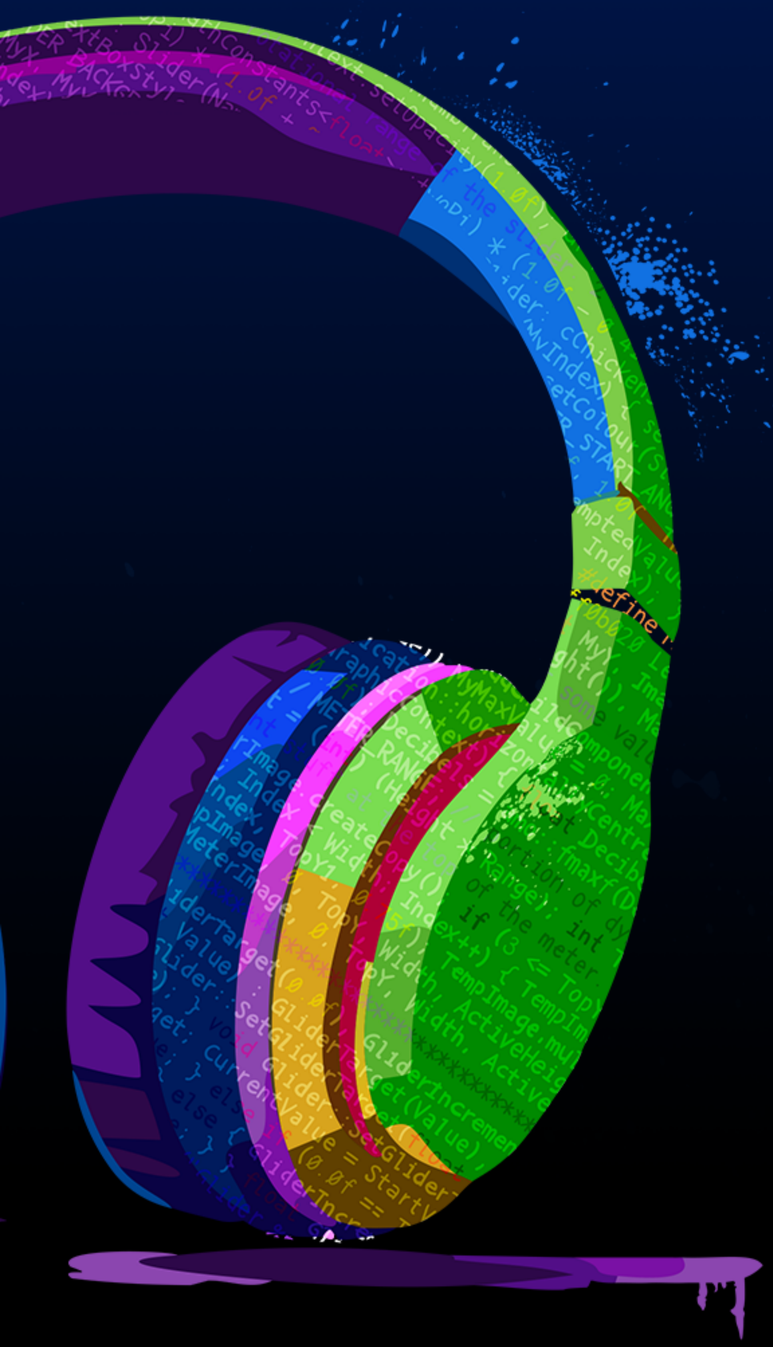

ADC24 Bristol

INTRODUCING NI-MIDI2

A MODERN C++ LIBRARY IMPLEMENTING MIDI2 UMP 1.1 AND MIDI CI 1.2

FRANZ DETRO

Overview

- Universal MIDI packet / MIDI messages
  - Data types
  - MIDI message creation and processing
  - How to support Per-Note Pitch
- MIDI CI messages (System Exclusive)
- Helper Functionalities

MIDI Message Data Types

midi::universal_packet

- Sequence of 1 to 4 32 bit words in native endianness
- Accessors for `group`, `type`, `status`, `channel` (if applicable), individual data bytes and words
- Helper to filter for MIDI protocol (`is_midi1_protocol_message()`)
- `enum class`es and `constexpr` for all UMP 1.1 packet types, statuses, controller numbers, protocols, etc.

MIDI Message Data Types

Strong Types for Message Properties

- Strong types `velocity`, `controller_value`, `controller_increment`, `pitch`, `pitch_bend` and `pitch_bend_sensitivity`
- All strong types work with high resolution (16 or 32 bit)
- Constructors and accessors for lower resolutions (7 or 14 bit)
- Conversion to `float` and `double` values (0..1, -1..1, 0..127.9999)
- Math with strong types is supported
- Recommendation: replace `byte`s and `int`s in you APIs with these strong types!

MIDI Message Data Types

Strong Type Code Examples

```
velocity velA { 0.87 }, velB { uint7_t { 98 } }, velC { uint16_t{ 0xA145 } };
controller_value valA { 0.5f }, valB { uint7_t { 100 } }, valC { uint32_t{ 0x40000000 } };
controller_increment inc { -1024 };

auto v = (valA + inc).as_double(); // get incremented controller value as double



pitch_7_9 pitch_attribute { 68.1f };
pitch_7_25 pitchA { 99.4 }, pitchB { note_nr_t{ 68 } };
pitch_bend bendA { -0.8 }, bendB { uint14_t{ 0x2000 } };
pitch_bend_sensitivity sensitivity { note_nr_t{ 2 } };

auto p = pitchA + bendA * pbSensitivity; // calculate pitch with pitch_bend and sensitivity
```

MIDI Message Creation

Types and Factory Functions

- There are classes available for all UMP Message Types, like `midi1_channel_voice_message`, `midi2_channel_voice_message`, `system_message`, `data_message` etc.
- Specific messages are created using message factory functions like `make_midi1_note_off_message`, `make_midi2_note_on_message`, `make_midi2_control_change_message`, new flex data `make_set_tempo_message`, etc.

MIDI Message Creation

Factory Function Examples

```
const group_t   group { 5 };
const channel_t channel { 0xA };
const note_nr_t note_nr { 68 };
const velocity  vel { 0.87 };
const pitch_7_9 pitch_attribute { 69.1f }; // middle A + 10 cent

make_midi1_note_off_message(group, channel, note_nr_t{ 58 });
make_midi2_note_on_message(group, channel, note_nr, vel);
make_midi2_note_on_message(group, channel, note_nr, vel, pitch_attribute);

const controller_value val { 0.9366 };


make_midi2_control_change_message(group, channel, controller_t{ 42 }, val);

make_system_message(group, system_status::clock);
make_song_position_message(group, uint14_t{ 1234 });
```

MIDI Message Processing

Filtering Messages

- Filter by UMP Message Type using `universal_packet::type()` or free-standing helpers like `is_midi2_channel_voice_message()`, `is_data_message()` etc.
- For Channel Voice Messages use MIDI protocol agnostic helpers like `is_note_on_message()`, `is_poly_pressure_message()`, `is_channel_pitch_bend_message()`, etc.
- Additionally, there are free helper functions to filter for specific messages, e.g. `is_sysex7_packet()`, `is_flex_data_message()`, `is_registered_per_note_controller_message()`, etc.

MIDI Message Processing

MIDI Message Data Views

- MIDI Message Data Views allow convenient and safe inspection of message properties (`midi1_channel_voice_message_view`, `midi2_channel_voice_message_view`, `system_message_view`, etc.)
- These view types can / should be used in APIs for type safe passing of UMPs
- Convience functions utilizing `std::optional` are available to safely create data views from UMPs (`as_midi1_channel_voice_message_view()`, `as_system_message_view()` etc.

MIDI Message Processing

Channel Voice Message Processing

- Properties of Channel Voice Messages can be retrieved with MIDI protocol agnostic free helper functions (e.g. `get_note_nr()`, `get_note_velocity()`, `get_controller_nr()`, `get_controller_value()`, etc.)
- These helper functions return instances of e.g. `velocity`, `controller_value`, `pitch`, `pitch_bend` and other properties for UMPs in both MIDI protocols
- This approach allows application code to easily handle Channel Voice Messages independent of their MIDI protocol, resulting in much cleaner and more readable code

MIDI Message Processing

Code Examples

```
if (auto msg = as_system_message_view(packet)) {
    if (msg->status() == system_status::song_position) {
        uint14_t pos = msg->get_song_position();
        // do_something_with(pos);
    }
}

if (is_note_on_message(packet)) {
    note_nr_t  note { get_note_nr(packet) };
    velocity   vel { get_note_velocity(packet) };
    pitch_7_25 pitch { get_note_pitch(packet) };
    // ... (trigger a new note with velocity and pitch)
}

if (is_control_change_message(packet)) {
    controller_t     c = get_controller_nr(packet);
    controller_value v = get_controller_value(packet);
    // do_something_with(c, v);
}
```

Per-Note Pitch Support

Note Attribute #3 Pitch 7.9

- Recap: Allows to provide an initial pitch with the Note On message
- For common cases simply use `get_note_pitch()` which deals with the existence of the Pitch 7.9 attribute in the message payload. If no Pitch 7.9 attribute is present the pitch is derived from the `note_nr` (assuming equal tempered scale default tuning as defined in the MIDI 1 specification).
- In case of alternative tunings one may utilize `is_note_on_with_pitch_7_9()`.
- If there is a need to lookup a sample based on the note message one might use `std::round(pitch.as_float())` as an index, as `note_nr()` may be an arbitrary value (fixed, note rotation, etc.)

Per-Note Pitch Support

Registered Per-Note Controller #3 Pitch

- Recap: allows to modify the pitch of a (sounding) note
- Per-Note Controller messages can be filtered using `is_registered_per_note_controller_message()` and check `get_per_note_controller_index()` against `registered_per_note_controller::pitch_7_25` or using the short version `is_registered_per_note_controller_pitch_message()`
- Use `get_controller_value()` to retrieve the 32 bit value payload and pass it into a `pitch_7_25` constructor
- Set the pitch of the addressed voice to the new value.

Per-Note Pitch Support

Code Examples

```
if (is_note_on_with_pitch_7_9(packet)) {
    note_nr_t  note { get_note_nr(packet) };
    velocity   vel { get_note_velocity(packet) };
    pitch_7_25 pitch { get_note_pitch(packet) };

    // ... (trigger a new note with velocity and pitch)
}
```

```
if (is_registered_per_note_controller_pitch_message(packet) {

    note_nr_t  note_nr { get_note_nr(packet) };
    pitch_7_25 new_absolute_pitch{ get_controller_value(packet) };

    // ... (modify pitch of note)
}
```

MIDI CI Messages (System Exclusive)

- MIDI 2 is not only a new protocol and new packet types, Sysex-based MIDI CI is an important part of the MIDI 2 ecosystem
- Traditionally, Sysex has been a second class citizen in OS APIs and MIDI libraries, dealing with Sysex has often been cumbersome
- In *ni-midi2* there is strong support for Sysex and especially MIDI CI 1.2 messages
- Foundation of the MIDI CI implementation in the library are the `sysex`, `universal_sysex::message` and `ci::message` classes

MIDI CI Message Creation

Factory Functions

- As with UMP messages, MIDI CI messages are created using factory functions, too.
- MIDI CI is designed to be a dialogue, so one typically sees pairs like `make_discovery_inquiry()` / `make_discovery_reply()`, `make_get_property_data_inquiry()`/ `make_get_property_data_reply()`
- The MIDI CI functionality also covers helpers like `device_identity`, `profile_id`, `property_exchange::header`, `property_exchange::chunk` and more

MIDI CI Message Creation

Code Examples

```
muid_t  my_muid, other_muid;
uint7_t request_id;


// discovery
const device_identity identity{ 0x002109, 0x2400, 61, 0x0100000 };
auto m1 = make_discovery_inquiry(my_muid, identity, category::property_exchange, 512);


// query resource ProgramList using request id 10
auto m2 = make_get_property_data_inquiry(my_muid, other_muid, "ProgramList", 10);


// reply resource request with status 200 and one chunk
const property_exchange::chunk c { std::string{ "this is my simple string payload" } };
auto m3 = make_get_property_data_reply(my_muid, other_muid, 200, 1, 1, c, request_id);
```

MIDI CI Message Processing

Filtering and Validation

- Other than UMPs, Sysex messages might declare a type in their header but then may not comply to the specified data structure
- This is especially dangerous in cases where the received message is too short
- Therefore MIDI CI Message Data Views provide basic message validation: `static bool validate(const sysex7&)` members are available in all CI Data View classes
- One can filter CI messages using `is_capability_inquiry_message()` and then attach a `capability_inquiry_view` for further processing

MIDI CI Message Processing

MIDI CI Message Data Views

- There are Data Views available for all types of MIDI CI messages (`ack` / `nak`, `discovery`, `profile_inquiry`, `property_exchange`, `process_inquiry`)
- Data View APIs provide convenient access to the properties of the message including
  - assembling multi-byte values (SysEx is 7 bit little endian)
  - constructing embedded CI types (`profile_id`, `device_identity`, Property Exchange `header` / `chunk`, etc.)
- `as<data_view_class>()` provides safe `std::optional` data view creation

MIDI CI Message Processing

Code Example

```
if (is_capability_inquiry_message(sx))
{
    capability_inquiry_view ci{ sx };

    if (ci.subtype() == subtype::get_property_data_reply)
    {
        auto msg = as<get_property_data_view>(ci); // calls validate
        if (msg)
        {
            auto request_id = msg->request_id();
            std::span<const uint7_t> header{ msg->header_begin(), msg->header_size() };

            uint14_t num_chunks = msg->number_of_chunks();
            uint14_t cur_chunk = msg->number_of_this_chunk();

            std::span<const uint7_t> chunk{ msg->chunk_begin(), msg->chunk_size() };
        }
    }
}
```

Helper Functionalities

- `sysex7_collector` assembles `sysex` instances from a stream of UMP Data Message packets. There is also a `sysex8_collector` available.
- Helper functions `send_sysex7` and `as_sysex7_packets` transform a `sysex` instance into a sequence of Data Message UMPs.
- There is `midi1_byte_stream_parser` class that converts a MIDI 1 Byte Stream into a UMP Stream, honoring Running Status and MIDI 1 System Exclusive rules.
- `universal_packet::is_midi1_protocol_message()` and `to_midi1_byte_stream()` help to interface with legacy MIDI 1 APIs

Helper Functionalities

Code Examples - UMPs

```
sysex7_collector c7{ [](const sysex7& s) { /* do something here */ } };

if (is_sysex7_packet(packet))
    c7.feed(packet);

sysex8_collector c8{ [](const sysex8&, uint8_t stream_id) { /* do something here */ } };

if (is_sysex8_packet(packet))
    c8.feed(packet);


sysex7 sx7{ manufacturer::native_instruments, { 1, 2, 3, 4, 5, 6, 7, 8, 9 } };

std::vector<data_message> p = as_sysex7_packets(sx7, group_t{ 4 });

auto send_packet = [](const universal_packet&) {};
send_sysex7(sx7, group_t{ 4 }, send_packet);
```

Helper Functionalities

Code Examples (MIDI 1 Byte Stream)

```
auto process_packet = [](midi::universal_packet) { /* do something here */ };
auto process_sysex7 = [](const midi::sysex7&) { /* do something here */ };
midi1_byte_stream_parser p(process_packet, process_sysex7);


const uint8_t data[] = { 0x90, 0x12, 0x34 };
const size_t num_bytes = sizeof(data);
p.feed(data, num_bytes);


uint8_t bytes[8]; // a sysex7 UMP may need up to 8 bytes
if (packet.is_midi1_protocol_message()) {
    auto cnt = to_midi1_byte_stream(packet, bytes);
}
```

Resources

midi2.dev

- github.com/midi2-dev/ni-midi2
- If you want to see *ni-midi2* in production take a look into AmeNote_Protozoa on *midi2.dev* (UUT_FreeRTOS/FreeRTOS_Tasks)
- This talk could only cover the basics of *ni-midi2*, targeting topics interesting for application development. The library covers all of UMP 1.1 and MIDI CI 1.2 including new UMP 1.1 *Stream* and *Flex Data* Messages and MIDI CI *Process Inquiry*
- Even if you may not want to use *ni-midi2*, this talk hopefully helped in pointing out key concepts and tips for MIDI 1 to MIDI 2 code migration

Q&A

© 2024 Franz Detro - Native Instruments