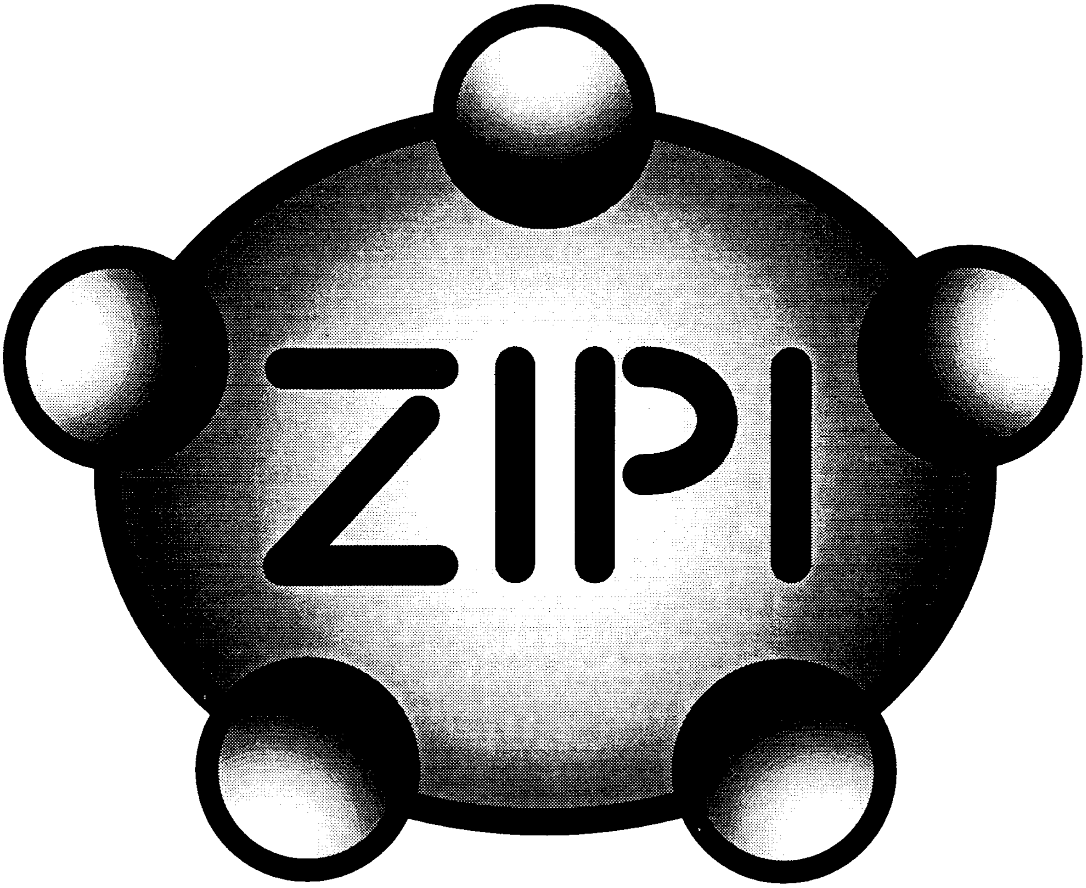


Proposal for a New Networking Interface for Electronic Musical Devices

presented by:

ZETA

Zeta Music Systems, Inc.
2230 Livingston St.
Oakland, CA 94606

CNMAT

C N M A T
Center for New Music & Audio Technologies
University of California • Department of Music
Genevieve McEnerney Hall
1750 Arch Street • Berkeley CA 94709

G-WIZ

Gibson Western Innovation Zone Labs
Gibson Guitar Corporation
2560 9th St., Suite 212
Berkeley, CA 94710

Zeta Music Systems, Inc.
2230 Livingston Street
Oakland, CA 94606 USA

January 12, 1994
Phone: (510) 649-6040
Fax: (510) 649-0278
Email: matt@cnmat.berkeley.edu

University of California • Department of Music
Center for New Music & Audio Technologies
Genevieve McEnerney Hall • 1750 Arch Street • Berkeley CA 94709
510 643-9990 • fax 510 642-7918 • cnmat@cnmat.berkeley.edu

ZIPI Documentation

The following four documents describe ZIPI, our proposed network protocol for music.

ZIPI Music Parameter Description Language is a description of the high-level protocol that will be used to transmit musical parameters such as pitch and loudness to a note or group of notes. It does not deal with any of the network or physical issues; it just talks about how to send music over ZIPI. This protocol could easily be sent over another lower level, such as ethernet or FDDI.

ZIPI Network Summary is a brief discussion of the ZIPI as a computer network. It ranges from physical details, such as a description of ZIPI cables, to network issues such as bandwidth and efficiency.

ZIPI Examples is a brief list of potential uses of ZIPI. It shows how to use ZIPI in some practical musical situations.

MIDI/ZIPI Comparison is a comparison between our protocol and MIDI, the de facto standard whose shortcomings we hope to address with ZIPI. If you have experience with MIDI, this might be a good document to read first, because it motivates many of the decisions made in the design of ZIPI.

There is also a very detailed technical specification of ZIPI that we are not including in this packet of documents. If you're interested in all the details of the network functionality, we'll be happy to provide you with the full document.

The purpose of these documents is to disseminate information about ZIPI, and to generate discussion that will lead to a better specification of ZIPI. Please feel free to make suggestions, comments, or criticism by email, postal mail, fax, phone, or in person. Thanks!

ZIPI Music Parameter Description Language

ZIPI Music Parameter Description Language

designed by Keith A. McMillen, David Wessel, and Matthew Wright
this document by Matthew Wright (version 1.0)[1]

This document describes ZIPI's Music Parameter Description Language. The MPDL is a new language for describing music. In a nutshell, it delivers musical parameters (such as articulation, brightness, etc.) to notes or groups of notes. It includes parameters that are well understood and universally implemented, such as loudness and pitch, with support for parameters such as brightness and noise amount that should be more common in the future.

The MPDL is just one of ZIPI's application layers; others include MIDI, data dumps, effect processing, virtual reality, etc.

As far as this document is concerned, ZIPI's lower levels deliver arbitrarily sized messages from any device on the network to any other device on the network; this document describes how to transmit music data via those messages. Note that this application layer could equally well run on some other lower network layer such as Ethernet or FDDI.

Byte Format

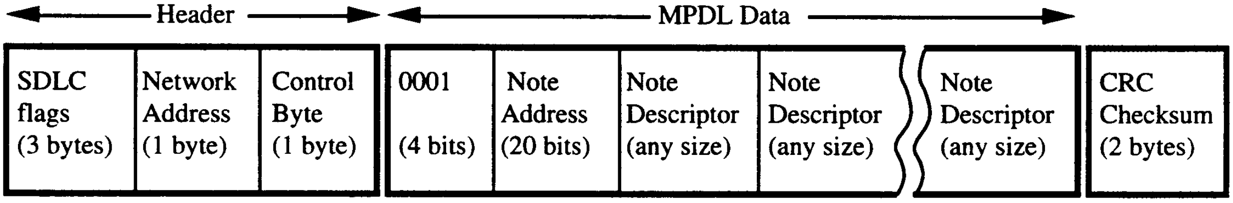
There are seven bytes of overhead in each ZIPI message that are not part of the application layer.[2] One of them selects a particular ZIPI device by number; therefore, a ZIPI device does not get interrupted by messages that are intended for other devices.

The first four bits of the application layer data say what kind of application layer data is included in the given message. The Music Parameter Description Language has the code 0001 in these first 4 bits.

The next 20 bits are a note address, saying where this message's data is headed, within the ZIPI device that received the message. (See the section "Address Space" below for an explanation of how ZIPI data is addressed.)

After the note address is selected, a message consists of any number of "note descriptors." A note descriptor gives a value for a musical parameter, e.g., "pan to the left" or "the pitch is Bb above middle C."

Here is a diagram of a typical ZIPI message:



Header			MPDL Data					
SDLC flags (3 bytes)	Network Address (1 byte)	Control Byte (1 byte)	0001 (4 bits)	Note Address (20 bits)	Note Descriptor (any size)	Note Descriptor (any size)	Note Descriptor (any size)	CRC Checksum (2 bytes)

[1] Thanks also to Chuck Carlson, Kim Flint, Guy Garnett, Mark Goldstein, and David Simon for their input.

[2] The first three bytes says "this is a new ZIPI message", the fourth byte is the network address of the device that the message is intended for, and the fifth byte says "this message contains application layer information." At the end of the message are two more bytes for the CRC error detection checksum.

Messages can contain multiple note descriptors to cut down on network overhead; multiple parameters can be updated in a single ZIPI message while only incurring the seven-byte overhead once.

A note descriptor consists of a one-byte ID, specifying which musical parameter is being set, and some number of data bytes, which are the new value for that parameter.[3]

ID (1 byte)	Data (any number of bytes)

The high-order two bits of the note descriptor ID say how many data bytes the note descriptor has:

High-order bits	Length
00	1 byte
01	2 bytes
10	3 bytes
11	other (see below)

Thus, there are 64 note descriptors that require one data byte, 64 note descriptors that require two data bytes, etc.

When the high-order bits are 11, i.e., "other," the message has more than three data bytes. In this case, the two bytes after the note descriptor ID aren't data bytes; instead they form a 16-bit unsigned integer that tells the number of data bytes.[4]

Address Space

A tremendous drawback of MIDI is that a note's address is the same as the note's pitch. If you want to specify which note to apply aftertouch to, or which note to release, you have to name that note by giving its pitch. In real life, though, a note's pitch might change over time, or there might be two notes played on the same instrument with the same pitch. Both of these situations are awkward to express in MIDI. Therefore, in ZIPI, notes have individual addresses that are unrelated to their pitch; any ZIPI note number can have any pitch.

Another drawback of MIDI is that many of the controllers are per-channel rather than per-note. For example, it's 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. Likewise, other MIDI messages, such as note-off, are always per-note, so there's no way to turn off all notes in a given channel with a single message. In general, it is desired to be able to send *any* control signal either to a particular note or to a group of notes.

[3] This is similar to a MIDI message; what they call "status byte" we call "ID".

[4] For example, a note descriptor that begins 0xE3001A would have the note descriptor ID 227 (0xE3). It would have 26 (0x001A) data bytes , for a total of 29 bytes altogether.

Therefore, ZIPI's note address space is a three-level hierarchy.[5] Our names for the layers of the hierarchy fit an orchestral metaphor: there are "notes" within "instruments," and the instruments are grouped into "families." Any message can be addressed to any level of the hierarchy: a pitch bend message could go to a single note, while a note release message could go to all of the notes of an instrument and a loudness message could go to an entire family.

The 20-bit address is interpreted as a 4-bit "family," an 8-bit "instrument," and an 8-bit "note".[6] So, for example, the address

```
0111  00011001 00010010  (hex 71912)
```

means that the information in the packet is addressed to note 18 (binary 10010) of instrument 25 (binary 11001) of family 7 (binary 0111).

Note 0 of any instrument is reserved to mean "the entire instrument," so the address

```
0111  00011001 00000000  (hex 71900)
```

means "instrument 25 of family 7." You can think of this as an abbreviation for "notes 1 through 255 of instrument 25 of family 7," since a message sent to that instrument is applied towards all the notes of that instrument.[7]

Likewise, instrument 0 of any family means "the entire family." So

```
0111 00000000 00000000  (hex 70000)
```

means "family 7." (If the instrument is zero, it doesn't matter what the note is; any address whose first 12 bits are 0111 0000 0000 means family 7.)

Finally, for messages that affect the entire address space, family zero means "all families." Sending a message to family zero is an abbreviation sending that message to each of the other 15 families.

All together, there are 15 ZIPI families, $15 \times 255 = 3825$ ZIPI instruments and $15 \times 255 \times 255 = 975375$ ZIPI note addresses.

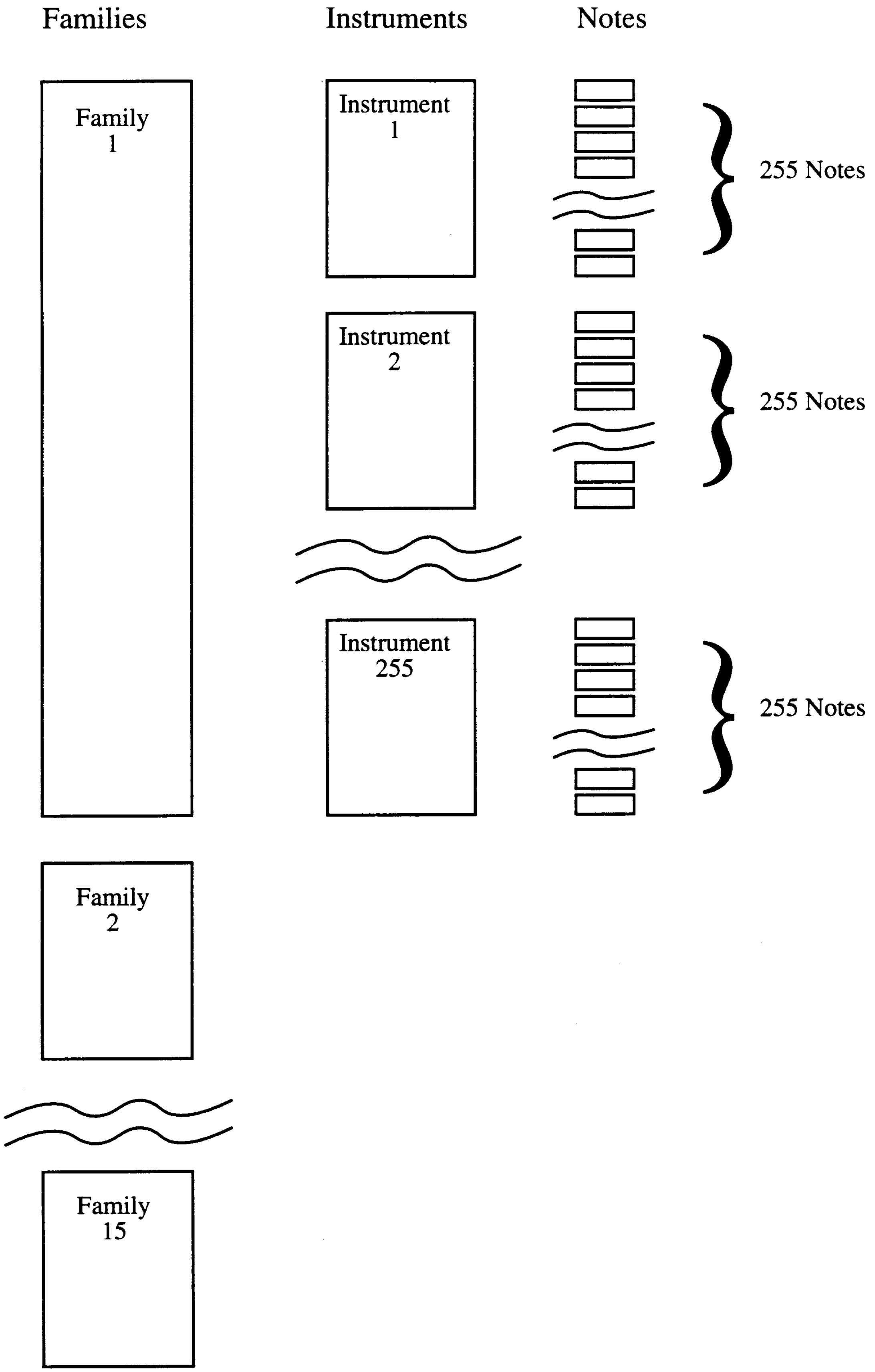
Note that an instrument cannot change its family, and can only belong to one family. If you like, you can think of the family number as the high-order four bits of the instrument. In other words, of the 3825 instruments, the first 255 are permanently in family 1, the next 255 are permanently in family 2, etc.

The diagram on the next page demonstrates ZIPI's address space pictorially.

[5] Don't confuse this address space with the one-byte network address that each ZIPI device has. This is for specifying a particular note within a given ZIPI instrument.

[6] These bit numbers come from the desire to have the parts of the address be naturally broken into byte boundaries. Probably 255 notes per instrument is too many and only 16 families is too few, but byte-alignment was deemed more important. A counter proposal is 6 bit family, 7 bit instrument, and 7 bit note number, giving 63 families, 127 instruments per family, and 127 notes per instrument.

[7] Sending a message to an instrument isn't exactly the same as sending the message to all 255 notes of that instrument; see the section "How the Levels Interact" below for more details.


Families
Instruments
Notes
Family 1
Family 2
Family 15
Instrument 1
Instrument 2
Instrument 255
255 Notes
255 Notes
255 Notes

Controlling Musical Instruments

The typical ZIPI situation will be to set up each ZIPI instrument as 255 voices of a particular timbre.[8] If you want to pitch bend an entire organ sound up, send a pitch note descriptor to the instrument, thus bending all notes of the organ. But if you want to pitch bend one of the notes differently from the others (e.g., because you're controlling the organ sound from a guitar), send a pitch update to only one of the notes.

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, with woodwinds and percussion in two more. Inside the string family we have instruments for first violins, second violins, violas, cellos, and basses. Each string of each of the violas can be assigned a note. Each of the notes is addressable with unique pitch, amplitude and timbre parameters. Here's a diagram of how we might set this up:

- 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
    - . . .

Since commands can be issued to control each level of the hierarchy, this setup gives us a conductor's control over the orchestra. The string family can be made louder and brighter. The woodwinds can all be made to go flat. Or within the woodwind family, instruments can be addressed. The oboes can get quiet while the clarinets overblow.

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 975375 note polyphony, so it must arbitrate the sound resource allocation.

[8] This corresponds to a particular MIDI channel, which would be connected to a single "patch" or "preset."

Musical Control Parameters

After the note address in a frame, the message may contain any number of note descriptors intended for that address. Here is a table of all the currently defined note descriptor IDs.[9] Note that 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.[10] The interpretation of these messages are described in the subheadings below. The last column of the table, "combine," indicates the way that these messages interact when sent to different levels of the address hierarchy; the values are described at the end of this document.

# of data bytes	**ID** (hex)	**Meaning**	**Default**	**Combining rule**
1	00	Articulation	see below	"and"
2	40	Pitch	Middle C	add
3	80	Absolute Frequency in Hertz	Middle C	overwrite
2	41	Loudness	zero	multiply
2	42	Amplitude	mid-scale	multiply
1	01	Brightness	mid-scale	multiply
1	02	Even/Odd Harmonic Balance	mid-scale	multiply
1	03	Pitched/Unpitched Balance	mid-scale	multiply
1	04	Roughness	mid-scale	multiply
1	05	Attack character	mid-scale	multiply
1	06	Pan Left/Right	center	multiply
1	07	Pan Up/Down	center	multiply
1	08	Pan Front/Back	center	multiply
2	43	Spatialization distance	10 meter	multiply
1	09	Spatialization azimuth angle	forward	add
1	0A	Spatialization elevation angle	zero	add
2	44	Multiple output levels	midscale	multiply
2	45	Program Change	silence	overwrite
1	0B	Timbre space X dimension	0	multiply
1	0C	Timbre space Y dimension	0	multiply
1	0D	Timbre space Z dimension	0	multiply
11	C0	Modulation info block	none	see below
3	81	Modulation rate	zero	see below
2	46	Modulation depth	zero	see below
257	C1	Modulation table	n/a	n/a
1	0E	Allocation priority	zero	multiply
3	82	New Address	n/a	n/a
n	C2	Overwrite	n/a	n/a

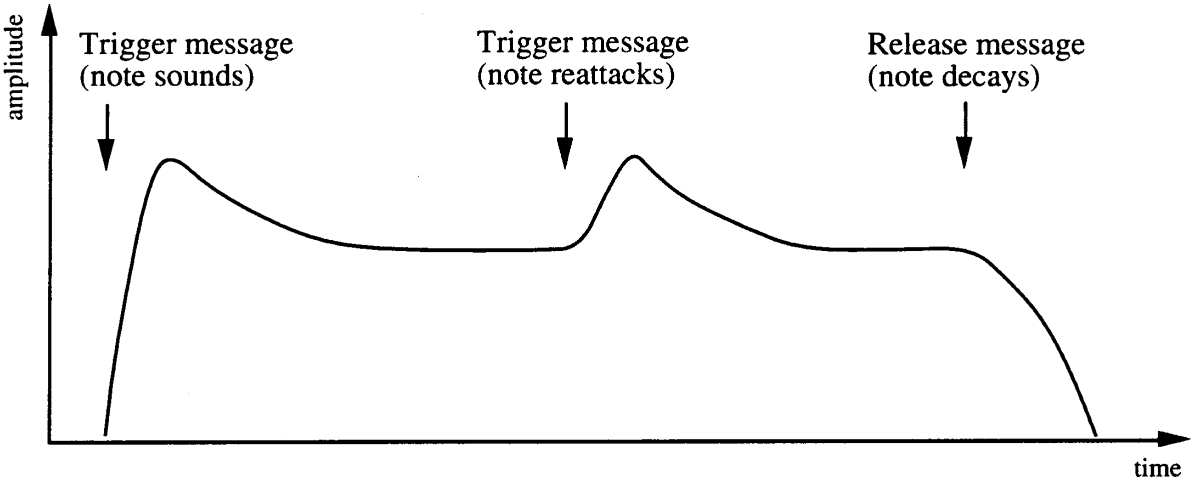
[9] We expect to define a few more note descriptors before completing the specification of ZIPI. We will leave at least half of the note descriptors free for future specification.

[10] Since a note descriptor's length is encoded as part of the ID byte, it should be easy for receiving synthesizers to ignore note descriptors that they don't implement, skip the correct number of bytes, and then examine the next note descriptor in the message.

Articulation

If the note descriptor ID is "articulation," the data byte can be can be trigger (hex 01), or release (hex 02). The trigger message starts a note; it's the equivalent of a note-on in MIDI. This note will have any parameters that were set for that note before the trigger message was sent. Note that pitch and loudness are not part of the trigger message! Before sending the trigger message you should set the pitch and loudness to the desired levels. Of course, it's possible to set the pitch, set the loudness, and trigger a note as three note descriptors in a single ZIPI frame.

The note then sounds indefinitely, until a release message is received; the release message ends a note. If a new trigger message comes before the release message comes, the note re-attacks with no release. This is useful for slurred notes.[11] Here is a diagram of what might happen to the amplitude of a tone as a note receives two trigger messages and then a release message:



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. Keep in mind that the default value for loudness is zero, so if you send a trigger message to a note that you've never sent a message to before it will be silent. The default pitch for a note, i.e., the pitch of a note that has never had its pitch set, is middle C, so if you send a loudness message and a trigger message to a note you'll get back a middle C.

Controlling a Sound's Basic Parameters

Pitch is given by a 16 bit log pitch word. The first 7 bits are nearest MIDI note number[12] and the remaining 9 bits are fractional semitonal values equal to about .2 cents. The word nnnn nnn1 0000 0000 is equal to MIDI note number nnn nnnn. The word nnnn nnn0 0000

[11] In MIDI, receipt of a second note-on for a given pitch while a previous note is still sounding does not have this behavior. In MIDI, receipt of a new note on leaves the old note sounding and starts a new, identical note, creating a chorus effect. If this situation occurs, a note-off will only turn off one of the two sounding notes. (It's not specified which one turns off.) A second note-off is needed for complete silence. ZIPI doesn't work like this! In ZIPI, the second trigger message doesn't allocate a new note, it just causes the already-sounding note to go back to the attack portion.

[12] So the high-order bits 011 1100, decimal 60, are middle C, C4, 261.624 Hertz. A440 is decimal 69, binary 100 0101.

0000 is MIDI note nnnn nnn, a quarter step flat, and nnnn nnn1 1111 1111 is MIDI note nnnn nnn, a quarter step sharp. (The rationale is that ZIPI to MIDI pitch conversion requires truncation instead of rounding.) Pitch for a note can be changed through the entire 10+ octave range without retriggering or loss of continuity of note. This is essential for instruments such as violin, flute, or even guitar.

Frequency in Hertz is an alternate way to specify pitch, as a 24-bit fixed point number. The first 16 bits are the number of Hertz, from 0 to 65535, and the last eight bits are the fractional part, giving a resolution of better than .004 Hertz. Receipt of a Hertz-frequency message overwrites the previous pitch message and vice versa; mixing the two kinds of messages is discouraged.

Amplitude describes the overall gain of a sound. Turning it up is like turning up the volume on a mixing board. Changing a sound's amplitude does not change its timbre.

Loudness corresponds to our subjective impression of intensity. It is influenced not only by amplitude but also by the temporal and spectral characteristics of the sound. Performing musicians are usually quite skilled in trading off the various acoustic parameters that contribute to the perception of loudness and are able, for example, to adjust a bright oboe note to be same musical dynamic or loudness as a mellow French horn tone. The key idea behind the ZIPI loudness parameter is that if one sends the same value to notes played on different instruments they will sound at the same subjective level. In order for the loudness parameter to function properly, the various instruments on a given synth 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 may 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 synths already points in this general direction in that different voices are adjusted so that a given MIDI velocity produces comparable loudness impressions.

Controlling a Note's Timbral Parameters

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 psychoacouticians. We have chosen to specify these more agreed upon aspects of timbre in MPDL. They are brightness, even/odd harmonic balance, pitched/unpitched balance, roughness, and attack character.

The impression we have of a tone's **Brightness** corresponds stongly to the amount of high-frequency content in the spectrum of the sound. One good measure of this is "spectral centroid," which is a weighted average frequency of the components of a sound. At a given pitch and loudness an oboe sounds brighter than a French horn and a look and 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 synthesis algorithms employ different procedures to manipulate a tone's brightness but most all provide a rather direct path to control this feature. In FM increasing the modulation index increases the high frequency content. Moving the cutoff freqency of a low pass filter upwards has a similar effect. With additive synthesis, detailed control of the spectral envelope is provided by direct specification the amplitudes of the partials.

Most pitched sounds can be thought of as a collection of "harmonics," which are sine waves spaced equally in frequency. The first harmonic is defined to be at the fundamental frequency, the second harmonic is at twice the fundamental frequency, the third harmonic at three times, etc. **Even/Odd Harmonic Balance** is provided by the relative contributions of the even partials and odd partials to the spectrum. Listening to only the odd harmonics of a tone gives a sound something like a square wave; listening to only the even harmonics of a tone sounds like a note played an octave higher on the same instrument.

This may seem like a strange parameter, but it's 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. 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. FM synthesis provides a natural mechanism by mixing simple FM patches with differing carrier-to-modulator ratios. Wave-shaping synthesis has its odd and even distortion function components and additive synthsis affords direct control over the spectral content. Even subtractive synthesis with poles and zeros allows for tight control over the Even/Odd Balance.

Most sounds can be thought of as containing a strongly pitched portion and an unpitched or noise portion. 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. MPDL's **Pitched/Unpitched Balance** measures the relative mix of these two portions of a tone.

Roughness has a direct intuitive meaning. Low values would correspond to very smooth tones, while high values would be rough. It might correspond to overblow on a saxophone. A considerable body of psychoacoustic 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 Hertz and when the depth of this amplitude fluctuation approaches 10% of the overall amplitude the sound quality becomes very rough. Beats among partials of a complex tones 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 ADSR 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; that is to say, 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 doesn't require the synthesizer to change anything about that note.

Controlling a Note's Position in Space

A sound's position in three-dimensional 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 hex 00 means "panned all the way to one direction," which would be left, up, or front. Hex ff means "panned all the way to the other direction," and hex 80 means "center." Synthesizers must implement equal energy panning.

For polar coordinates, there's **spatialization**, which describes the distance of the produced sound from the listener and the direction that the sound comes from. [13] The units of distance depend on sound being played by the synth. The **azimuth** is the angle between the sound source and "backwards," in a horizontal plane. Hex 0000 means "from behind," hex 4000 means "to the left," hex 8000 is "directly ahead," and hex C000 is "to the right." The **elevation** is the remaining dimension in polar coordinates, going from hex 0000, "down", to hex 8000, "on the same level", to hex ffff, "up."

Finally, there's a way to directly control 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. So a particular note might be coming mostly out of outputs 5 and 8, but also a little bit out of 1, 2, and 4. The **multiple output levels** note descriptor lets you set these amplitudes. It has two data bytes; the first selects one of the synth'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're 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 synth's outputs, so they're three ways to specify the same thing.)

Choosing an Instrument's Timbre

Program Change determines what basic timbre the synthesizer should produce: trumpet, bagpipes, timpani, etc.[14] It's selects a particular "patch" or "preset." It might be used to choose a set of sample files, or to load parameters for an FM synthesizer, or anything else.

It's expected that synthesizers will have some restrictions on the use of this parameter. For example, since each active timbre will probably take up a certain amount of memory or other resources, there might be a maximum number of different timbres 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.

Moving in a Timbre Space

Some synthesizers have the notion of a "timbre space," meaning that the timbres produceable by the instrument can be thought of as points in space. For example, there are some synthesizers that can produce a flute sound, a trumpet sound, or any sound along a continuum between the two. That would be an example of a one-dimensional timbre space, because you could imagine a one dimensional space, i.e., a line, with flute on one end and trumpet on another end and a continuous range of instruments in the middle.

[13] This is what psychoacousticians usually use to describe a sound's position in space.

[14] The first 127 default timbres will follow General MIDI, with further timbres to be defined.

A two dimensional timbre space would be like a piece of paper, with any point on the page corresponding to a particular tone. A three-dimensional space would be like a cube.

ZIPI has support for up to three-dimensional timbre spaces. To select a point within a one-dimensional space, use the **timbre space x dimension** note descriptor. To select a point in a two-dimensional space, also use **timbre space y dimension**. Add **timbre space z dimension** for three-dimensions.

It's assumed that the contents of the space itself, i.e., what sounds are where in the space, would be part of the patch selected by the "program change" note descriptor.

Modulating a Parameter with a Function or Table

Modulation is a generalization of the notion of vibrato. With vibrato, the pitch of a note varies a small amount around the overall pitch that the note is supposed to have. You can think of the pitch of a note as a function of time, e.g., a triangle wave. This function scales a default value, in this case, the underlying pitch that the vibrato is around. "Modulation" is the same idea: a parameter's value is given as a function of time, but any parameter, not just pitch, can vary.

It would be possible to modulate any parameter explicitly, by sending a stream of parameter values. For example, a good ZIPI violin would have fine-grained enough pitch detection to notice the vibrato played by the musician, and send it to a ZIPI synth 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, ZIPI has a way to specify a table or function to give values for a parameter over time. A single message says, for example, "start an exponential decay of volume that will go to complete silence in 1.6 seconds." After that message is sent, the receiving device computes the next values of that parameter with no additional ZIPI messages required.

There are 9 pre-defined functions:

- No modulation (turns off modulation)
- Sine wave
- Square Wave
- Sawtooth wave (the identity function)
- Exponential concave
- Exponential convex
- The function needed for equal-energy panning
- Random
- Triangle wave

Except for "random," these functions could be computed by table lookup from a 256 by 1-byte ROM table, sample values for which will be available in the future. Synth manufacturers may implement these functions in some other way, as long as the function being represented is the same. These functions are numbered 0 through 8.

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 of a power of two. Tables should be able to be at least 256 by 1-byte, but synth manufacturers are encouraged to provide larger ones. Synths are encouraged to have as many user-settable tables as possible, but can have at most 247. Synths should interpolate the values in these tables when necessary. Tables are numbered from 9 to 255.

To modulate a note descriptor, you must specify 9 things. ZIPI's **modulation info block** message consists of the values for these nine parameters, all in a single message. They are:

Byte Number	**Contents**
1	Which parameter you're modulating. (You specify this by giving that parameter's note descriptor ID
2	Which of the 9 functions are you using, or the number of a user-set table.
3+4	The rate at which you go through the table. This is the amount of time it takes to read through the table, with a resolution of 5ms.
high-order bit of 5	Whether you're looping through the table or just reading through it once. (For example, exponential decay of loudness, as a decrescendo, would just read the table once, but vibrato would loop indefinitely.)
low-order bit of 5	Whether to read through the table forwards or backwards.
6+7	Where in the table to start reading.
8+9	Where in the table to stop reading.
10	The depth of the modulation, a multiplier of the full-scale table values.
11	The offset of the modulation, as a signed value.

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 without updating any of the other parameters.

To send your own table as a ZIPI message, use a **modulation table** message. The first data byte is the number of the table you're setting, which must be at least 9. (Zero through eight are 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.[15] It doesn't 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 synth 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 just multiply through. Users that want to explicitly modulate pitch should probably turn off their synth's built-in vibrato. (Of course, if the vibrato is part of a sample, this isn't so easy.)

[15] Don't forget that ZIPI note descriptors include their own lengths, so you can tell how big the table is by how many data bytes are in the note descriptor is.

Housekeeping

Allocation priority describes the importance of a note, in case a synthesizer runs out of voices of polyphony and has to choose a note to turn off. As an example, important melodic notes might have a higher priority than sustaining inner voices in thick chords. A priority of zero means "never play this note." Note that zero is the default priority for a note. However, sending any message to a note with zero priority automatically increases that note's priority to 1.

Receipt of an allocation priority of zero allows a synth to deallocate all resources that were used by the note. It also resets all of the note's parameters back to their default values, as if no message had ever been sent to the note. Devices that send ZIPI information, e.g., controllers and samplers, should send zero values for allocation priority when they determine that a note's parameters will no longer need to be stored.

Naturally, ZIPI synths won't store note parameters by creating an array of size 975375. It's assumed that synths will take advantage of the allocation priority message to dynamically manage the memory needed to store note parameters. If they don't ever receive allocation priority messages, they can impose a reasonable limit on the number of notes whose parameters are stored, e.g., 1024.

New Address isn't a message addressed to a note; it's just a way to use ZIPI bandwidth more efficiently. Imagine that you'd 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 frames, one for each of the strings. But this is wasteful of network bandwidth, because each separate ZIPI frame has seven bytes of overhead.

Instead, it would be better to use the new address message. The MPDL portion of a ZIPI message must start with a 20-bit 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, if the message is

Note Address 00100	Pitch F#5	Pan 25% to the left	New Address 00200	Amplitude 25000	New Address 00300	Brightness 87

then the pitch and pan messages are for address 00100, while the amplitude message is for address 00200 and the brightness message is for address 00300.

The new address message has three data bytes, but ZIPI addresses are only 20 bits. So the high-order four bits of the first byte must be 0001, for symmetry with the 0001 just before the address at the opening of the MPDL portion of a ZIPI frame.

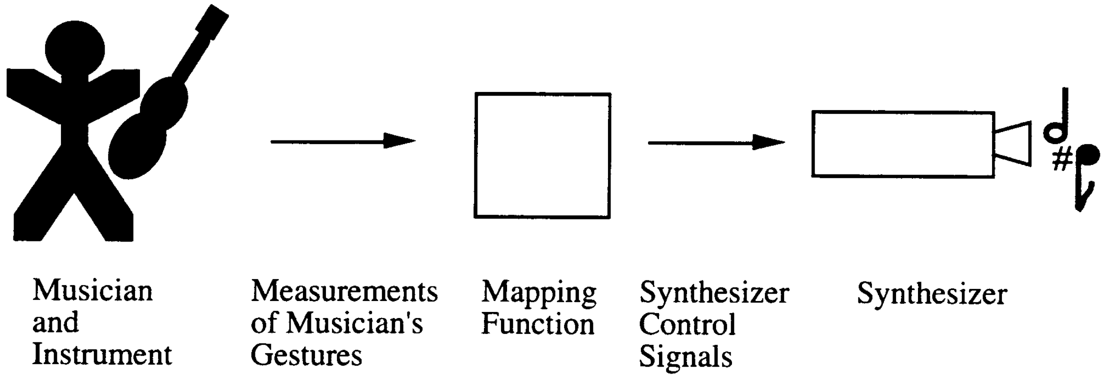
Overwrite has to do with the interaction of the three levels of the address hierarchy. Like Modulation, its first data byte is a note descriptor ID. The rest of the data bytes are a new value for that note descriptor. See the section below, "How the Levels Interact," for a complete explanation of the meaning of this message.

Controller Measurements

Real-time control of an electronic musical instrument involves three stages:

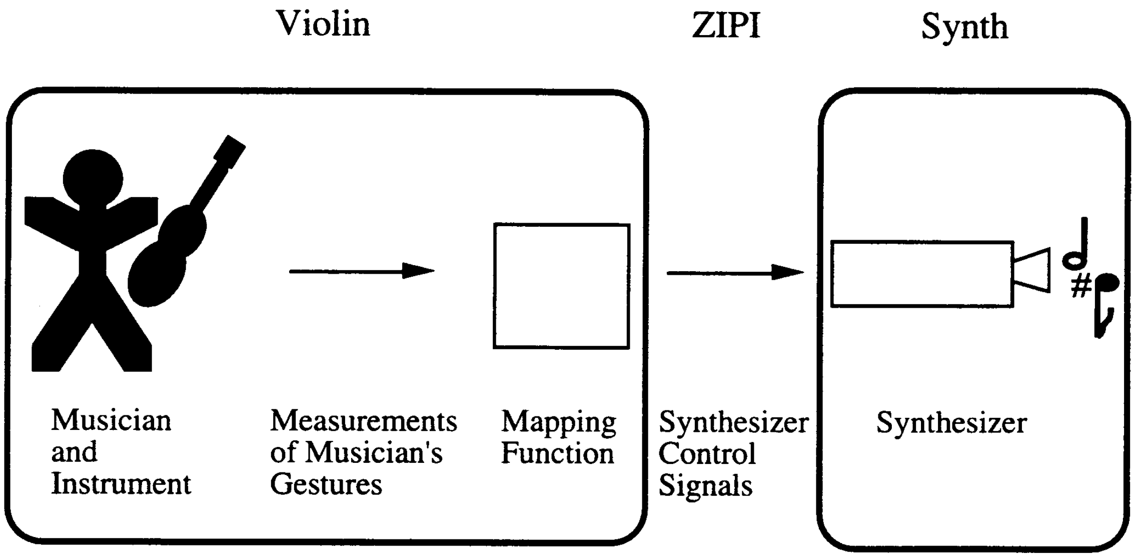
- Measuring the musician's gestures: which key was struck, how much air pressure was there, where on the fingerboard were the violinist's fingers, etc.
- Deciding how these gestures will translate into the electronic sounds produced
- Synthesizing a sound

The following picture demonstrates these stages:


Musician and Instrument
Measurements of Musician's Gestures
Mapping Function
Synthesizer Control Signals
Synthesizer

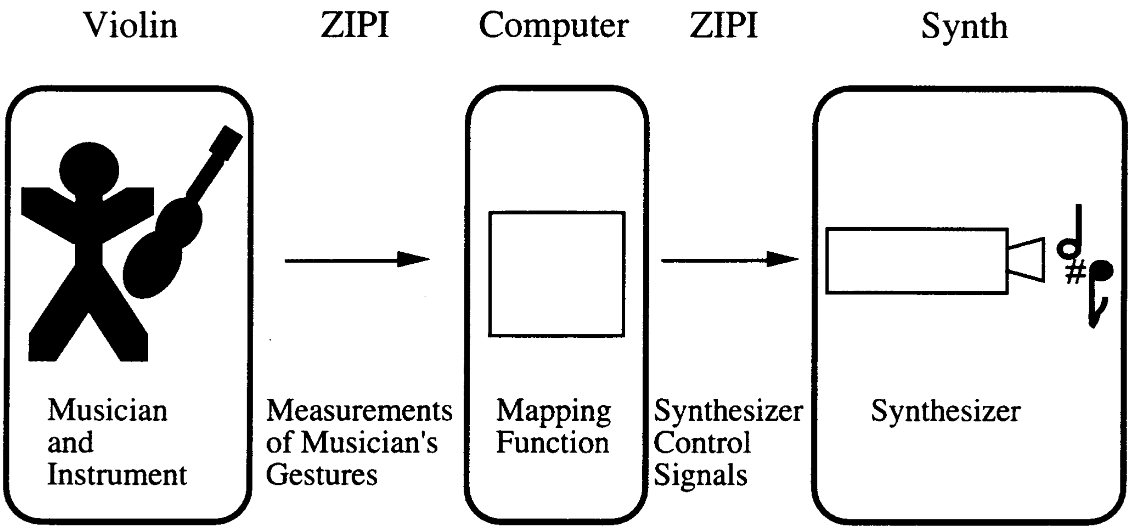
ZIPI draws a distinction between the first arrow, "measurements of musician's gestures," and the second arrow, "synthesizer control signals." All of the parameters listed in the table above are in the second category; they're descriptions of sound that tell a syntehsizer what to do.

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 as follows:



In this setup, the ZIPI instrument sends synthesizer control parameters, the ones described above.

However, people will not always want to use the mapping capabilities provided by their controller. For example, some people will want to write their own computer programs (e.g., in Max) to determine complex mappings. You might want to control the loudness of four families by finger position on the neck of a ZIPI violin. To support use-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's mapping the physical gestures into control information, sending the measurements of those gestures as uninterpreted data. That would divide the picture as follows:



ZIPI controllers' user interfaces should provide a way to switch between these two modes; sometimes the controller should do the mapping itself and sometime the controller should just send raw data.

Controller measurements are just another kind of note descriptor. Note that these measurements take up the higher ID numbers, while synth control parameters take up lower ID numbers and the middle numbers are undefined.[16]

Size	**ID** (hex)	**Meaning**
1	3f	Key Velocity
1	3e	Key Number
2	7f	Pitch Bend Wheel
2	7e	Mod Wheel 1
2	7d	Mod Wheel 2
2	7c	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	7b	Continuous pedal 1 (Volume)
2	7a	Continuous pedal 2
2	79	Continuous pedal 3

[16] As with the note descriptors for synth control, we expect to define a few more and leave most of them free for future specification.

2	78	Continuous pedal 4
1	39	Pick/Bow Velocity (signed!)
1	38	Pick Pressure
1	37	Pick/bow position
2	77	Fret/fingerboard position
1	36	Fret/fingerboard pressure
1	35	Wind Speed (breath controller)
1	34	embouchure (bite)
2	76	Wind controller keypads
1	33	Lip pressure
2	75	Lip frequency (buzz frequency for brass)
1	32	Drum head X position
1	31	Drum head Y position
1	30	Drum head distance from center
1	2f	Drum head angle
2	74	X position in space
2	73	Y position in space
2	72	Z position in space
2	71	Velocity in X dimension
2	70	Velocity in Y dimension
2	6f	Velocity in Z dimension
2	6e	Acceleration in X dimension
2	6d	Acceleration in Y dimension
2	6c	Acceleration in Z dimension
TBD	TBD	Support for data gloves

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, then an amplitude of 100 to the family containing that note? What's 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 table of note descriptors 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 the instrument's amplitude gets. If the instrument amplitude gets very high, then the two notes will have high amplitude and half as high amplitude. If the instrument gets very quiet the two notes will be very quiet and half that very quiet volume.

Note that what's being multiplied is offsets from 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 mid-scale brightness value for a flugelhorn will produce a much less bright sound than the mid-scale 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. Pitch, for instance, is taken as an offset from middle C, and the offsets accumulate additively. If a family receives a pitch message of hex 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 don't store their own values for the parameter. Instead, a message sent to an instrument or family overwrites the values of that parameter for each note of the instrument or family. Absolute frequency in hertz is a parameter with this rule. (Don't forget that absolute frequency is different from the pitch message.) Since there's no meaningful way to combine 440 Hz, 439.3 Hz and 1000 Hz together into a single frequency, a frequency received by a family sets the frequencies of all the notes of that family.

The "And" Rule for Articulation

The name "and" 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. It's possible to turn off all the notes in a family just by sending a release message to a family. If this happens, the notes still remember that they are turned on at the note and instrument levels, so if you re-trigger the family, all the notes that used to be sounding will sound again.

Here's 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 won't 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.

Family Zero

Family number zero is an abbreviation for families one through fifteen. It is not a fourth level of the hierarchy in the sense of storing yet another parameter value that must be multiplied through. It uses the "overwrite" rule; i.e., it changes the values stored for each of the fifteen families.

Overwriting a Large Group of Values

Usually the multiply or add rule does what you 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. But occasionally 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 instrument in this family to middle C."

The "overwrite" note descriptor handles cases like these. Its data bytes consist of a note descriptor ID, which specifies a parameter to be overwritten, and some data for that parameter, which specifies the new value for that parameter. 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.

What if you want to set the value of every note of every instrument in a family? You send an overwrite message to the family, and as the first data byte, repeat the ID for overwrite. That means, of course, "send the overwrite message to every instrument in this family".

Not In This Layer

It's important to mention some of the information that will be present in the ZIPI application layers other than the MPDL just described.

The data dump layer allows arbitrary information to pass between any pair of ZIPI devices, so it can be used analogously to MIDI's system exclusive data, sending data that doesn't fit the MPDL. For example, ZIPI devices would do memory or sample dumps via the data dump layer. Also, if ZIPI devices needed to transmit spectral envelopes, they could transmit them, as groups of parameters, over the data dump layer.

Also, another part of ZIPI will allow for time-tagging of certain messages and synchronizing a system-wide clock to within handfuls of microseconds, allowing ZIPI synths to impose a fixed latency on control information.

There is a way, via the data link layer, to request that certain messages be confirmed upon receipt, to ensure that they arrive intact. Any message sent by the MPDL layer can request this confirmation from the data link layer, so highly-critical messages such as "all notes off" could 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, notes of polyphony, etc.

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

ZIPI Network Summary

ZIPI Network Summary

by Keith A. McMillen, David E. Simon, and Matthew Wright (version 1.0)

This document is a technical introduction to the Zeta Instrumental Processor Interface (ZIPI). It summarizes the capabilities of the protocols and the full specification. It explains briefly

- What the ZIPI network does.
- How it works.
- How a network of ZIPI devices is connected.
- What is involved in building instruments to run in conjunction with ZIPI networks.

This is a "big picture" document; the details are available in the full specification.

Overview

The ZIPI network specification defines a collection of protocols that allow musical instruments, synthesizers, mixer boards, lighting controls, and other similar devices to communicate with one another.

The network is laid out as a star-shaped token passing ring. At the center of the star is an active hub (or a group of hubs) that has two functions: it connects the devices on the ring to one another and monitors the general health of the network.

The protocols constitute a stack that conforms to the OSI model.[1] The physical and data link layers have been fully defined. In addition a network-layer naming service is specified

[1] "OSI" stands for "Open Systems Interconnection." It was developed by the International Standards Organization as model for how to divide computer networks into various conceptual layers. OSI's seven layers are:

- Physical: transmits raw bits over a communication channel.
- Data link: gets frames from one device on the network to a device that's physically connected to it. Also serializes frames.
- Network: routes frames across a network that consists of multiple segments. (Not currently defined in ZIPI, we have only one physical segment per network, but hooks allow inclusion.)
- Transport: ensures reliable end-to-end communication, in sequential order, with confirmation, typically across multiple physical media. (Not currently defined in ZIPI; frames are acknowledged by the data link layer and have no way to get out of order.)

and various application-layer protocols have been defined. Routing, transport, session, and presentation protocols are not specified, as we see no immediate need for them, but hooks have been left to allow the inclusion of such protocols.

Features of the ZIPI Protocols

The following features of the ZIPI protocols are the most notable:

- **Peer-to-peer architecture** -- Any device can send messages directly to any other device on the ring, or to all devices at once. Up to 253 devices can be on the ring.
- **Low per-node cost** -- The hardware required to implement the network on a node consists of a serial controller chip, a small PAL, and a dual opto-isolator. The total cost is under $5.
- **Low development cost** -- Zeta will provide sample schematics for the hardware, the PAL equations or already-programmed PALS, and most of the software needed to implement an instrument . A manufacturer need only provide software to deal with the hardware-dependent aspects of his node.
- **Serial Communications Chip** -- Most of ZIPI's lower levels are taken care of by the 8530, an inexpensive, plentiful chip found in all Macintosh and SGI computers. It is a dual and can support both MIDI and ZIPI at the same time.
- **Compatibility** -- The protocol works with other protocols. Specific provision has been made to carry MIDI data over the ZIPI data-link protocol. Hooks have been left to run other protocols as the market requires.
- **Open architecture** -- Zeta has published the draft specifications for the protocols and will continue to publish revised specifications.
- **Efficiency** -- The protocols allow the most common information to be passed in very small frames. The token passing network scheme uses most of the network bandwidth for transmitting data. Typical applications would be on the order of 95% efficient.
- **Determinism** -- The token passing ring guarantees that each node will get a chance to send data whenever the token comes around the ring. Once a device has the token, the data that it transmits is guaranteed to arrive before a certain time, since no other device has permission to talk while another holds the token.

- Session: chooses a protocol for a network session, and deals with user logins and passwords, etc. (Not currently needed in ZIPI.)
- Presentation: takes care of various representations for network data, e.g., encryption. (Not currently needed in ZIPI.)
- Application: defines what the information sent over the network means, what it represents in the real world.

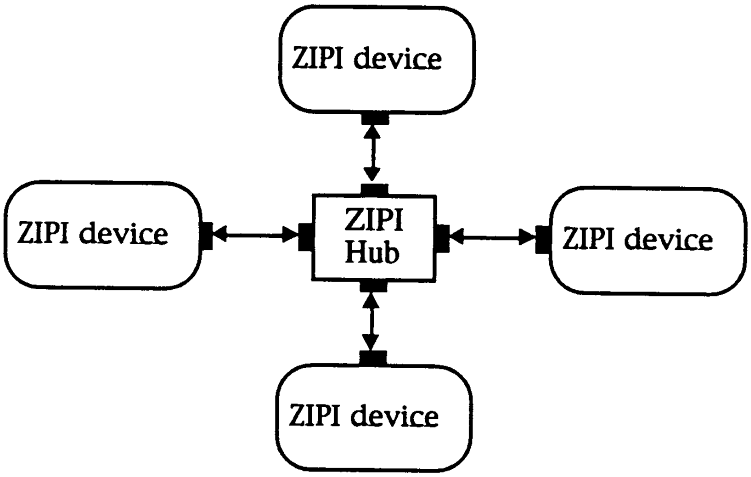
- **Fault tolerance** -- The network does not fail if one of the nodes fail; protection in the hubs will simply remove a failed node from the ring.
- **OSI** -- The protocols conform to the OSI model, allowing ZIPI to interface with other networks such as Ethernet or FDDI.
- **Expandability** -- ZIPI has been designed to allow adding new features to existing protocols or adding entirely new protocols.
- **Speed** -- The network operates at speeds from 250 KHz and up. (Presently available hardware will support up to 20 MHz.) Data rate is variable depending on the capability and need of all the devices on the network. Initial turn-on baud rate is 250K baud, but the software has a protocol for bringing the baud rate up to any speed that all devices can handle.
- **Small Processor Load** -- each host processor is interrupted only when it receives a message addressed to it. Only the monitor device must see every message.
- **Minimal cabling required** -- each ZIPI device needs only one cable to connect it to the network. ZIPI cables will be inexpensive.

Network Functionality Overview

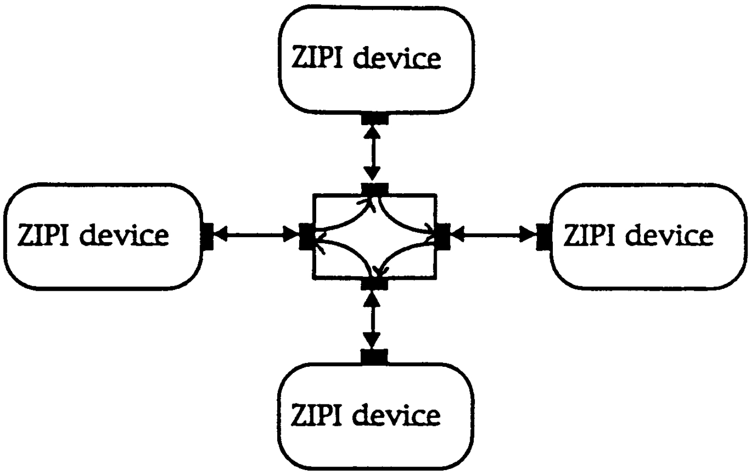
The ZIPI protocols are divided into layers according to the OSI model. The following sections summarize network operations at the various layers.

Physical Layer

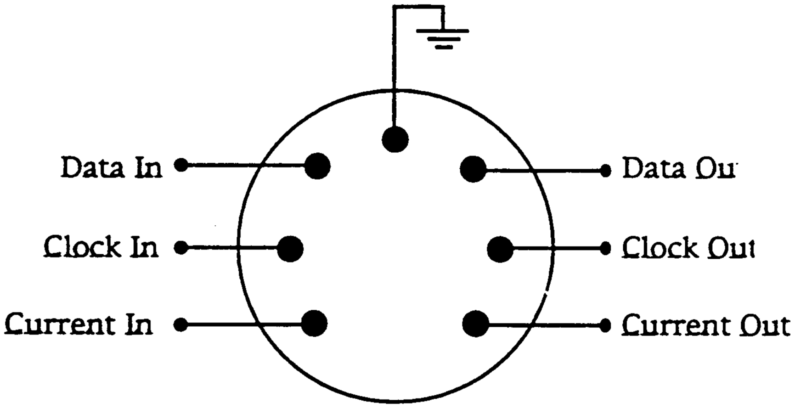
Logically, ZIPI devices are connected in a ring, in which each device passes data to the next one around the ring. Physically, the devices are connected in a star-shaped configuration in which each device is connected to an active "hub" at the center of the star.



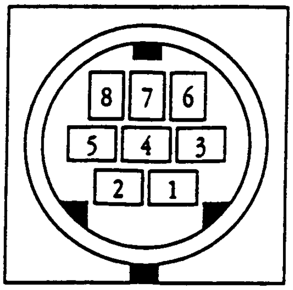
The hub maintains the logic of a ring by sending the data coming from each device out to the next device in the star.



Devices are connected to the hub by a 7-wire cable with two directions of ZIPI data flowing through it. Each direction has a clock, data, and current line; the seventh wire is for shielding the entire cable.[2] Each cable end is terminated with a 7 pin DIN plug.



ZIPI cables can also be terminated with an eight pin mini-DIN connector, to allow ZIPI interfaces to be built into laptop computers. In that case, the pin out is:



1. Clock out
2. Clock in
3. Current out
4. Shield
5. Current in
6. Data out
7. No connection
8. Data in

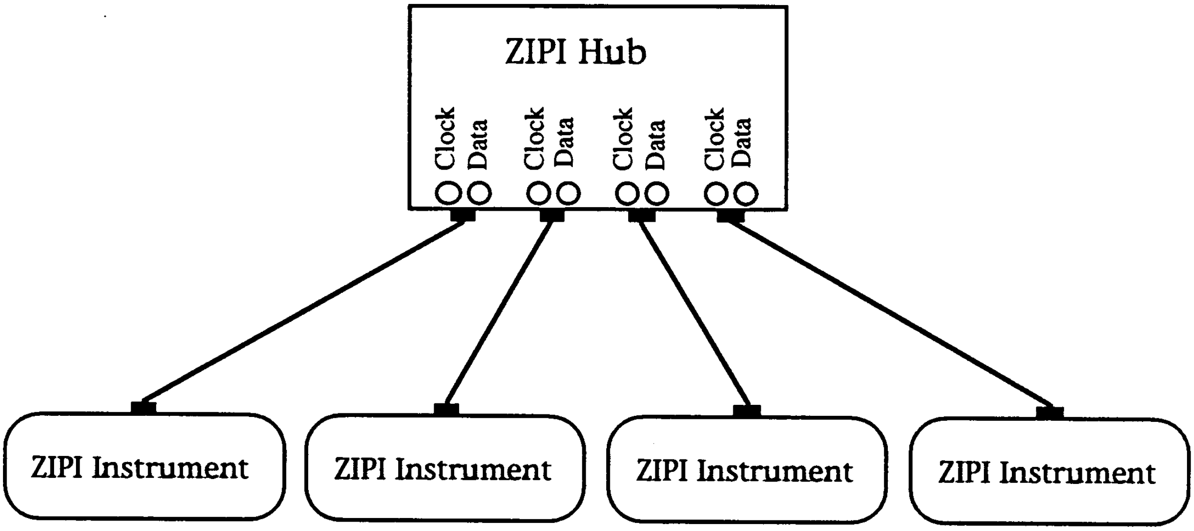
[2] It would be possible to encode the clock onto the data line, eliminating the need for the clock wire and associated opto-isolator. The receiving device could use a digital phase locked loop for clock recovery. However this would limit maximum data rates to under 800K, and reduces communications reliability. Using a clock line has distinct advantages.

By having one cable carry ZIPI information in two directions, each device will need only one female 7-pin DIN connector. ZIPI hubs will have multiple female 7-pin DIN connectors, all equivalent. A device can be attached to a hub using a male-to-male 7-pin DIN cable. Hubs can be attached to one another simply by cabling any connector on one to any connector on the other.

ZIPI hubs will have two LEDs per ZIPI plug, with the following meanings:

- A device is connected and sending a clock.
- The connected device is sending data.

These LEDs should make it very easy for musicians to debug problems in their ZIPI setup, such as malfunctioning devices, bad cables, or loose connections.



ZIPI uses an opto-isolated current loop, like MIDI.

Up to 253 devices can be connected on a single ring. The total distance from one device to the next one on the ring -- that is, the distance from one device to the hub and then back out to the next device -- can be up to 300 meters.

The preferred implementation of a ZIPI network device is based on the 8530, an inexpensive serial communications controller chip available from Zilog and from AMD, or on one of its close relatives. The 8530 running in "SDLC Loop Mode" implements most of the ZIPI physical layer protocol as well as some of the ZIPI data link layer protocol. The 8530 automatically handles most of the SDLC protocol: device addressing, host processor interrupts only for matched addresses, data framing, CRC error checking, and hardware arbitration of who gets to talk when. It has a four byte FIFO and supports DMA ("direct memory access").

Data is sent packaged in SDLC frames. A token circulates around the ring, and devices are allowed to transmit only when they receive the token. When a device has completed its transmission, it must pass the token on to the next device on the ring. Since each device gives up the token as soon as possible, the token goes all the way around the ring many times in a second, ensuring that it's never too long before any particular device gets a chance to talk.

In addition to providing connectivity, the hubs implement a monitoring function to keep the network running smoothly.[3] First, each connector on each hub has a set of relays to bypass that connector if no instrument is attached or if the attached instrument has failed or been turned off. Next, the hub provides a clock for all ring communications; devices on the ring can negotiate the speed of the clock. Last, the hub monitors the ring to ensure that the token is circulating and that no garbage circulates on the ring.

It is possible to have multiple monitor-capable devices; the software protocols elect a single monitor from among the qualified devices. It is also possible to have passive hubs that provide connectivity with no monitor capabilities.

When a ring of ZIPI devices is formed, an automatic start-up sequence begins. First, the ring monitor is elected. The clock rate is determined, and all the other overhead of the network is taken care of automatically within a second of connecting the ring. From the user's point of view, just turn everything on and plug everything in and ZIPI will work fine.

Data Link Layer

The data link layer provides the following services in addition to sending and receiving frames:

- It ensures that data has been received correctly by checking the CRC included with each frame and discarding frames on which the CRC is bad.
- It sends acknowledgments of received frames (at the option of the senders).
- It establishes an address for its device that is unique among devices on the network.
- It negotiates with the other devices on the network to determine the clock speed at which the network runs.

The data link layer in the device monitoring the ring also ensures that there is a token on the ring and that no garbage is on the ring, and notifies other devices when the clock speed is changing.

[3] One device on each ring must perform these monitoring functions. There is no technical reason why the monitor device has to be the same as the hub device, but we have chosen to make that restriction on non-technical grounds. Most manufacturers will choose the easier route of building a non-monitor-capable device, but each ZIPI network must include a monitor-capable device. The practical implication of this decision is that it won't be possible to directly connect two ZIPI devices; there must always be a hub in every network. (Of course, if there are three or more devices, a hub is required anyway just for connectivity.)

Application Layer

ZIPI will contain several application layer protocols. They include:

- The Music Parameter Description Language. The MPDL is a language for describing music. It delivers musical parameters (such as articulation, brightness, etc.) to notes or groups of notes. This language is fully described in another document.
- MIDI. The MIDI protocol can be sent through the data link layer and the data recovered at the other end.
- Data dump protocol. Unrestricted binary data, such as sample dumps, memory dumps, inter-computer communication, real-time digital audio, etc.
- Mixer Automation, lighting, and effects control (yet undefined).
- Machine Control/Synchronization (yet undefined).
- Virtual reality (yet undefined).

There is room to add more application layers in the future.

Other Layers

ZIPI includes a network-layer naming protocol to allow devices on the network to find one another either by name or by any of a large and expandable list of characteristics. For example, the naming protocol allows a device to search the network for "synthesizers capable of playing at least twelve notes and that can accept data at 1.0 Mb per second."

Hooks have been left for other protocol layers, but these protocols are not currently defined.

Building a ZIPI-Capable Device

A ZIPI device needs the capabilities of the 8530 running in SDLC loop mode. In general terms, these capabilities are to send and receive SDLC frames and to recognize and capture the token as it circulates around the ring.

Zeta will provide a sample schematic for the ZIPI network hardware. The required parts for a network device are:

- An 8530
- An inexpensive 22V10 PAL (the equations for which are available from Zeta).
- A fast dual opto-isolator (e.g., the TI HCPL2630) for the receive data and receive clock lines.

Zeta will provide the source code in C for the data link layer and some useful routines for implementing the upper layer protocols. The following software must be provided by the manufacturer:

- An interrupt routine for the 8530 that pushes the necessary registers, calls the routine provided by Zeta, and resets any interrupt control hardware other than that in the 8530.

- Software to set up a timer, handle the timer interrupt, push the registers, and call the routine provided by Zeta.

In addition, the source code from Zeta must be configured with such items as the address of the 8530 and the frequency of the timer interrupt.

Building a ZIPI Monitor

In addition to the functionality required of every ZIPI device, the monitor must implement the following:

- A 16-bit delay in the ring.

- Resynchronization of the data to deal with the problem that the clock the monitor uses to send data might not be in phase with the clock it uses to receive data.

- Recognition that the token has been lost and the ability to put a new token on the ring if it is.

- The protocol for electing a ring monitor when several monitor-capable devices are connected to the ring.

- The protocol for picking a clock speed.

ZIPI Examples

ZIPI Examples

by Matthew J. Wright (version 1.0)

This document gives some sample uses of ZIPI, the "Zeta Instrumental Processor Interface." It's recommended that you read it after you read a related document, "ZIPI Music Parameter Description Language," which describes the musical information that can be sent over ZIPI.

ZIPI Guitar

Zeta will soon release the "Infinity Box," which will be the first ZIPI musical instrument. The Infinity box is a sound to ZIPI converter; its inputs are arbitrary sound sources and its output is ZIPI control data. The Infinity Box is also a ZIPI synthesizer.

The Infinity Box will work with any instrument, but for now assume it's connected to a guitar with a hexaphonic[1] pickup. The Infinity Box will track pitch, loudness, even/odd ratio, pitched/unpitched ratio, and brightness information for each of the six strings of a guitar in real time, updating each parameter every 8-10ms. Also, of course, it will produce articulation information, noting when trigger and release events happen on each string.

The Infinity Box also will have a built-in sample playback synthesizer that can vary the pitch, loudness, even/odd ratio, pitched/unpitched ratio, and brightness amount of a sound. Thus, the sounds produced by the synthesis will be able to follow all these nuances of the acoustic sound, not just its pitch and volume.

This will give a guitar player unprecedented control over the sounds produced by a synthesizer. When the guitarist picks closer to the bridge, a synthesizer producing an organ sound would make it brighter. When the guitarist plays a harmonic, a trumpet sound would immediately shoot up an octave and get a pure bell-like tone. When the guitarist partially mutes the strings with his or her palm, the saxophone sound would have more of a breathy, noisy character. When the guitarist completely mutes the strings with his or her left hand and scratches rhythmically, a piano patch would produce just the sound of hammers hitting strings, with no pitched content.

And, of course, the Infinity Box will be able to do everything existing guitar controllers do, but better. Because of ZIPI's greater speed, there will be virtually no network delay when connecting two Infinity Boxes together, even when lots of notes are being played. The greater bandwidth also allows for much more frequent updates of pitch and amplitude than are possible under MIDI, so the sound of the synthesizer will be able to more closely track the guitarist's gestures. For example, instead of using the amplitude envelope of a synth patch, it will be easy to use the natural amplitude decay of the guitar strings to control the envelope of a synthesized tone.

As the default configuration, the Infinity Box will send the information from each guitar string to a separate ZIPI note address. Probably all six ZIPI notes would be in the same instrument because of controls such as pan that you might want to send to all six notes.

[1] "Hexaphonic" means one output channel per guitar string, instead of the usual case of the sound of all six guitar strings coming out of one output.

How much bandwidth does this require? Every 10ms the Infinity Box will send a ZIPI message that looks like this:

Address of note 1
pitch, loudness, brightness, even/odd ratio, noise ratio
new address: note 2
pitch, loudness, brightness, even/odd ratio, noise ratio
(and so on for the other four notes)

For each string, there are three bytes to select the address. Pitch and Loudness are three-byte messages (including the opcode); the other three are two-byte messages. So this requires 3+3+3+2+2+2=15 bytes per string, times six strings equals 90 bytes. There are also 5 overhead bytes at the beginning of each ZIPI message and 2 overhead bytes at the end, for a total of 97 bytes per update.

97 bytes every 10 milliseconds is 77.6K bits per second. That's under 1/3 of ZIPI's bandwidth at the slowest speed, and not quite 2.5 times the bandwidth of MIDI. This doesn't include articulation messages for triggering and releasing notes, but these would be much more than 10 milliseconds apart on average, and wouldn't take up any appreciable amount of bandwidth. Besides, not every parameter will be updated for every string every 10 milliseconds. Strings that aren't sounding wouldn't get timbral updates, and parameters such as brightness would only be re-sent when their values change appreciably.

What happens to the network if we add in lot of other ZIPI synthesizers to layer the sound? Nothing. ZIPI timbre modules only listen to the network; they never send their own messages.[2] So you could have 1 or 10 synths being controlled by an Infinity Box.

Sample dumps

Sample dumps would go through the general-purpose "data dump" application layer. In the data dump layer, there is no formatting of data, so the sending and receiving devices would have to agree that a sample dump is being sent. In most cases, sample dumps are a type of message that would be sent with "guaranteed delivery," meaning that the receiving device has to acknowledge successful receipt of the message.

The longest legal ZIPI message has 4096 data bytes, plus seven overhead bytes. After sending one of these messages, the receiving synth would have to send back an acknowledgment of around 8 bytes. Therefore, it takes 4111 bytes over the network to reliably transmit 4096 data bytes; that's 99.64% efficiency. (Of course, if there is data corruption it will be necessary to retransmit those data bytes, slowing down the process. But that's better than a garbled sample file!)

Assuming 16 bit mono PCM data, 4096 bytes is 2048 samples. At a 44.1K sampling rate, that would be about 46 milliseconds of sound.

In a ZIPI network running at 1M baud, it would take about 33 milliseconds to transmit 4109 bytes, so it would be about 40% faster than real-time to transmit CD-quality samples. Of course, a ZIPI network running at 8M baud would be 8 times faster, transmitting those 4109 bytes in 4.1 milliseconds, 11.2 times faster than real-time.

[2] Except in rare cases like memory dumps or setup when a ring of ZIPI devices is first formed.

Real-time Digital Audio

Real-time Digital Audio is similar to sample dumps, in that PCM audio data is sent at high speeds over ZIPI, but has a set of other issues that must also be resolved. First of all, obviously, the network must be able to send the data at faster than real-time!

Variations in network latency add complication to real-time digital audio. To solve them, it would be necessary for the receiving device to buffer data before playing it, so that a small loss of transmission rate won't be disastrous. This would also require that frames of audio data be time tagged, or at least sequentially numbered, so that they can be reassembled reliably into the buffer on the other end.

ZIPI Keyboard

ZIPI keyboards, like MIDI keyboards, would send messages whenever a key is pressed or released. Probably, a ZIPI keyboard would preallocate as many ZIPI notes as it has keys, sending each of them a pitch message only once as part of the setup. (For a keyboard, a note's address truly is the same as its pitch, so MIDI's model, in which the address is the same as pitch, applies.)

A "note-on" frame would have a loudness note descriptor computed from the key's velocity, followed by an articulation note descriptor ("trigger"). A "note-off" frame would just consist of a single articulation note descriptor, "release."

Like all ZIPI controllers, a ZIPI keyboard should be able to send raw controller measurements instead of mapping those measurements onto parameters like loudness and pitch. In this mode, key number would replace pitch, and key velocity would replace loudness.

It's possible that a ZIPI keyboard controller would want to implement its own note-stealing algorithm rather than to rely on that of the synthesizer being controlled. In that case, it could allocate as many notes as the receiving synthesizer has voices of polyphony. In each frame that contains a trigger message, there would also be a pitch message. Now, the keyboard knows that any note it asks the synthesizer to play will be played, and if a note needs to be turned off, the keyboard can choose which one.

On a ZIPI keyboard, alternate tunings would be a feature of the keyboard controller, not the synthesizer. Remember that pitch in ZIPI is a 16-bit quantity. The keyboard already has a table giving the mapping between key numbers and pitches; for example, it knows that the middle C key has the ZIPI pitch hex 7900. If the musician wanted alternate tunings, the keyboard could just use a different table, perhaps mapping middle C to ZIPI pitch hex 78e2 or hex 792a.

How to do Multis

Most MIDI timbre modules these days are "multi-timbral," meaning that the same synthesizer can produce pitches with different timbres on different MIDI channels. Since MIDI has only 16 channels, it's important to choose exactly which 16 timbres to use at a time. Therefore, many synthesizers have the concept of a "multi," which is a collection of 16 timbres. Sometimes it's possible to select a whole multi all at once, which is the equivalent to sending program change on all 16 MIDI channels. What's the equivalent mechanism in ZIPI?

In ZIPI, there are 3825 instruments in 15 families, so it's easy to set up an instrument with every timbre you're ever going to need, and then choose timbre just by selecting a particular instrument to trigger.

If you like the idea of "swapping in" a set of instruments, i.e., changing 16 timbres at once, it would still be easy via ZIPI. The ZIPI controller would have a data structure similar to a MIDI multi, but of any size, and possibly spanning multiple synthesizers. At the push of a button, the controller could send program change messages to all the appropriate instruments of all the appropriate synthesizers.

ZIPI Drums

Many MIDI drum machines allow you to pitch shift the drum samples. This can be useful to create what seems like a large number of instruments out of one single sample; it's especially effective for tom-toms and cymbals. But since MIDI's pitch is the same as its address, the usual situation is to think of each pitch as a different sound altogether, as in "middle C is ride cymbal, C# above that is closed hi-hat..." With this scheme, it's impossible to use MIDI's pitch mechanism to specify the pitch of a drum sound. Some MIDI drum machines get around this by letting you assign the same sample, with different pitches and pan locations, to multiple MIDI note numbers, but soon you run out of note numbers.

Drums under ZIPI would be worlds better, because pitch, pan, and address are all separate concepts, and because each note can have its own pan. A typical configuration would be to think of a drum kit as a family, with instruments like "snare drum," "timpani," "cowbell," etc. So selecting a percussion sound would be accomplished by choosing an instrument, and changing the pitch would be accomplished by sending a pitch message to that instrument.

This means that a ZIPI drum machine wouldn't have to provide so much structure for assigning sounds, pitches, and pans to each key number. Instead, all of the setup can be done over ZIPI. To get a new set of sounds, your controller or sequencer can just send program change, pan, and pitch messages to each instrument of the drum kit.

ZIPI also has note descriptors reserved for drum-specific control parameters like position on the drum head, and velocity and acceleration. Continuous hi-hat pedal position, varying steadily from fully depressed to fully open, would be encoded in "continuous pedal" messages. Hopefully, the next generation of drum pads and drum machines will take advantage of these parameters to give electronic drums a level of expressiveness closer to that of acoustic drums.

ZIPI Y-splitters and Mergers

In MIDI, two common tools are a Y splitter and a merger. The Y splitter has one MIDI input and multiple MIDI outputs, all of which are copies of the input signal. It's useful to send the control information from one computer or musical instrument to multiple synthesizers. The merger is just the opposite, taking some number of MIDI inputs and combining the MIDI messages logically into one single stream of MIDI data at the output. It's useful for controlling the same synthesizer or computer with two different MIDI instruments.

Neither of these is required in ZIPI. Any collection of ZIPI devices can be on the same network, and any of them can send a message to any other. If two devices want to send messages to the same synthesizer, they just both send data to the appropriate network device address. Likewise, if a device wants to send a message to two synthesizers, nothing needs to change in the ring configuration. The sending device can just send two messages addressed to the two devices, and both will reach their destinations. If the message is intended for *all* devices in the ring it can be broadcast, meaning that every devices sees the same message.

ZIPI-controlled Additive Synthesizer

Any sound can be represented as the sum of a number of sine waves, or "components," each differing in amplitude, pitch, and phase. The amplitude, pitch, and phase of 50 components, varying over time, might describe the sound produced by a trumpet.

An additive synthesizer produces a large number of sine waves with arbitrary amplitudes, pitches, and phases, and mixes them together, allowing a sound to be built piece by piece. How would an additive synthesizer be controlled by ZIPI?

In the common case, each instrument timbre on the synth would be a collection of descriptions of sine waves. The pitch message would scale the pitches of each component. Messages such as even/odd ratio and brightness would vary the relative amplitudes of the sine waves. There might be a model of loudness that corresponded to brightness as well as overall amplitude. With this system, the control is only over groups of sine waves, not individual sine waves, but the resulting instrument would be very expressive.

It would also be possible to control each of the components individually via ZIPI. In this case, the additive synth would only be able to produce one timbre, a sine wave. It would ignore program change messages, along with brightness, loudness, even/odd ratio, pitched/unpitched ratio, etc. It would only respond to articulation, pitch, frequency, and amplitude messages, and possibly some sort of way to specify phase. But instead of having 24 voices of polyphony, it would have more like 1000.

In this mode, controlling the additive synthesizer via ZIPI would be like programming the sounds on it, in real time. You would think of each ZIPI instrument address as a collection of components, and of each note address as a component. You'd specify the timbre by setting the relative pitches and amplitudes of each component, by sending pitch and amplitude messages to individual notes in an instrument. Pitch and amplitude messages sent to the instrument itself would then scale the values sent to each note, which would cause exactly the right effect, e.g., raising the overall volume of the sound while keeping the relative amplitudes of the partials the same. Articulation messages would also be sent at the instrument level.

Of course, you'd want the timbre to evolve over time, so you'd probably send lots of pitch and amplitude messages to the notes in order to capture this evolution.

What about phase? Since phase is so closely linked to time, it wouldn't work to try to update the phases of each partial in real-time. A network latency of merely a millisecond would render phase information meaningless. However, there is an operation on phase that would be possible via ZIPI. In many instruments there is some sort of physical action that re-aligns the phases of each component at fixed intervals. For example, in a clarinet, the action of the reed slapping against the wood causes the phase of each component to go to zero on every pitch period. It would be easy to accomplish this via ZIPI, assuming the synthesizer had a way to specify phase, by sending an "overwrite" message with a zero value for phase to an instrument. It would be OK for this realignment to happen somewhere in a window of 5ms, because it will happen to all of the components in an instrument at almost exactly the same time.

MIDI / ZIPI Comparison

MIDI/ZIPI Comparison

by Matthew J. Wright (version 1.0)

A main factor in the design of ZIPI was frustration with MIDI, the well-established standard for communication among electronic musical instruments. This document lists some of those frustrations, and explains the ways that ZIPI was designed to overcome them. It's recommended that you read this along with two related documents, *ZIPI Network Summary* and *ZIPI Music Parameter Description Language*, which describe ZIPI without reference to MIDI.

MIDI is Not a Real Network

Each MIDI device has separate MIDI plugs labeled "in," "out," or "thru." A MIDI user must think carefully about which devices will be sending information to which other devices, and arrange that the MIDI out of the device sending the information is connected with its own cable to the MIDI in of the device receiving the information, or that the signal is properly daisy-chained via the MIDI thru of intermediate devices.

Computer networks, in contrast, have the characteristic that any device connected to the network in any way can send and receive messages to and from any other device connected to the network. In Ethernet, for example, every device has only one plug, and single cable connects the computer to the entire network, allowing it to talk to any device.

In this respect, ZIPI is more like a computer network than like MIDI. Each ZIPI device has only one ZIPI plug, and a single ZIPI cable connects it to a hub. Any devices connected to the same hub can send messages to each other. If two hubs are connected with a ZIPI cable, than any device connected to either hub can talk to any other device on either hub.

This means that musicians won't ever have to rewire their ZIPI studios, unless they add or remove devices. If you want to use your ZIPI synthesizer as a keyboard controller one day and as a timbre module the next day, nothing has to change. You never have to worry about the number of ZIPI outputs your computer has, because connecting your computer's one ZIPI port to the hub will allow the computer to control any number of ZIPI synths. You never have to build complicated wiring structures with in/out/thru, because you don't have to think about which data will flow in what direction.

MIDI Has Very Low Bandwidth

MIDI has a data rate of 31.25 Kbaud, which it uses 80% efficiently.[1] This is fine for note on and note off events. Assume a musician is playing 10-voice 16th note chords at 120 BPM, which is certainly a worst case. That's 80 notes per second. A MIDI note on and

[1] MIDI has 10 bit bytes, with a start bit, 8 data bits, and a stop bit.

note off each take only 2 bytes to transmit (using running status), so that's 320 bytes, or 3,200 bits per second, which is just over a tenth of MIDI's bandwidth.

However, MIDI can't quite keep up with continuous controllers. A guitar controller soon to be released by Zeta tracks pitch, loudness, brightness, even/odd ratio, and noise amount on each of the six strings, updating every 10 milliseconds. That's

$$\frac{5\,parameters \times 6\,strings}{10\,ms} = \frac{3000\,parameters}{second}$$

Assuming these are each seven bit numbers[2], we could try to use MIDI's continuous controllers. Sending continuous controllers more or less rules out running status, so each of these would take three bytes. How much bandwidth does this require?

$$3000\,parameters \times 3\,bytes \times 10\,bits = 90,000\,baud$$

That's almost three times MIDI's data rate, without even considering note on and note off messages.

ZIPI's data rate is variable, with no maximum, so as technology improves and data rates increase, ZIPI will never be a bottleneck.[3] (ZIPI includes a mechanism for automatically picking the fastest speed that all connected devices can handle, so it's OK to mix ZIPI devices with different data rates.) ZIPI's minimum data rate is 250 Kbaud, eight times MIDI's rate, which is a comfortable speed even for this kind of continuous information.

MIDI Messages are Either for Notes or for Channels

MIDI messages fall into two categories. The first category consists of the messages whose first data byte specifies a particular note number: note on, note off, and polyphonic after touch. All other MIDI controller messages, such as Pitch bend, pan, and modulation, apply to an entire channel, not a single note.

Imagine that you're controlling a synthesizer from a guitar, via MIDI. Each of the six guitar strings might be bent by the guitarist by different amounts. So to have individual pitch-bend control of six voices, you'd have to put them on six different MIDI channels; all MIDI guitars do this. That's awkward and needlessly complicated, and it uses up over a third of the MIDI channels for one instrument.

In ZIPI, it's possible to address a pitch message to a single note instead to an entire channel. In fact, almost any message can be sent to a single note, so this entire category of problem can never arise.

[2] Actually, they're not; pitch and loudness are 16 bit values and the other three are 8 bit values.

[3] Currently available communication chips allow a maximum data rate of 20 Mbaud. The limiting factor right now isn't the communications hardware, but the speed with which host processors can deal with all that control information.

MIDI has the opposite problem too. It would be nice to turn off all of the notes on a channel all at once, but since note off is in the first category, this is impossible.[4] Every note off message has to be sent to only one note. For after touch, MIDI got around this problem by having two separate messages: polyphonic after touch, applicable only to a single note, and channel after touch, applicable only to an entire channel.

In ZIPI, *every* message can be sent either to a single note or to a group of notes. Anything you can tell a note to do you can also tell a group of notes to do.

MIDI Uses Pitch to Identify a Note

In MIDI, a note's address is the same as the note's pitch. If you want to specify which note to apply after touch to, or which note to release, you have to name that note by giving its pitch. You can't say "note number 55" without it meaning "the note whose pitch is G below middle C."

In real life, though, a note's pitch might change over time, or there might be two notes played on the same instrument with the same pitch. Both of these situations are awkward to express in MIDI. You can't say "that note that's G below middle C; slide it up a whole step to A below middle C." You can send a pitch-bend message to the channel containing that note, but then when you want to release the A you still have to call it a G, because the pitch is the name as well as the frequency.

Similarly, imagine a MIDI guitar controller in which the guitarist is fretting an E on the fifth fret of the B string, and also letting the open E ring on the high E string. The guitar is playing two notes at the same time, with the same pitch. But the note on the E string might be a lot quieter than the note on the B string, or the note on the B string might be bent up a half step, or one of them could end while the other keeps sounding. When you send a MIDI synth two note-on messages with the same pitch, you usually get the synth to play two copies of the same note. But then it's hard to send messages to a particular one of the two notes. If you send polyphonic after touch to MIDI note number 64 (the E being played by two strings), it might affect both the sounding notes, or just one of them, but there's no way to specify which one. If you send a note-off to note number 64, either note might release, even if one is much louder than the other.

It's possible to get around these problems by using separate MIDI channels for each note.[5] Then you could have a loud E on channel 1, and a quieter E, with after touch, on channel 2. But this solution is inelegant and awkward, and it soon leads to running out of MIDI channels.

In ZIPI, the notions of address and pitch are separate. ZIPI note number 64 doesn't have to be the E above middle C; it's just a number. When you want a note to sound, you pick an address, give it a pitch, loudness, etc., and tell it to start. Then whenever you want to

[4] For note on, this isn't really a problem. What would it mean to turn on all the notes of a MIDI channel? What pitches would the notes have?

[5] As mentioned above, you'd have to do this anyway to be able to control pitch independently.

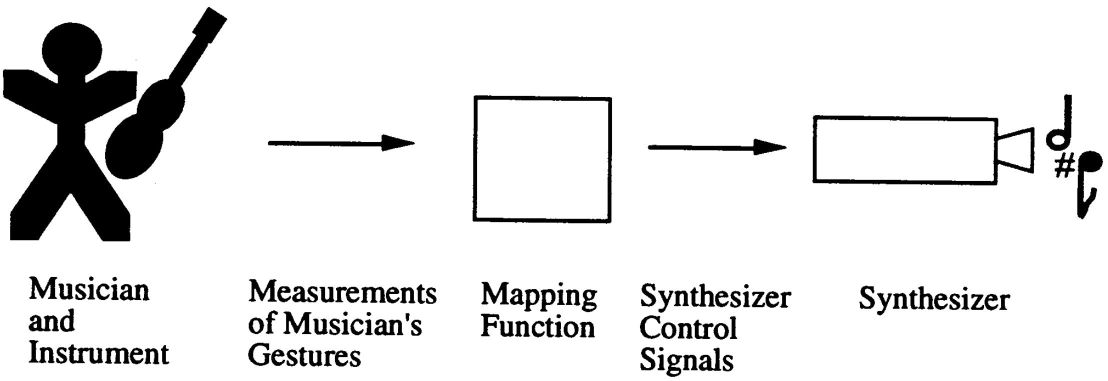
make changes to this note, you send the address of this note and the note descriptors that change it.

MIDI messages don't distinguish between controllers and synths

When a musician controls a synthesizer, there are 4 steps:

- The musician performs some action, like blowing into a mouthpiece or pressing keys.
- These gestures are somehow measured, producing parameters such as "how fast the key was going" and "which fret was fingered."
- These measurements are translated into parameters to control a synthesizer. For example, key velocity might map to amplitude and brightness, and fret position would map to pitch.
- A synthesizer takes these control parameters and produces sound.

This diagram illustrates these steps:


Musician and Instrument
Measurements of Musician's Gestures
Mapping Function
Synthesizer Control Signals
Synthesizer

Note that there are two streams of information. One is a stream of measurements about the musician's gestures; the other is a stream of control parameters for a synthesizer.

In MIDI, these two streams are confused. There's no way to directly set the pitch of a note in MIDI. You can say which key was pressed, and what the position of the pitch bend wheel is, but those are both descriptions of what the musician's hands are doing, not measurements of pitch. In other words, MIDI's notion of pitch only goes as far as describing the gestures produced by a keyboard player, not explicitly controlling a synthesizer.

Obviously, failing to make a distinction between these two ideas doesn't prevent music from being made with MIDI. For example, it's just understood that the way to send pitch via MIDI is to pretend that a keyboard player is pressing a certain key and holding the pitch bend wheel in a certain position, even if you'd rather control pitch directly. (Non-keyboard MIDI controllers start by knowing the desired pitch; then they have to go through extra steps to translate the desired pitch into a MIDI key number plus a pitch bend amount.) Likewise, people use the term "velocity," which is a measure of how fast a key is pressed, to mean loudness or amplitude.

ZIPI has a distinction between these two kinds of information. The standard type of message, which ZIPI synthesizers expect to see, is descriptions of sounds that should be produced, not descriptions of gestures that the musician is producing. So instead of having "key number" and "velocity," ZIPI has "pitch" and "loudness." But ZIPI also has a second set of parameters explicitly for describing musicians' gestures; these include keyboard measurements like key number and velocity, but also parameters that come from other controllers, e.g., bow position, wind pressure, and striking position on a drum head.

It's Hard to Control Drum Machines with MIDI

Many MIDI drum machines allow you to pitch shift the drum samples. This can be useful to create what seems like a large number of instruments out of one single sample; it's especially effective for tom-toms and cymbals. But since MIDI's pitch is the same as its address, the usual situation is to think of each pitch as a different sound altogether, as in "middle C is ride cymbal, C# above that is closed hi-hat..." With this scheme, it's impossible to use MIDI's pitch mechanism to specify the pitch of a drum sound. Some MIDI drum machines get around this by letting you assign the same sample, with different pitches and pan locations, to multiple MIDI note numbers, but soon you run out of note numbers.

Also, this mapping from MIDI note numbers to various drum sounds isn't standardized anywhere. This makes it difficult to control one drum machine from another via MIDI, because MIDI note number 37 might mean snare drum to one instrument and crash cymbal to another.

This can even be a nuisance when sequencing drum tracks from the same drum machine that will play them back. For example, suppose your drum machine lets you specify the pitch and pan of each note as you add it to a drum pattern. Once your pattern is complete you want to load it into your sequencer along with the keyboard parts. But on many drum machines, the MIDI note numbers chosen for outputted MIDI data are determined only by the instrument being played, not by the pitch of that instrument. So the process of sending your drum sequence via MIDI ruins the work you spent specifying the pitches of your drum samples.

In ZIPI, as mentioned above, pitch and address are separate ideas. So it would be easy for a drum machine to assign each drum sound to a separate ZIPI note number, and then assign a pitch to each of those notes. It would also be easy to change the pitch of a particular sound, simply by sending different pitch messages to its ZIPI note address.

MIDI Often Uses Too Few Bits

Each MIDI byte begins with a status bit that tells whether it's a data byte or a control byte. So each byte really only has 7 user-settable bits. 7 bits is not enough resolution for a variety of applications, but it's awkward to send larger amounts of information. It's possible to partition a 14 bit quantity into two separate MIDI controllers, but this is messy and rarely done. Also, even 14 bits isn't enough for many applications; it would take 3 MIDI bytes (30 bits) to send a 16 bit word.

Finally, the choice of four bits to encode a MIDI channel is pathetically small.

MIDI Has No High-Level Ways to Control Parameters

Suppose you're playing something on a multitimbral synthesizer via MIDI, and that you want to turn down the entire output of the synth via MIDI. The only way to do it is to send continuous controller 7, volume, to all 16 MIDI channels.

In ZIPI, there's a three level hierarchy of notes. A group of ZIPI notes is called an "instrument," and a group of instruments is called a "family." Messages can be sent to any level of this hierarchy, so it would be possible to turn down a group of instruments all at once (and with only one network message) by sending a loudness message to the family that contains those instruments. It's even possible to send a message to all families at once. This should make it unnecessary to duplicate the same ZIPI message many times to control different notes.

MIDI also requires a large number of messages to apply a simple function to a parameter. For example, suppose you'd like to exponentially decrescendo the volume of a MIDI channel. The only way is to send a stream of volume controller messages.

In ZIPI, it's possible to request that a certain function modulate a parameter. You could say "begin an exponential decay of loudness that takes 2.3 seconds to go to silence" in a single message, and then the decrescendo would happen without any further messages. There are eight useful predefined functions in ZIPI, and room for you to send your own tables over the network if you'd like to make up your own functions on the fly.