MIDI 1.5, So to Speak

Sean Luke[1] and **Luca Ludovico**[2]

[1]*Department of Computer Science*, **George Mason University**, Washington, DC, USA
[2]*Department of Computer Science*, **Università degli Studi di Milano**, Milano, Italy

ABSTRACT

In this position paper we argue that there may yet be more to squeeze out of MIDI 1.0 before adopting MIDI 2.0 wholesale. As MIDI 2.0 has been slow to roll out, it is worthwhile considering what kinds of backward-compatible improvements could be made to MIDI 1.0 as a stopgap to address some of the issues which drove MIDI 2.0 in the first place. In this paper we provide examples of six major improvements that could be made, as a proof of concept and strawman argument.

1. INTRODUCTION

In 1981 Ikutaro Kakehashi contacted Tom Oberheim, suggesting an international communication standard for music synthesizers, and Oberheim then recommended that Dave Smith be included in the informal discussion group. Smith's resulting proposal, called the Universal Synthesizer Interface, or USI, was first discussed at the Gakki Fair in 1981, and then after feedback from Japanese manufacturers it was presented at the following AES Conference [1]. After a few more iterations, Smith renamed the protocol MIDI and presented it at the NAMM show in 1982. Initial resistance from manufacturers was eventually won over, and MIDI became the de facto standard for control of electronic musical instruments [2].

That was over 40 years ago. MIDI has been very successful, and its weaknesses are also now well understood. In 2020 the MIDI Manufacturer's Association (MMA) formally adopted MIDI 2.0, a collection of six documents totaling 349 pages representing a major collaborative effort to update MIDI 1.0 for the modern age [3]. MIDI 2.0 tries to address most of MIDI 1.0's major problems and to provide room to grow.

MIDI 2.0 has been slow to roll out. Operating systems have only recently introduced MIDI 2.0 drivers despite likely having had significant runway prior to the 2020 announcement. Our private discussion with some major hardware manufacturers has not revealed enthusiasm for the protocol. Relatively few products have to date adopted MIDI 2.0, and almost no hardware devices have done so. This has been blamed on industry slowness due to the COVID pause, or the time needed to build drivers or software, and this may in fact be so. We have been assured by a member of the MIDI 2.0 committee that 2024 will be 2.0's *Annus Mirabilis*, and in truth we very much hope that it is!

However, it also may not be. We suspect that the chief cause of MIDI 2.0's slow adoption to date is simply that it is large and complex. And so we have explored what might be done in the interim. If MIDI 2.0 requires further revisions, what might be used as a stopgap measure? If small hardware manufacturers or users reject it for its complexity, what might be provided instead, to smooth the ramp until adoption?

Thus this position paper poses the question: **what would a MIDI 1.5 look like?** Are there improvements that we can *still* add to MIDI 1.0 which would be backward compatible and yet provide some of the features long sought after by MIDI 1.0 critics?

Though what follows might look like a specification proposal, it is in fact simply a strawman exercise, meant to show that many improvements *could* be made to MIDI 1.0 without jumping to the full features, and complexity, of MIDI 2.0, and to suggest that it might be worthwhile considering a stopgap measure if MIDI 2.0 needs time to live up to its promise.

1.1 Previous Work

1.1.1 Previous extensions and alternatives

While MIDI 1.0 never changed its version number prior to 2.0, it has had a great many revisions, and presently stands at Version 4.2 [4]. There have been many proposed extensions to, transports for, and alternatives to MIDI 1.0 for notation, song organization, performance, and music analysis: we refer the reader to *Beyond MIDI* [5] for a good summary of many proposals up to 1997 alone. Probably the most important alternative to MIDI 1.0 as a communications protocol has been *Open Sound Control* (or *OSC*) [6], which defines a full graph topology much like MIDI 2.0, albeit with an emphasis on portable and human-readable addressing and communication.

1.1.2 Previous Use of Reserved MIDI Status Bytes

Some of the solutions we propose use reserved MIDI status bytes. There were no doubt many previous proposals to use the four reserved status bytes, though perhaps not in response to MIDI 2.0! Here are two, both concerning the System Real-time message `F9`.

In 2016 Michael Lauter published an online whitepaper entitled "The case for MIDI real-time MARK message F9h" [7]. In it he argued that `F9` could be used to mark the

start of a new measure or musical phrase to make it easier to sync disparate devices.

The second is more curious. F9 has been incorrectly used as a *MIDI Tick* message to be emitted every 10 milliseconds to allow for a simple form of absolute time sync. This nonstandard usage has spread to various applications due to its inclusion in the standard Arduino MIDI library [8]. It is not clear what the origin of this usage was, but the earliest reference we can find appears to be from a MIDI fan website [9].

We have been informed that the MIDI 2.0 Committee also considered using status bytes early on, among many options, but rejected them. Among the chief reasons for rejection was the discovery that there existed hardware interfaces in the field, or operating system drivers or libraries, that did not behave properly when confronted with unexpected use of the reserved status bytes. Ultimately the committee chose to develop an entirely new protocol.

We find this argument unpersuasive. It is essentially saying that, because *some* nonconforming interfaces and software would need to be revised to work properly with Protocol A, we should abandon it and move to Protocol B, where *all* of them would need to be not only revised but redesigned from scratch. There are many compelling reasons to move to a better protocol: but we think this is not one of them.

We do note that a few specifications external to MIDI proper, such as MIDI RTP [10], have over the years improperly *banned* reserved status bytes rather than reserved them, and so might require revision.

2. MIDI 1.0 DESIDERATA

MIDI's failings are well known. Here are a few.

MIDI is Slow MIDI is a serial spec that runs at 31250 BPS, including a start and stop bit. This was chosen carefully: it is exactly 1/32 of 1MHz. In fact, originally it was to run at 19200 BPS until revised after suggestions from Japanese manufacturers [2]. It was thought that 31250 would be fast enough to "prevent objectionable delays between equipment" [11, p.7].

Parameter and Value Resolution are Poor MIDI pitch, velocity, and aftertouch resolution are only 7-bit: though the MIDI Tuning Standard allows for pitch changes with $100/2^{14}$ cents resolution. The value and parameter resolution of MIDI CC are both only 7-bit, and much of the CC parameter space is used up by official conventions and standards. One alternative route, 14-bit CC, increases, both the value space but dramatically reduces the parameter space. Another solution, RPN/NRPN, increases the value and parameter spaces to 14 bit, but is slower.

MIDI has Namespace and Polyphony Problems MIDI has a single namespace, namely its 16 channels, which is very limited from a modern perspective. This is remediated to some degree in software by increasing the number of MIDI ports or devices, each with its own set of 16 channels, but this is a poor fix. To make matters much worse, MIDI channels are not fully polyphonic, notably in per-note parameters and pitch bend. MPE was proposed to fix this, but its approach (usually making each channel a monophonic voice) means that MPE synthesizers can respond properly to at most 15 notes per MPE Zone, and there can be at most two of them, dividing those 15 notes amongst themselves. Furthermore in doing so this radically eats into the channel space allotted to other synthesizers.

Limited Discovery and Capability Inquiry MIDI only has a primitive discovery mechanism. Via system exclusive, devices can inform us that they exist but not what protocols they can respond to.

Patch Transfer is Undefined MIDI never defined how to transfer patches or update fined-grained patch data outside of CC or NRPN. Manufacturers, left to their own devices, have constructed a zoo of incompatible, buggy, and often profoundly ill-conceived patch transfer or revision schemes [12].

MIDI is Unidirectional MIDI was designed for a single device to send messages to multiple downstream devices. It was not meant for devices to communicate with one another in an arbitrary graph topology.

2.0.1 A Few Solutions

We are not so grandiose as to suggest that solutions to all these problems can be found with small modifications to MIDI 1.0. Certainly speed, for example, cannot be improved in general as MIDI 1.0 on serial hardware is fixed to 31250 BPS. However a number of improvements can be made while retaining backward compatibility with properly conforming devices. We refer to these improvements colloquially as *MIDI 1.5*.

MIDI 2.0 is also touted as backward compatible with MIDI 1.0: but only in the very weak sense that if the older device appears to not understand it, one could always revert to 1.0 instead. It cannot intermesh with properly conforming 1.0 devices. We think it is better placed in the "clean break" category along with approaches such as Open Sound Control [6].

How might we make MIDI 1.5 backward compatible with 1.0? First off, 1.0 has two options for official extension, Registered Parameter Numbers (RPN) and Universal System Exclusive messages. Less well known is that MIDI 1.0 also has *four* status bytes still not in use. Two of these status bytes (F9 and FD) are System Real-time messages and so are generally expected to be only one byte in length. The other two (F4 and F5) are System Common messages and do not have defined lengths. These status bytes could be defined for our purposes.

In this paper we outline six ways in which we could improve some of these issues while staying backward-compatible with MIDI 1.0, by using Universal System Exclusive and RPN messages and certain reserved status bytes. They are:

Discovery Given a bidirectional channel, we can determine if downstream devices are MIDI 1.5-Ready.

Sectors These are roughly analogous to MIDI 2.0's groups. This scheme extends MIDI's channels to 2048.

MPE Improvements We employ Sectors to extend MPE instruments to have more than 15 notes and more than two MPE zones.

Ports This scheme can be used instead of, or in addition to, CC or NRPN. Compared to NRPN, Ports have more parameters and often faster speed. Ports also fix NRPN's lack of atomicity, a major failing.

Patch Transfer We can use MIDI 2.0 Capability Inquiry (CI) to query devices for capabilities and parameters, and update the same, as is uses System Exclusive. But MIDI 2.0 CI is heavyweight, and so we also propose a basic patch dump standard.

Pitch Resolution We could improve the per-note pitch resolution with a CC.

3. DISCOVERY

We begin with methods to determine if a downstream device exists and is *MIDI 1.5 Ready.*

A few preliminaries. First, it has been said that MIDI 1.0 is unidirectional. But that is not really true! Bidirectionality just requires two MIDI cables. The fundamental distinction is that in MIDI 1.0, a device may not be the recipient of messages from two different senders (absent MIDI Merge). However MIDI 1.0 is perfectly capable of device discovery and capability inquiry if two cables or virtual devices are used. We presume bidirectionality for MIDI 1.5 Discovery, and later for Capability Inquiry and the Patch Dump standard (Section 7). However this is not absolutely necessary: if we only have a unidirectional channel, then we can still proceed with many MIDI 1.5 capabilities with care.

Second, several MIDI 1.5 messages are System Exclusive Common messages. These messages have a specific format: *F0 Type DeviceID SubID ... F7*. *Type* is 7E for non-real-time and 7F for real-time message categories. The *DeviceID* is set on the Device itself: all downstream devices have different DeviceIDs. A DeviceID of 7F means "all devices". The *SubID* is the namespace for each category of System Exclusive Common messages. MIDI 1.5's namespace is not yet defined: we will use the placeholder *M1.5* throughout.

3.0.1 MIDI 1.5 Readiness and Tolerance

A *MIDI 1.5 Ready* device understands MIDI 1.5 and how to respond to MIDI 1.5 properly. Using MIDI 1.5 Discovery, we can probe to determine if devices are MIDI 1.5 ready. In order to take advantage of MIDI 1.5, the endpoint devices must be MIDI 1.5 Ready.

A *MIDI 1.5 Tolerant* device is a device which is not MIDI 1.5 Ready, but ignores MIDI 1.5 messages and passes them through unmodified. In order to use MIDI 1.5 data, all devices must be at least MIDI 1.5 Tolerant. In particular, all drivers, interfaces, routers, and other in-between devices must be MIDI 1.5 Tolerant. MIDI 1.5 uses reserved status bytes from MIDI 1.0. It does not use any System Real-time status bytes. Thus it should be transferred to downstream devices by properly-adhering MIDI 1.0 operating system drivers, MIDI interfaces and routers, and so on. However there exist ones which filter out these status bytes, convert them, or otherwise behave badly when receiving them, and so will interfere with 1.5 even if the downstream device is MIDI 1.5 Ready. Thus we may query a downstream device and have it respond that it is MIDI 1.5 Ready, only to find that we cannot send 1.5 to it because of non-MIDI 1.5 Tolerant operating system driver.

It's possible for multiple devices to be on the same MIDI daisy-chain, and one is MIDI 1.0 (but 1.5 Tolerant) while another is MIDI 1.5 Ready. This would require that the MIDI 1.5 Ready device restrict itself to only operate within the channels available to it.

3.1 Existence Query and Response

We use MIDI System Exclusive (or *Sysex*) Device Inquiry to determine if bidirectional communications is working, and which devices are capable of responding. There may be more devices listening than are capable of responding. To send a Sysex Device Inquiry to a specific device, we send *F0 7E DeviceID 06 01 F7*.

Ideally (and always if the Device is MIDI 1.5 Ready) the Device should respond with a Sysex Device Inquiry Response: *F0 7E DeviceID 06 02 Manufacturer (1–3 Bytes) Device (4 Bytes) SoftwareRevision (4 Bytes) F7*. The *Device* consists of a 2-byte family code and a 2-byte family member code. We will use the *DeviceID, Manufacturer,* and *Device* later.

3.2 MIDI 1.5 Declaration and Query

The sender can declare that it is itself MIDI 1.5 Ready by broadcasting *F0 7E 7F M1.5 00* to all devices. Note that the sender does not itself have a DeviceID, though we could perhaps adapt 7E as a convention.

We next query a given device as to whether it is MIDI 1.5 Ready or not, with *F0 7E DeviceID M1.5 01 F7*. If the device is MIDI 1.5 Ready, it should respond with *F0 7E DeviceID M1.5 00 F7* .

If a device has responded with the Sysex Device Inquiry Response and a MIDI 1.5 Declaration, then we assume the following:

- The device exists.
- The device can respond to us.
- The device is MIDI 1.5 Ready: this includes being able to respond usefully to MPE messages.

If the device has responded with a Sysex Device Inquiry Response but not a MIDI 1.5 Declaration, then we should assume that the device is *not* MIDI 1.5 Ready, though it may be MIDI 1.5 tolerant and able to respond usefully to MPE Messages.

If the device has not responded with a Sysex Device Inquiry Response, then it may not exist, or may or may not be MIDI 1.5 Ready but is unable to respond to us. We must tread carefully.

4. SECTORS

Sectors are roughly analogous to MIDI 2.0's groups. They effectively extend the number of channels to 2048 and ultimately allow MPE to have more than 15 notes.

There are many reasons to want more than 16 channels. But one critical one is MPE, which has co-opted MIDI channels to provide per-voice parameterization. But because there are only 16 channels, MPE can support at most 15 voices. Furthermore, because MPE uses MIDI channels in the way it has, they can no longer be used in a multitimbral fashion if MPE is applied to all 15 voices.

4.1 Approach

A Sector is a group of 16 channels. There are 128 Sectors in all. Sectors can be used to provide 2048 individual streams of multitimbralism, and to support up to 256 MPE devices. Non-voiced messages are applied to all Sectors. All voiced messages are interpreted as applying only to the current Sector.

The current Sector is set with the Sector Declaration, *F5 Sector*. The default Sector is Sector 0, the "MIDI 1.0 Sector". Because there may exist MIDI 1.0 devices listening, Sectors must be careful to interoperate with them. To do this, the Sender must know (or suspect) which channels are being used by MIDI 1.0 devices, and assume that these channels are *always* Sector 0 regardless of the Sector declaration.

4.2 Sector and Channel Discovery

A MIDI 1.5 Ready device should respond to a Sector and Channel Request *F0 7E DeviceID M1.5 03 F7* to state which Sectors and channels it responds to. The response is *F0 7E DeviceID M1.5 02 [Sector Channel MPEInstrumentTag]* F7*. The *MPEInstrumentTag* is the MPE Instrument Tag of the given Channel: see Section 5. If *MPEInstrumentTag* is 7F, then the Channel has no MPE Instrument assigned, and is a plain Channel.

4.3 Alternatives

Sector Declarations could be less efficiently done in System Exclusive as *F0 7F 7F M1.5 00 SectorByte*.

4.4 Issues

Whereas declaring the channel of a voiced message is 3 bits, changing the Sector requires two bytes. Ideally per-Sector messages would be grouped together when possible to require only one Sector change message.

If a MIDI 1.5 Ready receiver has not heard a previous Sector message (perhaps he started listening late, or noise corrupted the message), he will apply follow-on messages to the wrong Sector. This is analogous to similar synchronicity issues in RPN and NRPN.

4.5 Mapping to MIDI 2.0

MIDI 2.0 introduces 16 *groups*, each of which may hold 16 channels, for a total of 256 channels. These could map to the first 16 Sectors. MIDI 2.0 CI also is compatible with groups, and thus we believe with the first 16 Sectors.

5. MPE IMPROVEMENTS

MPE is a hack that uses channels to enable per-note parameterization. Unfortunately, in doing so, MPE has three major limitations. First, it is limited to only fifteen voices. Second, it is limited to only two MPE instruments (that is, *MPE Zones*). Third, use of MPE consumes channels which could otherwise be usefully used by other devices.

We addressed the third issue with Sectors, discussed in Section 4. We address the other two here.

Our approach extends MPE Zones to *MPE Instruments.* An Instrument is a collection of MPE Zones grouped together into one voice pool. An MPE Instrument may have up to 128 voices shared across a number of MPE Zones. Each MPE Instrument has an *MPE Instrument Tag* and MPE Zones are each assigned a tag to indicate that they are part of a given Instrument. There can be up to 127 Instruments.

Because they use so many channels, realistically if there are any MPE devices listening, they must either all be, or none of them must be, MIDI 1.5 Ready. We further would recommend that MIDI 1.5 Ready devices only use MPE Low Zones.

5.1 MPE Capability Discovery

A device should respond to a MPE Capability Request to indicate which instruments it has available. The MPE Capability Request is *F0 7E DeviceID M1.5 05 F7*. The device would respond with *F0 7E DeviceID M1.5 04 NumMPEInstruments [MPEInstrumentTag MaxVoices]* F7*. Here, *NumMPEInstruments* indicates the maximum number of MPE instruments that may be created for the device. *MaxVoices*+1 is the maximum number of voices per instrument. This, combined with the Sector and Channel Request (Section 4.2) gives us which instruments are meant for which channels.

With or without discovery as useful additional information, we can proceed to define MPE Zones, per Sector, using the standard MPE RPN messages.

5.2 Issues

In MIDI 1.0, downstream devices decide on their own channels, but in MPE it is the *sender* which decides the structure and size of the MPE zones available, and the devices are left to fend for themselves after being told this structure. This may become more problematic with increasing numbers of channels, and zones.

6. PORTS

Ports are an attempt to achieve better speed than NRPN in certain common situations, to provide better parameter and value resolution than NRPN, and to overcome mistakes in the RPN/NRPN specification.

What mistakes? Since its inception [11] CC has been conventionally divided into two regions: 7-bit CC messages offering a resolution of 128 values, and optional CC message pairs offering 14-bits, or a range of 16384 values. In

the second case, a "most significant byte" (MSB) message is paired with a "least significant byte" (LSB) message. The number formed would be MSB×128+LSB. For example, to send a 14-bit value for parameter 2, you'd send the MSB of the value as CC 2, and the LSB of the value as CC 34. The MIDI designers felt that it was reasonable for a simple controller to just send the MSB, and so argued that the LSB should be thought of as an optional "fine tuning" of the MSB value. This was accompanied with the following text:

> If 128 steps of resolution is sufficient the second byte (LSB) of the data value can be omitted. If both the MSB and LSB are sent initially, a subsequent fine adjustment only requires sending the LSB. The MSB does not have to be retransmitted. If a subsequent major adjustment is necessary the MSB must be transmitted again. When an MSB is received, the receiver should set its concept of the LSB to zero.

This was a mistake. First off, a parameter's value range may not be 0–16383, but might more often be, say, 0–194. Following these instructions, manufacturers would have to awkwardly "stretch" the 194 range to 16384 so as to allow a bare MSB to be sent with no LSB. Many chose not to, and so the MSB would only be (in this example) 0 or 1, and the LSB would be the dominant factor.

Second and more problematic, imagine if the value is currently MSB=4, LSB=9. A new MSB=7 arrives. Is an LSB soon arriving to accompany it? If the controller were just a 7-bit controller, maybe not. The device is required to assume, for the interval between an MSB and an LSB, that the value is MSB×128+0, which is *wrong* if an LSB is arriving. 14-bit CC is not atomic.

These were reasonable oversights in the first MIDI draft, but then they were needlessly inherited in NRPN and RPN. Some manufacturers (like Sequential!) have abandoned NRPN "fine tuning" as unworkable, but others have retained it. Controller and DAW vendors must now offer both "fine" (LSB-primary) and "coarse" (MSB-primary) versions of 14-bit CC and NRPN.

Atomicity could have been fixed in 14-bit CC and RPN/NRPN by requiring that after an MSB the value not be changed until the next LSB appears. But that ship has sailed. We could have changed it here too, but it would not be backward-compatible with MIDI 1.0.

6.1 Approach

We introduce 65536 *Ports*, parameters like RPN/NRPN. A Port can be set to 16384 values.

Ports provide the full parameter or value in a single message, and so they are atomic. There is no notion of "coarse" versus "fine tuning", as neither the MSB nor LSB can be provided by itself. Unlike RPN and NRPN, ports can also auto-increment the parameter in order to easily perform dumps of parameter ranges.

Port parameters and values are set with the same message: `F4 Command MSB LSB`. If this message is meant to change the *Port parameter*, then *Command* takes the form `00hhcccc`, where *hh* are the high bits of the parameter, added before the MSB and LSB for a total of 16 bits or 65536 parameters; and *cccc* is the channel.

If the message changes the *Port value*, then *Command* takes the form `01ppcccc`. The MSB and LSB define the value with 14 bits, thus 16384 values. The *cccc* is the channel. The *pp* state how the value is to be set:

00 Set the value to the given amount.

01 Set the value to the given amount, then increment the current parameter. 65535 increments to 0 (wrapping around).

10 Increment the value by the given amount, with a ceiling at 16383.

11 Decrement the value by the given amount, with a floor at 0.

6.1.1 Efficiency Comparisons

The worst-case scenario in overhead for setting values in a Port require also changing the Port parameter each and every time. This would be a total of 8 bytes. But once a Port parameter is declared for a given channel, values may be repeatedly changed in that channel. Normally this would be 4 bytes per change. However if we were to extend running status to include F4, changing values in a Port would be 3 bytes amortized.

In comparison, the worst case scenario for NRPN is [Param-MSB Param-LSB Value-MSB Value-LSB]*. This is 12 bytes. The amortized case for NRPN is [Param-MSB Param-LSB Value-MSB Value-LSB [Value-MSB/LSB]*]*. Assuming running status, and that the user only changes the LSB (or only the MSB) this requires 2 bytes per change (else 4 bytes).

Ports have an advantage when making *bulk* changes to many parameters in a row. Ports can auto-increment the parameter, and so bulk changes are 3 bytes per change; whereas for NRPN they are 12 bytes.

Thus Ports are 1.5× faster for single parameter changes and 4× faster for bulk changes of multiple parameters, but are roughly 1.5× slower for multiple changes to the same parameter, such as a filter sweep. [1] There is of course no reason why NRPN and Ports could not be interchanged, assuming that their parameter numbers mapped to the same parameters.

7. PATCH TRANSFER

One of the MIDI's biggest failings is its lack of a standardized patch dump or editing standard. This is curious as the system exclusive message was, from the very beginning,

[1] We considered using `F4` to change the Port parameter as `F4 MSB LSB`; the value would be sent using Polyphonic Aftertouch. Port parameter 0 (the default) would be standard Polyphonic Aftertouch. This would give 16383 parameters, would yield a 2× improvement for single parameters and a 6× improvement for bulk changes. To do this trick we'd need *some* three-byte voiced channel message, and Polyphonic Aftertouch was the clear candidate as changing it would have the least impact and MPE forbids its standard use. However it could still cause problems with non-MIDI-1.5-Ready devices.

primarily meant for patch dumps [11]. As patch dumps were never standardized, this has led to a menagerie of patch dump protocols which can only be described as incredible in their variance and poor design [12]. Yet at the same time, MIDI proposed a standard *sample dump* protocol despite MIDI being a poor choice for sample dumps due to its speed.

It may be too late to propose a patch dump protocol, for MIDI 1.5 or even 2.0, because a number of manufacturers have begun to treat system exclusive as a trade secret or to abandon it entirely in favor of alternative USB-based byte protocols [12]. This is in stark contrast from the past, where manufacturers proudly published their system exclusive specifications in the name of openness that was the hallmark of MIDI.

That being said, a standard protocol would go a long ways towards fostering a better ecosystem. To this end, we propose the following set of conventions:

7.0.1 Command Syntax

System exclusive commands take the form *F0 Manufacturer (1–3 bytes) Payload...*, followed by a status byte, usually *F7*. *Manufacturer* is 1–3 bytes and sets the manufacturer ID, and thus defines a namespace. Manufacturer IDs are registered with the MMA.

Our proposed commands take one of two forms:

- Short Commands
  Header... Command (1 byte) Data (0–4 bytes) F7
- Long Commands *are usually of the form*
  Header... Command (1 byte) Length (2 bytes) Encoding Data... (any length) Checksum F7

...where the *Header...* is *F0 Manufacturer (1–3 bytes) Device (4 bytes) Sector*. The Device is four bytes, comprised of the device's family code and family member code, 2 bytes each. The *Sector* is the Sector (or just *a* Sector) to which the device is listening, or is 7F, that represents any Sector to which the device is listening.

The *Command* is the directive in question: the first 32 commands are reserved and 10 are defined as described below. The rest are up to the manufacturer's discretion. The *Data* is encoded using the given *Encoding* as described in Section 7.1. The *Length* is the length of the Data only, as a 14-bit number, and the *Checksum* is the sum of all the bytes, unsigned, of all bytes in the Payload except for *F0* and the Checksum, mod 128.

Patch data which uses the conventions in this Section should be fully, freely, and openly specified.

7.0.2 Actions

A Device must be able to issue or respond to the following twelve actions. Individual parameters and patches *could* be dumped with Ports, but we recommend the Dump commands here for a consistent sysex interface.

Version and Configuration This would be *Header... 00 Encoding Data... F7*. The specification for the *Data...* in this message must be publicly available.

Request Version and Configuration This would be *Header... 01 F7*. The device would respond with a Version and Configuration Message as described earlier.

Send Current Patch Load the patch into current working memory such that it can be used immediately by the musician. Do not write to patch memory, nor perform a program change. This would be *Header... 02 Chunk Of Length Encoding Data Checksum F7*. The data may be broken up into multiple pieces. *Of* indicates the number of pieces in total, and *Chunk* $(0...Of-1)$ represents the which piece this message is. Chunks must be sent in ascending order and do not take effect until the final Chunk is received.

Request Current Patch Request the patch presently in current working memory, that is, the one currently being used by the musician. Do not perform a program change. This would be *Header... 03 F7*. The device would respond with a *Send Current Patch* message.

Dump Patch Write the following patch to a given location in patch memory. Do not perform a program change to the patch. This would be *Header... 04 Chunk Length Encoding Location Data Checksum F7*. Note that the data may be broken up into multiple pieces. *Of* indicates the number of pieces in total, and *Chunk* $(0...Of-1)$ represents the which piece this message is. Chunks must be sent in ascending order and do not take effect until the final Chunk is received.

Request Dump Patch Request a patch from a given location in patch memory. Do not perform a program change. This would be *Header... 05 Location F7*. The device would respond with a *Dump Patch* message.

Change Patch This consists of an optional *Bank Select* message, followed by *Program Change* to any bank and patch. A Bank Select is sent with CC 32 for the LSB and CC 0 for the MSB; both the MSB and LSB must be sent, and the bank number is interpreted as MSB $\times 128+$ LSB. If there is only one bank, then Bank Select may be omitted.

Request All Patches in Bank This would be *Header... 06 Location F7*. Do not perform a program change to any patch. The device would respond by dumping each patch in turn from the current bank.

Change Mode If the Device is modal, the mode is changed. For example, a synthesizer might be playing in single-mode, multitimbral-mode, or drum mode, each with its own set of patches. After changing mode, the device may default to whatever patch in the mode it sees fit. This would be *Header... 07 Mode F7*.

Stop All Sound This kills all sound from the device, and is done with CC 120. The device is encouraged to also implement CC 123 but it is not required to.

Dump One Patch Parameter This is *Header... 09 Parameter (4 bytes) Value (4 bytes) Checksum F7*, where both the *Parameter* and *Value* are 4 MIDI bytes (the *Value* is encoded via Encoding #2 in Section 7.1); or by *Header... 09 Length*

Parameter (4 bytes) Encoding Value... Checksum F7, where the *Value* is encoded the provided *Encoding* as discussed in Section 7.1. Manufacturers are also encouraged to set parameters, to the degree that they can, via Ports or NRPN.

Query One Patch Parameter We query a parameter via *Header... 08 Parameter (4 bytes) Checksum F7*, where the *Parameter* is 4 MIDI bytes. The device responds with a Dump One Patch Parameter message.

7.1 Encodings

We propose five standard data encoding methods. The first three are obvious to anyone who has dealt with patch dump system exclusive messages in the past. The fourth is a logical extension of them. The fifth encoding is meant for situations where the size of some elements, or of every element, may vary arbitrarily.

0. Each element, 7 bits long, encoded in one byte.
1. Each element, 8 bits long, encoded by sending groups of 7 bytes at a time without their high bits, then a final byte with all 7 high bits. In this final byte, the low bit is the high bit of byte 1.
2. Each element, 14 bits long, encoded in two bytes, LSB sent first and MSB sent second.
3. Each element, 16 bits long, encoded by breaking it into two bytes, LSB first, then sending the resulting stream of bytes using Method 2.
4. Each element encoded in turn using Variable Length Encoding, as described below.

Prior to encoding, signed values should be converted to 0...*N* where 0 represents the lowest negative number and *N* represents the highest positive number.

7.1.1 Variable Length Encoding

There are many ways to do a variable length encoding. Here is one simple possibility that is easy and fast to implement. The encoding has a variable-length *header* followed by a certain amount of *data*. The header is zero or more 1-bits, followed by a single 0-bit, which together control the length of the data. The data appears immediately after this. The length of the message is equal to the number of bytes necessary to contain the header (padded with data bits), plus *N* additional bytes of data, where *N* is the number of 1-bits in the header. The encoding has at least an 85.7% efficiency.

Some examples. We pack up to 6 bits as 0ddddd, encoding the data *dddddd*. We may pack up to 12 bits as 10ddddd ddddddd, 18 bits as 110dddd dddddd dddddd, 24 bits as 1110ddd ddddddd ddddddd ddddddd, and so on. The header may cross byte boundaries, so we pack up to 42 bits as 1111110 [ddddddd]×6, but then 55 bits as 1111111 0ddddddd [ddddddd]×7 and 61 bits as 1111111 10ddddd [ddddddd]×8.

The low bit of the data should start at the far end of the encoding and proceed towards the front, zero-padding at the front if necessary. For example, the value 371, encoded in binary as 101110011, should be converted to 12 bits as 000101110011, and then packed as 1000010 1110011. It is not necessary to provide the full encoding if a value can be packed in less. For example, if we have a 32-bit parameter but we are setting its value to just 23, we could just encode it as 0010111.

7.2 MIDI 2.0 Capability Inquiry

MIDI 2.0 introduced MIDI 2.0 Capability Inquiry (CI), which allows a sender to determine the capabilities of a downstream device, and to query and change its parameters, given bidirectional communication. Unlike much of the rest of MIDI 2.0, this protocol can be done entirely with MIDI 1.0 System Exclusive messages. For that reason we could also adopt it here, and it would serve to replicate some of the functionality discussed in this Section. We note however that it has a complex specification totaling 183 pages spread over four documents (M2-100-U through M2-103-UM of [3]).

8. PITCH RESOLUTION

MIDI's pitch resolution is per-note. If you want a pitch whose value is in-between notes, you have three options in MIDI 1.0. First you can use the *MIDI Tuning Standard* (MTS), which specifies which pitch each key falls on: but you are stuck with 128 pitches. Second, you can send pitch bend prior to playing the note: but this will change the pitch of all other notes. Third, you can do the same thing with MPE, as it requires that bend not affect released notes, and there is normally one note per channel: but now bend can no longer be used to affect released notes! And, of course, it requires that you use MPE.

Ideally, the high resolution pitch of a note would accompany the note itself. MIDI Manufacturer's Association document CA-031 [13] proposed that prior to playing a note, one might send CC 88 as its *high resolution velocity*, that is, as the LSB of the note velocity. We suggest selecting another CC to do the counterpart: *high resolution pitch*. The LSB could be measured in 1/128 of a semitone; or perhaps it would be easier to simply measure it in cents, reserving the top 28 values.

We note three downsides. First, this is two to three times slower than a plain Note On, depending on Running Status. Second, the receiving device would have to support this CC. Third, the resulting 2^{14} total pitch range is less than MIDI 2.0's 2^{32}, 2^{21} for MTS, or even 2^{15} for MPE's default pitch bend.

9. CONCLUSION

We hope that MIDI 2.0 will be successful, but it has had a difficult roll-out. It is a large specification imposed on a small community. We have argued that in the meanwhile, or in the event that 2.0 is not fully adopted, MIDI 1.0 could still be usefully improved while maintaining backward compatibility and relative simplicity. We admit that not everything can be improved, such as raw serial hardware speed. But enhancements forming MIDI 1.5, so to speak, could have a major impact well worth the small cost.

10. REFERENCES

[1] D. Smith and C. Wood, "The 'USI', or Universal Synthesizer Interface," in *Audio Engineering Society Convention 70*, Oct 1981.

[2] MIDI Manufacturers Association, "MIDI history chapter 6 — MIDI begins 1981–1983," https://www.midi.org/midi-articles/midi-history-chapter-6-midi-begins-1981-1983, as of 1/1/2024.

[3] *MIDI 2.0 Specification*, MIDI Manufacturers Association, 2023.

[4] *The Complete MIDI 1.0 Detailed Specification (Version 96.1, Third Edition)*, MIDI Manufacturers Association, 1996.

[5] E. Selfridge-Field, Ed., *Beyond MIDI: The Handbook of Musical Codes*. MIT Press, 1997.

[6] M. Wright and A. Freed, "Open SoundControl: a new protocol for communicating with sound synthesizers," in *International Computer Music Conference (ICMC)*, 1997.

[7] M. Lauter, "The case for MIDI real-time MARK message F9h," http://lauterzeit.com/tech/midi/case_for_MIDI_F9h_Aug2016.pdf, as of 1/1/2024.

[8] Forty-Seven Effects, "Arduino MIDI library," https://github.com/FortySevenEffects/arduino_midi_library/, as of 1/1/2024.

[9] J. Glatt, "Tick," http://midi.teragonaudio.com/tech/midispec/tick.htm, original site, now defunct, was http://www.borg.com/~jglatt. As of 1/1/2024.

[10] J. Lazzaro and J. Wawrzynek, "RTP payload format for MIDI," https://datatracker.ietf.org/doc/html/rfc6295, 2011, as of 1/1/2024.

[11] S. Jungleib, *The Complete SCI MIDI*, Sequential Circuits, Inc., 1983.

[12] S. Luke, "So you want to write a patch editor," Available at http://cs.gmu.edu, George Mason University, Tech. Rep. GMU-CS-TR-2023-1, 2018.

[13] *CC #88 High Resolution Velocity Prefix (CA-031)*, MIDI Manufacturers Association, 1996.