, such as the Session Initiation Protocol (SIP, [14]) and the Real Time Streaming Protocol (RTSP, [16]). RTP interacts with session management tools via another standard, the Session Description Protocol (SDP, [6]).

SDP is a textual format for specifying session descriptions. Session descriptions specify the network transport and media encoding for RTP streams. SIP and RTSP coordinate the exchange of session descriptions between participants. In SIP, session descriptions also support negotiation [15].

Below, we show session description examples for native (Section 6.1) and mpeg4-generic (Section 6.2) streams. In Section 6.3, we introduce session configuration tools that may be used to customize streams.

6.1 Session Descriptions for Native Streams

The session description below shows a minimal session description for a native stream sent over unicast UDP transport.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
a=rtpmap: 96 rtp-midi/44100
```

The rtpmap attribute line uses the rtp-midi MIME type to specify a native stream. If the session parties send and receive RTP packets, the streams form bi-directional MIDI connections, suitable for use by the MIDI System commands that use handshaking protocols [1].

We describe this session description as minimal, because it does not customize the stream. Without such customization, a native stream has these characteristics:

1. If the stream uses unreliable transport (unicast UDP, multicast UDP, ...) the recovery journal system is in use, and the RTP payload contains both the MIDI command section and the journal

section. If the stream uses reliable transport (TCP, TLS, ...), the stream does not use journalling, and the payload contains only the MIDI command section (Section 2.2).

2. If the stream uses the recovery journal system, the recovery journal system uses the default sending policy and the default journal semantics (Section 4).

3. In the MIDI command section of the payload, command timestamps use the default semantics (Section 3).

4. The media time encoded by an RTP packet may range from 0 to 200 ms, and the RTP timestamp difference between sequential packets in the stream may be arbitrarily large (Section 2.1).

5. If more than one minimal rtp-midi stream appears in a session, the MIDI name spaces for these streams are independent: channel 1 in the first stream does not reference the same MIDI channel as channel 1 in the second stream.

6. The rendering method for the stream is not specified.

6.2 Session Description for mpeg4-generic Streams

An mpeg4-generic stream uses an MPEG 4 Audio Object Type to render MIDI into audio [3]. Three Object Types are compatible with MIDI:

- General MIDI (Audio Object Type ID 15), based on the General MIDI rendering standard [1].

- Wavetable Synthesis (Audio Object Type ID 14), based on the Downloadable Sounds Level 2 (DLS 2) rendering standard [9].

- Main Synthetic (Audio Object Type ID 13), based on Structured Audio and the programming language SAOL [5].

The session description below shows a minimal session description for an mpeg4-generic stream sent over unicast UDP transport. This example uses the General MIDI Audio Object Type under Synthesis Profile @ Level 2.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP6 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP6 FF1E:03AD::7F2E:172A:1E24
a=rtpmap: 96 mpeg4-generic/44100
a=fmtp: 96 streamtype=5; mode=rtp-midi; profile-level-id=12;
```

```
 a=fmtp: 96 config=7A124D5468640000000060000000100604D547
                   26B0000000400FF2F000
```

(The linebreak in the second fmtp line accommodates memo formatting restrictions; SDP does not have continuation lines.)

The fmtp attribute lines code the parameters that MUST appear in a mpeg4-generic session description [4]. The "streamtype" parameter MUST be set to 5, and the "mode" parameter MUST be set to "rtp-midi". The "profile-level-id" parameter MUST be set to the MPEG-4 Profile Level.

The "config" parameter MUST appear in the session description. The config value is a hexadecimal encoding [4] of the AudioSpecificConfig data block [7] for the stream. AudioSpecificConfig encodes the Audio Object Type for the stream, and also encodes initialization data (SAOL programs, DLS 2 wave tables, etc). Standard MIDI Files encoded in the AudioSpecificConfig MUST be ignored by the receiver.

We describe this session description as minimal, because it does not customize the stream. In Section 6.1, we describe the behavior of a minimal native stream, as a numbered list of characteristics. Items 1-4 on that list also describe the minimal mpeg4-generic stream, but items 5 and 6 require restatements, as listed below:

5. If more than one minimal mpeg4-generic stream appears in a session, each stream uses an independent instance of the Audio Object Type coded in the config parameter value.

6. A minimal mpeg4-generic stream encodes the AudioSpecificConfig as an inline hexadecimal constant. If session description is sent over UDP, it may be impossible to transport large AudioSpecificConfig blocks, as the Maximum Transmission Size (MTU) of the underlying network limits the UDP packet size (for Ethernet, the MTU is 1500 octets).

6.3 Session Configuration Tools

This section introduces the session configuration tools for RTP MIDI sessions. The tools add features to the minimal streams described in Sections 6.1-2, and support several types of services:

- Journal customization. The j_sec and j_update parameters configure the use of the payload journal section. The ch_default, ch_unused, ch_never, and ch_anchor parameters configure the semantics of the recovery journal chapters. These fmtp parameters are described in Appendix C.1, and override the default stream behaviors 1 and 2 listed in Section 6.1 and referenced in Section 6.2.

- MIDI command timestamp semantics. The tsmode, octpos, mperiod, and linerate parameters customize the semantics of timestamps in the MIDI command section. These parameters let RTP MIDI accurately encode the implicit time coding of MIDI 1.0 DIN cables. These fmtp parameters are described in Appendix C.2, and override default stream behavior 3 listed in Section 6.1 and referenced in Section 6.2

- Media time. The standard SDP attributes ptime and maxptime define the media time encoded by a packet. The guardtime fmtp parameter sets the minimum sending rate of stream packets. These tools are described in Appendix C.3, and override default stream behavior 4 listed in Section 6.1 and referenced in Section 6.2.

- Multiple streams. The musicport parameter labels the MIDI name space of multi-stream sessions. The zerosync parameter supports synchronization in multi-stream sessions. These fmtp parameters are described in Appendix C.4, and override default stream behavior 5 in Sections 6.1 and 6.2.

- MIDI rendering. Several fmtp parameters specify the MIDI rendering method of a stream. These parameters are described in Appendix C.5, and override default stream behavior 6 in Sections 6.1 and 6.2.

7. Extensibility

The payload format defined in this memo exclusively encodes all commands that may legally appear on a MIDI 1.0 DIN cable.

Many worthy uses of MIDI over RTP do not fall within the narrow scope of the format. For example, the format does not support the direct transport of Standard MIDI File (SMF) meta-event and metric timing data. As a second example, the format does not define transport tools for user-defined commands (apart from tools to support System Exclusive commands [1]).

The format does not provide an extension mechanism to support new features of this nature, by design. Instead, we encourage the development of new payload formats for specialized musical applications. The IETF session management tools [15] [16] support codec negotiation, to facilitate the use of new formats in a backward-compatible way.

However, the payload format does provide several extensibility tools, which we list below:

- Rendering. The payload format may be extended to support new MIDI renderers (Appendix C.5.1). The extension mechanism uses the standard MIME registration process [20].

- Journalling. As described in Appendix C.1, new token values for the j_sec and j_update fmtp parameters may be defined in IETF standards-track documents. This mechanism supports the design of new journal formats and the definition of new journal sending policies.

- Undefined opcodes. [1] reserves 4 MIDI System opcodes for future use (0xF4, 0xF5, 0xF9, 0xFD). If updates to [1] define the reserved opcodes, IETF standards-track documents may be defined to provide resiliency support for the commands. Opaque LEGAL fields appear in System Chapter D for this purpose (Appendix B.1.1).

A final form of extensibility involves the inclusion of the payload format in framework documents. Framework documents describe how to combine protocols to form a platform for interoperable applications. For example, a network musical performance [12] framework might define how to use SIP [14], SDP [6] and RTP/AVP [2] [3] to support real-time performances between geographically-distributed players.

8. Congestion Control

RTP MIDI has congestion control issues that are unique for an audio payload format. In applications such as network musical performance [12], the packet rate is linked to the gestural rate of a human performer.

Senders MUST monitor the MIDI command source for patterns that result in excessive packet rates, and take actions during RTP transcoding to reduce the RTP packet rate. [18] offers implementation guidance on this issue.

A. The Recovery Journal Channel Chapters

A.1 Recovery Journal Definitions

This Appendix defines the terminology and the coding idioms that are used in the recovery journal bitfield descriptions in Section 5 (journal header structure), Appendices A.2-9 (channel journal chapters) and Appendices B.1-5 (system journal chapters).

We assume that the recovery journal resides in the journal section of an RTP packet with sequence number I ("packet I") and that the Checkpoint Packet Seqnum field in the top-level recovery journal header refers to a packet with sequence number C. Unless stated otherwise, algorithms are assumed to use modulo 2^16 arithmetic for calculations on 16-bit sequence numbers and modulo 2^32 arithmetic for calculations on 32-bit extended sequence numbers.

Several bitfield coding idioms appear throughout the recovery journal system, with consistent semantics. Most recovery journal elements begin with an "S" (Single-packet loss) bit. S bits are designed to help receivers efficiently parse through the recovery journal hierarchy in the common case of the loss of a single packet.

By default, all S bits MUST be set to 1. If a recovery journal element in packet I encodes data about a command stored in the MIDI command section of packet I - 1, its S bit MUST be set to 0. If a recovery journal element has its S bit set to 0, all higher-level recovery journal elements that contain it MUST also have S bits that are set to 0, including the top-level recovery journal header.

Other consistent bitfield coding idioms are described below:

- R flag bit. R flag bits are reserved for future use. Senders MUST set R bits to 0. Receivers MUST ignore R bit values.

- LENGTH field. All fields named LENGTH (as distinct from LEN) code the number of octets in the structure that contains it, including the header it resides in and all hierarchical levels below it. If a structure contains a LENGTH field, a receiver MUST use the LENGTH field value to advance past the structure during parsing, rather than use knowledge about the internal format of the structure.

We now define normative terms used to describe recovery journal semantics, grouped by order of appearance in Appendices A.2-9.

- Checkpoint history. The checkpoint history of a recovery journal is the concatenation of the MIDI command sections of packets C

through I - 1. The last command in the MIDI command section for packet I - 1 is considered the most recent command; the first command in the MIDI command section for packet C is the oldest command. If command X is less recent than command Y, X is considered to be "before Y". A checkpoint history with no commands is considered to be empty. The checkpoint history never contains the MIDI command section of the packet I (the packet containing the recovery journal), so if C == I, the checkpoint history is empty by definition.

- Session history. The session history of a recovery journal is the concatenation of MIDI command sections from the first packet of the session up to packet I - 1. The definitions of command recency and history emptiness follow those in the checkpoint history. The session history never contains the MIDI command section of packet I, and so the session history of the first packet in the session is empty by definition.

- Finished/unfinished commands. If all octets of a MIDI command appear in the session history, the command is defined to be finished. If some but not all octets of a command appear in the session history, the command is defined to be unfinished. Unfinished commands occur if segments of a SysEx command appear in several RTP packets. For example, if a SysEx command is coded as 3 segments, with segment 1 in packet K, segment 2 in packet K + 1, and segment 3 in packet K + 2, the session histories for packets K + 1 and K + 2 contain unfinished versions of the command.

- Active commands. Active command are MIDI commands that do not appear before one of the following commands in the session history: System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7).

- N-active commands. N-active commands are MIDI commands that do not appear before one of the following commands in the session history: System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7), MIDI Control Change numbers 123-127 (numbers with All Notes Off semantics) or 120 (All Sound Off).

- C-active commands. C-active commands are MIDI commands that do not appear before one of the following commands in the session history: System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7), MIDI Control Change number 121 (Reset All Controllers).

- Parameter system. A MIDI feature that provides two sets of 16,384 parameters to expand the 0-127 controller number space. The Registered Parameter Names (RPN) system and the Non-Registered Parameter Names (NRPN) system each provides 16,384 parameters.

- Parameter system transaction. The value of RPNs and NRPNs are changed by a series of Control Change commands that form a parameter system transaction. A transaction begins with two Control Change commands to set the parameter number (controller numbers 98 and 99 for NRPNs, controller numbers 100 and 101 for RPNs). The transaction continues with an arbitrary number of Data Entry (controller numbers 6 and 38) and Data Button (controller numbers 96 and 97) Control Change commands to set the parameter value. The transaction ends with a second pair of (98, 99) or (100, 101) Control Change commands. These terminal commands are considered a part of the transaction. In addition, the terminal commands start a second parameter system transaction. Thus, the commands belong to two transactions.

- Initiated parameter system transaction. An initiated parameter system transaction is a transaction whose (98, 99) or (100, 101) initial Control Change command pair appears in the session history. Unpaired Control Change commands for controller numbers 98-101 do not form an initiated transaction. The termination of a transaction does not change the "initiated" status of the transaction.

The chapter definitions in Appendices A.2-9 and B.1-5 reflect the default recovery journal behavior. The ch_default, ch_unused, ch_never, and ch_anchor parameters modify these definitions, as described in Appendix C.1.3.

The chapter definitions specify if data MUST be present in the journal. Senders MAY also include non-required data in the journal. This optional data MUST comply with the normative chapter definition. For example, if a chapter definition states that a field codes data from the most recent active command in the session history, the sender MUST NOT code inactive commands or older commands in the field.

Finally, we note that channel journals only encode information about MIDI commands appearing on the MIDI channel the journal protects. All references to MIDI commands in Appendices A.2-9 should be read as "MIDI commands appearing on this channel."

A.2 Chapter P: MIDI Program Change

A channel journal MUST contain Chapter P if an active Program Change

(0xC) command appears in the checkpoint history. Figure A.2.1 shows the format for Chapter P.

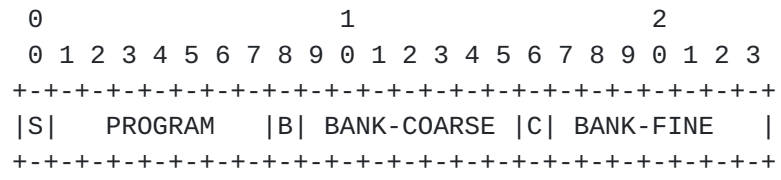

Figure A.2.1 -- Chapter P format

The chapter has a fixed size of 24 bits. The PROGRAM field indicates the data value of the most recent active Program Change command in the session history. By default, the B, BANK-COARSE, C, and BANK-FINE fields MUST be set to 0.

However, if an active Control Change (0xB) command for controller number **0 (Bank Select Coarse) appears before the Program Change command in the** session history, the B bit MUST be set to 1, and the BANK-COARSE field MUST code the data value of the Control Change command. If this Control Change command is also C-active, the C bit MUST be set to 1.

If the B bit is set to 1, the BANK-FINE field MUST code the data value of the most recent Control Change command for controller number 32 (Bank Select Fine) that preceded the Program Change command coded in the PROGRAM field and followed the Control Change command coded in the BANK-COARSE field. If no such Control Change command exists, the BANK-FINE field MUST be set to 0.

A.3 Chapter W: MIDI Pitch Wheel

A channel journal MUST contain Chapter W if an active MIDI Pitch Wheel (0xE) command appears in the checkpoint history. Figure A.3.1 shows the format for Chapter W.

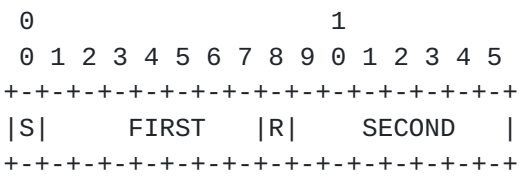

Figure A.3.1 -- Chapter W format

The chapter has a fixed size of 16 bits. The FIRST and SECOND fields are the 7-bit values of the first and second data octets of the most recent active Pitch Wheel command in the session history.

A.4 Chapter N: MIDI NoteOff and NoteOn

In this Appendix, we consider NoteOn commands with zero velocity to be NoteOff commands. Readers may wish to review the Appendix A.1 definition of "N-active commands" before reading this Appendix.

A channel journal MUST contain Chapter N if an N-active MIDI NoteOn (0x9) or NoteOff (0x8) command appears in the checkpoint history. Figure A.4.1 shows the format for Chapter N.

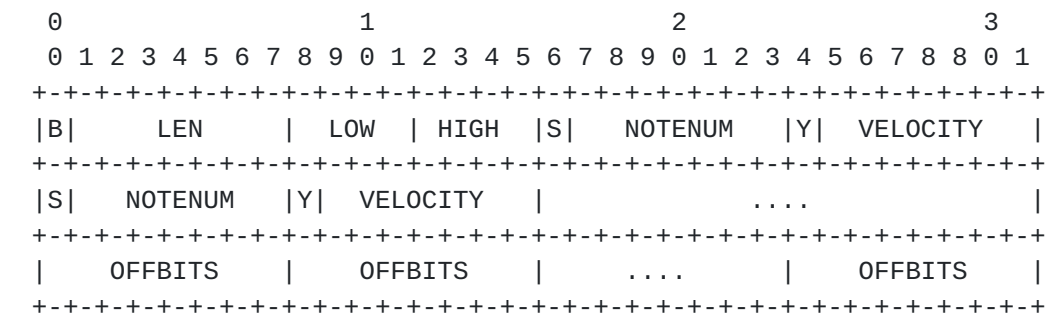

Figure A.4.1 -- Chapter N format

Chapter N consists of a 2-octet header, followed by least one of the following data structures:

- A list of note logs to code NoteOn commands.
- A NoteOff bitfield structure to code NoteOff commands.

The note log list MUST contain an entry for all note numbers whose most recent checkpoint history appearance is in an N-active NoteOn command. The NoteOff bitfield structure MUST contain a set bit for all note numbers whose most recent checkpoint history appearance is in an N-

active NoteOff command.

A note number MUST NOT be coded in both structures. All note logs and NoteOff bitfield set bits MUST code the most recent N-active NoteOn or NoteOff reference to a note number in the session history.

A.4.1 Header Structure

The header for Chapter N, shown in Figure A.4.2, codes the size of the note list and bitfield structures.

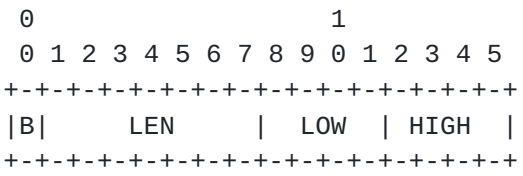

Figure A.4.2 -- Chapter N header

The LEN field, a 7-bit integer value, codes the number of 2-octet note logs in the note list. Zero is a valid value for LEN, and codes an empty note list.

The 4-bit LOW and HIGH fields code the number of OFFBITS octets that follow the note log list. LOW and HIGH are unsigned integer values. If LOW <= HIGH, there are (HIGH - LOW + 1) OFFBITS octets in the chapter. The value pairs (LOW = 15, HIGH = 0) and (LOW = 15, HIGH = 1) code an empty NoteOff bitfield structure (i.e. no OFFBITS octets). Other (LOW > HIGH) value pairs MUST NOT appear in the header.

The B bit provides S-bit functionality (Appendix A.1) for the NoteOff bitfield structure. By default, the B bit MUST be set to 1. However, if the MIDI command section of the previous packet (packet I - 1, with I as defined in Appendix A.1) includes a NoteOff command for the channel, the B bit MUST be set to 0. If the B bit is set to 0, the higher-level recovery journal elements that contain Chapter N MUST have S bits that are set to 0, including the top-level journal header.

The LEN value of 127 codes a note list length of 127 or 128 note logs, depending on the values of LOW and HIGH. If LEN = 127, LOW = 15, and HIGH = 0, the note list holds 128 note logs, and the NoteOff bitfield structure is empty. For other values of LOW and HIGH, LEN = 127 codes that the note list contains 127 note logs. In this case, the chapter has (HIGH - LOW + 1) NoteOff OFFBITS octets if LOW <= HIGH, and has no OFFBITS octets if LOW = 15 and HIGH = 1.

A.4.2 Note Structures

Figure A.4.3 shows the 2-octet note log structure.

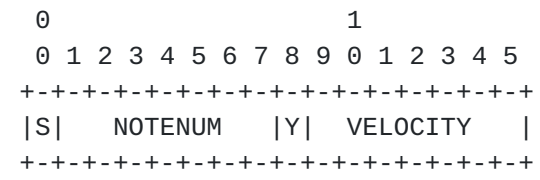

Figure A.4.3 -- Chapter N note log

The 7-bit NOTENUM field codes the note number for the log. A note number MUST NOT be represented by multiple note logs in the note list. The 7-bit VELOCITY field codes the velocity value for the most recent N-active NoteOn command for the note number in the session history. VELOCITY is never zero; NoteOn commands with zero velocity are coded as NoteOff commands in the NoteOff bitfield structure.

The note log does not code the execution time of the NoteOn command. However, the Y bit codes a hint from the sender about the NoteOn execution time. The Y bit codes a recommendation to play (Y = 1) or skip (Y = 0) the NoteOn command recovered from the note log. In a normative sense, the Y bit is set to 1 if the sender considers the command coded by the log to be simultaneous with the RTP timestamp of the packet that contains the log. In all other cases, the Y bit is set to 0.

Figure A.4.1 shows the NoteOff bitfield structure, as the list of OFFBITS octets at the end of the chapter. A NoteOff OFFBITS octet codes NoteOff information for eight consecutive MIDI note numbers, with the most-significant bit representing the lowest note number. The most-significant bit of the first OFFBITS octet codes the note number 8*LOW; the most-significant bit of the last OFFBITS octet codes the note number 8*HIGH.

A set bit codes a NoteOff command for the note number. In the most efficient coding for the NoteOff bitfield structure, the first and last octets of the structure contain at least one set bit. Note that Chapter N does not code NoteOff velocity data.

A.5 Chapter A: MIDI Poly Aftertouch

A channel journal MUST contain Chapter A if an N-active Poly Aftertouch (0xA) command appears in the checkpoint history. Figure A.5.1 shows the

format for Chapter A.

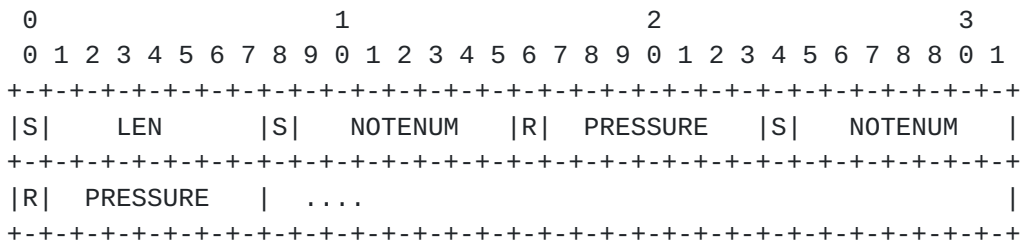

Figure A.5.1 -- Chapter A format

The chapter consists of a 1-octet header, followed by a variable length list of 2-octet note logs. A note log MUST appear for a note number if an N-active Poly Aftertouch command for the note number appears in the checkpoint history. A note number MUST NOT be represented by multiple note logs in the note list.

The 7-bit LEN field codes the number of note logs in the list, minus one. Figure A.5.2 reproduces the note log structure of Chapter A.

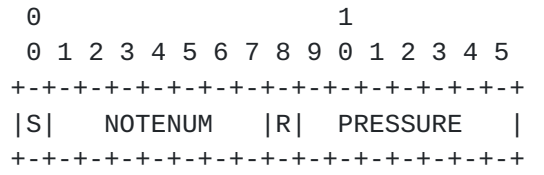

Figure A.5.2 -- Chapter A note log

The 7-bit PRESSURE field codes the pressure value of the most recent N-active Poly Aftertouch command in the session history. The MIDI note number for this command is coded in the 7-bit NOTENUM field.

A.6 Chapter T: MIDI Channel Aftertouch

A channel journal MUST contain Chapter T if an N-active MIDI Channel Aftertouch (0xD) command appears in the checkpoint history. Figure A.6.1 shows the format for Chapter T.

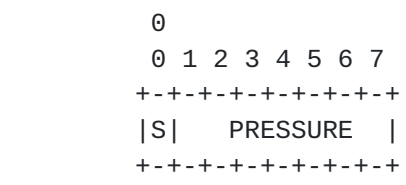

Figure A.6.1 -- Chapter T format

The chapter has a fixed size of 8 bits. The 7-bit PRESSURE field holds the pressure value of the most recent N-active Channel Aftertouch command in the session history.

A.7 Chapter C: MIDI Control Change

Readers may wish to review the Appendix A.1 definition of "C-active commands" before reading this Appendix.

Figure A.7.1 shows the format for Chapter C.

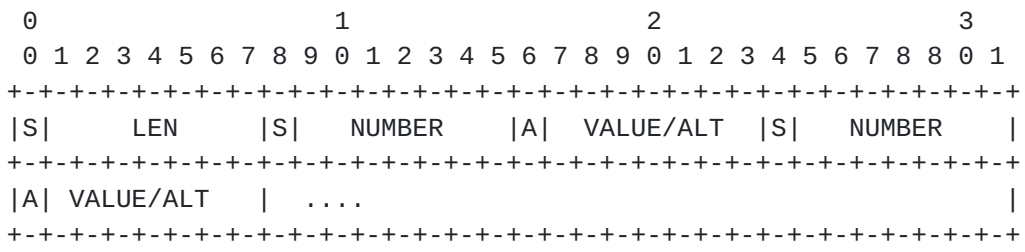

Figure A.7.1 -- Chapter C format

The chapter consists of a 1-octet header, followed by a variable length list of 2-octet controller logs. The list MUST contain at least one controller log. The 7-bit LEN field codes the number of controller logs in the list, minus one.

A channel journal MUST contain Chapter C if the rules defined in this Appendix require that one or more controller logs appear in the list.

A.7.1 Log Inclusion Rules

The list MUST contain an entry for controller numbers 0-119 (excepting controller numbers 0, 6, 32-63, 96-101, and 124-127) if a C-active Control Change command for a number appears in the checkpoint history. In addition, the list MUST contain an entry for a controller numbers 120-123 if an active Control Change command for a number appears in the

checkpoint history.

A special rule applies to streams that transmit 14-bit controller values using paired MSB (controller numbers 0-31) and LSB (controller numbers 32-63) Control Change commands. In this case, if the most recent C-active Control Change command in the session history for a 14-bit controller uses the MSB controller number, the Chapter C MUST NOT code the associated LSB controller number.

Apart from this exception, and apart from exceptions for controller numbers 32 (described below) and 38 (described in Appendix A.7.4), the controller list MUST contain an entry for controller numbers 32-63 if a C-active Control Change command for a number appears in the checkpoint history.

If C-active Control Change commands for controller numbers 0 (Bank Select Coarse) or 32 (Bank Select Fine) appear in the checkpoint history, the most recent commands for these controller numbers MUST appear as entries in the controller list, with a single exception: if the command instances are also coded in the BANK-COARSE and BANK-FINE fields of the Chapter P (Appendix A.2), Chapter C MAY omit the controller logs for the commands. Note that for this exception to apply, the C bit for Chapter P MUST be set to 1.

Several controller numbers pairs are defined to be mutually exclusive. Controller numbers 124 (Omni Off) and 125 (Omni On) form a mutually exclusive pair, as do controller numbers 126 (Mono) and 127 (Poly).

If active Control Change commands for one or both members of a mutually exclusive pair appear in the checkpoint history, exactly one controller log MUST appear in controller list to code the pair.

If active Control Change commands for one or both members of a mutually exclusive pair appear in the session history, at most one controller log MAY appear in controller list to code the pair.

In both cases, the controller log that appears in the controller list MUST code the controller number of the most recent Control Change command of the pair in the session history.

A.7.2 Log Coding Rules

Figure A.7.2 shows the controller log structure of Chapter C.

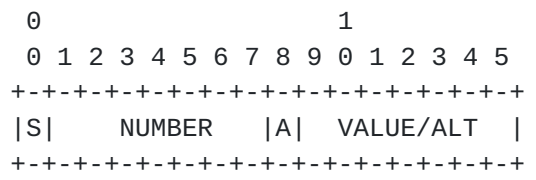

Figure A.7.2 -- Chapter C controller log

The 7-bit NUMBER field identifies the controller number. Controller logs for controller numbers 120-127 MUST appear at the start of the Chapter C controller list, in ascending NUMBER field order. Logs for controller numbers 0-119 MUST follow the 120-127 logs, also in ascending NUMBER field order.

The 7-bit VALUE/ALT field codes recovery information for the most recent C-active (controller numbers 0-119) or active (controller numbers 120-127) Control Change command in the session history.

Chapter C provides three tools for coding recovery information for a command in the VALUE/ALT field: the value tool, the toggle tool, and the count tool. Implementations may choose among the tools to code a Control Change command.

In the value tool, the 7-bit VALUE field codes the control value of the most recent C-active (controller numbers 0-119) or active (controller numbers 120-127) Control Change command in the session history. This tool works best for controllers that code a continuous quantity, such as number 1 (Modulation Wheel). If the value tool is chosen, the A bit is set to 0.

The A bit is set to 1 to code the toggle or count tool. These tools work best for controllers that code discrete actions. Figure A.7.3 shows the controller log for these tools.

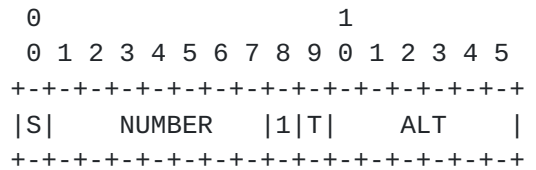

Figure A.7.3 -- Controller log for ALT tools

The T flag is set to 1 to code the toggle tool; T is set to 0 to code the count tool. Both methods use the 6-bit ALT field as an unsigned integer.

The toggle tools works best for controllers that act as on/off switches, such as 64 (Hold Pedal). These controllers code the "off" state with control values 0-63 and the "on" state with 64-127. The ALT field codes the total number of toggles (off->on and on->off) due to Control Change commands in the session history, including toggle events caused by MIDI Control Change number 121 (Reset All Controllers).

Toggle counting is performed modulo 64. The toggle count is reset at the start of a session, and whenever a System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), or General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7) appears in the session history. When these reset events occur, the toggle count for a controller is set to 0 (for controllers whose default value is 0-63) or 1 (for controllers whose default value is 64-127).

The Hold Pedal controller illustrates the benefit of the toggle tool over the value tool for switch controllers. As often used in piano applications, the "on" state of the Hold Pedal lets notes resonate, while the "off" state immediately damps notes to silence. The loss of the "off" command in an "on->off->on" sequence results in ringing notes that should have been damped silent. The toggle tool lets receivers detect this lost "off" command but the value tool does not.

The count tool is similar to the toggle tool, but is optimized for controllers whose value octet is ignored, such as 123 (All Notes Off). For the count tool, the ALT field codes the total number of Control Change commands in the session history. Command counting is performed modulo 64.

The command count is set to 0 at the start of the session, and is reset to 0 whenever a System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), or General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7) appears in the session history.

In most situations, a controller number SHOULD be coded by a single tool (and thus, a single controller log). However, a few controller numbers require several tool types (and thus, several controller logs) to code correctly.

For example, controller number 121 (Reset All Controllers) may require two tools to code correctly. Active commands for controller number 121 in the checkpoint history MUST be coded with the count tool. If the most recent command has a non-zero data octet, a second log MUST also appear in the controller list, and this log MUST use the value tool. This rule supports renderers (such as [9]) that use the data octet to code the reset semantics.

Multiple logs for the same controller number that use the same tool type

MUST NOT appear in the controller list.

A.7.3 Portamento Control

Controller number 84 (Portamento Control) codes that a pitch glide effect should be used for the next NoteOn command on the note number coded in the Control Change data octet. These semantics are a poor match for Chapter C, as a long-lived log entry may introduce a spurious portamento effect after each packet loss event, and thus create an indefinite artifact.

Note that by its nature, a lost Portamento Control command can not cause an indefinite artifact, as it only affects a single note instance. Rather, it is the attempt at recovery that causes the artifact. However, banning controller number 84 from Chapter C is not an option, as the Portamento Control semantics are a recent MIDI addition, and thus number 84 is often used as a generic controller.

This situation motivates the following rule. If a C-active Control Change command for controller number 84 appears in the checkpoint history, the controller list MUST contain at least 2 entries for the number. One entry MUST use the value tool, and one entry MUST use the count tool. The presence of both value and count tools lets a receiver detect long-lived log entries, and avoid the "indefinite portamento effect" problem.

A.7.4 The Parameter System

Appendix A.9 defines Chapter M, the MIDI Parameter chapter, to provide resiliency for the MIDI registered/non-registered parameter system. Here, we define the Chapter C rules for coding Control Change commands related to the parameter system. These rules serve to minimize redundancy with Chapter M.

Control Change commands for controller numbers 6 and 38 (Data Slider) and 96 and 97 (Data Button) may be used as part of the parameter system, or may be used as general-purpose controllers. Control Change commands for controller numbers 6, 38, 96, or 97 in the session history that are used in the parameter system MUST NOT appear as entries in the controller list.

However, if C-active Control Change commands for controller numbers 6, 38, 96, or 97 appear in the checkpoint history, and these commands are used as general-purpose controllers, the most recent general-purpose command instance for these controller numbers MUST appear as entries in the controller list.

Likewise, if C-active Control Change commands for controller numbers 6,

38, 96, or 97 appear in the session history, and these commands are used as general-purpose controllers, the most recent general-purpose command instance for these controller numbers MAY appear as entries in the controller list. In Chapter C, the controller number pair (6, 38) adheres to the 14-bit log inclusion rules defined in Appendix A.7.1, and controllers numbers 96 and 97 are coded as 7-bit controllers, independent of each other and from the controller pair (6, 38).

A parameter system transaction begins with paired Control Change commands for controller numbers 98 and 99 (Non-Registered Parameter LSB and MSB) or 100 and 101 (Registered Parameter LSB and MSB). Chapter M codes these paired Control Change commands. The Chapter C rule below acts to code "unpaired" commands for these controller numbers, that appear in the checkpoint history if a (98, 99) or (100, 101) pair is split across the MIDI command sections of two RTP packets.

If the most recent C-active Control Change command for controller 98, 99, 100, or 101 in the session history is part of a (98, 99) or (100, 101) command pair that begins a parameter system transaction, the command MUST NOT appear in the controller list.

However, if the most recent C-active Control Change command for controller 98, 99, 100, or 101 in the checkpoint history does not form part of a (98, 99) or (100, 101) command pair, an entry MUST appear in the controller list. Likewise, if the most recent C-active Control Change command for controller 98, 99, 100, or 101 in the session history does not form part of a (98, 99) or (100, 101) command pair, an entry MAY appear in the controller list.

A.8 Chapter E: MIDI Reset All Controllers

As defined in [1], the Control Change (0xB) command for controller number 121 (Reset All Controllers) resets controller numbers 0-119 to the "ideal initial state" for the device. As a consequence, the definition of Chapter C (Appendix A.7) limits the inclusion of Control Change commands for controller numbers 0-119 to C-active commands (Appendix A.1). This rule ensures that receivers do not incorrectly set controllers to "stale" values during recovery from a loss event.

However, rendering standards may define certain controller numbers in the 0-119 range to be unaffected by Reset All Controllers commands. For example, DLS 2 [9] declares controller numbers 7, 10 and 11 to be unaffected by a Reset All Controller command whose data octet is null. Thus, Chapter C would not protect controller numbers 7, 10, and 11 if a packet loss event occurred at an inopportune moment in a stream.

Chapter E is designed to close this coverage gap in the recovery

journal. Chapter E uses the same bitfield format as Chapter C: a 1-octet header, followed by a variable length list of 2-octet controller logs, as shown in Figure A.7.1.

A channel journal MUST contain Chapter E if (1) an active Control Change command for controller number 121 (Reset All Controllers) appears in the checkpoint history, and if (2) the rules we define below yield a non-empty list of controller logs. Otherwise, a channel journal MUST NOT contain Chapter E.

The controller log inclusion rules for Chapter E are identical to the inclusion rules for Chapter C, except that:

- All uses of the term "C-active" in Appendix A.7 are replaced with the term "active".
- Control Change commands that occur after the most recent Reset All Controllers command in the session history MUST NOT be coded in Chapter E.
- Controller numbers 120-127 MUST NOT be coded in Chapter E.
- If a controller number appears in Chapter C of a channel journal, the number MUST NOT appear in Chapter E of the channel journal. In addition, if the MSB controller numbers 0-31 appears in Chapter C, the associated LSB controller numbers 32-63 MUST NOT appear in Chapter E.
- If the channel journal contains Chapter P, and the BANK-COARSE field of Chapter P codes a Control Change command that occurred after the most recent Reset All Controllers command in the session history, Chapter E MUST NOT code controllers 0 and 32.
- Chapter E MUST NOT code Control Change commands for controller numbers 0 or 32 if the BANK-COARSE or BANK-FINE fields of Chapter P code the same command instances.

As defined by these rules, Chapter E codes a snapshot of the active Control Change commands for controllers 0-119 that appear in the checkpoint history before the most recent Reset All Controllers command. As the Control Change commands that follow the Reset All Controllers command make part of the snapshot irrelevant, formerly REQUIRED controller logs in Chapter E are removed from the controller list.

The normative text in this Appendix reflects the default recovery journal behavior. In most situations, session participants know which controller numbers (if any) require Chapter E support. Parties SHOULD use this knowledge to minimize the size of the Chapter E bitfields, by

using the session configuration tools defined in Appendix C.1.3.

A.9 Chapter M: MIDI Parameter System

Readers may wish to review the Appendix A.1 definitions for "parameter system", "parameter system transaction", and "initiated parameter system transaction" before reading this Appendix.

Chapter M protects RPN and NRPN parameters. Figure A.9.1 shows the variable-length format of Chapter M.

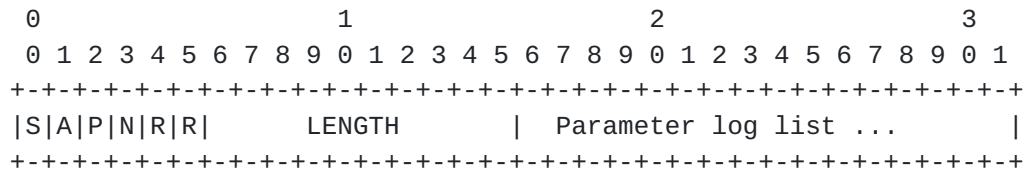

Figure A.9.1 -- Top-level Chapter M format

Chapter M consists of a 2-octet header, followed by a list of variable-length parameter logs. The list MUST contain at least one parameter log. The 10-bit LENGTH field codes the size of Chapter M, and conforms to semantics described in Appendix A.1.

A channel journal MUST contain Chapter M if the rules defined in this Appendix require that one or more parameter logs appear in the list.

A.9.1 Log Inclusion Rules

Parameter logs code recovery information for a specific RPN or NRPN parameter. Multiple logs for the same RPN or NRPN parameter MUST NOT appear in the list.

By default, a parameter log MUST appear in the list if a C-active command that forms part of an initiated transaction for the parameter appears in the checkpoint history. If Chapter M uses these default semantics, the A header bit MUST be set to 0.

During session configuration, Chapter M may be customized to require that a parameter log MUST appear in the list if an active (as opposed to C-active) command that forms part of an initiated transaction for the parameter appears in the checkpoint history. If Chapter M uses these modified semantics, the A header bit MUST be set to 1.

In both configurations, a log MAY appear in the list if an active

command associated with an initiated transaction for the parameter appears in the session history.

Parameter logs MUST be ordered with respect to the relative recency of transactions for the parameter. The first log in the list codes the parameter most recently involved in a transaction, the second log codes a parameter whose most recent transaction occurred before the most recent transaction of the first list parameter, etc.

The N and P header bits signal the presence of ongoing RPN and NRPN transactions in the session history. If the session history does not include commands to terminate the most recent initiated transaction for the first RPN parameter log in the list, P MUST be set to 1. Otherwise, P MUST be set to 0. Likewise, if the session history does not include commands to terminate the most recent initiated transaction for the first NRPN parameter log in the list, N MUST be set to 1. Otherwise, N MUST be set to 0.

Transactions for the RPN and NRPN null parameter (0x3FFF) MUST NOT appear in the list. Null parameter transactions are coded implicitly, by the N and P header bits and by the ordering of parameter log list.

A.9.2 Log Coding Rules

Figure A.9.2 shows the parameter log structure of Chapter M.

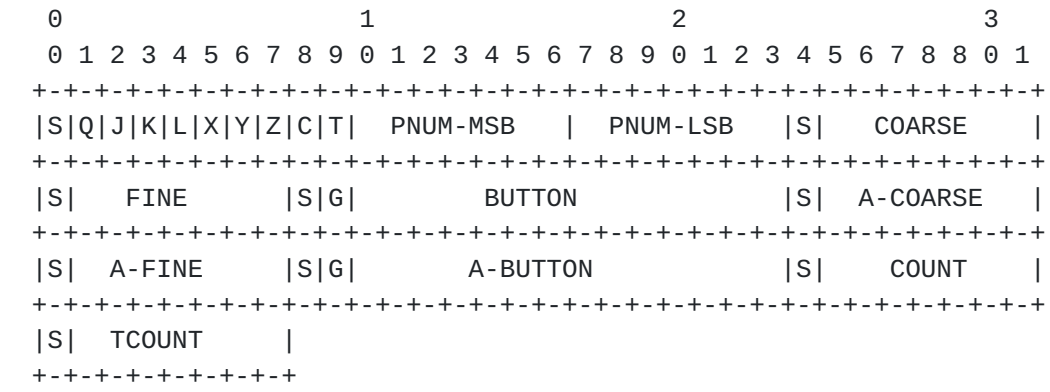

Figure A.9.2 -- Chapter M parameter log

The log begins with a 3-octet header (10 flag bits, followed by the PNUM-MSB and PNUM-LSB fields). If the Q header bit is set to 0, the log encodes an RPN parameter. If Q = 1, the log encodes an NRPN parameter. The 7-bit PNUM-LSB and PNUM-MSB fields code the parameter number, and reflect the Control Change command data values for controller numbers 98-99 (for NRPNs) or 100-101 (for RPNs).

The J, K, L, X, Y, Z, C and T header bits form a Table of Contents (TOC) for the log, and signal the presence of fixed-sized fields that optionally follow the header. Figure A.9.2 shows a fully-populated log (coded by setting all TOC bits to 1). The ordering of fields in the log follows the ordering of the header bits in the TOC. A set header bit codes the presence of a field in the log.

Each field acts as a coding tool to protect the parameter. If the rules in Appendix A.9.1 state that a log for a given parameter MUST appear in Chapter M, the log MUST include the subset of fields necessary to protect the parameter for loss events, given the semantics of the parameter. A safe (but inefficient) option is to use all possible fields for each coded parameter log.

A.9.2.1 COARSE and FINE Fields

The J bit codes the presence of the 7-bit COARSE field and its associated S bit. The COARSE field codes the data value of the most recent C-active Control Change command for controller number 6 (Data Entry MSB) in the session history that appears in a transaction for the log parameter.

The K bit codes the presence of the 7-bit FINE field and its associated S bit. The FINE field codes the data value of the most recent C-active Control Change command for controller number 38 (Data Entry LSB) in the session history that appears in a transaction for the log parameter. However, the FINE field MUST NOT appear in the log if the Control Change command associated with FINE field precedes the Control Change command associated with the COARSE field in the session history.

Note that the FINE field MAY appear in the log even if the COARSE field does not appear in the log. In this situation, FINE does not have C-active semantics, but may have active semantics if the A-COARSE field is present in the log (Appendix A.9.2.3).

A.9.2.2 The BUTTON Field

The B bit codes the presence of the 14-bit BUTTON field and its associated S and G bits. The fields code the use of Control Change commands for controller numbers 96 and 97 (Data Button Increment and Data Button Decrement) in transactions for the log parameter. BUTTON is interpreted as an unsigned integer, and the G bit codes the sign of the integer (G = 1 for positive, G = 0 for negative).

If the parameter log does not use the COARSE field, the BUTTON and G fields code a signed count of the number of C-active Data Button Increment and Decrement Control Change commands in the session history that appear in a transaction for the log parameter.

If the log uses the COARSE field but not the FINE field, the BUTTON and G fields code a signed count of the number of C-active Data Button Increment and Decrement Control Change commands in the session history that are more recent than the Control Change command associated with the COARSE field, and that appear in a transaction for the log parameter.

If the log uses the FINE field, the BUTTON and G fields code a signed count of the number of C-active Data Button Increment and Decrement Control Change commands in the session history that are more recent than the Control Change command associated with the FINE field, and that appear in a transaction for the log parameter. If necessary, the value of the COARSE or the A-COARSE field MUST be adjusted to reflect the presence of C-active Data Button Increment and Decrement Control Change commands between the Control Change command associated with the COARSE or A-COARSE field and the Control Change command associated with the FINE field.

To compute and code the count value, initialize the count value to 0, add 1 for each qualifying Data Button Increment command, subtract 1 for each qualifying Data Button Decrement command, and limit the magnitude of the final count to 16383. The G bit codes the sign of the count, and the BUTTON field codes the magnitude of the count.

A.9.2.3 The A-COARSE, A-FINE, and A-BUTTON Fields

The X header bit codes the presence of the 7-bit A-COARSE parameter and its associated S bit. The Y bit codes the presence of the 7-bit A-FINE field and its associate S bit. The Z bit codes the presence of the 14-bit A-BUTTON field and its associated G and S bits.

The rules we define below let the A-COARSE, A-FINE, and A-BUTTON fields code a snapshot of the parameter value at the moment before the appearance of the most recent active Control Change command for controller number 121 (Reset All Controllers) in the session history. This snapshot, in combination with the COARSE, FINE, and BUTTON fields, lets receivers who ignore Reset All Controllers Control Change commands for an RPN or NPRN parameter recover from packet loss events.

The A-COARSE, A-FINE, and A-BUTTON fields MUST NOT appear in the log if an active Control Change command for controller number 121 (Reset All Controllers) does not appear in the session history. The A-COARSE, A-FINE, and A-BUTTON fields MUST NOT appear in the log if Appendix A.9.2.1 permits the COARSE field to appear in the log. The A-FINE and A-BUTTON fields MUST NOT appear in the log if Appendix A.9.2.1 permits the FINE field to appear in the log.

The A-COARSE field codes the data value of the most recent active Control Change command for controller number 6 (Data Entry MSB) in the

session history that appears in a transaction for the log parameter, and that precedes the most recent Reset All Controllers Control Change command in the session history. If a Control Change command for controller number 6 that meets this criteria does not exist, the A-COARSE field MUST NOT appear in the log.

The A-FINE field codes data value of the most recent active Control Change command for controller number 38 (Data Entry LSB) in the session history that appears in a transaction for the log parameter, and that precedes the most recent Reset All Controllers Control Change command in the session history. If a Control Change command for controller number **38 that meets this criteria does not exist, the A-FINE field MUST NOT** appear in the log. The A-FINE field MUST NOT appear in the log if the A-COARSE field does not appear in the log, or if the command associated with the A-FINE field precedes the command associated with the A-COARSE field in the session history.

If the log does not use the A-COARSE field, the A-BUTTON and its associated G bit code a signed count of the number of active Data Button Increment and Decrement Control Change commands in the session history that appear in a transaction for the log parameter, and that precede the most recent Reset All Controllers Control Change command in the session history.

If the log uses the A-COARSE field but not the A-FINE field, the A-BUTTON and its associated G bit code a signed count of the number of active Data Button Increment and Decrement Control Change commands in the session history that are more recent than the Control Change command associated with the A-COARSE field, that appear in a transaction for the log parameter, and that precede the most recent Reset All Controllers Control Change command in the session history.

If the log uses the A-COARSE and A-FINE fields, the A-BUTTON and its associated G bit code a signed count of the number of active Data Button Increment and Decrement Control Change commands in the session history that are more recent than the Control Change command associated with the FINE field, that appear in a transaction for the log parameter, and that precede the most recent Reset All Controllers Control Change command in the session history. If necessary, the value of the A-COARSE field MUST be adjusted to reflect the presence of Data Button Increment and Decrement Control Change commands between the Control Change command associated with the A-COARSE field and the Control Change command associated with the A-FINE field.

To compute and code the count value, initialize the count value to 0, add 1 for each qualifying Data Button Increment command, subtract 1 for each qualifying Data Button Decrement command, and limit the magnitude of the final count to 16383. The G bit codes the sign of the count, and

the A-BUTTON field codes the magnitude of the count.

A.9.2.4 The COUNT and TCOUNT Fields

The C bit codes the presence of the 7-bit COUNT field and its associated S bit. The COUNT field codes the number of active Control Change commands for controller numbers 6, 38, 96, and 97 in the session history that appear in a transaction for the log parameter.

The T bit codes the presence of the 7-bit TCOUNT field, and its associated S bit. The TCOUNT field codes the number of initiated transactions for the parameter in the session history that contain at least one active Control Change command, including commands for controller numbers 98-101 that initiate a transaction, but excluding commands for controller numbers 98-101 that terminate the transaction.

COUNT and TCOUNT counting is performed modulo 128. COUNT and TCOUNT are set to 0 at the start of a session, and are reset to 0 whenever a System Reset (0xFF), General MIDI System Enable (0xF0 0x7E 0xcc 0x09 0x01 0xF7), or General MIDI System Disable (0xF0 0x7E 0xcc 0x09 0x00 0xF7) appears in the session history.

B. The Recovery Journal System Chapters

B.1 System Chapter D: Simple System Commands

The system journal MUST contain Chapter D if an active MIDI Reset (0xFF), MIDI Tune Request (0xF6), MIDI Song Select (0xF3), undefined MIDI System Common (0xF4 and 0xF5), or undefined MIDI System Real-time (0xF9 and 0xFD) command appears in the checkpoint history.

Figure B.1.1 shows the variable-length format for Chapter D.

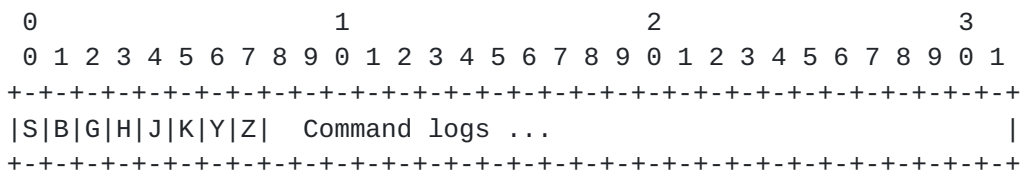

Figure B.1.1 -- System Chapter D format

The chapter consists of a 1-octet header, followed by one or more command logs. Header flag bits indicate the presence of command logs for the Reset (B = 1), Tune Request (G = 1), Song Select (H = 1), undefined System Common 0xF4 (J = 1), undefined System Common 0xF5 (K =

1), undefined System Real-time 0xF9 (Y = 1), or undefined System Real-time 0xFD (Z = 1) commands.

Command logs appear in a list following the header, in the order that the flag bits appear in the header.

Figure B.1.2 shows the 1-octet command log format for the Reset and Tune Request commands.

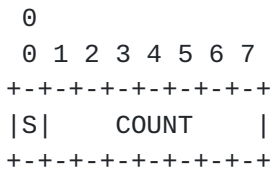

Figure B.1.2 -- Command log for Reset and Tune Request

Chapter D MUST contain the Reset command log if an active Reset command appears in the checkpoint history. The 7-bit COUNT field codes the total number of Reset commands (modulo 128) present in the session history.

Chapter D MUST contain the Tune Request command log if an active Tune Request command appears in the checkpoint history. The 7-bit COUNT field codes the total number of Tune Request commands (modulo 128) present in the session history.

Figure B.1.3 shows the 1-octet command log format for the Song Select command.

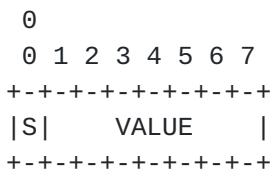

Figure B.1.3 -- Song Select command log format

Chapter D MUST contain the Song Select command log if an active Song Select command appears in the checkpoint history. The 7-bit VALUE field codes the song number of the most recent active Song Select command in the session history.

B.1.1 Undefined System Commands

In this section, we define the Chapter D command logs for the undefined System opcodes. [1] reserves the undefined System opcodes 0xF4, 0xF5, 0xF9, and 0xFD for future use. At the time of this writing, any MIDI command stream that uses these opcodes is non-compliant with [1]. However, future versions of [1] may define these opcodes, and a few products do use these opcodes in a non-compliant manner.

Figure B.1.4 shows the variable length command log format for the undefined System Common commands (0xF4 and 0xF5).

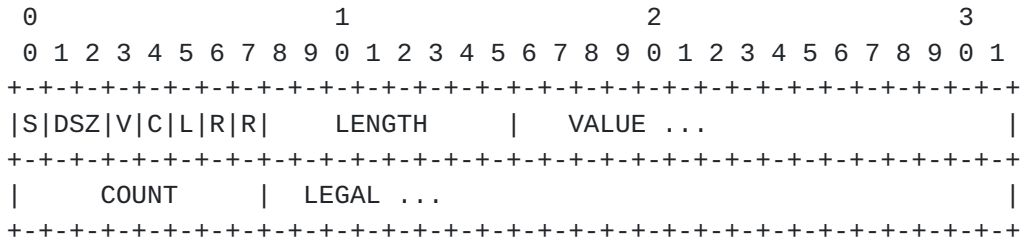

Figure B.1.4 -- Undefined System Common command log format

The command log codes a single opcode type (0xF4 or 0xF5, not both). Chapter D MUST contain a command log if an active 0xF4 command appears in the checkpoint history, and MUST contain an independent command log if an active 0xF5 command appears in the checkpoint history.

Chapter D consists of a two-octet header followed by a variable number of data fields. Header flag bits indicate the presence of the VALUE field (V = 1), the COUNT field (C = 1), and the LEGAL field (L = 1). The 8-bit LENGTH field codes the size of the command log, and conforms to semantics described in Appendix A.1.

The 2-bit DSZ field codes the number of data octets in the command instance that appears most recently in the session history. If DSZ = 0-2, the command has 0-2 data octets. If DSZ = 3, the command has 3 or more command data octets.

We now define the default rules for the use of the VALUE, COUNT, and LEGAL fields. The session configuration tools defined in Appendix C.1.3 may be used to override this behavior.

If the DSZ field is set to 0, the command log MUST include the COUNT field. The 8-bit COUNT field codes the total number of opcode commands present in the session history, modulo 256.

If the DSZ field is set to 1-3, the command log MUST include the VALUE field. The variable-length VALUE field codes a verbatim copy the data

octets for the most recent use of the opcode in the session history. The most-significant bit of the final data octet MUST be set to 1, and the most-significant bit of all other data octets MUST be set to 0.

The LEGAL field is reserved for future use. If an update to [1] defines the 0xF4 or 0xF5 opcode, an IETF standards-track document MAY define the LEGAL field to protect the opcode. Until such a document appears, senders MUST NOT use the LEGAL field, and receivers MUST use the LENGTH field to skip over the LEGAL field.

Figure B.1.5 shows the variable length command log format for the undefined System Real-time commands (0xF9 and 0xFD).

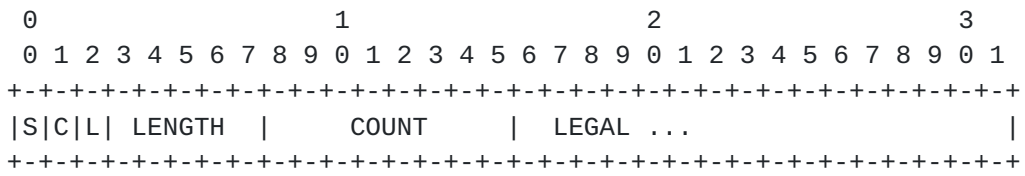

Figure B.1.5 -- Undefined System Real-time command log format

The command log codes a single opcode type (0xF9 or 0xFD, not both). Chapter D MUST contain a command log if an active 0xF9 command appears in the checkpoint history, and MUST contain an independent command log if an active 0xFD command appears in the checkpoint history.

Chapter D consists of a one-octet header followed by a variable number of data fields. Header flag bits indicate the presence of the COUNT field (C = 1) and the LEGAL field (L = 1). The 5-bit LENGTH field codes the size of the command log, and conforms to semantics described in Appendix A.1.

We now define the default rules for the use of the COUNT and LEGAL fields. The session configuration tools defined in Appendix C.1.3 may be used to override this behavior.

The 8-bit COUNT field codes the total number of opcode commands present in the session history, modulo 256. By default, the COUNT field MUST be present in the command log.

The LEGAL field is reserved for future use. If an update to [1] defines the 0xF9 or 0xFD opcode, an IETF standards-track document MAY define the LEGAL field to protect the opcode. Until such a document appears, senders MUST NOT use the LEGAL field, and receivers MUST use the LENGTH field to skip over the LEGAL field.

Finally, we note that some non-standard uses of the undefined System Real-time opcodes act to implement non-compliant variants of the MIDI sequencer system. In Appendix B.3.1, we describe resiliency tools for the MIDI sequencer system that provide some protection in this case.

B.2 System Chapter V: Active Sense Command

The system journal MUST contain Chapter V if an active MIDI Active Sense (0xFE) command appears in the checkpoint history. Figure B.2.1 shows the format for Chapter V.

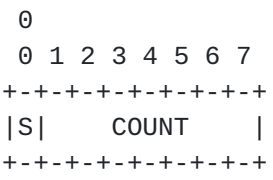

Figure B.2.1 -- System Chapter V format

The 7-bit COUNT field codes the total number of Active Sense commands (modulo 128) present in the session history.

B.3 System Chapter Q: Sequencer State Commands

This Appendix describes Chapter Q, the system chapter for the MIDI sequencer commands.

The system journal MUST contain Chapter Q if an active MIDI Song Position Pointer (0xF2), MIDI Clock (0xF8), MIDI Start (0xFA), MIDI Continue (0xFB) or MIDI Stop (0xFC) command appears in the checkpoint history. Figure B.3.1 shows the variable-length format for Chapter Q.

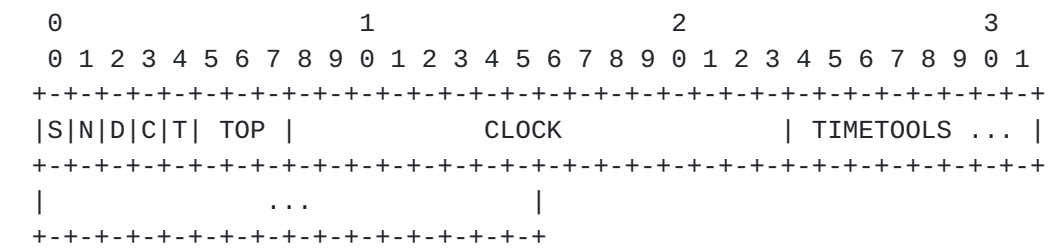

Figure B.3.1 -- System Chapter Q format

Chapter Q encodes the most recent state of the sequencer system. Receivers use the chapter to re-synchronize the sequencer after a packet loss episode. Chapter fields encode the position of the sequencer pointer, the presence of the downbeat, and the on/off state of the sequencer.

Chapter Q consists of a 1-octet header followed by several optional fields, in the order shown in Figure B.3.1. Header flag bits signal the presence of the 16-bit CLOCK field (C = 1) and the 24-bit TIMETOOLS field (T = 1).

The N header bit encodes the relative occurrence of the Start, Stop, and Continue commands in the session history. If an active Start or Continue command appears most recently, the N bit MUST be set to 1. If an active Stop appears most recently, or if no active Start, Stop, or Continue commands appear in the session history, the N bit MUST be set to 0.

The D header bit encodes the presence of the downbeat. If N is set to 1, and if at least one Clock command follows the most recent Start or Continue command in the session history, the D bit MUST be set to 1. In all other cases, the D bit MUST be set to 0.

If N is set to 0 (coding a stopped sequence), or if N is set to 1 and D is set to 0 (coding a sequence on the verge of beginning), Chapter Q MUST encode the starting song position of the sequence. The C flag, the TOP field, and the CLOCK field act to code the starting song position:

- If C = 0, the song position is at the beginning of the song.

- If C = 1, the 3-bit TOP header field and the 16-bit CLOCK field are combined to form the 19-bit unsigned quantity 65536*TOP + CLOCK. This value encodes the song position in units of clocks (24 clocks per quarter note).

If the N and D header bits are both set to 1, the sequence is playing, and Chapter Q MUST encode the current song position in the sequence.

The current song position is coded using the same fields and methods as the starting song position (65536*TOP + CLOCK, with C set to 1).

B.3.1 Non-compliant Sequencers

The Chapter Q description in this Appendix assumes that the sequencer system counts off time with Clock commands, as mandated in [1]. However, a few non-compliant products do not use Clock commands to count off time, but instead use non-standard methods.

Chapter Q uses the TIMETOOLS field to provide resiliency support for these non-standard products. By default, the TIMETOOLS field MUST NOT appear in Chapter Q, and the T header bit MUST be set to 0. The session configuration tools described in Appendix C.1.3 may be used to select TIMETOOLS coding.

Figure B.3.2 shows the format of the 24-bit TIMETOOLS field.

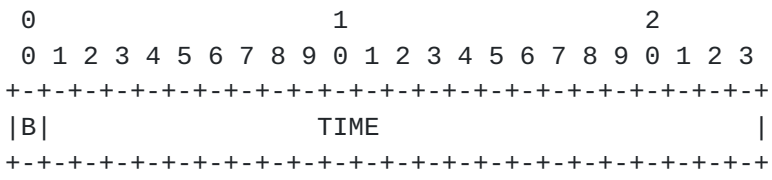

Figure B.3.2 -- TIMETOOLS format

The TIME field is a 23-bit unsigned integer quantity, with units of milliseconds. TIME codes an additive correction term for the song position coded by the TOP, CLOCK, C fields. TIME is coded in network byte order (big-endian).

A receiver computes the correct song position by converting TIME into units of MIDI clocks and adding it to 65536*TOP + CLOCK (assuming C = 1). Alternatively, a receiver may convert 65536*TOP + CLOCK into milliseconds (assuming C = 1) and add it to TIME.

The B bit encodes the presence of the downbeat in the non-standard command stream. If the N header bit is set to 1, and if at least one non-standard command that counts off time follows the most recent Start or Continue command in the session history, the B bit MUST be set to 1. In all other cases, B MUST be set to 0.

B.4 System Chapter F: MIDI Time Code Tape Position

This Appendix describes Chapter F, the system chapter for the MIDI Time

Code (MTC) commands. Readers may wish to review the Appendix A.1 definition of "finished/unfinished commands" before reading this Appendix.

The system journal MUST contain Chapter F if an active System Common Quarter Frame command (0xF1) or an active finished System Exclusive (Universal Real Time) MTC Full Frame command (F0 7F cc 01 01 hr mn sc fr F7) appears in the checkpoint history.

Figure B.4.1 shows the variable-length format for Chapter F.

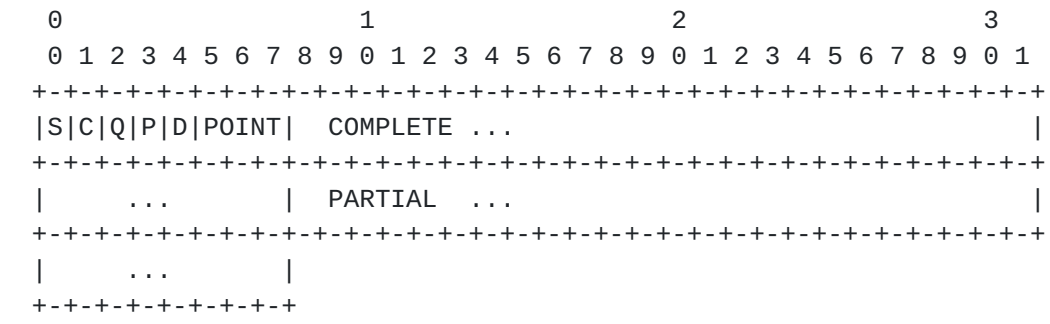

Figure B.4.1 -- System Chapter F format

Chapter F holds information about the most recent MTC tape position coded in the session history. Receivers use Chapter F to re-synchronize the MTC system after a packet loss episode.

Chapter F consists of a 1-octet header followed by several optional fields, in the order shown in Figure B.4.1. Header flag bits signal the presence of the 32-bit COMPLETE field (C = 1) and the 32-bit PARTIAL field (P = 1).

Chapter F MUST include the COMPLETE field if an active finished Full Frame command appears in the checkpoint history, or if an active Quarter Frame command that completes the encoding of a frame value appears in the checkpoint history.

The COMPLETE field encodes the most recent active complete MTC frame value that appears in the session history. This frame value may take the form of a series of 8 active Quarter Frame commands (0xF1 0x0n through 0xF1 0x7n for forward tape movement, 0xF1 0x7n through 0xF1 0x0n for reverse tape movement), or may take the form of an active finished Full Frame command.

If the COMPLETE field encodes a Quarter Frame command series, the Q header bit MUST be set to 1, and the COMPLETE field MUST have the format

shown in Figure B.4.2. The 4-bit fields MT0 through MT7 code the binary data nibble for the Quarter Frame commands for Message Type 0 through Message Type 7 [1]. These nibbles encode a complete frame value, in addition to fields reserved for future use by [1].

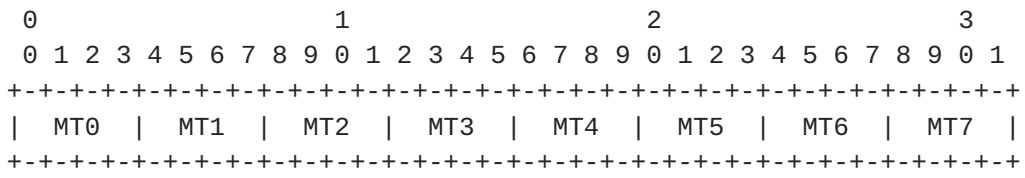

Figure B.4.2 -- COMPLETE field format, Q = 1

In this usage, the frame value encoded in the COMPLETE field MUST be offset by 2 frames, relative to the frame value encoded in the Quarter Frame commands, if the tape is moving in the forward direction. This offset compensates for the two frame latency of the Quarter Frame encoding. No offset is applied if the tape is moving in reverse.

Alternatively, the most recent active complete MTC frame value may be encoded by an active finished Full Frame command. In this case, the Q header bit MUST be set to 0, and the COMPLETE field MUST have format shown in Figure B.4.3. The HR, MN, SC, and FR fields correspond to the hr, mn, sc, and fr data octets of the Full Frame command.

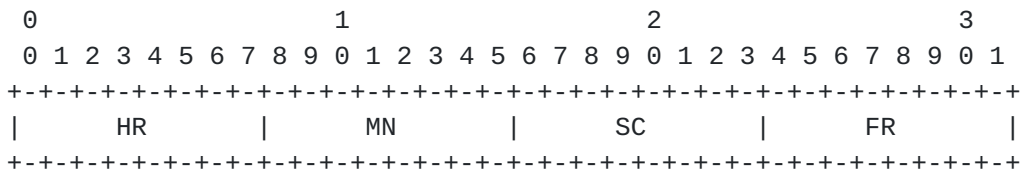

Figure B.4.3 -- COMPLETE field format, Q = 0

B.4.1 Partial Frames

The most recent active session history command that encodes MTC frame value data may be a Quarter Frame command other than a forward-moving 0xF1 0x7n command (which completes a frame value for forward tape movement) or a reverse-moving 0xF1 0x1n command (which completes a frame value for reverse tape movement).

We define this type of Quarter Frame command as being associated with a partial frame value encoding. This definition only holds if the partial frame value is well-formed: the Quarter Frame sequence MUST start at

Message Type 0 and increment contiguously to an intermediate value, or start at Message Type 7 and decrement contiguously to an intermediate value).

Chapter F MUST include a PARTIAL field if the most recent active command in the checkpoint history that encodes MTC frame value data is a Quarter Frame command that is associated with a partial frame value.

The PARTIAL field MUST have the format shown in Figure B.4.2. The D and POINT header fields (Figure B.4.1) qualify the contents of the PARTIAL field, as we now describe.

The D header bit reflects the direction of tape movement coded by the Quarter Frame command (D = 0 for forward movement, D = 1 for reverse movement). The 3 bit POINT header field encodes the unsigned integer value formed by the lower 3 bits of the upper nibble of the data value of the most recent active Quarter Frame command in the session history.

If D = 0, POINT may take on the values 0-6. If D = 1, POINT may take on the values 1-7. If D = 0, MT fields (Figure B.4.2) in the inclusive range 0 to the POINT value encode the partial frame value, and all other MT fields MUST be ignored. If D = 1, MT fields in the inclusive range 7 down to the POINT value encode the partial frame value, and all other MT fields MUST be ignored.

Senders MUST NOT add a 2-frame offset to the partial frame value encoded in the PARTIAL field. Unlike the COMPLETE field, an offset is not necessary because the D bit encodes the tape direction.

The header field value pairs (D = 0, POINT = 7) and (D = 1, POINT = 0) are reserved for future use. Senders MUST NOT use these value pairs and receivers MUST ignore the PARTIAL field if these value pairs appear in the chapter header.

B.5 System Chapter X: System Exclusive

This Appendix describes Chapter X, the system chapter for MIDI System Exclusive (SysEx) commands (opcode 0xF0). Readers may wish to review the Appendix A.1 definition of "finished/unfinished commands" before reading this Appendix.

The system journal may code multiple Chapter X chapters. Chapter X journal chapters are ordered with respect to the recency of the SysEx command coded by the chapter. The chapter coding the most recent SysEx command in the session history appears first in the system journal, followed by a chapter coding an older command, followed by a chapter coding an even older command, etc.

The system journal MUST contain at least one Chapter X chapter if an active SysEx command (excluding a finished MTC Full Frame command) appears in the checkpoint history. A SysEx command "appears" in the checkpoint history if the history contains a verbatim encoding of the SysEx command, or if the history contains at least one segment of a segmental encoding of the SysEx command.

Chapter X is optimized for the small SysEx commands that signal real-time events, not the large SysEx commands used for bulk data. Bulk data commands SHOULD be sent over reliable transport. Appendix C.4 defines session configuration tools for splitting a MIDI name space into streams that are carried on different transports.

B.5.1 Chapter Format

Figure B.5.1 shows the variable length format for System Chapter X.

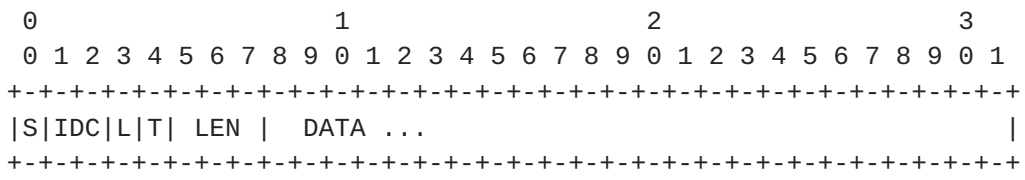

Figure B.5.1 -- System Chapter X format

Chapter X consists of a 1-octet header, following by an arbitrary length DATA field. The DATA field encodes a modified version of the data octets of a SysEx command. The leading 0xF0 and trailing 0x7F SysEx octets never appear in the DATA field.

The DATA field encodes all command data octets that appears in the session history (as distinct from the checkpoint history). This distinction is relevant for the coding of commands whose segments appear across multiple packets. In this case, the DATA field MUST include the starting segments for the command, even if these segments no longer appear in the checkpoint history.

If the Manufacturer ID value of the SysEx command (coded in the first octet of the MIDI command) has the values 0x00, 0x7E, or 0x7F, the DATA field begins with the second data octet of the SysEx command; for all other Manufacturer ID values, the DATA field begins with the first data octet of the SysEx command. The 2-bit IDC header field codes 0x00, 0x7E, and 0x7F ID values, using the method shown in Figure B.5.2.

IDC	Manufacturer ID	First DATA octet is:
0x0	0x7E (Universal Real-Time)	2nd SysEx data octet
0x1	0x7F (Universal Non-Real-Time)	2nd SysEx data octet
0x2	0x00 (Extension Escape Code)	2nd SysEx data octet
0x3	in the range 0x01--0x7D	1st SysEx data octet

Figure B.5.2 -- IDC header field encoding

The 3-bit LEN header field codes the exact length of short, complete SysEx commands, and signals alternative coding techniques for longer commands and truncated commands.

The LEN values 0x0 through 0x5 indicate that the length of the DATA field is 1-6 octets. For these LEN values, the DATA field encodes a complete SysEx command, as a verbatim copy of the SysEx data octets (possibly skipping the first octet, per Figure B.5.2).

The LEN value 0x6 indicates that the DATA field contains 7 or more octets. The DATA field encodes a complete SysEx command, as a verbatim copy of the data octets of the SysEx command (possibly skipping the first octet, per Figure B.5.2). To code the field length, the most-significant bit of the final octet MUST be set to 1, and the most-significant bit of all other octets MUST be set to 0.

The LEN value 0x7 indicates that the SysEx command is truncated. This coding option is used for SysEx commands encoded using the segmented method, in the case where not all segments appear in the session history. The DATA field encodes a verbatim copy of the data octets of the command segments that appear in the session history, ordered from the first segment to the last segment, using the coding methods defined for the 0x6 LEN value.

B.5.2 Coding Tools

The L and T header flags (Figure B.5.1) indicate the coding tool for the Chapter X chapter. The coding tool sets the inclusion semantics for a subset of SysEx commands, which we call a type.

If the L bit is set to 1 (the list tool), all active commands that appear in the checkpoint history of the type coded in the DATA field MUST be coded by a chapter. If L is set to 0 (the recency tool), the

most recent active command that appears in the checkpoint history of the type coded in the DATA field MUST be coded by a chapter.

For each command type, an implementation may choose either the list tool or the recency tool. Simple implementations may use the list tool for all command types; sophisticated implementations may reduce bandwidth by using the recency tool for some command types.

The T flag defines the nature of the type. The T flag has different semantics for MIDI Universal SysEx commands (Manufacturers ID 0x7E and 0x7F) and for generic SysEx commands (all other Manufacturers ID values).

We first define the T flag for Universal SysEx commands. The first four data octets of Universal commands are defined in [1], using the syntax: ID cc SubID SubID1. If T is set to 0, all Universal commands with the same ID, cc, SubID, and SubID1 values are considered the same type. If T is set to 1, all Universal commands with the same ID, cc, and SubID values are considered the same type.

For generic SysEx commands (all Manufacturers ID values except 0x7E and 0x7F), we define the T flag as follows. The first data octet of a generic SysEx command is the Manufacturers ID; the remaining data octets may have an arbitrary organization, but often have a set of octets coding device and sub-command, followed by data octets for the command.

If T is set to 0, all generic SysEx commands with the same ID value are considered to be of the same type. If T is set to 1, the command is assumed to have a device/sub-command/data organization, and all commands with the same ID value, device, and sub-command values are considered to be of the same type. If the command has a multi-level sub-command structure, these semantics require identical sub-command values at all levels.

C. SDP Session Configuration Tools

In the main text, we show minimal session descriptions for native (Section 6.1) and mpeg4-generic (Section 6.2) streams. In this Appendix, we describe how to customize (and perhaps negotiate [15]) stream behavior through the use of the standard SDP attributes and the payload format fmtp parameters.

The Appendix is divided into 5 sections, each devoted to parameters that affect a particular aspect of stream behavior:

- Appendix C.1 describes the journalling system (ch_anchor, ch_default, ch_never, ch_unused, j_sec. j_update).
- Appendix C.2 describes MIDI command timestamp semantics (linerate, mperiod, octpos, tsmode).
- Appendix C.3 describes media time (guardtime, maxptime, ptime).
- Appendix C.4 describes multi-stream sessions (musicport, zerosync).
- Appendix C.5 describes MIDI rendering (chanmask, cid, inline, render, rinit, smf_cid, smf_info, smf_inline, smf_url, url).

Appendix C.5.4 defines the MIME type "audio/asc", a stored object for initializing mpeg4-generic renderers. RTP stream semantics are not defined for "audio/asc". Therefore, "asc" MUST NOT appear on the rtpmap line of a session description.

Appendix D defines the Augmented Backus-Naur Form (ABNF, [10]) syntax for the parameters listed above. Appendix H provides information to the Internet Assigned Numbers Authority (IANA) on the MIME types and parameters defined in this document.

C.1 SDP Definitions: The Journalling System

In this Appendix, we define the session description parameters that configure stream journalling and the recovery journal system.

The j_sec parameter (Appendix C.1.1) sets the journalling method for the stream. The j_update parameter (Appendix C.1.2) sets the recovery journal sending policy for the stream. Appendix C.1.2 also defines the sending policies of the recovery journal system.

Appendix C.1.3 defines several parameters that modify the recovery journal semantics. These parameters change the default recovery journal semantics as defined in Section 5 and Appendices A-B.

C.1.1 The j_sec Parameter

Section 2.2 defines the default journalling method for a stream. Streams that use unreliable transport (such as UDP) default to using the recovery journal. Streams that use reliable transport (such as TCP) default to not using a journal.

The fmtp parameter j_sec may be used to override this default. This memo defines two symbolic values for j_sec: "none", to indicate that all

stream payloads MUST NOT contain a journal section, and "recj", to indicate that all stream payloads MUST contain a journal section that uses the recovery journal format.

For example, the j_sec parameter might be set to "none" for a UDP stream that travels between two hosts on a local network that is known to provide reliable datagram delivery.

The session description below configures a UDP stream that does not use the recovery journal:

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
a=rtpmap: 96 rtp-midi/44100
a=fmtp: 96 j_sec=none;
```

Other IETF standards-track documents may define alternative journal formats. These documents MUST define new symbolic values for the j_sec parameter to signal the use of the format. If a session description uses a j_sec value unknown to the recipient, the recipient MUST NOT accept the description.

Special j_sec issues arise when sessions are managed by the Real Time Streaming Protocol (RTSP, [16]). In many streaming applications, the session description in the response to the DESCRIBE method does not code the transport details (such as UDP or TCP) for the session. Instead, server and client negotiate transport details using the SETUP method.

In this scenario, the use of the j_sec parameter may be ill-advised, as the server does not yet know the transport type for the session. In this case, the session description SHOULD configure the journalling system using the parameters defined in the remainder of Appendix C.1, but SHOULD NOT use j_sec to set the journalling status. Recall that if j_sec does not appear in the session description, the default method for choosing the journalling method is in effect (no journal for reliable transport, recovery journal for unreliable transport).

However, in situations where the server knows journalling is always required (such as pre-recorded streams that contain packet loss events) or never required (such as UDP streams sent over a reliable network), the session description returned by the DESCRIBE method SHOULD use the j_sec parameter.

C.1.2 The j_update Parameter

In Section 4, we use the term "sending policy" to describe the method a sender uses to choose the checkpoint packet identity for each recovery journal in a stream. In the sub-sections that follow, we normatively define three sending policies: anchor, closed-loop, and open-loop.

As stated in Section 4, the default sending policy for a stream is the closed-loop policy. The fmtp parameter j_update may be used to override this default.

We define three symbolic values for j_update: "anchor", to indicate that the stream uses the anchor sending policy, "open-loop", to indicate that the stream uses the open-loop sending policy, and "closed-loop", to indicate that the stream uses the closed-loop sending policy. See **Appendix C.1.3 for examples session descriptions that use the j_update** parameter.

Other IETF standards-track documents may define additional sending policies for the recovery journal system. These documents MUST define new symbolic values for the j_update parameter to signal the use of the new policy. If a session description uses a j_update value unknown to the recipient, the recipient MUST NOT accept the description.

C.1.2.1 The anchor Sending Policy

In the anchor policy, the sender uses the first packet in the stream as the checkpoint packet for all packets in the stream. The anchor policy satisfies the recovery journal mandate (Section 4), as the checkpoint history always covers the entire stream.

The anchor policy does not require the use of the Real Time Control Protocol (RTCP, [2]) or other feedback from receiver to sender. Senders do not need to take special actions to ensure that received streams start up free of artifacts, as the recovery journal always covers the entire history of the stream. Receivers are relieved of the responsibility of tracking the changing identity of the checkpoint packet, because the checkpoint packet never changes.

The main drawback of the anchor policy is bandwidth efficiency. Because the checkpoint history covers the entire stream, the size of the recovery journals produced by this policy usually exceeds the journal size of alternative policies. For single-channel MIDI data streams, the bandwidth overhead of the anchor policy is often acceptable (see **Appendix A.4 of [12]). For dense streams, the closed-loop or open-loop** policies may be more appropriate.

C.1.2.2 The closed-loop Sending Policy

The closed-loop policy is the default policy of the recovery journal

system. For each packet in the stream, the policy lets senders choose the smallest possible checkpoint history that satisfies the recovery journal mandate. As smaller checkpoint histories generally yield smaller recovery journals, the closed-loop policy reduces the bandwidth of a stream, relative to the anchor policy.

The closed-loop policy relies on feedback from receiver to sender. The policy assumes that a receiver periodically informs the sender of the highest sequence number it has seen so far in the stream, coded in the 32-bit extension format defined in [2]. In sessions that use RTCP, receivers transmit this information in the Extended Highest Sequence Number Received (EHSNR) field of Receiver Report (RR) packets. However, applications MAY use any method of feedback to implement the closed-loop policy.

The sender may safely use receiver sequence number feedback to guide checkpoint history management, because Section 4 requires receivers to repair indefinite artifacts whenever a packet loss event occur.

We now normatively define the closed-loop policy. At the moment a sender prepares an RTP packet for transmission, the sender is aware of R >= 0 receivers for the stream. Senders may become aware of a receiver via RTCP traffic from the receiver, via RTP packets from a paired stream sent by the receiver to the sender, via messages from a session management tool, or by other means. As receivers join and leave a session, the value of R changes.

Each known receiver k (1 <= k <= R) is associated with a 32-bit extended packet sequence number M(k), where the extension reflects the sequence number rollover count of the sender.

If the sender has received at least one feedback report from receiver k, M(k) is the most recent report of the highest RTP packet sequence number seen by the receiver, normalized to reflect the rollover count of the sender.

If the sender has not received a feedback report from the receiver, M(k) is the extended sequence number of the last packet the sender transmitted before it became aware of the receiver. If the sender became aware of this receiver before it sent the first packet in the stream, M(k) is the extended sequence number of the first packet in the stream.

Given this definition of M(), we now state the closed-loop policy. When preparing a new packet for transmission, a sender MUST choose a checkpoint packet with extended sequence number N, such that M(k) >= (N - 1) for all k, 1 <= k <= R, where R >= 1. The policy does not restrict sender behavior in the R == 0 (no known receivers) case.

Under the closed-loop policy as defined above, a sender may transmit packets whose checkpoint history is shorter than the session history (as defined in Appendix A.1). In this event, a new receiver that joins the stream may experience indefinite artifacts.

For example, if a Control Change (0xB) command for the channel volume (controller number 7) was sent early in a stream, and later a new receiver joins the session, the closed-loop policy may permit all packets sent to the new receiver to use a checkpoint history that does not include the channel volume Control Change command. As a result, the new receiver experiences an indefinite artifact, and play all notes on a channel too loudly or too softly.

To address this issue, the closed-loop policy states that whenever a sender becomes aware of a new receiver, the sender MUST determine if the receiver would be subject to indefinite artifacts under the closed-loop policy. If so, the sender MUST ensure that the receiver starts the session free of indefinite artifacts. In satisfying this requirement, senders MAY infer the initial MIDI state of the receiver from the session description. For example, the stream example in Section 6.2 has the initial state defined in [1] for General MIDI.

In some types of sessions, a receiver may have access to stream packets before the sender is aware of the receiver. In this case, the restrictions the closed-loop policy places on the sender may not protect the receiver from indefinite artifacts.

To address this issue, the closed-loop policy states that if a receiver participates in a session where it may have access to a stream before the sender is aware of the receiver, the receiver MUST take actions to ensure that its rendered MIDI performance does not contain indefinite artifacts. The receiver MUST NOT discontinue these protective actions until it is certain that the sender is aware of its presence.

The final set of normative closed-loop policy requirements concern how senders drop receivers from a stream. As defined earlier in this section, the closed-loop policy states that a sender MUST choose a checkpoint packet with extended sequence number N, such that M(k) >= (N - 1) for all k, 1 <= k <= R, where R >= 1. If the sender has received at least one feedback report from receiver k, M(k) is the most recent report of the highest RTP packet sequence number seen by the receiver, normalized to reflect the rollover count of the sender.

If this receiver k stops sending feedback to the sender, the M(k) value used by the sender reflects the last feedback report from the receiver. As time progresses without feedback from receiver k, this fixed M(k) value forces the sender to increase the size of the checkpoint history, and thus increases the bandwidth of the stream.

At some point, the sender may need to take action in order to limit the bandwidth of the stream. The closed-loop policy states that if this situation occurs, and if the nature of the session permits a sender to stop transmitting packets to the offending receiver, the sender MUST stop transmitting packets to this receiver. In other words, it is not permissible for a sender to no longer use M(k) in computing the checkpoint packet identity but still send the stream to receiver k, if it is possible for the sender to actively cut off receiver k from the stream.

In certain types of sessions, it may not be possible for a sender to actively stop sending packets to a particular receiver. The closed-loop policy states that if receivers participate in a session where senders are unable to stop sending packets to a particular receiver of the stream, the receiver MUST monitor the RTP stream, and any other sources of information, to determine if the sender is no longer using the M(k) feedback from the receiver to choose each checkpoint packet. If the receiver detects this condition, it MUST leave the session, and close down the rendered MIDI performance in a manner that is free of indefinite artifacts.

Finally, we note that the closed-loop policy is suitable for use in RTP/RTCP sessions that use multicast transport. However, aspects of the closed-loop policy do not scale well to sessions with large numbers of participants. The sender state scales linearly with the number of receivers, as the sender needs to track the identity and M(k) value for each receiver k. The average recovery journal size is not independent of the number of receivers, as the RTCP reporting interval backoff slows down the rate of a full update of M(k) values. The backoff algorithm may also increase the amount of ancillary state used by implementations of the normative sender and receiver behaviors defined in Section 4.

C.1.2.3 The open-loop Sending Policy

The open-loop policy is suitable for sessions that are not able to implement the receiver-to-sender feedback required by the closed-loop policy, and are also not able to use the anchor policy because of bandwidth constraints.

The open-loop policy does not place constraints on how a sender chooses the checkpoint packet for each packet in the stream. In the absence of such constraints, a receiver may find that the recovery journal in the packet that ends a loss event has a checkpoint history that does not cover the entire loss event. We refer to loss events of this type as uncovered loss events.

To ensure that uncovered loss events do not compromise the recovery journal mandate, the open-loop policy assigns specific recovery tasks to

senders, receivers, and the creators of session descriptions. The underlying premise of the open-loop policy is that the indefinite artifacts produces during uncovered loss events fall into two classes.

One class of artifacts are recoverable indefinite artifacts. Receivers are able to repair recoverable artifacts that occur during an uncovered loss event without intervention from the sender, at the potential cost of unpleasant transient artifacts.

For example, after an uncovered loss event, receivers are able to repair indefinite artifacts due to NoteOff (0x8) commands that may have occurred during the loss event, by executing NoteOff commands for all active NoteOns commands. This action causes a transient artifacts (a sudden silent period in the performance), but ensures that no stuck notes sound indefinitely. We refer to MIDI commands that are amenable to repair in this fashion as recoverable MIDI commands.

A second class of artifacts are unrecoverable indefinite artifacts. If this class of artifact occurs during an uncovered loss event, the receiver is not able to repair the stream.

For example, after an uncovered loss event, receivers are not able to repair indefinite artifacts due to Control Change (0xB) channel volume (controller number 7) commands that have occurred during the loss event. A repair is impossible because the receiver has no way of determining the data value of a lost channel volume command. We refer to MIDI commands that are fragile in this way as unrecoverable MIDI commands.

The open-loop policy does not specify how to partition the MIDI command set into recoverable and unrecoverable commands. Instead, it assumes that the creators of the session descriptions are able to come to agreement on a suitable recoverable/unrecoverable MIDI command partition for an application.

Given these definitions, we now state the normative requirements for the open-loop policy.

In the open-loop policy, the creators of the session description MUST use the ch_unused or ch_anchor fmtp parameters (defined in Appendix C.1.3) to protect all unrecoverable MIDI command types from indefinite artifacts.

In a general sense, the ch_anchor parameter changes the recovery journal semantics to use the anchor checkpoint policy (Appendix C.1.2.1) for a command, and the ch_unused parameter acts to exclude a command type from the stream. These options act to shield command types from artifacts during an uncovered loss event.

In the open-loop policy, receivers MUST examine the Checkpoint Packet Seqnum field of the recovery journal header after every loss event, to check if the loss event is an uncovered loss event. Section 5 shows how to perform this check. If an uncovered loss event has occurred, a receiver MUST perform indefinite artifact recovery for all MIDI command types that are not shielded by ch_anchor and ch_unused parameter assignments in the session description.

The open-loop policy does not place specific constraints on the sender. However, the open-loop policy works best if the sender manages the size of the checkpoint history to ensure that uncovered losses occur infrequently, by taking into account the delay and loss characteristics of the network. Also, as each checkpoint packet change incurs the risk of an uncovered loss, senders should only move the checkpoint if it reduces the size of the journal.

C.1.3 Recovery Journal Chapter Inclusion Parameters

The recovery journal chapter definitions (Appendices A-B) specify under what conditions a chapter MUST appear in the recovery journal. In most cases, the definition states that if a certain command appears in the checkpoint history, a certain chapter type MUST appear in the recovery journal to protect the command.

In this section, we describe the chapter inclusion fmtp parameters. These parameters modify the conditions under which a chapter appears the journal.

These parameters are essential to the use of the open-loop policy (Appendix C.1.2.3), and may also be used to simplify multicast implementations of the closed-loop policy (Appendix C.1.2.2).

The parameters also serve to signal the types of MIDI commands that are not in use in a session. In this role, the parameters may be used with streams that do not use journalling.

Each parameter represents a type of chapter inclusion semantics. An assignment to a parameter declares which chapters (or chapter subsets) obey the inclusion semantics. We describe the assignment syntax for these parameters later in this section.

Below, we normatively define the semantics of the chapter inclusion parameters. For clarity, we define the action of parameters on complete chapters. If a parameter is assigned a subset of a chapter, the definition applies only to the chapter subset.

- ch_unused. If a chapter is assigned to the ch_unused parameter, the command types encoded by the chapter MUST NOT appear in

the MIDI command sections of stream packets. As a consequence, the chapter MUST NOT appear in the recovery journal.

In contrast with ch_unused, if a chapter is assigned to the parameters we define below, the command types encoded by the chapter MAY appear in the MIDI command section of stream packets.

- ch_never. A chapter assigned to the ch_never parameter MUST NOT appear in the recovery journal.

- ch_default. A chapter assigned to the ch_default parameter MUST follow the default semantics for the chapter, as defined in Appendices A-B.

- ch_anchor. A chapters assigned to the ch_anchor MUST obey a modified version of the default chapter semantics. In the modified semantics, all references to the checkpoint history are replaced with references to the session history, and all references to the checkpoint packet are replaced with references to the first packet sent in the stream.

Parameter assignments obey the following syntax (see Appendix D for ABNF):

<parameter> = [channel list]<chapter list>[field list];

The chapter list is mandatory; the channel and field lists are optional. Multiple assignments to parameters have a cumulative effect, and are applied in the order of parameter appearance in a media description.

The chapter list specifies the channel or system chapters for which the parameter applies. The chapter list is a concatenated sequence of one or more of the letters corresponding to the chapter types (ACDEFMNPQTVWX). In addition, the list may contain one or more of the letters for the sub-chapter types (BGHJKYZ) of System Chapter D. Assignments to sub-chapters of Chapter D override assignments to Chapter D. **The letters in a chapter list MUST be upper case, and MUST appear in** alphabetical order.

The channel list specifies the channel journals for which this parameter applies; if no channel list is provided, the parameter applies to all channel journals. The channel list takes the form of a list of channel numbers (0 through 15) and dash-separated channel number ranges (i.e. 0-5, 8-12, etc). Dots (i.e. "." characters) separate elements in the channel list.

A few system channels use special semantics for the channel list, which we now define.

For the J and K Chapter D sub-chapters (undefined System Common), the digit 0 codes that the parameter applies to the LEGAL field of the associated command log (Figure B.1.4 of Appendix B.1), the digit 1 codes that the parameter applies to the VALUE field of the command log, and the digit 2 codes that the parameter applies to the COUNT field of the command log.

For the Y and Z Chapter D sub-chapters (undefined System Real-time), the digit 0 codes that the parameter applies to the LEGAL field of the associated command log (Figure B.1.5 of Appendix B.1) and the digit 1 codes that the parameter applies to the COUNT field of the command log.

For Chapter Q (Sequencer State Commands), the digit 0 codes that the parameter applies to the default Chapter Q definition, which forbids the TIME field. The digit 1 codes that the parameter applies to the optional Chapter Q definition, which supports the TIME field.

For Chapter X (System Exclusive), the channel list specifies the types of System Exclusive commands to which the parameter applies. The digit **0 corresponds to Universal Real-Time commands (Manufacturer ID 0x7E),** the digit 1 corresponds to Universal Non-Real-Time (Manufacturer ID 0x7F), and the digit 2 corresponds to internal use (Manufacturer ID 0x7D). The digit 3 corresponds to real-time commands for all other Manufacturer ID numbers, and the digit 4 corresponds to non-real-time commands for all other Manufacturer ID numbers.

The syntax for field lists follows the syntax for channel lists. If no field list is provided, the parameter applies to all controller or note numbers.

For Chapters C and E, the field list codes the controller numbers for which the parameter applies.

For Chapter M, the field list consists of a single digit. The digit 0 codes that a log MUST appear for a parameter in Chapter M if a C-active command that forms part of an initiated transaction for the parameter appears in the checkpoint history, and that the A-COARSE, A-FINE, and A-BUTTON fields MUST NOT appear in parameter logs. The digit 1 codes that a log MUST appear for a parameter in Chapter M if an active (as opposed to C-active) command that forms part of an initiated transaction for the parameter appears in the checkpoint history.

For Chapters N and A, the field list codes the note numbers for which the parameter applies. For sub-chapters J and K of Chapter D, the field list consists of a single digit, which specifies the number of data octets that follow the command octet.

The example session description below illustrates the use of the chapter

inclusion parameters:

```
v=0
o=lazzaro 2520644554 2838152170 IN IP6 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP6 FF1E:03AD::7F2E:172A:1E24
a=rtpmap: 96 rtp-midi/44100
a=fmtp: 96 j_update=open-loop; ch_unused=ABDEFGHJKMQTVWXYZ;
a=fmtp: 96 ch_anchor=P; ch_anchor=C7.64;
a=fmtp: 96 ch_never=4.11-13N;
```

The j_update parameter codes that the stream uses the open-loop policy. Most chapters are assigned to ch_unused, a typical MIDI usage pattern of a low-bandwidth stream.

To guard against indefinite artifacts, the MIDI Program Change command and several MIDI Control Change controller numbers are assigned to ch_anchor. Note that the ordering of the ch_anchor chapter C assignment after the ch_unused command acts to override the ch_unused assignment for the listed controller numbers (7 and 64).

Chapter N for several MIDI channels is assigned to ch_never; in practice, this assignment pattern would reflect knowledge about a resilient rendering method in use for certain channels. In this example, Chapter N for MIDI channels other than 4, 11, 12, and 13 may appear in the recovery journal, per the default behavior.

C.2 SDP Definitions: Command Execution Semantics

The MIDI command section of the payload format consists of a list of commands, each with an associated timestamp. Section 3.1 defines the default semantics for command timestamps. These semantics work well for transcoding Standard MIDI Files (SMFs), but are problematic for transcoding MIDI sources (such as MIDI 1.0 DIN cables [1]) that use implicit "time-of-arrival" coding.

In this Appendix, we define session configuration tools for customizing the timestamp semantics of the MIDI command section.

The fmtp parameter "tsmode" specifies the timestamp semantics for a stream. The parameter takes on one of three token values: "comex", "async", or "buffer".

The "comex" value specifies the default semantics. The "async" value selects an asynchronous sampling algorithm for time-of-arrival sources

(Appendix C.2.1). The "buffer" value selects an alternative synchronous sampling algorithm (Appendix C.2.2).

Ancillary fmtp parameters may follow tsmode in a media description. One such parameter is "linerate". This parameter codes the timespan of one MIDI octet on the transmission medium of the MIDI source to be sampled (such as a MIDI 1.0 DIN cable). The parameter has units of nanoseconds, and takes on integral values. For MIDI 1.0 DIN cables, the correct linerate value is 320000 (this value is also the default value for the parameter). Other ancillary fmtp parameters are defined in Appendices C.2.1-2 below.

C.2.1 The async Algorithm

The "async" tsmode value specifies the asynchronous sampling of a MIDI time-of-arrival source. In asynchronous sampling, the moment an octet is received from a source it is labelled with a wall-clock time value. The time value has RTP timestamp units.

The "octpos" ancillary fmtp parameter defines how RTP command timestamps are derived from octet time values. If octpos has the token value "first", a timestamp codes the time value of the first octet of the command. If octpos has the token value "last", a timestamp codes the time value of the last octet of the command. If the octpos parameter does not appear in the media description, a timestamp MAY reflect the time value of any octet of the command.

The octpos semantics refer to the first or last octet of a command as it appears on a time-of-arrival source, not as it appears in the RTP packet. This distinction is significant for segmented SysEx commands. This distinction is also significant for sources that use running status coding, as the RTP encoding does not always preserve running status. The P header bit of the MIDI command section may be used to ascertain accurate command timing in this case (Section 3).

We now show a session description example for the async algorithm. Consider a sender that is transcoding a MIDI 1.0 DIN cable source into RTP. The sender runs on a computing platform that assigns time values to every incoming octet of the source, and the sender uses the time values to label the first octet of each command in the RTP packet. This session description describes the transcoding:

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
```

```
a=rtpmap: 96 rtp-midi/44100
a=fmtp: 96 tsmode=async;linerate=320000;octpos=first;
```

C.2.2 The buffer Algorithm

The "buffer" tsmode value specifies the synchronous sampling of a MIDI time-of-arrival source.

In synchronous sampling, octets received from a source are placed in a holding buffer upon arrival. At periodic intervals, the RTP sender examines the buffer. The sender removes complete commands from the buffer, and codes those commands in an RTP packet. The command timestamp reflects the actual moment of buffer examination, expressed in RTP timestamp units. Note that several commands may have the same timestamp value.

The "mperiod" ancillary fmtp parameter defines the nominal periodic sampling interval. The parameter takes on positive integral values, and has RTP timestamp units.

The "octpos" ancillary fmtp parameter, defined in Appendix C.2.1 for asynchronous sampling, plays a different role in synchronous sampling. In synchronous sampling, the parameter specifies the timestamp semantics of a command whose octets span several sampling periods.

If octpos has the token value "first", the timestamp reflects the arrival period of the first octet of the command. If octpos has the token value "last", the timestamp reflects the arrival period of the last octet of the command. If the octpos parameter does not appear in the media description, the timestamp MAY reflect the arrival period of any octet of the command. The octpos semantics refer to the first or last octet of the command as it appears on a time-of-arrival source, not as it appears in the RTP packet.

We now show a session description example for the buffer algorithm. Consider a sender that is transcoding a MIDI 1.0 DIN cable source into RTP. The sender runs on a computing platform that places source data into a buffer upon receipt. The sender polls the buffer 1000 times a second, extracts all complete commands from the buffer, and places the commands in an RTP packet. This session description describes the transcoding:

```
v=0
o=lazzaro 2520644554 2838152170 IN IP6 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP6 FF1E:03AD::7F2E:172A:1E24
```

```
a=rtpmap: 96 rtp-midi/44100
a=fmtp: 96 tsmode=buffer;linerate=320000;octpos=last;mperiod=44;
```

The mperiod value of 44 is derived by dividing the srate (44100 Hz) by the 1000 Hz buffer sampling rate, and rounding to the nearest integer. Command timestamps might not increment by exact multiples of 44, as the actual sampling period might not precisely match the nominal mperiod value.

C.3 SDP Definitions: Timing Tools

In this Appendix, we describe session configuration tools for customizing the temporal behavior of MIDI streams.

C.3.1 ptime and maxptime

Senders code the temporal nature of a stream by choosing the amount of media time encoded in each packet. Short media times (20 ms or less) often imply an interactive session. Longer media times (100 ms or more) usually indicate a content streaming session. The AVP profile permits audio packet media times to range from 0 to 200 ms.

An RTP receiver dynamically senses the media time of packets in a stream, and chooses the length of its playout buffer to match the stream. A receiver typically sizes its playout buffer to fit several audio packets, and adjusts the buffer length to reflect the network jitter and the sender timing fidelity.

Alternatively, the packet media time may be statically set during session configuration. The standard "ptime" attribute sets the typical packet media time for a session. The standard "maxptime" attribute sets the maximum packet media time for a session [6].

0 ms is a reasonable media time value for MIDI packets. In a packet with a 0 ms media time, all commands execute at the instant coded by the packet timestamp. Prohibitions in [15] against 0 ms ptime values are not relevant for MIDI streams, and may be ignored.

The session description example below defines a stream suitable for use in low-latency interactive applications.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
```

```
a=rtpmap: 96 rtp-midi/44100
a=ptime:0
a=maxptime:0
```

C.3.2 The guardtime Parameter

RTP/AVP permits a sender to stop sending audio packets for an arbitrary period of time during a session. When sending resumes, the RTP sequence number series continues unbroken, and the RTP timestamp value reflects the media time silence gap.

This RTP/AVP feature has its roots in telephony, but is also well matched to interactive MIDI sessions, as players may fall silent for several seconds during (or between) songs.

Certain MIDI applications benefit from a slight enhancement to this RTP/AVP feature. In interactive applications, receivers may use on-line network models to guide heuristics for handling lost and late RTP packets. These models may work poorly if a sender ceases packet transmission for long periods of time.

Session descriptions may use the fmtp parameter "guardtime" to set a minimum sending rate for a media session. The value assigned to guardtime codes the maximum separation time between two sequential packets, as expressed in RTP timestamp units. Typical guardtime values are 500-2000 ms.

Below, we show a session description that uses the guardtime parameter.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP6 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP6 FF1E:03AD::7F2E:172A:1E24
a=rtpmap: 96 rtp-midi/44100
a=ptime:0
a=maxptime:0
a=fmtp: 96 guardtime=44100;
```

C.3.3 MIDI Time Code Issues

RTP defines tools to synchronize the playout of multiple RTP media streams. Appendix C.4 shows how to use these tools in MIDI streams.

In content-creation applications, it may be necessary to synchronize stream playout with media that are not sent over RTP. For example,

analog video may be marked with SMPTE 12M timecode, and an application may need to synchronize MIDI playout the video using timecode.

The MIDI standard includes the MIDI Time Code (MTC) commands for SMPTE 12M timecode [1]. An application MAY use MTC to send timecode data (including offsets and user data) in the MIDI command stream for heterogeneous synchronization purposes.

C.4 SDP Definitions: Multiple Streams

Several MIDI streams may appear in a session description. By default, the MIDI name space (16 voice channels + systems) for each stream is unique, and the rendering for each stream proceeds independently. The audio outputs of the streams are presented in a synchronized fashion.

In this Appendix, we define two fmtp parameters for use in sessions with several streams. These parameters ("musicport" and "zerosync") add three features to RTP MIDI:

1. Several streams may target the same MIDI name space.

2. Several streams may be bundled to form a larger MIDI name space, that a single rendering system may treat as an ordered entity.

3. Streams may specify relative timebase offsets, to support synchronization with zero sync-lock delay.

In Appendices C.4.1-2, we define the musicport and zerosync parameters. In Appendix C.4.3, we show session description examples.

Other payload formats MAY define musicport and zerosync fmtp parameters. Formats would define these parameters so that their streams could be bundled into RTP MIDI name spaces. The parameter definitions MUST be compatible with the musicport and zerosync semantics defined in this Appendix.

C.4.1 The musicport Parameter

The musicport parameter codes an arbitrary identification number for the MIDI name space (16 voice channels + systems) of an RTP stream. The musicport parameter may take on integer values between 0 and 429496729.

If several MIDI streams in a session share the same musicport value, the streams target the same MIDI name space. We refer to this relationship as the identity relationship.

If several MIDI streams in a session have contiguous musicport values (i.e. i, i+1, ... i+k), the name spaces of the streams form an ordered entity. In this case, the streams in the entity are said to share an ordered relationship.

Note that a stream may participate in both an identity and an ordered relationship. For example, a stream in an identity relationship may have a musicport value that forms part of an ordered relationship. If the musicport values of two streams are not part of an ordered or identity relationship, the two streams are independent, and have independent MIDI name spaces.

RTP MIDI streams in an ordered or identity relationship MUST be all native streams or all mpeg4-generic streams. Thus, we refer to relationships as being native relationships or mpeg4-generic relationships.

For native relationships, at most one stream may specify MIDI renderers (using the tools described in C.5). Each MIDI rendering type may define its own semantics with regard to identity and ordered relationships.

For mpeg4-generic relationships, at most one stream in an identity or ordered relationship may have a config parameter value other than the empty string. In this case, the config value configures the stream. Alternatively, all config parameters may be set to the empty string. In this case, exactly one stream in the relationship MUST define the configuration using the tools described in Appendix C.5.

For both native and mpeg4-generic relationships, an exception to the "one stream defines the rendering" rule applies to relationships that exclusively contain sendonly and recvonly streams (as defined in [6]). In this case, a stream in each direction may define a renderer.

In an identity relationship, the sender partitions a MIDI name space (16 voice channels + systems) into several RTP streams. Receivers may process these streams independently, or may merge the streams to reconstitute the original MIDI command stream. We now specify receiver and sender responsibilities to ensure the robust transmission of identity relationships.

Receivers that merge identity relationship streams into a single MIDI command stream MUST maintain the structural integrity of the MIDI commands coded in each stream during the merging process, in the same way that software that merges traditional MIDI 1,0 DIN cable flows is responsible for creating a merged command flow compatible with [1].

Senders MUST partition the name space so that the rendered MIDI performance does not contain indefinite artifacts (as defined in Section

4). This responsibility holds even if all streams are sent over reliable transport, as imperfect synchronization of reliable streams may yield indefinite artifacts. For example, stuck notes may occur in a performance split over two TCP streams, if NoteOn commands are sent on one stream and NoteOff commands are sent on the other.

Senders MUST NOT split a Registered Parameter Name (RPN) or Non-Registered Parameter Name (NRPN) transaction appearing on a MIDI channel across multiple identity relationship streams. Receivers MUST assume that the RPN/NRPN transactions that appear on different identity relationship streams are independent, and MUST preserve transactional integrity during the MIDI merge.

A simple way to safely partition voice channel commands is to place all MIDI commands for a particular voice channel into the same stream. Safe partitions of systems commands may be more complex for streams that extensively use System Exclusive commands.

C.4.2 The zerosync Parameter

The RTP timestamp of the first packet in a stream is not set to zero. Instead, [2] mandates that the RTP timestamp is initialized to a randomly chosen value, to guard against plaintext attacks on encrypted streams. As a consequence, a receiver cannot directly use RTP timestamps to play back two RTP streams in sync.

The Real Time Control Protocol (RTCP), a low-bandwidth feedback channel that is paired with each RTP stream, provides synchronization services. Certain types of RTCP packets code the current time in two forms: the format of the RTP timestamp, and the 64-bit Network Time Protocol (NTP) format. A receiver may examine the NTP timestamps of several RTCP streams, and use this information to deduce the temporal relationship between the RTP streams associated with the RTCP streams. This method assumes that the NTP timestamps coded by all streams derive from a common clock source.

For many applications, this RTCP-based method is a good way to synchronize streams. In some applications, however, this method is not optimal, because of the synchronization time delay at the start of the session.

The zerosync parameter provides an alternative mechanism for stream synchronization. The zerosync parameter codes the RTP timestamp offsets for each stream, so that streams generated in a synchronized fashion may be played back in sync without using RTCP feedback.

The use of the zerosync parameter weakens the security of RTP, as discussed in Appendix G of this memo.

The zerosync parameter supports two synchronization mechanisms. One mechanism potentially synchronizes all streams within a given relationship. Media descriptions code this mechanism with a zerosync parameter whose value is in the range 1-429496729. We refer to this mechanism as the non-zero behavior.

A second mechanism potentially synchronizes all RTP MIDI streams in a session. Media descriptions code this mechanism with a zerosync parameter whose value is set to 0. We refer to this mechanism as the zero behavior.

A media description may contain, at most, one zerosync parameter assignment. Thus, a stream may participate in a non-zero behavior or a zero behavior, but not both. In both zero and non-zero behaviors, all media descriptions synchronized by the behavior MUST have identical srate values.

In a non-zero behavior, all streams within a relationship share an underlying timebase, but the randomly chosen initial timestamp value for each stream obscures this commonality. To unmask the similarity, each media description in the relationship MAY include a zerosync parameter whose non-zero value codes its initial timestamp value. In this scheme, the underlying timestamp for a packet is computed by subtracting (modulo 2^32) the zerosync value from the packet timestamp.

In a zero behavior, all affected streams share an underlying timebase AND the same initial timestamp value (in direct violation of [2]). Thus, the packet timestamps code the "true" timestamp directly.

C.4.3 Multi-stream examples using musicport and zerosync.

This section shows several session description examples that use the musicport and zerosync parameters.

Our first session description example shows two mpeg4-generic streams that drive the same General MIDI decoder.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 61
c=IN IP4 192.0.2.94
a=rtpmap: 61 mpeg4-generic/44100
a=fmtp: 61 streamtype=5; mode=rtp-midi; profile-level-id=12;
a=fmtp: 96 config=7A124D5468640000000060000000100604D547
                  26B0000000400FF2F000
a=fmtp: 61 musicport=12;zerosync=1726
```

```
 m=audio 5006 RTP/AVP 62
 c=IN IP4 192.0.2.94
 a=rtpmap: 62 mpeg4-generic/44100
 a=fmtp: 62 streamtype=5; mode=rtp-midi; config="";
 a=fmtp: 62 profile-level-id=12; musicport=12; zerosync=726;
```

(The linebreak in the second fmtp line accommodates memo formatting restrictions; SDP does not have continuation lines.)

The musicport values indicate the streams share an identity relationship, and the zerosync values code the non-zero behavior.

A variant on this example, whose session description is not shown, would use two streams in an identity relationship driving the same MIDI renderer, each with a different transport type. One stream would use UDP, and would be dedicated to real-time messages. A second stream would use TCP, and would be used for SysEx bulk data messages.

In the next example, two mpeg4-generic streams form an ordered relationship to drive a Structured Audio decoder with 32 MIDI voice channels.

```
 v=0
 o=lazzaro 2520644554 2838152170 IN IP6 first.example.net
 s=Example
 t=0 0
 m=audio 5004 RTP/AVP 61
 c=IN IP6 FF1E:03AD::7F2E:172A:1E24
 a=rtpmap: 61 mpeg4-generic/44100
 a=fmtp: 61 streamtype=5; mode=rtp-midi; config="";
 a=fmtp: 61 profile-level-id=13; musicport=5; zerosync=0;
 m=audio 5006 RTP/AVP 62
 c=IN IP6 FF1E:03AD::7F2E:172A:1E24
 a=rtpmap: 62 mpeg4-generic/44100
 a=fmtp: 62 streamtype=5; mode=rtp-midi; config=""; profile-level-id=13;
 a=fmtp: 62  profile-level-id=13; musicport=6; zerosync=0;
 a=fmtp: 62 render=synthetic; rinit="audio/asc";
 a=fmtp: 62 url="http://example.com/cardinal.asc";
 a=fmtp: 62 cid="azsldkaslkdjqpwojdkmsldkfpe";
```

The sequential musicport values for the two streams establishes the ordered relationship. The musicport=5 stream maps to Structured Audio extended channels range 0-15, the musicport=6 stream maps to Structured Audio extended channels range 16-31. The zerosync values code the zero behavior.

Both config strings are empty. The configuration data is specified in the final two fmtp lines of the second media description. We define

this configuration method in Appendix C.5.

C.5 SDP Definitions: MIDI Rendering

This Appendix defines the session configuration tools for rendering.

The "render" fmtp parameter specifies a rendering method for a stream. The inclusion of a render parameter in a media description acts to override the default rendering semantics (defined in Sections 6.1-2) for the stream.

The render parameter is assigned a token value that signals the top-level rendering type. This memo defines two token values for render: "synthetic" and "api". A "synthetic" renderer transforms the MIDI stream into audio output (or sometimes, into stage lighting changes or other actions). An "api" renderer presents the command stream to applications via an Application Programmer Interface (API).

Other fmtp parameters follow the render parameter in the media description, and define the exact nature of the renderer. The "rinit" fmtp parameter (defined in Appendix C.5.1) specifies the MIME subtype for the renderer, and the "inline", "url", and "cid" fmtp parameters (defined in Appendix C.5.2) specify renderer initialization data.

Other IETF standards-track documents MAY define additional token values for the render parameter. If a receiver is not aware of the token value assigned to a render parameter, the receiver MUST ignore the renderer the parameter defines.

A media description MAY contain several render parameters. This syntax requests synchronized rendering of the stream by each renderer, if possible. Renderers appear in a media description in order of decreasing priority. A receiver with limited resources SHOULD use the priority to decide which renderer(s) to retain in a session.

C.5.1 The rinit Parameter

The "rinit" fmtp parameter defines the nature of the renderer declared by the render parameter. Exactly one rinit parameter MUST follow the render parameter in a media description.

The value assigned to the rinit parameter MUST be a MIME type/subtype [8] that defines a renderer. Authors of rendering systems and MIDI APIs SHOULD register [20] a MIME subtype for use with RTP MIDI.

A renderer that directly produces audio output SHOULD be registered under the "audio" MIME type. API presentation renderers, and renderers

that control non-audio devices, SHOULD be registered under the "application" MIME type.

The subtype registration for a renderer MAY define a data object. For renderers that directly produce audio or control output, the data object usually codes initialization data for the rendering algorithm. The data object may also encapsulate an SMF, so that the data object may be used as a format for stored performances.

For API presentation renderers, the role of the data object varies. In some cases, the data object describes the hardware device that generates the stream (manufacturer, model, etc). In other cases, the data object follows the semantics of audio renderer data objects.

If a renderer MIME registration defines a data object, additional fmtp parameters MAY follow the rinit parameter to encode the object. We define these parameters in Appendix C.5.2.

By default, if a data object is encoded in an RTP MIDI media description, SMFs encapsulated in the data object MUST be ignored by the receiver. We define fmtp parameters to override this default in Appendix C.5.3.

Special rules apply to using the rinit parameter in an mpeg4-generic stream. We define these rules in Appendix C.5.4.

The rinit parameter MAY be assigned the "application/octet-stream" or "audio/octet-stream" values. These values code an opaque rendering type, whose rendering semantics and data object format has been defined outside the scope of this memo.

C.5.2 Encoding rinit Data Objects

The "inline", "url", and "cid" fmtp parameters MAY follow the rinit parameter in a media description. These parameters encode the initialization data object for the renderer.

The "inline" parameter supports the inline encoding of the data object. The parameter is assigned a double-quoted Base64 [8] encoding of the binary data object, with no line breaks.

The "url" parameter is assigned a double-quoted string representation of a Uniform Resource Locator (URL) for the data object. If the URL points to a MIME object, the object MUST have the MIME type/subtype value coded by the rinit parameter.

The "cid" parameter supports data object caching, and MAY follow the url parameter in the media description. The parameter is assigned a double-

quoted string value that encodes a globally unique identifier for the data object. If the url string points to a MIME object, the cid string MUST match the Content-ID header [8] value of the object.

In most cases, one inline parameter or one url/cid parameter pair follows the rinit parameter in the media description. The correct receiver interpretation of multiple data objects SHOULD be defined in the renderer MIME registration.

C.5.3 MIDI Channel Mapping

In this Appendix, we specify how to map MIDI name spaces (16 voice channels + systems) onto a renderer.

In the general case:

- A session may define an ordered relationship (Appendix C.4) that presents more than one MIDI name space to a renderer.

- A renderer may accept an arbitrary number of MIDI name spaces, or may expect a fixed number of MIDI name spaces.

A session description SHOULD define mappings of streams to renderers that are name-space compatible. If a receiver detects a name-space mismatch in a session description, extra stream name spaces MUST be discarded, and extra renderer name spaces MUST NOT be driven with MIDI data.

If a media description defines several renderers, each renderer processes the presented name space(s) in parallel. However, the "chanmask" fmtp parameter may be used to mask out selected voice channels to each renderer. We define "chanmask" and other channel management fmtp parameters in the sub-sections below.

C.5.3.1 The smf_info fmtp Parameter

The smf_info parameter MAY appear after the rinit parameter (Appendix C.5.1) in a media description. The parameter defines the use of all SMFs encapsulated in renderer data objects.

We define token values for smf_info: "sdp_start" and "ignore". The "sdp_start" value codes that SMF rendering MUST begin upon the acceptance of the session description. The "ignore" value codes that SMF files MUST be discarded (the default behavior). Below, we define the semantics for the "sdp_start" token value.

SMFs share the MIDI name spaces of the RTP streams. SMF commands and RTP stream commands are merged and presented to the renderer. The

indefinite artifact responsibilities for merged MIDI streams defined in Appendix C.4.1 **also apply to merging RTP streams and SMFs.**

If the data object encapsulates multiple SMFs, the SMF name spaces are presented as an ordered entity to the renderer. The first encapsulated SMF in the data object maps to the first renderer name space, the second encapsulated SMF maps to the second renderer name space, etc. If the associated RTP streams form an ordered relationship, the first SMF is merged with the first name space of the relationship, the second SMF is merged to the second name space of the relationship, etc.

Unless the streams and the SMFs both use MIDI Time Code, the time offset between SMF and stream data is unspecified. This restriction may limit the use of SMFs to applications where synchronization is not critical, such as the transport of System Exclusive commands for renderer initialization, or human-SMF interactivity.

C.5.3.2 The smf_inline, smf_url, and smf_cid fmtp Parameters

In some applications, the renderer data object may not encapsulate SMFs, but an application may wish to use SMFs in the manner defined in Appendix C.5.3.1.

The "smf_inline", "smf_url", and "smf_cid" fmtp parameters address this situation. These parameters use the syntax and semantics of the inline, url, and cif parameters defined in C.5.2, except that the encoded data object is an SMF.

If several "smf_inline" or "smf_url" parameters appear in a media description, the order of the parameter defines the SMF name space ordering.

If smf_url points to a MIME object, the "application/octet-stream" type/subtype SHOULD be used for the object.

C.5.3.3 The chanmask fmtp Parameter

The chanmask fmtp parameter instructs the renderer to ignore all MIDI voice commands for certain channel numbers. The parameter value is an concatenated string of "1" and "0" digits. Each string position maps to a MIDI voice channel number (system channels may not be masked). A "1" instructs the renderer to process the voice channel; a "0" instructs the renderer to ignore the voice channel.

The string length of the chanmask parameter value MUST be 16 (for a single stream or an identity relationship) or a multiple of 16 (for an ordered relationship).

The chanmask parameter appears after the render parameter, and describes the final MIDI name spaces presented to the renderer. The SMF and stream components of the MIDI name spaces may not be independently masked.

C.5.4 The audio/asc MIME Type

In Appendix H.3, we register the audio/asc MIME type. The data object for audio/asc is a binary encoding of the AudioSpecificConfig data used to configure mpeg4-generic streams (Section 6.2 and [7]).

An mpeg4-generic media description MAY use audio/asc for renderer configuration. Several restrictions apply to the use of the render parameter with mpeg4-generic streams:

- An mpeg4-generic media description that uses the render parameter MUST assign the empty string ("") to the mpeg4-generic "config" parameter.
- The render parameter MUST be assigned the value "synthetic". Other token values for render MUST NOT appear in an mpeg4-generic media description.
- The rinit parameter MUST be assigned the value "audio/asc". Other token values for rinit MUST NOT appear in an mpeg4-generic media description.
- The streamtype, mode, and profile-level-id parameters MUST be used as defined in Section 6.2, and the AudioSpecificConfig data MUST encode one of the MPEG 4 Audio Object Types defined for use with mpeg4-generic in Section 6.2.

In addition, several restrictions apply to the use of the audio/asc MIME type in RTP MIDI.

- A native stream MUST NOT assign the "audio/asc" value to rinit.
- The audio/asc MIME type defines a stored object type; it does not define semantics for RTP streams. Thus, audio/asc MUST NOT appear on an rtpmap line of a session description.

Below, we show session description examples for audio/asc. The session description below uses the inline parameter to code the AudioSpecificConfig block for a mpeg4-generic General MIDI stream.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
s=Example
```

```
 t=0 0
 m=audio 5004 RTP/AVP 61
 c=IN IP4 192.0.2.94
 a=rtpmap: 61 mpeg4-generic/44100
 a=fmtp: 61 streamtype=5; mode=rtp-midi;
 a=fmtp: 61 config=""; profile-level-id=12; render=synthetic;
 a=fmtp: 61 rinit="audio/asc";
 a=fmtp: 61 inline="ehJNVGhkAAAABgAAAAEAYE1UcmsAAAAEAP8vAAA="
```

The session description below uses the url fmtp parameter to code the AudioSpecificConfig block for the same General MIDI stream:

```
 v=0
 o=lazzaro 2520644554 2838152170 IN IP4 first.example.net
 s=Example
 t=0 0
 m=audio 5004 RTP/AVP 61
 c=IN IP4 192.0.2.94
 a=rtpmap: 61 mpeg4-generic/44100
 a=fmtp: 61 streamtype=5; mode=rtp-midi;
 a=fmtp: 61 config=""; profile-level-id=12;
 a=fmtp: 61 render=synthetic; rinit="audio/asc";
 a=fmtp: 61 url="http://example.net/oski.asc";
 a=fmtp: 61 cid="xjflsoeiurvpa09itnvlduihgnvet98pa3w9utnuighbuk";
```

D. Parameter Syntax Definitions

In this Appendix, we define the syntax for the RTP MIDI fmtp parameters in Augmented Backus-Naur Form (ABNF, [10]). Parameters appear in the fmtp lines of session descriptions for native or mpeg4-generic streams. A fmtp line may be defined as:

```
;
; SDP fmtp line definition
;

fmtp = "a=fmtp:" token 1*(param-assign ";") CRLF
```

where <token> codes the RTP payload type. Below, we define <param-assign> as a set of incremental rules for the custom parameters defined in Appendix C.

```
;
;
; top-level definition for all parameters
```

```
;
;

;
; Parameters defined in Appendix C.1

param-assign  = "j_sec"      "=" ("none" / "recj" / *ietf-extension)

param-assign /= "j_update"   "=" ("anchor" / "closed-loop" / "open-loop"
                                  / *ietf-extension)

param-assign /= "ch_default" "=" ([channel-list] chapter-list [f-list])

param-assign /= "ch_unused"  "=" ([channel-list] chapter-list [f-list])

param-assign /= "ch_never"   "=" ([channel-list] chapter-list [f-list])

param-assign /= "ch_anchor"  "=" ([channel-list] chapter-list [f-list])

;
; Parameters defined in Appendix C.2

param-assign /= "tsmode"     "=" ("comex" / "async" / "buffer")

param-assign /= "linerate"   "=" nonzero-four-octet

param-assign /= "octpos"     "=" ("first" / "last")

param-assign /= "mperiod"    "=" nonzero-four-octet

;
; Parameter defined in Appendix C.3

param-assign /= "guardtime"  "=" nonzero-four-octet

;
; Parameters defined in Appendix C.4

param-assign /= "musicport"  "=" four-octet

param-assign /= "zerosync"   "=" four-octet

;
; Parameters defined in Appendix C.5

param-assign /= "chanmask"   "=" 1*( 16( "0" / "1" ) )

param-assign /= "cid"        "=" double-quote cid-block double-quote
```

```
param-assign /= "inline"     "=" double-quote base-64-block double-quote

param-assign /= "render"     "=" ("synthetic" / "api" / *ietf-extension)

param-assign /= "rinit"      "=" mime-type "/" mime-subtype

param-assign /= "smf_cid"    "=" double-quote cid-block double-quote

param-assign /= "smf_info"   "=" ("ignore" / "sdp_start" )

param-assign /= "smf_inline" "=" double-quote base-64-block double-quote

param-assign /= "smf_url"    "=" double-quote uri-element double-quote

param-assign /= "url"        "=" double-quote uri-element double-quote

;
; list definitions for the ch_ chapter-list
;

chapter-list       = chapter-part1 chapter-part2 chapter-part3

chapter-part1      = 0*1"A" 0*1"B" 0*1"C" 0*1"D" 0*1"E" 0*1"F" 0*1"G"

chapter-part2      = 0*1"H" 0*1"J" 0*1"K" 0*1"M" 0*1"N" 0*1"P" 0*1"Q"

chapter-part3      = 0*1"T" 0*1"V" 0*1"W" 0*1"X" 0*1"Y" 0*1"Z"

;
; list definitions for the ch_ channel-list
;

channel-list       = midi-chan-element *("." midi-chan-element)

midi-chan-element  = midi-chan / midi-chan-range

midi-chan-range    = midi-chan "-" midi-chan

                   ; decimal value of left midi-chan
                   ; MUST be strictly less than decimal
                   ; value of right midi-chan

midi-chan          = %d0-15

;
; list definitions for the ch_ field list (f-list)
;
```

```
f-list             = midi-field-element *("." midi-field-element)

midi-field-element = midi-field / midi-field-range

midi-field-range   = midi-field "-" midi-field
                   ;
                   ; decimal value of left midi-field
                   ; MUST be strictly less than decimal
                   ; value of right midi-field

midi-field         = %d0-127

;
; definitions for rinit fmtp parameter
;

mime-type          = type
                   ;
                   ; as defined on page 12 in [8]

mime-subtype       = subtype
                   ;
                   ; as defined on page 12 in [8]

;
; generic rules
;

ietf-extension     = token
                   ;
                   ; token as defined in reference [6].
                   ; ietf-extension may only be defined in
                   ; standards-track RFCs (Section 7).

four-octet         = %d0-429496729
                   ; unsigned encoding of 32-bits

nonzero-four-octet = %d1-429496729
                   ; unsigned encoding of 32-bits, ex-zero

uri-element        = uri
                   ; as defined in reference [6].

base-64-block      = base64
                   ; as defined in reference [6].

double-quote       = %x22
                   ; the double-quote (") character
```

```
cid-block          = msg-id
                   ; as discussed in Section 7 of
                   ; reference [8]

;
; End of ABNF
```

The mpeg4-generic RTP payload [4] defines a "mode" parameter that signals the type of MPEG stream in use. We add a new mode value, "rtp-midi", using the ABNF rule below:

```
;
; mpeg4-generic mode parameter extension
;

mode               /= "rtp-midi"
                   ; as described in Section 6.2 of this memo
```

E. A MIDI Overview for Networking Specialists

This Appendix presents an overview of the MIDI standard, for the benefit of networking specialists new to musical applications. Implementors should consult [1] for a normative description of MIDI.

Musicians make music by performing a controlled sequence of physical movements. For example, a pianist plays by coordinating a series of key presses, key releases, and pedal actions. MIDI represents a musical performance by encoding these physical gestures as a sequence of MIDI commands. This high-level musical representation is compact but fragile: one lost command may be catastrophic to the performance.

MIDI commands have much in common with the machine instructions of a microprocessor. MIDI commands are defined as binary elements. Bitfields within a MIDI command have a regular structure and a specialized purpose. For example, the upper nibble of the first command octet (the opcode field) codes the command type. MIDI commands may consist of an arbitrary number of complete octets, but most MIDI commands are 1, 2, or 3 octets in length.

Name	Bitfield Pattern
NoteOff (end a note)	1000cccc 0nnnnnnn 0vvvvvvv
NoteOn (start a note)	1001cccc 0nnnnnnn 0vvvvvvv
PTouch (Polyphonic Aftertouch)	1010cccc 0nnnnnnn 0aaaaaaa
CControl (Controller Change)	1011cccc 0xxxxxxx 0yyyyyyy
PChange (Program Change)	1100cccc 0ppppppp
CTouch (Channel Aftertouch)	1101cccc 0aaaaaaa
PWheel (Pitch Wheel)	1110cccc 0xxxxxxx 0yyyyyyy
System (sub-opcode is xxxx)	1111xxxx ...

Figure E.1 -- MIDI command chart

Figure E.1 shows the MIDI command family. There are two major classes of commands: voice commands (opcode field values in the range 0x8 through 0xE) and system commands (opcode field value 0xF). Voice commands code the musical gestures for each timbre in a composition. Systems commands perform housekeeping functions, such as System Reset (the one-octet command 0xFF).

E.1 Commands Types

Voice commands execute on one of 16 MIDI channels, as coded by its 4-bit channel field (field cccc in Figure E.1). In most applications, notes for different timbres are assigned to different channels. To support applications that require more than 16 channels, MIDI systems use several MIDI command streams in parallel, to yield 32, 48, or 64 MIDI channels.

As an example of a voice command, consider a NoteOn command (opcode 0x9), with binary encoding 1001cccc 0nnnnnnn 0aaaaaaa. This command signals the start of a musical note on MIDI channel cccc. The note has a pitch coded by the note number nnnnnnn, and an onset amplitude coded by note velocity aaaaaaa.

Other voice commands signal the end of notes (NoteOff, opcode 0x8), map a specific timbre to a MIDI channel (PChange, opcode 0xC), or set the value of parameters that modulate the timbral quality (all other voice

commands). The exact meaning of most voice channel commands depends on the rendering algorithms the MIDI receiver uses to generate sound. In most applications, a MIDI sender has a model (in some sense) of the rendering method used by the receiver.

E.2 Running Status

All MIDI command bitfields share a special structure: the leading bit of the first octet is set to 1, and the leading bit of all subsequent octets is set to 0. This structure supports a data compression system, called running status [1], that improves the coding efficiency of MIDI.

In running status coding, the first octet of a MIDI voice command may be dropped if it is identical to the first octet of the previous MIDI voice command. This rule, in combination with a convention to consider NoteOn commands with a null third octet as NoteOff commands, supports the coding of note sequences using two octets per command.

E.3 Command Timing

The bitfield formats in Figure E.1 do not encode the execution time for a command. Timing information is not a part of the MIDI command syntax itself; different applications of the MIDI command language use different methods to encode timing.

For example, the MIDI command set acts as the transport layer for MIDI 1.0 DIN cables [1]. MIDI cables are short asynchronous serial lines that facilitate the remote operation of musical instruments and audio equipment. Timestamps are not sent over a MIDI 1.0 DIN cable. Instead, the standard uses an implicit "time of arrival" code. Receivers execute MIDI commands at the moment of arrival.

In contrast, Standard MIDI Files (SMFs, [1]), a file format for representing complete musical performances, add a explicit timestamp to each MIDI command, using a delta encoding scheme that is optimized for statistics of musical performance. SMF timestamps usually code timing using the metric notation of a musical score. SMF meta-events are used to add a tempo map to the file, so that score beats may be accurately converted into units of seconds during rendering.

F. Acknowledgements

We thank the networking, media compression, and computer music community members who have commented or contributed to the effort, including Steve Casner, Paul Davis, Robin Davies, Joanne Dow, Dominique Fober, Adrian Freed, Philippe Gentric, Chris Grigg, Todd Hager, Michel Jullian, Phil Kerr, Young-Kwon Lim, Jessica Little, Jan van der Meer, Colin Perkins,

Charlie Richmond, Herbie Robinson, Larry Rowe, Dave Singer, Martijn Sipkema, Kent Terry, David Wessel, Magnus Westerlund, Tom White, Matt Wright, Jim Wright, and Giorgio Zoia.

G. Security Considerations

Authentication of incoming RTP and RTCP packets is RECOMMENDED. Without such protections, attackers could forge MIDI commands into an ongoing stream, damaging speakers and eardrums. An attacker could also craft RTP and RTCP packets to exploit known bugs in the client, and take effective control of a client machine.

Session management tools SHOULD use authentication on all session descriptions. Session descriptions may code initialization data inline, using the inline (Appendix C.5.2) and smf_inline (Appendix C.5.3.2) fmtp parameters. If an attacker inserts bogus initialization data into a session description, he can corrupt the session or forge an client attack.

Session descriptions may code renderer initialization data by reference, via the url (Appendix C.5.2) and smf_url (Appendix C.5.3.2) parameters. If the coded URL is spoofed, both session and client are open to attack.

The zerosync fmtp parameter (described in Appendix C.4.2) impairs a security feature of RTP. In standard RTP, the RTP timestamp is initialized to a randomly chosen value, to reduce the predictability of the header. If zerosync is used in a media description, this security feature is partially (for non-zero zerosync values) or totally (if zerosync is set to zero) disabled.

H. IANA Considerations

In this Appendix, we register the audio/rtp-midi and audio/asc MIME types, and we extend the audio/mpeg4-generic MIME type [4]. The audio/rtp-midi and audio/asc registrations are in the IETF tree.

H.1 rtp-midi MIME Registration

This section registers rtp-midi as a MIME subtype for the audio type.

MIME media type name:

audio

MIME subtype name:

rtp-midi

Required parameters:

rate: The RTP timestamp clock rate, as specified in the rtpmap line. See Sections 2.1 and 6.1 for usage details.

Optional parameters:

Standard SDP attributes:

maxptime:	See Appendix C.3 for usage details.
ptime:	See Appendix C.3 for usage details.

Non-extensible parameters:

ch_anchor:	See Appendix C.1 for usage details.
ch_default:	See Appendix C.1 for usage details.
ch_never:	See Appendix C.1 for usage details.
ch_unused:	See Appendix C.1 for usage details.
chanmask:	See Appendix C.5 for usage details.
cid:	See Appendix C.5 for usage details.
guardtime:	See Appendix C.3 for usage details.
inline:	See Appendix C.5 for usage details.
linerate:	See Appendix C.2 for usage details.
musicport:	See Appendix C.4 for usage details.
mperiod:	See Appendix C.2 for usage details.
octpos:	See Appendix C.2 for usage details.
rinit:	See Appendix C.5 for usage details.
tsmode:	See Appendix C.2 for usage details.
smf_cid:	See Appendix C.5 for usage details.
smf_info:	See Appendix C.5 for usage details.
smf_inline:	See Appendix C.5 for usage details.
smf_url:	See Appendix C.5 for usage details.
url:	See Appendix C.5 for usage details.
zerosync:	See Appendix C.4 for usage details.

Extensible parameters:

j_sec, j_update:

See Appendix C.1 for usage details. The parameters may only be extended via an IETF standards-track document.

render:

See Appendix C.5 for usage details. The parameter may only be extended via an IETF standards-track document.

Encoding considerations:

This type is only defined for real-time transfers of MIDI streams via RTP. Stored-file semantics for rtp-midi may be defined in the future.

Security considerations:

See Appendix G of this memo.

Interoperability considerations:

None.

Published specification:

This memo and [1] serve as the normative specification. In addition, references [12], [13], and [18] provide non-normative implementation guidance.

Applications which use this media type:

Audio content-creation hardware, such as MIDI controller piano keyboards and MIDI audio synthesizers. Audio content-creation software, such as music sequencers, digital audio workstations, and soft synthesizers. Computer operating systems, for network support of MIDI Application Programmer Interfaces. Content distribution servers and terminals may use this media type for low bit-rate music coding.

Additional information:

None.

Person & email address to contact for further information:

John Lazzaro <lazzaro@cs.berkeley.edu>

Intended usage:

COMMON.

Author/Change controller:

John Lazzaro <lazzaro@cs.berkeley.edu>

H.2 mpeg4-generic MIME Registration

The mpeg4-generic MIME type [4] permits extensions to support new modes. The registration below defines mode rtp-midi for mpeg4-generic, to support the MPEG Audio codecs [5] that use MIDI.

MIME media type name:

audio

MIME subtype name:

mpeg4-generic

Required parameter extensions:

We extend the mpeg4-generic required parameter mode, by adding the value=parameter syntax:

mode=rtp-midi

to the list of legal mode values defined in [4]. See Section 6.2 for usage details.

rate: In mode rtp-midi, rate is a required parameter. Rate specifies the RTP timestamp clock rate on the rtpmap line. See Sections 2.1 and 6.2 for usage details.

Optional parameters:

Standard SDP attributes:

maxptime: See Appendix C.3 for usage details.
ptime: See Appendix C.3 for usage details.

Non-extensible parameters:

ch_anchor: See Appendix C.1 for usage details.
ch_default: See Appendix C.1 for usage details.
ch_never: See Appendix C.1 for usage details.
ch_unused: See Appendix C.1 for usage details.
chanmask: See Appendix C.5 for usage details.
cid: See Appendix C.5 for usage details.
guardtime: See Appendix C.3 for usage details.
inline: See Appendix C.5 for usage details.
linerate: See Appendix C.2 for usage details.
musicport: See Appendix C.4 for usage details.
mperiod: See Appendix C.2 for usage details.
octpos: See Appendix C.2 for usage details.
rinit: See Appendix C.5 for usage details.
tsmode: See Appendix C.2 for usage details.
smf_cid: See Appendix C.5 for usage details.
smf_info: See Appendix C.5 for usage details.
smf_inline: See Appendix C.5 for usage details.
smf_url: See Appendix C.5 for usage details.
url: See Appendix C.5 for usage details.
zerosync: See Appendix C.4 for usage details.

Extensible parameters:

j_sec, j_update:

See Appendix C.1 for usage details. The parameters may only be extended via an IETF standards-track document.

render:

See Appendix C.5 for usage details. The parameter may only be extended via an IETF standards-track document.

Encoding considerations:

Only defined for real-time transfers of audio/mpeg4-generic RTP streams with mode=rtp-midi.

Security considerations:

See Appendix G of this memo.

Interoperability considerations:

Except for the marker bit (Section 2.1), the packet formats for audio/rtp-midi and audio/mpeg4-generic (mode rtp-midi) are identical. The formats differ in use: audio/mpeg4-generic is for MPEG work, audio/rtp-midi is for all other work.

Published specification:

This memo, [1], and [5] are the normative references. In addition, references [12], [13], and [18] provide non-normative implementation guidance.

Applications which use this media type:

MPEG 4 servers and terminals that support [5].

Additional information:

None.

Person & email address to contact for further information:

John Lazzaro <lazzaro@cs.berkeley.edu>

Intended usage:

COMMON.

Author/Change controller:

John Lazzaro <lazzaro@cs.berkeley.edu>

H.3 asc MIME Registration

This section registers asc as a MIME subtype for the audio type.

MIME media type name:

audio

MIME subtype name:

asc

Required parameters:

none

Optional parameters:

none

Encoding considerations:

This type is only defined for data object (stored file) transfer. The native form of the data object is binary data padded to an octet boundary. The most common transports for the type are HTTP and SMTP.

Security considerations:

See Appendix G of this memo.

Interoperability considerations:

None.

Published specification:

The audio/asc data object is the AudioSpecificConfig binary data structure, which is normatively defined in [7].

Applications which use this media type:

MPEG 4 Audio servers and terminals which support audio/mpeg4-generic RTP streams for mode rtp-midi.

Additional information:

None.

Person & email address to contact for further information:

John Lazzaro <lazzaro@cs.berkeley.edu>

Intended usage:

COMMON.

Author/Change controller:

John Lazzaro <lazzaro@cs.berkeley.edu>

I. References

I.1 Normative References

[1] MIDI Manufacturers Association. "The Complete MIDI 1.0 Detailed Specification", 1996.

[2] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson. "RTP: A transport protocol for real-time applications", work in progress, draft-ietf-avt-rtp-new-12.txt.

[3] Schulzrinne, H., and S. Casner. "RTP Profile for Audio and Video Conferences with Minimal Control", work in progress, draft-ietf-avt-profile-new-13.txt.

[4] van der Meer, J., Mackie, D., Swaminathan, V., Singer, D., and P. Gentric. "RTP Payload Format for Transport of MPEG-4 Elementary Streams", work in progress, draft-ietf-avt-mpeg4-simple-07.txt.

[5] International Standards Organization. "ISO/IEC 14496 MPEG-4", Part 3 (Audio), Subpart 5 (Structured Audio), 2001.

[6] Handley, M., Jacobson, V., and C. Perkins. "SDP: Session Description Protocol", work in progress, draft-ietf-mmusic-sdp-new-12.txt.

[7] International Standards Organization. "ISO 14496 MPEG-4", Part 3 (Audio), 2001.

[8] Freed, N. and N. Borenstein. "MIME Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[9] MIDI Manufacturers Association. "The MIDI Downloadable Sounds Specification", v98.2, 1998.

[10] Crocker, D. and P. Overell. "Augmented BNF for Syntax Specifications: ABNF.", RFC 2234, November 1997.

[11] Bradner, S. "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

I.2 Informative References

[12] Lazzaro, J. and J. Wawrzynek. "A Case for Network Musical Performance", 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001) June 25-26, 2001, Port Jefferson, New York.

[13] Fober, D., Orlarey, Y. and S. Letz. "Real Time Musical Events Streaming over Internet", Proceedings of the International Conference on WEB Delivering of Music 2001, pages 147-154.

[14] Rosenberg, J, Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler. "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[15] J. Rosenberg and H. Schulzrinne. "An Offer/Answer Model with SDP", RFC 3264, June 2002.

[16] Schulzrinne, H., Rao, A., and R. Lanphier. "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998.

[17] Clark, D. D. and D. L. Tennenhouse. "Architectural considerations for a new generation of protocols", SIGCOMM Symposium on Communications Architectures and Protocols , (Philadelphia, Pennsylvania), pp. 200--208, IEEE, Sept. 1990.

[18] Lazzaro, J., and J. Wawrzynek. "An Implementation Guide for RTP MIDI", work in progress,

draft-lazzaro-avt-mwpp-coding-guidelines-02.txt.

[19] Braden, R. et al. "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification", RFC 2205, September 1997.

[20] Freed, N., Klensin, J., and J. Postel. "MIME Part Four: Registration Procedures", RFC 2048, November 1996.

J. Author Addresses

John Lazzaro (corresponding author)
UC Berkeley
CS Division
315 Soda Hall
Berkeley CA 94720-1776
Email: lazzaro@cs.berkeley.edu

John Wawrzynek
UC Berkeley
CS Division
631 Soda Hall
Berkeley CA 94720-1776
Email: johnw@cs.berkeley.edu

K. Intellectual Property Rights Statement

L. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

M. Change Log for <draft-ietf-avt-mwpp-midi-rtp-08.txt>

[Note to RFC Editors: this Appendix, and its Table of Contents listing, should be removed from the final version of the memo]

Chapter M (Appendix A.9) has been redesigned, to follow the semantic design of Chapters C and E. Several definitions in Appendix A.1 have been changed to reflect this change, as have the chapter inclusion semantics for Chapter M in Appendix C.1.3.

Many small editorial changes throughout the document, to correct grammatical errors and improve phrasing.