The ZIPI Music Parameter Description Language

Keith McMillen,* David L. Wessel,† and Matthew Wright*†
*Zeta Music/Gibson Western Innovation Zone
2560 Ninth St., Suite 212
Berkeley, California 94710 USA
†Center for New Music and Audio Technologies (CNMAT)
Department of Music, University of California, Berkeley
1750 Arch St.
Berkeley, California 94720 USA
McMillen, Wessel,
Matt@CNMAT.Berkeley.edu

ZIPI's Music Parameter Description Language (MPDL) is a new language for describing music. It delivers musical parameters, such as articulation and brightness, to notes or groups of notes. It includes parameters that are well understood and universally implemented, such as loudness and pitch, and supports parameters that should be more common in the future, such as brightness and noise amount. A large number of parameters remain unspecified, thus ensuring expandability and flexibility. The MPDL is just one of ZIPI's application layers; others will include MIDI, data dumps, and digital audio.

This article does not address any of the low-level networking issues associated with ZIPI. We will assume that ZIPI's lower levels deliver arbitrary-sized data packets from any device on the network to any other device on the network; this article describes how to transmit music data via those packets. This application layer could run equally well on some other lower network layer, such as Ethernet or FDDI.

ZIPI's Music Parameter Description Language was designed by Keith A. McMillen, David Wessel, and Matthew Wright.

The Shape of MPDL Packets

Figure 1 shows the format of MPDL packets. The low-level network that carries the MPDL packets will impose some overhead bytes; this will include a network address indicating which ZIPI device should receive the packet. The other overhead bytes depend on the particular network that carries MPDL; we will not discuss the details here. Note that overhead bytes are at the beginning and/or the end of a network packet, but the MPDL data are contiguous. The MPDL data consist of a *note address* followed by an arbitrary number of *note descriptors;* these are described in the sections below.

Computer Music Journal, 18:4, pp. 52–73, Winter 1994

Address Space

When you send a message, such as "become louder," you also need to specify what it is that should become louder. We call this "what" the *address* of a message. In MIDI, you might send continuous controller number 7 (which typically means volume) to channel 3; the address of the message is thus "channel 3." If you want to send a MIDI message to a single note, rather than to an entire channel, you must name that note by giving its pitch as well as its channel, as in, "apply aftertouch to middle C on channel 2; release the G above middle C on channel 1."

One weakness of MIDI is that there are many musical situations that are awkward to express when a note's address corresponds directly to its pitch. For example, a note's pitch might change over time, or there might be two notes played on the same instrument with the same pitch. Therefore, in MPDL, notes have individual addresses

Figure 1. Format of ZIPI MPDL packets.

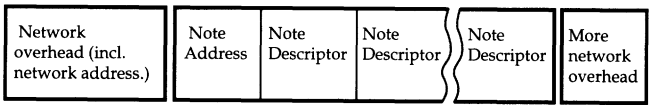
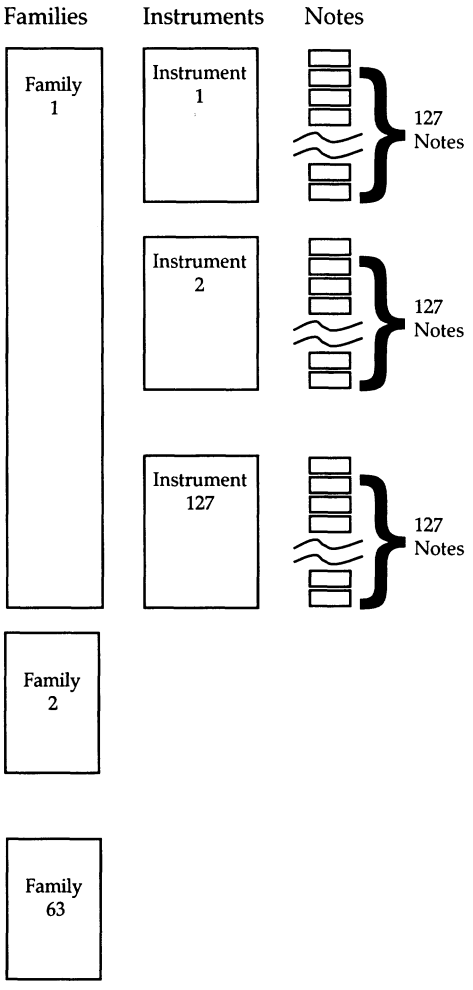


Figure 2. MPDL's address space hierarchy.



that are unrelated to their pitch. An MPDL note number is simply a number used to identify that note; any MPDL note number can have any pitch.

MIDI's address space is organized as a two-level hierarchy—notes within channels. It is sometimes useful, however, to control groups of groups of notes, rather than just groups of notes. For this reason, MPDL's note address space uses a three-level hierarchy. Our names for the layers of the hierarchy fit an orchestral metaphor; there are *notes* within *instruments*, and the instruments are grouped into *families*. (One might ask why we do not have a four-level hierarchy or even a general *n*-level hierarchy. Such schemes take up synthesizer resources, so we have tried to balance generality with ease of implementation.)

Another weakness of MIDI is that each kind of message must be addressed either to an entire channel or to a single note, but not both. For example, it is impossible to individually pitch-bend one of the notes of a MIDI channel; any pitch-bend information will affect all of the notes sounding on that channel. Pan, volume, and any other "continuous controller" data must also apply to an entire channel. Likewise, other MIDI messages, such as note-on, are always per note, so there is no way to articulate an entire chord with a single message. In general, one would like to be able to send any control signal either to a particular note or to a group of notes. Therefore, in MPDL, any message can be addressed to any level of the hierarchy. For example, a pitch message could go to a single note, a note release message could go to all of the notes of an instrument, and a loudness message could go to an entire family.

There are 63 MPDL families, each of which has 127 instruments. Each of these 8,001 MPDL instruments has 127 notes, for a total of 1,016,127 MPDL note addresses. This hierarchical organization is shown in Figure 2. Families, instruments, and notes are numbered from 1, not 0. There is also a way to send a message to all families. This is not a fourth level of the hierarchy, but rather an abbreviation for sending the same message to each of the 63 families; its effect is exactly the same as if the message were sent 63 times. An instrument belongs to exactly one family and cannot change its family. The orchestral metaphor is just a metaphor; the purpose of an MPDL address is to specify uniquely the note or group of notes to which a message applies.

The device sending the note information must keep track of which notes are sounding and which are not, so that it can update parameters of already sounding notes. The device receiving the note information, of course, will not be capable of 1,016,127-note polyphony, so it must manage the allocation of the available synthesis resources.

(Nor will it be capable of storing parameters for 1,016,127 different notes. It is expected that ZIPI controllers will, in practice, only use a small subset of the address space; algorithms can take advantage of that expectation to store and record parameter values very quickly and without using very much memory.)

Each ZIPI device has its own address space; all of the families, instruments, and notes just discussed are within a single ZIPI device. In addition to the note address contained in the MPDL packet, ZIPI packets also include a *network address* that indicates which device should receive the MPDL message. Delivery of the MPDL packet to the appropriate device is handled by the network levels below MPDL.

Controlling Musical Instruments

Musicians will typically set up each MPDL instrument as 127 voices of a particular timbre. (In a sense, an MPDL instrument configured in this way is like a MIDI channel, which is always associated with a single "patch" or "preset.") Messages sent to this instrument will influence all of the notes played by the instrument, yet other messages can be addressed to individual notes within the instrument. For example, one could "pitch-bend" all of the notes played by a given instrument by sending a pitch message to the instrument, yet an individual note can be bent by sending a pitch message to that note.

Furthermore, it sometimes makes sense to group collections of instruments to be controlled together. To create a ZIPI orchestra, we might put all of the strings in one family, the brass in another, and woodwinds and percussion instruments in two more. In the string family, we would have instruments for first violins, second violins, violas, cellos, and basses. Each string of each of the violas would be a note. Table 1 shows how this might be set up.

Because commands can be issued to control each level of the hierarchy, this setup gives the user a conductor's control over the orchestra. The string family can be made louder and brighter. The woodwinds can all be panned slightly to the left. Within the woodwind family, individual instruments can be addressed; for example, the oboes can get quieter while the clarinets overblow.

Table 1. The Address Space for a ZIPI Orchestra

Family 1: Strings
- Instrument 1: First violins
- Instrument 2: Second violins
- Instrument 3: Violas
  - Note 1: C string
  - Note 2: G string
  - Note 3: D string
  - Note 4: A string
- Instrument 4: 'Cellos
- Instrument 5: Basses

Family 2: Brass
- . . .

Family 3: Woodwinds
- Instrument 1: Flutes
- Instrument 2: Clarinets
- Instrument 3: Oboes
- Instrument 4: Bassoons

Family 4: Percussion
- . . .

Musical Control Parameters

After the note address has been chosen, a ZIPI packet may contain any number of *note descriptors* intended for that address. A note descriptor gives a new value for a parameter, such as "pitch is B flat 2" or "pan hard left." The note descriptor consists of a note descriptor identifier (ID), which indicates which parameter is being updated, and some number of data bytes, which give the new value for that parameter.

Tables 2 through 6 list the currently defined note descriptors. We expect to define a few more note descriptors before completing the specification of the MPDL, but we will leave at least half of them explicitly undefined (Zicarelli 1991). The interpretations of these messages are described in

Table 2. Synthesizer Control Parameters

No. of Data Bytes	*ID (Hex)*	*Meaning*	*Default*	*Combining Rule*
1	01	Articulation	See below	"And"
2	40	Pitch	Middle C	Add (see below)
4	80	Frequency in Hz	Middle C	Overwrite
2	41	Loudness	Mezzo forte	Multiply
2	42	Amplitude	Midscale	Multiply
1	02	Brightness	Midscale	Multiply
1	03	Even/odd harmonic balance	Midscale	Multiply
1	04	Pitched/unpitched balance	Midscale	Multiply
1	05	Roughness	Midscale	Multiply
1	06	Attack character	Midscale	Multiply
1	07	Inharmonicity (signed)	0	Multiply
1	08	Pan left/right	Center	Multiply
1	09	Pan up/down	Center	Multiply
1	0A	Pan front/back	Center	Multiply
2	43	Spatialization distance	10 meters	Multiply
1	0B	Spatialization azimuth angle	Forward	Add
1	0C	Spatialization elevation angle	0	Add
2	44	Multiple output levels	Midscale	Multiply
2	45	Program change immediately	0 (silence)	Overwrite
2	46	Program change future notes	0 (silence)	Overwrite
1	0D	Timbre space X dimension	0	Add
1	0E	Timbre space Y dimension	0	Add
1	0F	Timbre space Z dimension	0	Add

Table 3. Higher-Order Messages

No. of Data Bytes	*ID (Hex)*	*Meaning*	*Default*	*Combining Rule*
11	C0	Modulation info block	None	See below
3	81	Modulation rate	0	See below
2	47	Modulation depth	0	See below
n	C1	Modulation table	N/A	N/A
n	C2	Segment info block	None	See below
n	C3	Segment table	N/A	N/A

the sections below. The last column of each table, the "combining rule," indicates the way that values of these parameters interact when sent to different levels of the address hierarchy; this is described in the section, "How the Levels Interact," below. Note descriptor ID zero is illegal.

ZIPI instruments are not required to respond to every one of these messages, although they are encouraged to respond to as many of them as possible. A note descriptor's byte length is encoded as part of the ID number, therefore, receiving synthesizers can ignore note descriptors that they do not

Table 4. Housekeeping Messages

No. of Data Bytes	*ID (Hex)*	*Meaning*	*Default*	*Combining Rule*
1	10	Allocation priority	0	Multiply
3	82	New address	N/A	N/A
n	C4	Overwrite	N/A	N/A
n	C5	Query	N/A	N/A
n	C6	Query response	N/A	N/A
n	C7	Text/comment	N/A	N/A

Table 5. Time-tagging Messages

No. of Data Bytes	*ID (Hex)*	*Meaning*	*Default*	*Combining Rule*
4	83	Time tag	N/A	N/A
4	84	Desired minimum latency	0	N/A

Table 6. Undefined MPDL Synthesizer Control Parameters

No. of Data Bytes	*ID (Hex)*	*Meaning*	*Default*	*Combining Rule*
1	11-1F	Undefined 1-byte controllers	Undefined	Undefined
2	48-5F	Undefined 2-byte controllers	Undefined	Undefined
3-4	85-9F	Undefined 4-byte controllers	Undefined	Undefined
n	C8-DF	Undefined n-byte controllers	Undefined	Undefined

implement, skip the correct number of bytes, and then examine the next note descriptor in the packet.

Logically, all of the note descriptors in a single MPDL packet apply to the same instant of time. This means that the order of note descriptors within a MPDL packet does not matter.

Table 7. MPDL Articulation Types

High-Order Bits	*Articulation Type*
11	Trigger
10	(Not used)
01	Reconfirm
00	Release

Articulation

If the note descriptor ID is *articulation*, the two high-order bits of the data byte specify one of the three articulation types that are defined in Table 7. *Trigger* messages start a note; the new note will have any parameters that were set for that note before the trigger message was sent. Pitch and loudness are not part of the trigger message, so before sending the trigger message, or in the same MPDL packet as the trigger message, you should set pitch and loudness to the desired levels. Remember that the order of note descriptors within an MPDL packet does not matter, so pitch, loudness, and trigger could come at any position in the packet.

The note then sounds until a *release* message is received. If a new trigger message comes before the release message comes, the note reattacks with no release. This is useful for legato phrasing. Figure 3 shows what might happen to the amplitude of a tone as a note receives two trigger messages and then a release message.

Figure 3. Amplitude of a tone being articulated by MPDL.

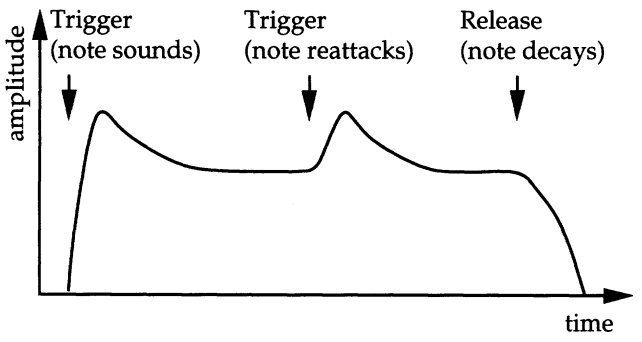


Table 8. Types of Release Messages

Low-Order Bits	Behavior
000001	Release the note naturally
000010	Instantly silence the note
000011	Release the note naturally, unless it is still in the attack portion of the tone, in which case complete the attack portion and then release naturally

A note retains its parameters after a release message; receipt of a new trigger message will articulate a new note with the same parameters as before. (The section below on the "Allocation Priority" message describes when a note loses its parameters.)

Keep in mind that the default timbre is silence, so unless you have sent a program change message that affects a note, triggering it will have no effect. The default pitch for a note, i.e., the pitch of a note that has never had its pitch set, is middle C, and the default loudness is *mezzo forte*, so if you set the timbre of a note and trigger it, the synthesizer will play a *mezzo forte* middle C.

The *reconfirm* message is a reminder from the controller that it thinks the note should still be sounding. It is not needed under most circumstances because notes that have been triggered but have not yet been released are assumed to be still sounding. In cases of network failure, however, this message can be used to reestablish the notes that should still sound. If a network failure occurs, all controllers should reconfirm all sounding notes to the synthesizers. Synthesizers that do not receive a reconfirmation within a certain amount of time should shut off those notes, assuming that the release message was lost.

These three messages can be understood in terms of the gate and trigger bits that were used to articulate synthesizers in the "old days." The high-order bit is like the trigger bit, and the next bit is like the gate bit. Thus, trigger means asserting both the gate and trigger bits, release means deasserting the gate bit, and reconfirm is like asserting the gate bit but not the trigger bit.

The remaining 6 bits of the articulation data byte specify exactly what kind of articulation occurs. In music, "articulation" can mean a lot more than "on and off." There are a large number of instrument-specific articulation styles, e.g., hammer-ons for guitar, lip slurs for brass instruments, and heavily tongued attacks for reed instruments (Piston 1955; Blatter 1980). The problem with encoding these articulation types is that they are meaningful only in the context of certain instruments; it is difficult to say how to implement a hammer-on on a clarinet. Therefore, we are working to define abstract articulation categories, expressed in a way that does not refer to a particular instrument. We hope that these will be in a future version of the MPDL as the possible values for the remaining 6 bits.

For the release message, we have specified three possible behaviors, as shown in Table 8.

Controlling Pitch

In MIDI, pitch is specified by key number, which in practice almost always maps to equal-tempered semitones. For finer resolution, e.g., to convey vibrato or pitch-bend or to use alternate tunings, it is necessary to use a pitch-bend controller, which provides an additional 7 or 14 bits of precision. Attempts have been made to retrofit a better tuning system onto MIDI (Scholz 1991), but nothing has been officially adopted (Rona 1991).

In MPDL, pitch information is given by a 16-bit

logarithmic pitch word. The first 7 bits are the nearest MIDI note number, and the remaining 9 bits are fractional semitonal values in units of about 0.2 cents. The binary word *nnnn nnn*1 0000 0000 is equal to MIDI note number *nnn nnnn*. The word *nnnn nnn*0 00 00 0000 is MIDI note *nnnn nnn*, a quarter tone flat, and *nnnn nnn*1 1111 1111 is MIDI note *nnnn nnn*, a quarter tone sharp. (The rationale is that MPDL-to-MIDI pitch conversion thus requires truncation instead of rounding.)

Frequency in Hz is an alternate way to specify pitch, as a 32-bit fixed-point number. The first 16 bits are the number of Hertz, from 0 to 65,535, and the last 8 bits are the fractional part, giving a resolution of better than 0.000016 Hz. Receipt of a Hertz-frequency message overwrites the previous pitch message and vice versa; mixing the two kinds of messages is discouraged.

Loudness

The *loudness* parameter corresponds to our subjective impression of intensity. The units of loudness are musical dynamic markings; the interpretations of loudness are given in Table 9.

Loudness is influenced not only by amplitude, but also by the temporal and spectral characteristics of the sound (Moore 1989; Pierce 1992). Performing musicians are usually quite skilled in trading off the various acoustic parameters that contribute to the perception of loudness. For example, they are able to adjust a bright oboe note to be the same musical dynamic or loudness as a mellow French horn tone. The key idea behind the MPDL loudness parameter is that if one sends the same value to notes played on different instruments, they will sound at the same subjective level. For the loudness parameter to function properly, the various instruments on a given synthesizer must be carefully matched throughout their pitch and dynamic ranges so that a given loudness value consistently produces the same musical dynamic.

Admittedly, as loudness is defined in terms of a subjective impression, there will be some differences of opinion among different listeners. What we are asking for is a good approximation to matched loudnesses. Good voicing practice on MIDI synthesizers already points in this general direction in that different voices are adjusted so that a given MIDI velocity produces comparable loudness impressions.

Table 9. Interpretations of MPDL Loudness Values

MPDL Loudness Value (Hex)	*Musical Dynamic*
0000	pppp
1000	ppp
2000	pp
4000	p
6000	mp
8000	mf
A000	f
C000	ff
E000	fff
FFFF	ffff

Amplitude

Amplitude describes the overall gain of a sound. Increasing it is like turning up the level on a mixing board. Changing a sound's amplitude does not change its timbre. We might describe here the result of changing the amplitude of instrument timbres already adjusted for loudness. High amplitude with low loudness on a piano sound would give the effect of a pianist playing very softly through a loud amplification system. Conversely, low amplitude with high loudness would sound like someone banging hard on a piano with the sound played very quietly through speakers.

Even/Odd Harmonic Balance

Most pitched sounds can be thought of as a collection of harmonics or overtones, which are sine waves spaced equally in frequency. The first harmonic is defined to be at the fundamental frequency, the second harmonic at twice the fundamental frequency, the third harmonic at

three times, etc. (Pierce [1992] contains a good explanation of this.) The *even/odd harmonic balance* parameter is a measure of the overall amplitude of the first, third, fifth, etc., harmonics versus the overall amplitude of the second, fourth, sixth, etc. Listening to only the odd harmonics of a tone gives a sound something like a square wave; listening to only the even harmonics sounds somewhat similar to a note played an octave higher with the same spectrum.

This may seem like a strange parameter, but it is actually quite meaningful. Many acoustic musical instruments have different balances of odd and even harmonics, and this balance can fluctuate dramatically over the time course of the note. This even/odd balance and its variation over time have potent effects on tone quality (Krimphoff, McAdams, and Winsberg 1994). For example, the timbral difference that comes from picking a guitar at different points on the string has a lot to do with this balance.

Most synthesis algorithms make it easy to manipulate this balance. In physical modeling synthesis, one can make models of open and closed tubes or strings plucked or bowed at various critical points along the string. Frequency modulation (FM) synthesis provides a natural mechanism by mixing simple FM patches with differing carrier-to-modulator ratios. Waveshaping synthesis has its odd and even distortion function components, and additive synthesis affords direct control over the spectral content. Even subtractive synthesis with poles and zeros allows for tight control over the even/odd balance (Smith 1993).

Pitched/Unpitched Balance

Many sounds can be thought of as containing a pitched portion and an unpitched or noise portion (Serra and Smith 1990). The sound of a piano, for example, consists of a "thud" made by the sound of the hammer hitting the string, along with the pitched sound of the vibrating strings. The *pitched/unpitched balance* parameter measures the relative volume of these two portions of a tone.

Inharmonicity

Most pitched musical sounds are harmonic or nearly harmonic; that is, the partial frequencies are nearly exact integer multiples of the fundamental. For some instruments, however, these ratios are inexact. Pianos, bells, tympani, and other instruments have partials whose frequencies are not always integer multiples of the fundamental and are sometimes nowhere near integer multiples.

The *inharmonicity* parameter describes the amount that partials deviate from perfect harmonicity. Inharmonicity is signed; zero, the center value, means "the usual inharmonicity of the sound." As there are many ways to produce a deviation from the harmonic series, the interpretation of this parameter may vary when it is nonzero. Some synthesizers might interpret negative inharmonicity to mean "more perfectly harmonic than the original sound" and positive inharmonicity to mean more inharmonic. Others might interpret negative inharmonicity as "squeeze," causing the partial frequencies to be spaced closer together than usual, with positive inharmonicity as "stretch," making the partial frequencies spaced farther apart than usual. Other synthesizers might take the absolute value of this parameter, ignoring the sign.

Controlling a Note's Position in Space

A sound's position in 3-D space can be described in either rectangular or polar form. For rectangular form, there are three *pan* variables: left/right, up/down, and front/back. A value of hexadecimal 00 means "panned all the way to one direction," which would be left, up, or front. Hexadecimal FF means "panned all the way to the other direction," and hexadecimal 80 means "center." Synthesizers should implement equal energy panning.

For polar coordinates, there is *spatialization*, which describes the distance of the produced sound from the listener and the direction from which the sound comes. Psychoacousticians have noted that human spatial perception is described by separate mechanisms for angular orientation

and distance (Blauert 1982), so spatialization is perceptually more meaningful than pan. The azimuth is the angle between the sound source and "backward," in a horizontal plane. Hexadecimal 0000 means "from behind," 4000 means "to the left," 8000 is "directly ahead," and C000 is "to the right." The elevation is the remaining dimension in polar coordinates, going from hexadecimal 0000, meaning "down," to 8000 for "on the same level," to FFFF for "up."

Finally, there's a way to control directly the amplitude that a note has out of each of the outputs of a ZIPI timbre module. In the most general case, the synthesizer has up to 256 separate outputs, and any note can be directed to any output with any volume. A particular note might be coming mostly out of outputs 5 and 8 but also a little bit out of 1, 2, and 4, for example. The *multiple output levels* note descriptor lets you set these amplitudes. It has 2 data bytes; the first selects one of the synthesizer's outputs, and the second sets a level for the given sound at that output.

ZIPI controllers should not mix these three types of positioning information; they are meant to be three separate systems to specify the same thing. (That is to say, the low-level effect of pan or spatialization messages will be to control how much of each sound comes out of each of the synthesizer's outputs.)

Program Change

Program change determines which program or preset the synthesizer should use as an instrument—trumpet, bagpipes, timpani, etc. Patch 0, the default, is defined to be silence. Presets 1 through 127 should follow the General MIDI assignments (MMA 1991). Because the program is specified by two bytes (giving 65,536 possible values), the default set can be largely expanded while keeping a large segment free for arbitrary use. This message might cause a synthesizer to choose a set of sample files, load FM synthesis parameters, or do anything else.

It is expected that synthesizers will have some restrictions on the use of this parameter. For example, because each active patch will probably take up a certain amount of memory, processing power, or other resources, there might be a maximum number of different patches that can be selected at once. Also, it might take a certain amount of time to set up a new timbre, so this message should be sent before the new timbre actually has to sound.

There are two reasonable behaviors for a program change message with regard to the timbre of the notes sounding at the time the message is received. The *program change immediately* message requests that all currently sounding notes change their program to the new one. The *program change future notes* message requests that currently sounding notes retain their programs but that newly articulated notes use the new program. For example, if note 3 of some instrument is sounding a flute and that instrument receives a program change future notes message to trumpet, note 3 will continue to play a flute until it is released. When note 3 receives another trigger message, it will sound as a trumpet.

Controlling Timbre

Though timbre is a complex and subjective attribute of musical tones, there are some aspects of timbre about which there is considerable agreement among both musicians and psychoacousticians (Risset and Wessel in press). We have chosen to specify these more agreed-on aspects of timbre in MPDL. They are brightness, roughness, and attack character.

The impression we have of a tone's *brightness* corresponds strongly to the amount of high-frequency content in the spectrum of the sound. One good measure of this is the "spectral centroid," which is the average frequency of the components of a sound, weighted by amplitude. At a given pitch and loudness, an oboe sounds brighter than a French horn, and a look at the spectrum of each tone shows the oboe to have more high-frequency components than the French horn. Computing the spectral centroid would show the oboe to have a higher value than the French horn. Different syn-

thesis algorithms employ different procedures to manipulate a tone's brightness, but almost all provide a rather direct path to control this feature. In FM, increasing the modulation index increases the high-frequency content. Moving the cutoff frequency of a low-pass filter upward has a similar effect. With additive synthesis, detailed control of the spectral envelope is provided by direct specification of the amplitudes of the partials.

Roughness has a direct intuitive meaning. Low values would correspond to very smooth tones, whereas high values would be rough. Roughness might, for example, correspond to overblow on a saxophone. A considerable body of psychoacoustics research shows it to be related to amplitude fluctuations in the tone's envelope. When the envelope of a tone fluctuates at a rate of 25 to 75 Hz and the depth of this amplitude fluctuation approaches 10 percent of the overall amplitude, the sound quality becomes very rough. Beats among partials of a complex tone can produce such fluctuations and give a roughness to the sound. As with other timbral parameters, there are a variety of ways to implement roughness control in different synthesis algorithms.

Attack character describes, intuitively, how strong of an attack a note should have. As a first approximation, it might correspond to the attack rate in a traditional attack-decay-sustain-release envelope. It might also correspond to a louder maximum volume value during the attack, a noisier attack, a brighter attack, etc. The value for this parameter is "sampled" at a note's trigger time; whatever value this parameter has when a note is triggered specifies the attack character for the note. Changing attack character in the middle of a note does not require the synthesizer to change anything about that note.

Moving in a Timbre Space

A timbre space (Wessel 1985) is a geometric model wherein different sound qualities or timbres are represented as points. Similar sounding timbres are proximate, and dissimilar ones are distant from one another. A timbre space is a fairly general model for representing the important perceptual relationships among different timbres and provides an intuitive control scheme based on interpolation. Timbral control is exercised by making trajectories in the space.

MPDL provides for up to 3-D timbre spaces. For higher-dimensioned timbre spaces, it is always possible to use MPDL's undefined note descriptors.

The timbre space *x*, *y*, and *z* coordinate controls are like other continuous controller note descriptors. The contents of the space and the scheme for interpolation are part of the patch that is selected by the program change note descriptor.

Higher-Order Messages

MPDL provides for modulation messages. Vibrato is an example of pitch modulation, in which the pitch of a note varies around the central pitch of the note. You can think of the vibrato as a function of time (e.g., a triangle or sine wave) modifying a default value (i.e., the underlying pitch that the vibrato is around). In the MPDL, modulation means that an additive offset or a multiplicative scale factor can be applied to a parameter value as a function of time. Any parameter, not just pitch, can vary.

It would be possible to modulate any parameter explicitly by sending a stream of explicit parameter values. For example, a good ZIPI violin would have fine-grained enough pitch detection to notice the vibrato played by the musician and would send it to a ZIPI synthesizer as a series of very accurate pitches all close to a central value.

On the other hand, consider a computer program playing a symphony on a collection of ZIPI timbre modules. Individually specifying the vibrato of each stringed instrument via frequent updates of the pitch parameter would be impractical in terms of the amount of data transmitted. Instead, the MPDL has a way to specify a table or function to give values for a parameter over time. A single message says, for example, "start a triangle-wave-shaped modulation of the pitch parameter with depth 10 cents and frequency 6 Hz." After that message is sent, the receiving device computes the

Table 10. MPDL Modulation Functions

Number	*Function*
0	No modulation (i.e., f(t) = constant)
1	Sine wave
2	Square wave
3	Saw-tooth wave
4	Triangle wave
5	Random
6–255	User-defined tables

Table 11. MPDL Segment Functions

Number	*Function*
0	Linear
1	Exponential concave
2	Exponential convex
3	S-shaped (e.g., sigmoid or logistic)
4–255	User-defined tables

subsequent values of that parameter with no additional ZIPI messages required. Furthermore, because the depth and frequency of a modulation are MPDL parameters like any other, they can be updated by a stream of control values.

Segments provide a similar high-level control but have semantics more like that of decrescendo. With segments, one can say, for example, "start an exponential decay of loudness that will go to pianissimo in 1.6 sec." Here, what is specified is a parameter (loudness), a target value (pianissimo), a time to reach that target (1.6 sec), and a shape for the function to use on the way to that value (exponential).

Modulation is for signed indefinite repeating functions moving back and forth around a center value; the possible modulation functions are given in Table 10. Segments are for functions with a "goal"; they are used to get from one parameter value to another in a set amount of time. The possible segment functions are given in Table 11.

Synthesizer manufacturers may implement these functions in a variety of ways as long as the functions behave as specified. (Except for "random," these functions could be computed by table lookup from a 256 by 1-byte table, sample values for which will be made available.)

Table 12. Format of Modulation Information Block Message

Byte No.	*Contents*
1	Note descriptor ID of parameter being modulated
2	Which function (from Table 10)
3 + 4	Modulation rate, from –255 to 255 Hz, with .008 Hz resolution
5 + 6	Modulation depth
7 + 8	Loop begin point
9 + 10	Loop end point

There should also be user-settable tables that can be loaded with any values over the network. Tables can contain 1-, 2-, 3-, or 4-byte numbers and must have a size that is a power of 2. Tables should be able to be at least 256-byte by 1-byte, but synthesizer manufacturers are encouraged to provide larger ones. Synthesizers are encouraged to have as many user-settable tables as possible. Synthesizers should interpolate the values in these tables when necessary.

To modulate a note descriptor, you must specify six things. The MPDL's *modulation info block* message consists of the values for these six parameters, all in a single message. The format of this message is shown in Table 12.

When the modulation rate is negative, it means that the synthesizer is to read backward through the table (or the equivalent if the function is implemented in a manner other than a table). This is useful for changing the shape of a sawtooth wave, for example.

The loop begin and end points specify which portion of the table is read through during the course of modulation. Typically, they will specify the entire table, but they can be used, for example, to alter the duty cycle of a square wave.

Instead of sending a new modulation info block, you can send a *modulation rate* or *modulation depth* message to specify the rate or depth of the modulation for a particular note descriptor with-

out updating any of the other parameters.

To send your own modulation table as an MPDL message, use a *modulation table* message. The first data byte is the number of the table you are setting, which should not conflict with any of the predefined functions. The second byte gives the size of the values in the table, in bytes, e.g., 1, 2, or 4. The remaining data bytes are the actual contents of the table. (Because MPDL note descriptors include their own lengths, the size of a transmitted table is unambiguous.) It does not matter what note address a modulation table message is sent to.

Function zero, "no modulation," stops modulation of a given parameter, freeing all resources used for the modulation. ZIPI controllers should explicitly turn off modulation of a parameter rather than just setting the depth to zero, so that the synthesizer will be able to free these resources.

Modulation scales whatever other changes happen to a particular parameter, using the same combining rule as with messages sent to various levels of the address space hierarchy (see below). Exponential loudness decay on a piano sample will scale the natural exponential decay from the real piano. Imposing sinusoidal vibrato on a patch with built-in vibrato will simply combine through. Therefore, users who want to modulate pitch explicitly should probably turn off their synthesizer's built-in vibrato. (Of course, if the vibrato is part of a sample, this is not so easy.)

The *segment info block* message gives all of the parameters necessary to apply a segment to a parameter, as shown in Table 13. The starting value for the segment is not specified. Whatever value the given parameter currently has is the start point. This makes it much easier to chain segments together. If you would like the value of the given parameter to jump to a new value and then go gradually to a second value, just precede the segment info block message with an explicit value for that parameter.

The function begin and end points are like the loop begin and end points for modulation; they specify a portion of the function to use. The target value must be a legal value for the given note descriptor; the byte length depends on the note descriptor.

The segment chaining byte indicates how the given segment fits in with other segments. If the value is 0, it means that this segment should override any other segments that are affecting the given parameter for the given address, "taking over" control of that parameter. If the value is 1, it means that the synthesizer should put this segment at the end of a queue of segments for this parameter, to take effect after the current segment finishes. This allows complex envelopes to be built out of these segments.

Table 13. Format of Segment Information Block Message

Byte No.	*Contents*
1	Note descriptor ID of parameter being manipulated
2	Which function (from Table 11)
3	Segment chaining byte
4 + 5	Time to reach desired value, in msec
6 + 7	Function begin point
8 + 9	Function end point
10–n	Target value

The *segment table* allows you to specify your own table for use by a segment. Its syntax is exactly like that of the modulation table message.

The segment mechanism is exactly equivalent to sending a stream of values for a parameter, so the old value of that parameter is overwritten with values produced by the segment, and when the segment is done, the last value produced by the segment is the current value of that parameter. What happens when an explicit value is set for the parameter while a segment is still running? That segment (and all segments in the queue) terminate, and the parameter gets the explicitly set value. In other words, sending a parameter change message is equivalent to killing a segment.

Housekeeping

Allocation priority describes the importance of a note in case a synthesizer runs out of resources and has to choose a note to turn off. As an example, important melodic notes might have a higher prior-

ity than sustaining inner voices in thick chords. In MIDI, there is no way to tell the synthesizer which notes are important, so MIDI synthesizers typically just turn off the oldest sounding note, which is not always desired (Loy 1985).

Zero is the default priority for a note, but sending any message to a note with zero priority automatically increases that note's priority to one. Setting a note's priority to zero means "reset." If the note is sounding, receipt of a zero priority silences it immediately. It also resets all of the parameters associated with the note, as if no message had ever been sent to the note. This allows a synthesizer to deallocate the memory used to remember parameter values that had been sent to that note. Therefore, devices that send MPDL information, such as controllers and sequencers, should send zero values for allocation priority when they determine that a note's parameters will no longer need to be stored.

Setting an instrument's priority to zero resets the entire instrument; all of the notes inside the instrument shut off, all of the parameters associated with the instrument are reset, and all of the parameters associated with all of the notes inside the instrument are reset. Likewise, allocation priority of zero for a family shuts off all notes in all instruments of the family, resets all parameters associated with that family, and resets all parameters of all instruments and notes inside the family.

New address is not a message addressed to a note; it is just a way to use ZIPI bandwidth more efficiently. Imagine that you would like to update certain timbral parameters for a whole group of notes. For example, a ZIPI guitar might provide continuous information about the pitch, loudness, and brightness of each of the six strings. One way to do this would be to send six ZIPI packets, one for each of the strings. However, this is wasteful of network bandwidth because each separate ZIPI packet has 7 bytes of overhead.

It would be better to use the new address message. The MPDL portion of a ZIPI message must start with an address to which further note descriptors apply. However, if one of the note descriptors is "new address," it specifies a different address to which subsequent note descriptors apply. For example, for the packet shown in Figure 4, the pitch and pan messages are for note 1 of instrument 3 of family 1, while the amplitude message is for note 2 of instrument 2 of family 1, and the brightness message is for instrument 1 of family 1.

Overwrite has to do with the interaction of the three levels of the address hierarchy. See the section below, "How the Levels Interact," for an explanation of this message.

Querying a Synthesizer

Because ZIPI communication can always be two-way, there is a mechanism for asking questions of a synthesizer. The *query* message asks some question; it is a request for the synthesizer to respond with a *query response* message answering the question.

The first data byte of the query message is the question ID, as given in Table 14. The remaining data bytes qualify the question. For example, for question 4 ("Do you respond to the given note descriptor?") there is one additional data byte, a note descriptor ID. (We also rely on the data link layer to include the network address of the querying device as the return address for the response.)

The first byte of the query response message is the ID of the question being answered, and the next bytes are the qualifying data that were asked in the question. The remaining bytes are the actual response to the question. For example, a sequencer program might ask a synthesizer whether note 2 of instrument 1 of family 1 is sounding. It would select the given address as the address of an MPDL packet, then include a query note descriptor. The first data byte would be 2 ("What's the combined value of this MPDL parameter for the given note?"), and the second data byte would be 1 (the note descriptor ID for articulation).

The synthesizer's response would first select note 2 of instrument 1 of family 1 as the address, then include a query response message. The data bytes of the query response message would be 2 (the question ID), 1 (the note descriptor ID for articulation), and then something like 11000000 (an

Figure 4. Example of a MPDL packet with "new address" message.

Note Address note 1, inst 3, family 1	Pitch F#5	Pan 25% to the left	New Address note 2, inst 2, family 1	Amplitude half normal	New Address inst 1, family 1	Brightness twice normal

Table 14. MPDL Query Question IDs

ID	*Meaning*
1	What is the value of this MPDL parameter at the given level of the address space?
2	What is the combined value of this MPDL parameter for the given note?
3	Please send me a menu of all patch names.
4	Do you respond to the given note descriptor?
5	How many voices of polyphony do you have?
6	How many voices of polyphony do you have left?
...	Undefined
255	(Indicates that the next two bytes specify the question ID)

articulation value for "trigger") or 00000000 (an articulation value for "release").

Comments

The comment note descriptor's data bytes are ASCII characters; they have no meaning to a synthesizer. In recorded files of MPDL control information (see Appendix B), one might want to add comments to certain note descriptors, such as "Start of second movement." For these comments to be incorporated consistently with the other messages, they are part of the MPDL itself. (This implies that when an MPDL file is transmitted via ZIPI, the comments will remain intact.)

Time Tags

No network can provide instantaneous transmission of information. In ZIPI, network latency (the amount of time it takes a packet to be delivered) will be very small under normal conditions, usually in the range of 0.5 to 5 msec. Variability in network latency, sometimes called "jitter," can be worse than the latency itself if music is involved, however. It is less annoying to play a synthesizer that delays every attack by exactly 10 msec than one that delays every attack by a random amount of time between 3 and 9 msec (Moore 1988.)

Therefore, the MPDL has a way to impose a fixed minimum delay on each packet. If it arrives earlier than expected, the receiving synthesizer can wait a short amount of time before carrying out the instructions in packet. (Anderson and Kuivila [1986] discuss this principle, but in the context of algorithmic composition rather than networking.) To accomplish this, there is a way to put a time tag in each MPDL packet. The *time tag* note descriptor indicates the exact time that the packet applies to, according to the sender's clock. The *desired minimum latency* note descriptor tells a ZIPI device what minimum latency to impose on all time-tagged network packets.

What does the receiving device do with this information? For now, let us assume that the receiving device has its own clock, which has been synchronized very closely to the sender's clock. (ZIPI's data link layer can do this synchronization.) The receiving device can compare an incoming message's time stamp to the current value of its own clock to see how much delay has occurred already. If this is less than the requested minimum latency, the receiving device simply buffers the message for the appropriate amount of time until the minimum latency has been reached. (Dannenberg [1989] gives a variety of efficient algorithms for this.) If the delay already incurred is greater than the requested minimum, the message is already too late, so the receiving device should deal with it immediately.

What if the two clocks are not synchronized? For example, if the MPDL runs over FDDI or Ethernet, no system-wide clock synchronization will be provided. In that case, there are still algorithms to impose a minimum latency on network packets, although they are more complicated and slightly less effective.

These time tags are useful in other situations

Figure 5. A musician controlling a synthesizer.

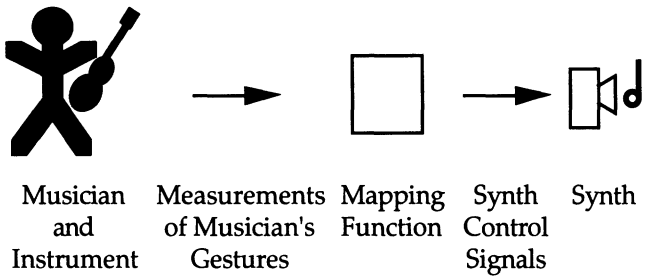


Figure 6. A musician controlling a synthesizer: instrument providing the mapping function.

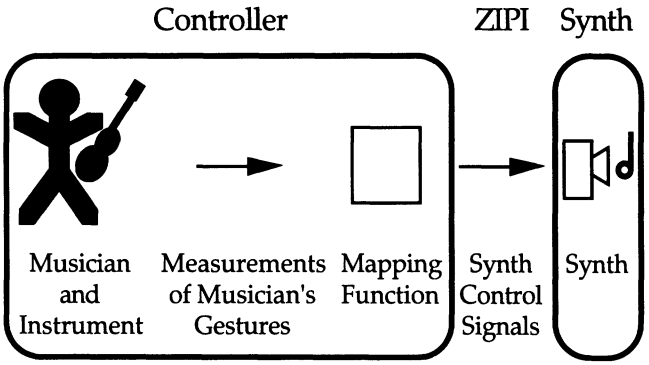


Figure 6

Figure 7. A musician controlling a synthesizer: mapping function done elsewhere.

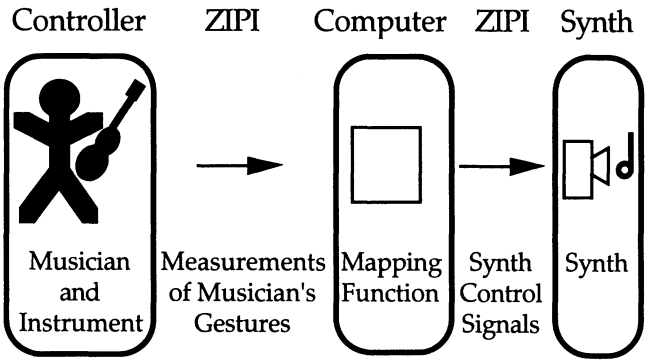


Figure 7

besides real-time control of a synthesizer. For example, take the case of recording into a ZIPI sequencer. MIDI sequencers must record the time that each message was received, in order to store timing information in the recorded file. The time that the sequencer receives the information is the time that it was played, plus some network delay. In ZIPI, if a controller time tags outgoing messages, the network delay will have no effect on the recorded sequence.

Sequencer playback benefits from time tags also. If the sequencer program uses the time stamps stored in the file and requests a sufficiently large minimum delay, all of the delays incurred by the sequencer, including disk latency, processing time, and network delay, can be eliminated as well.

As another example, imagine the data from some ZIPI controller being fed into a computer, which applies some transformation to the data and then gives it to a synthesizer. The computer program does not have to wait for a fixed latency before processing each controller message; it can start manipulating the data as soon as it arrives, but if it preserves the time stamps produced by the controller, rather than producing new ones, the jitter introduced by the transformation process can be eliminated at the synthesis end.

Time stamps are 4 bytes, using 50-μsec units, giving a range of 2.5 days expressible in an MPDL time stamp. The desired minimum latency note descriptor has the same data format.

Controller Measurements

Real-time control of an electronic musical instrument involves three stages: measuring the musician's gestures—which key was struck, how much air pressure was used, where the violinist's fingers were on the fingerboard, etc.; deciding how these gestures will translate into the electronic sounds produced; and synthesizing a sound. Figure 5 demonstrates these stages. In the MPDL, we draw a distinction between the first arrow, "measurements of musician's gestures," and the second arrow, "synthesizer control signals." All of the parameters listed in Table 2 above are in the second category; they are descriptions of sound that tell a synthesizer what to do.

Typically, ZIPI controllers will provide both the measurements of the gestures and a way to map those gestures onto parameters required to produce a sound. For example, a ZIPI violin might measure the bow's distance from the bridge and use it to determine brightness. That would divide the above picture according to Figure 6. In this setup, the ZIPI instrument sends synthesizer control parameters, such as the ones described above.

Table 15. Controller Measurement Parameters

Size	*ID (Hex)*	*Meaning*
1	3f	Key velocity
1	3e	Key number
2	7f	Key pressure
2	7e	Pitch-bend wheel
2	7d	Mod wheel 1
2	7c	Mod wheel 2
2	7b	Mod wheel 3
1	3d	Switch pedal 1 (sustain)
1	3c	Switch pedal 2 (soft pedal)
1	3b	Switch pedal 3
1	3a	Switch pedal 4
2	7a	Continuous pedal 1 (volume)
2	79	Continuous pedal 2
2	78	Continuous pedal 3
2	77	Continuous pedal 4
1	39	Pick/bow velocity (signed)
1	38	Pick pressure
1	37	Pick/bow position
2	76	Fret/fingerboard position
1	36	Fret/fingerboard pressure
1	35	Wind flow or pressure (breath controller)
1	34	Embouchure (bite)
2	75	Wind controller keypads
1	33	Lip pressure
2	74	Lip frequency (buzz frequency for brass)
1	32	Drum head striking point X position (rectangular coordinates)
1	31	Drum head striking point Y position (rectangular coordinates)
1	30	Drum head striking point distance from center (polar form)
1	2f	Drum head striking point angle from center (polar form)
2	73	X position in space
2	72	Y position in space
2	71	Z position in space
2	70	Velocity in X dimension
2	6f	Velocity in Y dimension
2	6e	Velocity in Z dimension
2	6d	Acceleration in X dimension
2	6c	Acceleration in Y dimension
2	6b	Acceleration in Z dimension

Musicians will not always want to use the mapping capabilities provided by their controller, however. For example, some people will want to write their own computer programs, e.g., in the Max language (Puckette 1991), to determine complex mappings. One might want to control the loudnesses of four families by finger position on the neck of a ZIPI violin. To support user-defined mappings, we recommend that ZIPI controllers be able to send their raw physical measurements directly, without mapping them onto synthesizer control parameters. In other words, it should be possible to turn off the software in the controller that is mapping the physical gestures into control information, sending the measurements of those gestures as uninterpreted data. That would divide the picture as shown in Figure 7. The user interfaces of ZIPI controllers should provide a way to switch between these two modes; sometimes the controller should do the mapping itself, and sometimes the controller should send out the "raw" data.

Controller measurements are just another kind of note descriptor, listed in Table 15. Note that these measurements take up the higher ID numbers, synthesizer control parameters take up the lower ID numbers, and the middle numbers are undefined. (As with the note descriptors for synthesizer control, we expect to define a few more and leave most of them free for future specification.)

Note that not all ZIPI controllers will work by physically measuring a musician's gestures. Another class of ZIPI controllers consists of acoustic instruments whose sound output is measured and analyzed by a computer and converted into control information. In this kind of instrument, for example, a real-time pitch tracker would examine the sound produced by a flute and convert it to MPDL pitch messages. Digital signal analysis

could be used to compute the spectral centroid of the flute's sound, which would produce MPDL brightness messages. In this case, the measurements of the musician's gestures are done with the same musical control parameters that the mapping function produces to control a synthesizer. (In this case, however, the identity function is a perfectly good mapping function; it will cause the synthesizer to mimic timbrally what the musician is playing, which will probably be the most commonly desired situation.)

How the Levels Interact

What happens if you send an amplitude of 1 to a note, then an amplitude of 10 to the instrument containing that note, and then an amplitude of 100 to the family containing that note? What is the actual amplitude of the sound produced?

There are four different ways to combine parameter values passed to different levels of the hierarchy. Each parameter uses one of these four rules. They are "and," "multiply," "add," and "overwrite." The "Combine" column in the tables of note descriptors (Tables 2 through 6) tells which of these four rules each parameter uses. Only articulation uses the "and" rule, which is described below.

Most parameters use the "multiply" rule, meaning that each level of the hierarchy (notes, instruments, and families) stores its most recent value for the parameter, and the actual value that comes out is the product of these three numbers. Amplitude is an example of a parameter with this rule. If two notes of an instrument have amplitudes of 20 and 10, they will have a relative amplitude ratio of 2:1, no matter how high or low the instrument's amplitude gets.

Note that what are being multiplied are scale factors for a base value and that the base value depends on the particular patch being played on the synthesizer. A flugelhorn will naturally be less bright than an oboe, so the midscale brightness value for a flugelhorn will produce a much less bright sound than the midscale brightness value for an oboe.

The "add" rule is just like the "multiply" rule except that the three values for the parameter are added together instead of multiplied together. Coordinates in a timbre space are combined in the MPDL with this rule. The combining rule for pitch is a special case of the "add" rule; pitch is taken as an offset from middle C, and the offsets accumulate additively. If a family receives a pitch message of hexadecimal 7F00, which would be middle C#, the effect will be to transpose everything played by that family up a half step.

The last rule is "overwrite." For these parameters, the instrument and family do not store their own values for the parameter. Instead, a message sent to an instrument or family overwrites the values of that parameter for each active note of the instrument or family. Program change is a parameter with this rule. Because there is no easy way to combine three synthesizer programs into a single patch, a program change message received by a family sets the program of all the notes of that family.

The "And" Rule for Articulation

The name of this rule comes from Boolean logic. In this context, it means that a note only sounds if it has been triggered at the note level, the instrument level, and the family level. By default, everything is triggered at the family and instrument levels, so sending a trigger message to any note turns on that note. That is the normal case that people will use most of the time.

It is possible to turn off all of the notes in a family just by sending a release message to a family. If this happens, the previously sounding notes still remember that they are turned on at the note and instrument levels. Therefore, if you retrigger the family, those notes will sound again.

Here is an example of how to take advantage of this. First, send a release message to an instrument, preventing any notes from sounding on that instrument. Then, send pitch and trigger messages to a group of notes in that instrument to form a chord. Those notes do not sound yet because the instrument is switched off. Finally, you can send a trigger message to the instrument, which will trigger all of the notes of that instrument, playing the

chord you set up earlier. Now you can turn the chord on and off by sending articulation messages to the instrument. If you want to add or delete notes from the chord, send articulation messages at the note level.

Sending a Message to All Families

Sending a message to the "all families" address is an abbreviation for sending the message repeatedly to families 1 through 63. It is not a fourth level of the hierarchy in the sense of storing yet another parameter value that must be added or multiplied through. Instead, it changes the values stored for each of the 63 families.

Overwriting a Large Group of Values

Usually, the multiply or add rule does what one would want; it makes sense to have the oboes louder than the flutes by the same relative amount no matter how quiet or loud the wind section gets. Occasionally, however, the overwrite rule is desired even for parameters that typically use the multiply or add rules. For example, suppose all of the instruments in a family are playing different pitches, but now you want them to play in unison. If you send a pitch message to the family, it will transpose all of the instruments of the family, leaving their relative pitches the same. Instead, you want a way to say "individually set the pitch of each note of each instrument in this family to middle C."

The "overwrite" note descriptor handles cases like this. Its data bytes consist of a note descriptor ID, which specifies a parameter to be overwritten, and some data for that parameter, which specifies a new value for it. If you send overwrite to a family, it sets the values for every instrument in that family, throwing away each instrument's old value for the parameter. If you send an overwrite message to an instrument, it sets the values for each note in that instrument. If you send an overwrite message to a family and, as the first data byte, repeat the ID for overwrite, the next data byte gives a parameter that should be reset for every note of every instrument of the family.

Not in This Layer

It is important to mention some of the information that will be present in ZIPI's lower-level network layers and in application layers other than the MPDL.

There will be separate application layers for sample dumps, patch dumps, and raw binary data, which can be used analogously to MIDI's "system-exclusive" messages, sending data that does not fit the MPDL.

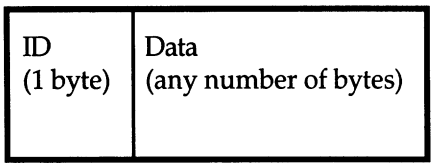
ZIPI's data link layer will provide a way for all of the devices on a ZIPI network to synchronize their clocks to within 50 μsec in order to provide a common time base to the time stamp messages described above. Strictly, systemwide clock synchronization is not required to benefit from time-tagged data. There are algorithms to reduce network delay jitter even if the sending and receiving devices have different time bases, but implementations of the MPDL over the ZIPI data link layer will have the advantage of synchronized system clocks.

There is a way, via the data link layer, to request that certain packets be confirmed upon receipt to ensure that they arrive intact. Any packet sent by the MPDL layer can request this confirmation from the data link layer. In this manner, highly critical messages such as "all notes off" can be guaranteed to arrive.

A lower network layer provides a way for ZIPI devices to identify their characteristics to other devices on the network, to query devices about their characteristics, and to look for devices with certain characteristics. These characteristics include instrument name, manufacturer, possible ZIPI speeds, etc.

ZIPI's data link layer provides a way to send a packet to all devices on the network and a way for a device to listen to all packets on the network, regardless of the device for which the packet is intended. Both of these features are available for MPDL data.

Figure 8. Note descriptor byte format.



A separate application layer for machine control will handle issues of synchronization (complying with SMPTE and other standards) and sequencer control.

There will be an application layer for MIDI messages, carried over a ZIPI network.

There will be an application layer for error messages. ZIPI devices with limited user interfaces can send ASCII-encoded error messages, which will be picked up and displayed to the user by another device, such as a computer.

References

Anderson, D., and R. Kuivila. 1986. "Accurately Timed Generation of Discrete Musical Events." *Computer Music Journal* 10(3):48–56.

Blatter, A. 1980. *Instrumentation/Orchestration.* New York: Schirmer Books.

Blauert, J. 1982. *Spatial Hearing,* trans. John S. Allen. Cambridge, Massachusetts: MIT Press.

Dannenberg, R. 1989. "Real-Time Scheduling and Computer Accompaniment." In M. V. Mathews and J. R. Pierce, eds. *Current Directions in Computer Music Research.* Cambridge, Massachusetts: MIT Press, pp. 225–261.

Krimphoff, J., S. McAdams, and S. Winsberg. 1994. "Caractérisation du timbre des sons complexes. II: Analyses acoustiques et quantification psychophysique." *Proceedings of the 3rd French Congress of Acoustics, Toulouse.*

Loy, D. G. 1985. "Musicians Make a Standard: The MIDI Phenomenon." *Computer Music Journal* 9(4):8–26.

MIDI Manufacturers Association (MMA). 1991. *General MIDI System—Level 1.* Los Angeles: IMA.

Moore, B. 1989. *Introduction to the Psychology of Hearing,* 3rd ed. London: Academic Press.

Moore, F. R. 1988. "The Dysfunctions of MIDI." *Computer Music Journal* 12(1):19–28.

Pierce, J. R. 1992. *The Science of Musical Sound,* rev. ed. New York: W. H. Freeman.

Piston, W. 1955. *Orchestration.* New York: W. W. Norton.

Puckette, M. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer Music Journal* 15(3):58–67.

Risset, J. C., and D. Wessel. in press. "Analysis-Synthesis Methods for Sound Synthesis and the Study of Timbre." In D. Deutsch, ed. *The Psychology of Music,* 2nd ed. London: Academic Press.

Rona, J. 1991. "Proposed MIDI Extension" (letter). *Computer Music Journal* 15(3):15.

Scholz, C. 1991. "A Proposed Extension to the MIDI Specification Concerning Tuning." *Computer Music Journal* 15(1):49–54.

Serra, X., and J. Smith. 1990. "Spectral Modeling Synthesis: A Sound Analysis Synthesis System Based on a Deterministic plus Stochastic Decomposition." *Computer Music Journal* 14(4):12–24.

Smith, R. 1993. *Morpheus User's Manual.* Scotts Valley, California: E-Mu Systems.

Wessel, D. L. 1985. "Timbre Space as a Musical Control Structure." In C. Roads and J. Strawn, eds. *Foundations of Computer Music.* Cambridge, Massachusetts: MIT Press, pp. 640–657.

Zicarelli, D. 1991. "Communicating with Meaningless Numbers." *Computer Music Journal* 15(4):74–77.

Appendix A: ZIPI MPDL Byte Format

In ZIPI's current low-level protocol, there are 7 bytes of overhead in each ZIPI packet that are not part of the application layer. (The first 3 bytes say "this is a new ZIPI packet," the fourth byte is the network address of the device for which the packet is intended, and the fifth byte says "this packet contains application layer information." At the end of the packet are two more bytes for the CRC error detection checksum.) It is important that one of these bytes selects a particular ZIPI device by number so that a ZIPI device is not interrupted by packets that are intended for other devices.

The first byte of the application layer data indicates the application layer to which the given packet applies. If the first 4 bits are 0000 (i.e., if the number represented by the byte is less than 16), it means that the packet is for the Music Parameter Description Language, regardless of the

Figure 9. Typical ZIPI MPDL packet.

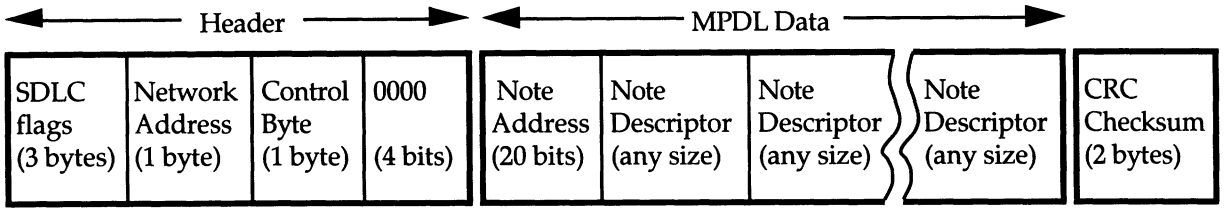


Figure 9

value of the next 4 bits. The other 240 possible 1-byte values indicate different application layers.

The remainder of the first byte, plus the next 2 bytes, make up a 20-bit note address. The format of these 20 bits is described below.

After the note address is selected, a packet consists of any number of note descriptors. A note descriptor consists of a 1-byte ID (as given in Tables 2 through 6 and 15) and some number of data bytes. (This is similar to a MIDI message; what MIDI calls "status byte" we call "ID") See Figure 8 for an illustration of this.

A packet can contain multiple note descriptors to cut down on network overhead; multiple parameters can be updated in a single ZIPI MPDL packet while only incurring the 7-byte overhead once. Furthermore, with the "new address" message described above, a single ZIPI MPDL packet can contain note descriptors for an arbitrary number of different addresses. Figure 9 shows the structure of a ZIPI MPDL packet, including the 7 overhead bytes required for the lower network layers.

Byte Format of ZIPI MPDL Addresses

The 20-bit address is interpreted as a 6-bit family, a 7-bit instrument, and a 7-bit note, as shown in Figure 10. For example, the binary address 000111 0011001 0010010 means that the information in the packet is addressed to note 18 (binary 10010) of instrument 25 (binary 11001) of family 7 (binary 111).

Note 0 of any instrument means "the entire instrument," so the address 000111 0011001 0000000 means "instrument 25 of family 7." Likewise, instrument 0 of any family means "the entire family," so the address 000111 0000000 0000000 means "family 7." (If the instrument bits

Figure 10. Bit format of MPDL note address.

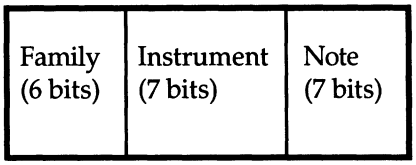


Figure 10

are zero, it does not matter what the note bits are; any address whose first 13 bits are 000111 0000000 means family 7.) Finally, for messages that affect the entire address space, family zero means "all families." As mentioned above, this is just an abbreviation for a message sent to each of the 63 families. If the first 6 bits of an address are zero, it does not matter what the other 14 bits are.

The new address message has 3 data bytes, but MPDL addresses are only 20 bits. So the high-order 4 bits of the first byte must be 0000, to be consistent with the 0000 at the beginning of MPDL portion of a ZIPI packet.

Note Descriptor Length

The high-order 2 bits of the note descriptor ID say how many data bytes the note descriptor has, according to Table 16. Thus, there are 64 note descriptors that have 1 data byte, 64 note descriptors that have 2 data bytes, etc. Note descriptors that only require data bytes (e.g., new address) have IDs that begin with binary 10. These note descriptors actually have 4 data bytes; the fourth is simply ignored. (So the 4 bytes for new address are 0000, followed by the 20-bit address, followed by 8 more bits to be ignored.)

When the high-order bits are binary 11 (i.e., "other"), the message has more than 4 data bytes.

Figure 11. Byte format for note descriptors with "other" length.

Figure 12. Example of a note descriptor with "other" length.

Figure 13. Complete MPDL packet—Articulating a C Major Triad.

ID	Length	Data
11xxxxxx	xxxxxxxx xxxxxxxx	xxxxxxxx xxxxxxxx ...

Figure 11

ID	Length	Data
11001010	00000000 00000101	00000001 00000010 00000011 00000100 00000101

Figure 12

Note Address		
Family	Instrument	Note
000001	0000001	0000001

Note Descriptor	
ID: Loudness	Data: midscale
01000001	10000000 00000000

Note Descriptor	
ID: Pitch	Data: middle C
01000000	01111001 00000000

Note Descriptor	
ID: Articulation	Data: trigger
00000001	11000000

Note Descriptor	
ID: New Address	Data: 1.1.2
10000010	000001 0000001 0000010

Note Descriptor	
ID: Loudness	Data: Under mid
01000001	01110000 00000000

Note Descriptor	
ID: Pitch	Data: Middle E
01000000	10000001 00000000

Note Descriptor	
ID: Articulation	Data: trigger
00000001	11000000

Note Descriptor	
ID: New Address	Data: 1.1.3
10000010	000001 0000001 0000011

Note Descriptor	
ID: Loudness	Data: Over mid
01000001	10010000 00000000

Note Descriptor	
ID: Pitch	Data: Middle G
01000000	10000111 00000000

Note Descriptor	
ID: Articulation	Data: trigger
00000001	11000000

Figure 13

Figure 14. MPDL file format.

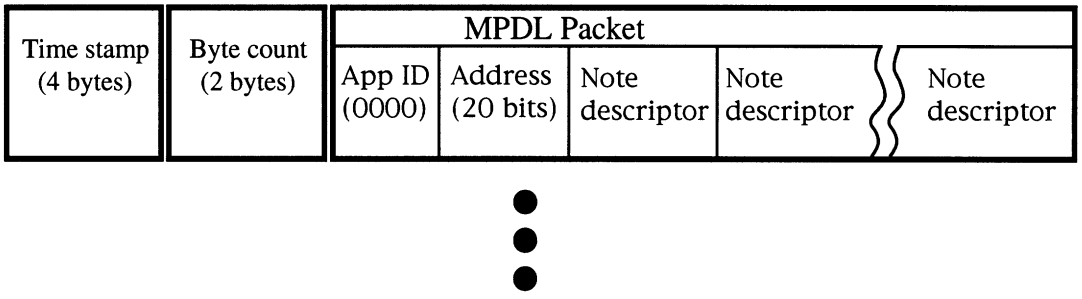


Table 16. Number of Data Bytes of Note Descriptors, Based on ID

High-Order Bits of ID	*Length*
00	1 byte
01	2 bytes
10	4 bytes
11	Other

In this case, the 2 bytes after the note descriptor ID are not data bytes; instead they form a 16-bit unsigned integer that tells the number of data bytes. Figure 11 shows this pictorially. Figure 12 shows an example. Because the note descriptor begins with "11," the length is given by the next 2 bytes. The second and third bytes form the number 5, meaning that there are 5 data bytes, for a total of 8 bytes altogether in the note descriptor. The 5 data bytes are decimal 1, 2, 3, 4, and 5.

Complete Example

Figure 13 shows every single byte of the MPDL portion of a typical ZIPI packet. This packet corresponds to playing a close-voiced root position C major triad on a keyboard, with all three notes being sent simultaneously.

Appendix B: ZIPI MPDL File Format

Files containing ZIPI MPDL data are logically sequences of MPDL frames. The file must also include a time stamp for each frame; these time stamps have the same format as MPDL time stamps (an unsigned 4-byte integer, in units of 50 μsec). The file must also include the number of bytes of each frame. (Remember that there can be multiple note descriptors in a single MPDL packet; we know the length of the MPDL data only because the lower network levels know when the entire packet ends.) This count is an unsigned 2-byte integer.

A ZIPI MPDL file thus consists of an arbitrary number of repetitions of (1) a 4-byte time tag; (2) a 2-byte count; and (3) an MPDL packet consisting of the given number of data bytes. Figure 14 demonstrates this format pictorially.

We have intentionally used the same format for MPDL file time tags as for the MPDL time tag note descriptor; this makes it trivial to convert sequences of time-tagged MPDL packets into MPDL files. In this case, there is no reason to store the time tag note descriptor in the file because that information would be redundant. When storing non-time-tagged MPDL data into a file, the process creating the file will have to supply its own time stamps.