Midi Device Schema Design

Version 2

Contents

[Midi Device Schema Design 2](#_Toc377664957)

[Midi Schema Mapping 2](#_Toc377664958)

[Midi Data Convertors 4](#_Toc377664959)

[Midi Message Checksums 9](#_Toc377664960)

[Midi Message Address Map 10](#_Toc377664961)

[Midi Message Exchange 12](#_Toc377664962)

[Midi Message Type Detection 19](#_Toc377664963)

# Midi Device Schema Design Version 2

Refactoring to a more balanced design and aiming to detect / find bugs earlier.

**Q**: What parts of the schema are expressed in logical terms and what parts are expressed in physical terms?  
**A**: The GroupConverters manipulate the physical stream. They can alter the ordering and combine or split multiple 7-bit bytes into one or more 8-bit bytes. These adapting-streams are stacked in the same order as the inheritance chain. The DataConverters (fields) represent the logical data that are specified in terms of stream content of how the GroupConverter transformed it.

**Q**: How to combine different trades of a Converter in a transparent way by the mechanism – not each in a separate converter?  
Want to indicate Little/BigEndian, read/write-only and signed/unsigned values with attributes in the schema xml.  
**A**: Make the GroupConverter and DataConverter a placeholder that can hold a stack of ‘processes’. That way the mechanism can inject any processes that implements any of these attributes.

Device Schema is the bases for how the midi data stream is processed.

|  |  |  |
| --- | --- | --- |
| **Area** | **Function** | **Class** |
| Midi Device Schema | Load Xml |  |
|  | Parse to Object Model |  |
| Convertors | Match all Fields with Convertors | FieldConvertorPair |
| BinaryMap | Create a map that connects nodes based on schema | BinaryFieldNode |
|  | Maintain relevant data on FieldNode | FieldConverterPair, FieldKeyPath, LogicFieldInfo, FieldConverterMap |
|  | Use BinaryMap to determine DynamicConvertors | AddressMapGroupConverter \* |
| Transformation | ToLogical |  |
|  | ToPhysical |  |

\*) If we use the BinaryMap as a basis for stream/schema processing/transformation and it contains fixed FieldConverterPairs how do we deal with DynamicConverters (the AddressMap is in the BinaryMap)? Do we need something else that is used for the actual transformation process? Not if the AddressMapGroupConverter manages the whole thing. DynamicConverters are only alive during the transformation.

**Q**: How to reuse the AddressMap from the BinaryMap for multiple messages? BinaryMap is only for one message.   
**A**: Make a node type that delegates its links to a contained reference. Links going out from the AddressMap must also be intercepted… Or split the structure from the information? That still leaves a lot of nodes that are duplicate. Low prio.

# Midi Device Schema Design Version 1

Contains copies of notes made previously.

## Midi Schema Mapping

Physical Midi Data

-------------------

Physical Midi data is midi as it is received from or sent to a midi device. The

device dictates for SysEx messages the content and structure and meaning for

each message. Some devices implement some weird mapping algorithms to go from a

normal 8-bit byte to a 7-bit SysEx data byte. Or one physical SysEx data byte

is home to more than one (logical) setting (usually for small enumerators or

boolean values).

Logical Midi Data

------------------

Logical Midi data is semantically equivalent to the physical midi data. The

logical SysEx data lists each device setting as a separate field. Software

clients to midi SysEx data will find Logical SysEx easier to work with than

physical SysEx data.

Midi Device Schema

-------------------

In order to come to a generic SysEx implementation for "all" devices, we need

to be able to describe the SysEx content. Once a description for all SysEx

for a specific device is available we can fully control that device with the

same code that would control any other device. Such a logical device

description we call a Midi Device Schema.

A Midi Device Schema is built up of records. Each record can have sub records

and so on. A record indicates total size and/or byte offset. A record can occur

one or many times. A record is the smallest entity (and the biggest) that can

be sent individually. A record can also contain fields. Each field is a logical

setting for the device. A field is always sent as part of a record, never

alone. The data type of the field signals how big the data is (in bits or

bytes) and how this field is represented in physical midi data. Ordering of

fields is important because logical fields can be merged into one physical

byte.

We (mis)use Xml Schema (xsd) to describe a SysEx schema. There are certain

slightly different interpretations to the Xml Schema elements.

Xml Schema | Midi Device Schema

-------------+-------------------

simpleType | data type

complexType | logical field record

element | logical field

union | mask, shift and bit-or a byte (most significant first)

fixed | literal (protocol) byte

annotation | documentation

abstract | record placeholder

facet | physical constraints on logical fields

The Midi Device Schema 'language' also provides several base Types that are to

be used as simpleType base classes. These Midi Types serve two purposes. First

they define standard (constant) values for the Midi Protocol (such as Note On)

and the layout of (standard) midi messages. And second they are hardwired to

code that performs the transformation to go from Logical to Physical Midi data

and visa versa. Sometimes a conversion needs to be performed on the data value

(such as true/false for a Boolean bit 0/1). Midi Types can also perform these

conversions.

Midi Types

-----------

These types represent the mapping from logical to physical midi data. The Midi

Types are represented in code by 'operators'. Types of operators are Mask,

Shift and ChangeType (convert). Operators can be extended by using a custom

attribute to specify a custom operator CLR class. [TODO: this now works with

Converters and ConverterFactories.]

A transformation of byte data from logical midi to physical midi can involve

one or more operations. Also one or more logical positions can map to one

physical position (bits that represent logical fields) or one or more physical

positions can map to one logical position (a logical field is stored in

multiple bytes). When more than one logical fields map to more than one

physical bytes special block operators are called to do the translation (some

devices use byte packing algorithms to fit 8bit bytes into 7 bit sysex data

bytes).

## Midi Data Convertors

Midi Converter classes

-----------------------

The conversion from logical midi to physical midi and visa versa is done

through a chain of converters. The data-types of each field in the Midi Device

Schema implies a specific data converter. All of the midi types available in

the schema language (expressed in Xml schema) is hardwired to a specific

converter implementation class (CLR type).

The order of converter execution is exactly the same as the field definition in

the midi device schema (based on the data-types of those fields).

The types of conversion that should be supported listed below are a result of a

short study of the different flavors of sys ex implementations of various

manufacturers. Note that bit 7 of a sys ex data byte must always be zero.

During conversion of logical midi to physical or visa versa the conversion

context maintains the status and conversion info between converter calls.

The context has a carry byte and a field that marks what data of the carry has

been read. The carry byte is cleared when the last converter read the final

bit(s). Converters have free access to the carry byte. Especially the bit

converters should check the carry byte before accessing the stream.

NOTE: This is now handled transparently through a reader and writer impl.

Hierarchy of Converters

-----------------------

The hierarchy of the record and field definitions is followed by the

converters. For all records (RecordType) a group converter is created and its

children are the converters for the fields (DataType) of that record. This

structure can be nested. Each (group) converter saves the writer's stream

position of its value or the start of its value-block. This way each converter

(or other converters walking the hierarchy) can always rewind the stream and

find the converters value (NOTE: we need to keep the converters themselves

stateless, so we may need to build a transaction record into the data context).

TODO: When RecordTypes derive from one another their respective GroupConverters

should be nested in some way (containment). All calls for logical transformation

will go through the 'top' GroupConverter. Each nested GroupConverter is able

to manipulate the (physical) 'stream' before it is used by the GroupConverter

that is nesting it. This means that a 'little endian' GroupConverter can be nested

inside a 'Split Nibble' GroupConverter to implement a split nibble little endian

GroupConverter. Currently there is no easy place to chain together these stream

processors - a change in the interfaces may be required.

One key-aspect to analyze is the use of the Carry. Does each GroupConverter need

its own Carry instance or should we have only one Carry - but at which level is

it used?

- My 'guess' is that each level could need a Carry. Whenever the GroupConverter

reads 'complex' bit patterns it benefits from the Carry. If it doesn't, the Carry

does not hinder either. For instance reading repeated lower nibble data would

automatically flush the Carry with each read.

Also, how do the GroupConverters logic impact the logic of their fields?

For instance if the field is a string of a certain length (say 10), inside a

ConverterGroup that processes split nibbles, does this value mean characters or

bytes? It is 'logical' (less close to the implementation details) to let it mean

characters. But how about other data type behaviors? How are they affected when

the GroupConverter pre-processes the physical stream.

BTW: we never process (pre- or post) the logical stream.

Also, should GroupConverters be allowed to take part in the creation process of

the Converters of their fields? This appears to be useful for implementing

automatic message recognition where a few identifying bytes are read from the

physical stream and their values determine what (Dynamic) RecordType should be

used.

-------------------------------------------------------------------------------

The following problems must be solved by the converters. These problems were

discovered during a short analysis of several midi devices (Roland, Yamaha and

other). At this time it is not understood if some of these problems should be

solved in the device's vocabulary definitions. Also standard Midi (Bank,

Program and Control changes) needs to be incorporated. MMA has a standard for

this.

1) Address Map - Address based indexing. Address info is in header of the sysex

message and has to be calculated based on the fields that has to be sent or

interpreted when received when repeating records are used.

2) Repeating Records - More than one record of the exact same type (layout of

fields) represent multiple instances of a setting or setting group. (for

instance voice settings- for each voice a record exists). This only applies to

address mapped schemas.

3) String - Multiple bytes that contain ascii characters. The string has a max

length which is the same as the number of physical bytes. See also Composite

Value.

4) Dummy - One or more bytes that are ignored and have no meaning. Used as

fillers. Dummies will always use zero as a value. A special data type indicates

the field is to be treated as a dummy.

5) Composite Value - Multiple bytes that make up one logical value. Not all

bits of these bytes have to be used (for instance there might be 3 bytes that

each contains a (low) nibble and these three nibbles make up a 12bit value).

The data types of each byte will be specified as a union in a new data type

used for the composite field.

6) Boolean Value - One bit at any position within the byte that represents a

logical Boolean value. See also Enumerated Value. One of eight data types that

each represent a bit in the midi byte can be used to indicate the Boolean

value.

7) Constrained Value - The range of the value is constrained using a max value.

A custom dataType derived from one of the standard midi data types can specify

a restriction (min- max values and length).

8) Merged Value - One byte that contains multiple logical values spanning one

or more bits. See also Composite Value. Each field specifies a midi data type

to indicate the bits. Carry functionality in the converters will not advance

the stream position until the full byte is read (or until a field uses a data

type that does not support a Carry or already read bits are requested again).

TODO: An issue might exist here to escape Carry-mode and go back to byte-mode.

9) Logical Value Transform - A transformation on the physical value to come to

a logical value. For instance go from a zero-based value to a one-based value.

Will this function be implemented by the vocabulary item for this field?

10) Enumeration - A sequence of values is represented as named options. A

custom data type derived from a standard midi data type can specify enumeration

facets. The id of each facet can be used to display a 'name' for each value.

The vocabulary item for this field can provide a localized (translated) name

based on the id.

Will this function be implemented by the vocabulary item for this field?

11) Block shift - Multiple 8-bit bytes are packed into multiple 7-bit bytes.

Shifted: |x|a7|a6|a5|a4|a3|a2|a1| |x|a0|b7|b6|b5|b4|b3|b2| |x|b1|b0|c6|c...

Grouped: |x|a6|a5|a4|a3|a2|a1|a0| |x|b6|b5|b4|b3|b2|b1|b0| |x|---|b7|a7|

NibbleSplit:|x|x|x|x|a7|a6|a5|a4| |x|x|x|x|a3|a2|a1|a0| |x|x|x|x|b7|b6|...

- or - |x|x|x|x|a3|a2|a1|a0| |x|x|x|x|a7|a6|a5|a4| |x|x|x|x|b3|b2|...

This converter is implemented at the group level. This group converter will

intercept the stream used by its child converters and apply the shifting logic

on those.

TODO: Does this mean that the "bit-7 must be cleared" rule has to be enforced

by (group) converters and not by the mechanism itself?

12) Checksum - A math operation on a portion of the sysex message byte values.

The checksum field has to be defined in the parent to calculate the checksum

for. All siblings of the Checksum field are included in the checksum. Each

checksum converter implementation defines a custom data type derived from a

standard data type and a converter factory. Whenever the custom data type is

encountered the converter factory will create a checksum converter instance.

-Or the checksum converter can be implemented as a group converter and intercept

the streams passed to the child converters.

13) Schema Validation - The validity of the data bytes is checked against the

midi device schema fields. Enumerations (in the schema), fixed values and min-

and max values. Each converter implementation provides a Validate (both for

logical and physical values) to check the data type's constraints.

## Midi Message Checksums

Several sysex checksum calculations exist:

------------------------------------------

Universal SysEx Checksum calculation:

-------------------------------------

The checksum field, where present, is calculated by successively XOR'ing the

bytes in the message, excluding the F0, F7, and the checksum field. The

resulting value is then AND'ed with 7F, to create a 7 bit value.

Roland SysEx Checksum calculation:

----------------------------------

Add the 'data' bytes in the message (starting after the command Id up to -but

not including the checksum) and devide by 80(h). Then subtract the value from

80(h) to calculate the checksum value.

Official Roland Documentation:

The checksum is a value derived by adding the address, size and checksum itself and

inverting the lower 7 bits. Here's an example of how the checksum is calculated. We will

assume that in the exclusive message we are transmitting, the address is aa bb and the data

or size is ccH.

aa + bb + cc = sum

sum / 128 = quotient ... remainder

128 - remainder = checksum

<Example> Setting DETUNE to DEPTH 4

According to the “Parameter Address Map,” the DETUNE Address is 01 20H, and DEPTH 4

is a value of 40H. Thus,

F0 41 00 1A 12 01 20 40 ?? F7

(1) (2) (3) (4) (5) address data checksum (6)

(1) Exclusive Status, (2) ID (Roland), (3) Device ID (UNIT#-1),

(4) Model ID (C-80), (5) Command ID (DT1), (6) End of Exclusive

\* UNIT# must be set to the same channel as the basic channel. The basic channel is ch. 1 in

the example message above.

Next we calculate the checksum.

01H + 20H + 40H = 1 + 32 + 64 = 97 (sum)

97 (sum) / 128 = 0 (quotient) ... 97 (remainder)

checksum = 128 - 97 (remainder) = 31 = 1FH

This means that F0 41 00 1A 12 01 20 40 1F F7 is the message we transmit.

## Midi Message Address Map

Dynamic Type Creation

---------------------

When an address and a size are to be de-referenced to a specific RecordType it

is possible that this RecordType is not defined as is in the original device

schema. Assuming the ToPhysical and ToLogical conversions operate based on a

RecordType, we need to create a RecordType on the fly containing the Fields and

other (sub) RecordTypes requested for by the address/size combination.

Certain rules should apply to generating a RecordType dynamically. One problem

is the type of the RecordType(s) that contain the fragmented content (Fields).

Say we request through an address/size value a sequence of Fields that span

three RecordTypes as defined in the device schema, we may need to modify the

first and the last RecordType in such a way that they only contain the Fields

that were requested. Consider the following example where the a: is the start

of the address requested and s: is the end of the size (including).

<RecordType1>

<Field1.1>

a: <Field1.2>

<Field1.3>

</RecordType1>

<RecordType2>

<Field2.1>

s: <Field2.2>

<Field2.3>

</RecordType2>

Clearly we cannot use the RecordType1 and RecordType2 type without

modification. We should create a new RecordType that contains all the fields.

Or perhaps two RecordTypes that contain the fields to be included from

RecordType1 and RecordType2. Below the adjusted types:

<RecordType1'>

a: <Field1.2>

<Field1.3>

</RecordType1'>

<RecordType2'>

<Field2.1>

s: <Field2.2>

</RecordType2'>

NOTE: Below is not feasible or wanted. This has to do with byte-unpacking logic

that has been assigned to the GroupConverter (RecordType). The fact that records

need to be chopped up does not excuse the burden of group processing.

NOTE: This is the current implementation (which need to change).

Creating two types in this scenario is not necessary if we can prohibit address

map RecordTypes to be 'complex'. No derivation or special processing (and no

extensibility). That would allow us to create one dynamic RecordType that

simply contains all the Fields requested through the address/size.

<DynaRecordType>

a: <Field1.2>

<Field1.3>

<Field2.1>

s: <Field2.2>

</DynaRecordType>

Fields will still be extensible but group-level extensibility will not be

possible in this case because we cannot combine the logic of multiple

RecordType-GroupConverters to one dynamic RecordType/GroupConverter.

## Midi Message Exchange

Midi Sysex Message Exchange

---------------------------

Idea: use workflow to implement message exchange patterns?

Even the simplest midi sysex implementations have a concept of a request-

response message exchange. Although a dump (fire and forget) exchange usually

also exists, the other types of message exchange needs to be described.

Message exchange patterns live between the device schema definitions and the

device's vocabulary definitions. An application must be able to use a simple

mechanism for knowing what type of message will result in sending another type

of message.

Basically three types of sysex message exchange patterns have been identified.

Important issues in message exchange are: 1) A wait timeout in between multiple

message of data. This gives the receiving device time to clear the buffers. Not

used for handshake exchange patterns.

2) A request for data does not necessary lead to only one response. Multiple

messages may be sent to the device requesting data.

1) Dump (Fire and forget)

A device just sends its information to another device in one or more messages.

2) Request - Response

A device first sends a request for data to another device. This message holds

information on what type of information is requested. The other device responds

by sending one or more messages containing the requested information.

3) Handshake

The handshake exchange pattern is a more elaborate request-response exchange

that aims to increase reliability and data transfer speed. An Acknowledge

message is used to give the other device a sign that all is well and the next

message can be sent. For sending data a "Want to send" message first ask the

receiving device if its ready. When a device is unable to make sense of a

message it can ask for a "Retry" of the last message or sent a "Rejection"

(or Abort) message to cancel the whole data transfer.

Each type of message supported by the device is defined in its Midi Device

Schema. Additionally we have to define the exchange patterns that exist for

those messages. The same message types might participate in multiple exchange

patterns. A message exchange pattern in general looks like a (sequential)

workflow with alternate paths and error handling. But it is more likely we just

predefine the known message flows and allow the user to assign messageTypes for

each class of message used in the exchange pattern.

Message Header

--------------

Each manufacturer chooses its own message header structure to identify the

sysex message sent or received (besides the SOX/EOX). These message headers can

contain calculated data when -for instance- the device uses an address map.

Although the structure of these message headers can be defined using the Midi

Device Schema definitions, extra processing is probably necessary for correctly

generating the values for these dynamic header fields. For now we will not try

to make this extensible and bake an address map provider into the system.

The MessageExchangeManager is initialized at the beginning of each midi

data transfer. The MessageExchangePattern dictates what possible messages can

be sent or received and possibly an order. The intended message exchange

pattern is 'played out' after the initial message has been sent or received.

Each instance of the MessageExchangeManager represents one instance of a

MessageExchangePattern. An event is raised when an instance receives its first

message (for an incoming pattern) and the application should react by sending a

message back. The correct Device Schema information is passed with the event.

Each received message is checked against the message types in the device schema

to determine the exact type. In order to check the message identifying fields

must be read from the physical stream and interpreted. Only fixed values in the

midi device schema are considered a message header and compared against the in-

coming message data (except for the sysex-channel). This special process of

matching incoming messages to device declarations is done by a MessageProvider.

The messaging system works with two concepts: envelopes and records. The record

contains the actual data the sysex message is trying to communicate. The

envelope contains header/meta information concerning the message payload. The

minimum envelope always contains the F0 and F7 midi protocol bytes.

For some addressable devices (Roland and Yamaha) the envelope and record schema

types are different. The envelope in a Roland sysex message would contain

information about the type of message (Data Request or Data Set, etc.). The

record schema type would contain information about the payload inside the

envelope. Some message might not even contain any record data at all. This type

of messages are usually control messages that are part of the MessageExchange-

Pattern. Examples are Acknowledge, Retry, Abort and Error messages.

For simpler devices only one or two sysex messages are defined and it is

therefore possible that the envelope schema type also contains the record data

declarations. Although it is still possible to separate these into envelope and

record types, this should not be forced upon the schema author.

The MessageProvider should be able to fill in the message envelope based on the

record schema type and the actual record data. This means that for addressable

messages that are received the MessageProvider should be able to do the math on

the addresses and field sizes. For incoming messages might only contain address

and size information and from that a record schema type must be 'calculated'

and visa versa (for sending).

The MessageProvider also decides on how to interpret a minimum valid record.

For Roland (and Yamaha?) devices almost all settings can be addressed

separately. But some settings are required to be changed in batch (voice

reservations is a typical batch). For this purpose a record schema type can be

marked as such. Without such a mark a MessageProvider will allow individual

Fields to be used as data record. This means that a data record schema type

identification should contain a root, a (leave) record and an optional field.

A message exchange pattern can be defined by a state table in xml. All the

states for all exchanged messages for one device is defined in one table.

Normal state table concepts would map to device messages as follows:

state - send a message

event - received a message

next-state - refers to another state

There is one central/neutral state that an instance of a message exchange

pattern can be in. This is the starting point when the input is listening for

messages to come in. It is also the state that is selected when each handshake

message exchange is finished. This neutral state contains the definitions for

all events that can be responded to, which start of new state "paths".

When a message needs to be sent (the program takes the initiative to start the

message exchange) the current state is changed to the state that represents the

corresponding message (to be sent). From that point on the state's events

dictate what messages can be received and how the exchange proceeds.

A special condition is when the message exchange is interrupted by either an

(user) abort or an (programmatic) error.

TODO: How to define retry logic (on for instance a checksum error)?

TODO: How to define data transfer using multiple messages?

It is probable that manufactures of devices that support a 'complex' handshake

protocol only use one message exchange pattern state table. For example Roland

uses a set of standard messages that are implemented by all devices they make.

So once this state table is defined it is reusable for all devices -or at least

can be the default fallback state table if a device specific state table was

not declared.

<state name="idle"> // root state represents the idle/neutral/listening state

<events>

<event name="WSD" next-state="DS" msg-id="" />

<event name="RQD" next-state="DS" msg-id="" />

</events>

<states>

<state name="DS" msg-id="" >

<events>

<event name="ACK" msg-id="" /> // TODO: send next data message?

<event name="ERR" msg-id="" /> // TODO: possible retry or abort.

</events>

<states>

<state name="EOD" msg-id="" /> // a state without events is terminating.

</states>

</state>

</states>

</state>

TODO:

- how to define multiple messages of the same type coming in one after the other?

- how to define the basic handshake messages in order for the manager to perform

some tasks automatically. Such as retries on error or throttling when processing is slow.

MessageExchangeManager

.ctor(mepConfig);

Xxxx CurrentState { get; }

Receive(MidiDeviceSchemaType type);

Send(MidiDeviceSchemaType type);

Cancel();

The MessageExchangeManager needs to provide some way for the application logic

to indicate what message to send after a message was received. For instance

when an error occurs during parsing of the sysex message or a checksum error,

the MessageExchangeProvider can automatically send out a retry request for the

last message to be resent. But it must also be possible when a bulk is sent

from the device in successive data messages for the application to request a

wait or a pause in order to process the information. Although its very unlikely

that the modern PC will not be fast enough to keep up with a Midi SysEx

message. This throttling could also be automatic and build into the manager

itself. At least the Abort function should be present on the manager in order

for the application to cancel a lengthy handshake message exchange.

MessageProvider

.ctor(MidiDeviceSchema schema);

MidiDeviceSchemaType DetermineEnvelope(Stream physicalStream);

MidiDeviceSchemaType DetermineRecord(Stream physicalStream);

Stream WriteEnvelope(MidiDeviceSchemaType type, Stream physicalStream);

But before this can be brought in place we need to cover the following topics:

1) Automatic detection of the schema of the incoming message.

The manufacturer ID of the message determines the type of AIP used. Other

fields inside the message may yield more information on the actual message

RecordType or Field. Once the provider is determined it has to identify the

message type. It returns a RecordType or Field of the contained in a

general purpose GroupConverter. The Provider also has to determine the type of

Address Map to use based on the type of device (Model ID).

2) Allow the ToLogical and ToPhysical operations to be performed based on a

Field (instead of a RecordType). Or allow RecordTypes to be created

dynamically to contain the Field(s) that need to be processed.

TODO: analyze the case of universal sysex messages. Where "all" messages have

the same layout/structure and only some the values determine what type of

universal sysex message it really is.

NOTE

An orchestration/workflow type approach might be more suitable. Also taking

into account that some 'steps' also require sending normal midi messages (PC's)

or displaying text to the user to perform some manual action.

## Midi Message Type Detection

Auto Detect Message Type

------------------------

When an unknown (physical) SysEx message comes in, some sort of detection

mechanism has to determine what message RecordType it represents. Certain

marker fields in the message RecordType can be tagged with special attributes

and indicate what conditions must be fulfilled by the incoming message to

select that particular message RecordType.

Each manufacturer (or even model) uses its own message structure. So we need to

have a provider model where each provider implements the specifics for each

type of mechanism. This provider has the job of interpreting the physical data

stream and coming up with the identifying attributes that would select \*one\*

message RecordType. Currently this provider is called an IMessageProvider.

The schema manager has to be extended with search functionality. With a list of

identifying attributes, the schema manager should be able to locate the correct

message record type. The message provider specifies the device schemas to the

schema manager it wants to use in order to satisfy the identifying attributes.

The order of the identifying attributes is important as well as at what fields

these attributes can be found. Perhaps it is possible to declare templates that

define a certain order and structure to these attributes. For some schemas it

is simple: for example Roland specifies a command id in all of its messages. So

just read up to the command id and you'll know exactly what RecordType to pick.

The identifying attributes are declared as xml attributes on the fields of a

device schema. Some default attributes will be supplied, but ultimately the

message provider has to generate the identifying attributes that match up with

those specified in the device schema (or not). So not all devices will use

these default attributes.

manufacturer - the field that identifies the manufacturer (can be a composite field).

model/device - the field that identifies the model of the device (can be a composite field).

sysexchannel - the field that identifies the device instance.

command - the field(s) that identify the command (sysex type id).

All bytes read by the message providers from the incoming sysex message are

buffered in order to be able to rewind the stream before it is passed to the

next message provider. The interpretation of the incoming physical message can

be done through the use of Converters and the MidiDeviceDataContext. The

logical reader is supplied by the message provider and it interprets the

values. A message provider might also decide to interpret the raw physical

bytes directly (faster).

The first message provider that is able to resolve the physical stream of the

incoming message, thru identifying attributes to a message RecordType is

selected to provide the MidiDeviceMessageInfo structure.

The MidiDeviceMessageStructure contains the identifying attributes as well as

two RecordType references. One RecordType is to identify the envelope of the

message. Some manufacturers (Roland and Yamaha) use a structured messaging

scheme where "all" devices use the same envelopes. The envelope usually

contains identifying information and other meta data. The other RecordType

identifies the body (or payload) of the message. This is the part that contains

the actual information.

Note that the requirement of needing to identify two RecordTypes can lead to

multiple searches in the schema manager by the message provider.

Universal SysEx Message Structure:

F0H SOX Fixed value

UID 7EH/7FH Non-Realtime/Realtime

DEV DeviceID SysEx Channel/Device number

CMD1 CommandID Identification of the message (type)

CMD2 SubCommandID Sub-identification of the message (type) Optional.

...

F7H EOX Fixed value

Roland Message Structure:

F0H SOX Fixed value

41H Manufacturer Fixed value (for Roland)

DEV DeviceID SysEx Channel/Device number

MDL ModelID Identification of the device model

CMD CommandID Identification of the message (type)

...

F7H EOX Fixed value