RMR

Release 0.1.8

6r1d

Mar 10, 2021

CONTENTS:

Hello and welcome to RMR documentation.

RMR is a rewrite of RtMIDI project's ALSA[1] part from C++ to C with GLib (it uses both GArray and GAsyncQueue for input). It is made as a personal experiment for use on embedded devices.

It is using the Sphinx documentation generator with Hawkmoth extension.

[1] I will probably rewrite Jack support, as well, but I don't have access and motivation for rewriting Mac / Windows parts.

CHAPTER

ONE

CONTENTS

1.1 Starting

There is a wrapper function for creating each port type, `start_port()`. It accepts a mode argument and calls 4 lower-level wrapper functions.

1.1.1 Types of MIDI ports

There are virtual and non-virtual MIDI ports.

Think about virtual MIDI input and ouptut in terms of endpoints you connect to. "Normal" or "non-virtual" MIDI ports connect to those.

Client—server model, where a client[1] connects to a server is a nice analogy, as well. In this analogy, a server is a "virtual port" and a client is "just a port".

[1] Alsa has its own client term , that has a different meaning: an Alsa seq client that has one or more ports as endpoints to communicate. Each port can be in a mode to read or write data and have other options.

1.1.2 Calling a wrapper

There is a single wrapper that is made of two parts: a port configurator (*setup_port_config()*) and a port starter (*start_port()*).

The table below shows some init examples and there's more info in the examples section.

MIDI port type	Use	Init function
Input	Connect to	`setup_port_config(&port_config, MP_IN);` `start_port(&amidi, port_config);`
Output	Connect to	`setup_port_config(&port_config, MP_OUT);` `start_port(&amidi, port_config);`
Virtual input	Create an endpoint	`setup_port_config(&port_config, MP_VIRTUAL_IN);` `start_port(&amidi, port_config);`
Virtual output	Create an endpoint	`setup_port_config(&port_config, MP_VIRTUAL_OUT);` `start_port(&amidi, port_config);`

1.1.3 Installation

Just clone the repo and include the library as the examples show.

1.1.4 Requirements

- Alsa — should be already installed in your Linux distro
- GNU Make — generally available in your Linux distro
- glib-2.0 — **libglib2.0-dev** in Ubuntu

1.2 Examples

This page shows several examples of using RMR.

1.2.1 Picking a mode

If you are writing a code that other programs will find and connect to, create a **virtual** port.

Otherwise, connect to software or devices using normal **input** or **output** ports.

1.2.2 Building

Open each directory, type "make" and it should be enough to get an executable file in that directory.

"Virtual input" is compatible with output, run "virtual input" first.

"Virtual output" is compatible with input, run "virtual output" first.

1.2.3 Virtual input

```
#include <stdio.h>
#include <stdbool.h>
// Keeps process running until Ctrl-C is pressed.
// Contains a SIGINT handler and keep_process_running variable.
#include "util/exit_handling.h"
#include "util/output_handling.h"
#include "util/midi_parsing.h"
// Main RMR header file
#include "midi/midi_handling.h"

Alsa_MIDI_data * data;
MIDI_in_data * input_data;

MIDI_message * msg;
error_message * err_msg;

RMR_Port_config * port_config;

int main() {
    // Allocate a MIDI_in_data instance, assign a
    // MIDI message queue and an error queue
    prepare_input_data_with_queues(&input_data);

    // Create a port configuration with default values
    setup_port_config(&port_config, MP_VIRTUAL_IN);
```

(continues on next page)

(continued from previous page)

```
    // Start a port with a provided configruation
    start_port(&data, port_config);

    // Assign amidi_data to input_data instance
    assign_midi_data(input_data, data);

    // Open a new port with a pre-set name
    open_virtual_port(data, "rmr", input_data);

    // Don't exit until Ctrl-C is pressed;
    // Look up "output_handling.h"
    keep_process_running = 1;

    // Add a SIGINT handler to set keep_process_running to 0
    // so the program can exit
    signal(SIGINT, sigint_handler);

    // Run until SIGINT is received
    while (keep_process_running) {
        while (g_async_queue_length_unlocked(input_data->midi_async_queue)) {
            // Read a message from a message queue
            msg = g_async_queue_try_pop(input_data->midi_async_queue);
            if (msg != NULL) {
                print_midi_msg_buf(msg->buf, msg->count);
                free_midi_message(msg);
            }
        }
        while (g_async_queue_length_unlocked(input_data->error_async_queue)) {
            // Read an error message from an error queue,
            // simply deallocate it for now
            err_msg = g_async_queue_try_pop(input_data->midi_async_queue);
            if (err_msg != NULL) free_error_message(err_msg);
        }
    }

    // Close a MIDI input port,
    // shutdown the input thread,
    // do cleanup
    destroy_midi_input(data, input_data);

    // Destroy a port configuration
    destroy_port_config(port_config);

    // Exit without an error
    return 0;
}
```

1.2.4 Virtual output

```
#include <stdio.h>
#include <stdbool.h>
// Needed for usleep
#include <unistd.h>
// Needed for nanosleep
#include <time.h>
// Main RMR header file
#include "midi/midi_handling.h"
// Keeps process running until Ctrl-C is pressed.
// Contains a SIGINT handler and keep_process_running variable.
#include "util/exit_handling.h"
#include "util/timing.h"
#include "test_data.h"

Alsa_MIDI_data * amidi_data;

RMR_Port_config * port_config;

// Send "note on" or "note off" signal
bool msg_mode = false;
// Last recorded time in milliseconds
double timer_msec_last;

int main() {
    // Record initial time to a timer
    timer_msec_last = millis();

    // Create a port configuration with default values
    setup_port_config(&port_config, MP_VIRTUAL_OUT);
    // Start a port with a provided configruation
    start_port(&amidi_data, port_config);

    // Send out a series of MIDI messages.
    send_midi_message(amidi_data, MIDI_PROGRAM_CHANGE_MSG, 2);
    send_midi_message(amidi_data, MIDI_CONTROL_CHANGE_MSG, 3);

    // Add a SIGINT handler to set keep_process_running to 0
    // so the program can exit
    signal(SIGINT, sigint_handler);
    // Don't exit until Ctrl-C is pressed;
    // Look up "output_handling.h"
    keep_process_running = 1;

    // Run until SIGINT is received
    while (keep_process_running) {
        if (millis() - timer_msec_last > 100.) {
            timer_msec_last = millis();
            // Send a Note On message
            if (msg_mode) send_midi_message(amidi_data, MIDI_NOTE_ON_MSG, 3);
            // Send a Note Off message
            else          send_midi_message(amidi_data, MIDI_NOTE_OFF_MSG, 3);
            printf("mode: %d\n", msg_mode);
            msg_mode = !msg_mode;
            fflush(stdout);
            usleep(10);
```

(continues on next page)

(continued from previous page)

```
        }
    }

    // Destroy a MIDI output port:
    // close a port connection and perform a cleanup.
    if (destroy_midi_output(amidi_data, NULL) != 0) slog("destructor", "destructor␣
↪error");

    // Destroy a port configuration
    destroy_port_config(port_config);

    // Exit without an error
    return 0;
}
```

Note: inconsistent intervals

I had inconsistent note intervals while making this example.

I've used **millis()** and **usleep(N)** calls. It's possible to feed usleep values that make these intervals inconsistent. Current intervals are **100**, but I'd have to play with this idea more.

```
// Returns the milliseconds since Epoch
double millis() {
        struct timeval cur_time;
        gettimeofday(&cur_time, NULL);
        return (cur_time.tv_sec * 1000.0) + cur_time.tv_usec / 1000.0;
}
```

1.2.5 Input

```
#include <stdbool.h>
// Keeps process running until Ctrl-C is pressed.
// Contains a SIGINT handler and keep_process_running variable.
#include "util/exit_handling.h"
#include "util/output_handling.h"
// Main RMR header file
#include "midi/midi_handling.h"

Alsa_MIDI_data * data;
MIDI_in_data * input_data;

MIDI_port * current_midi_port;

RMR_Port_config * port_config;

MIDI_message * msg;
error_message * err_msg;

int main() {
    // Allocate a MIDI_in_data instance, assign a
    // MIDI message queue and an error queue
    prepare_input_data_with_queues(&input_data);
```

(continues on next page)

(continued from previous page)

```

    // Create a port configuration with default values
    setup_port_config(&port_config, MP_IN);
    // Start a port with a provided configruation
    start_port(&data, port_config);

    // Allocate memory for a MIDI_port instance
    init_midi_port(&current_midi_port);

    // Assign amidi_data to input_data instance
    assign_midi_data(input_data, data);

    // Count the MIDI ports,
    // open if a port containing "Synth" is available
    if (find_midi_port(data, current_midi_port, MP_VIRTUAL_OUT, "rmr") > 0) {
        print_midi_port(current_midi_port);
        open_port(MP_IN, current_midi_port->id, current_midi_port->port_info_name,␣
↪data, input_data);
        // Don't exit until Ctrl-C is pressed;
        // Look up "output_handling.h"
        keep_process_running = 1;
    }

    // Add a SIGINT handler to set keep_process_running to 0
    // so the program can exit
    signal(SIGINT, sigint_handler);

    // Run until SIGINT is received
    while (keep_process_running) {
        while (g_async_queue_length_unlocked(input_data->midi_async_queue)) {
            // Read a message from a message queue
            msg = g_async_queue_try_pop(input_data->midi_async_queue);
            if (msg != NULL) {
                // Print and deallocate a midi message instance
                print_midi_msg_buf(msg->buf, msg->count);
                free_midi_message(msg);
            }
        }
        while (g_async_queue_length_unlocked(input_data->error_async_queue)) {
            // Read an error message from an error queue,
            // simply deallocate it for now
            err_msg = g_async_queue_try_pop(input_data->midi_async_queue);
            if (err_msg != NULL) free_error_message(err_msg);
        }
    }

    // Close a MIDI input port,
    // shutdown the input thread,
    // do cleanup
    destroy_midi_input(data, input_data);

    // Destroy a port configuration
    destroy_port_config(port_config);

    // Exit without an error
    return 0;
}
```

1.2.6 Output

```
#include <stdio.h>
#include <stdbool.h>
// Needed for usleep
#include <unistd.h>
// Needed for nanosleep
#include <time.h>
// Main RMR header file
#include "midi/midi_handling.h"
// Keeps process running until Ctrl-C is pressed.
// Contains a SIGINT handler and keep_process_running variable.
#include "util/exit_handling.h"
#include "util/timing.h"
#include "test_data.h"

Alsa_MIDI_data * data;
MIDI_port * current_midi_port;

RMR_Port_config * port_config;

// Send "note on" or "note off" signal
bool msg_mode = false;
// Last recorded time in milliseconds
double timer_msec_last;

int main() {
    // Record initial time to a timer
    timer_msec_last = millis();

    // Create a port configuration with default values
    setup_port_config(&port_config, MP_VIRTUAL_OUT);
    // Start a port with a provided configruation
    start_port(&data, port_config);

    // Allocate memory for a MIDI_port instance
    init_midi_port(&current_midi_port);

    // Count the MIDI ports,
    // open if a port containing a certain word is available
    if (find_midi_port(data, current_midi_port, MP_VIRTUAL_IN, "rmr") > 0) {
        print_midi_port(current_midi_port);
        open_port(MP_OUT, current_midi_port->id, current_midi_port->port_info_name,␣
↪data, NULL);
        // Don't exit until Ctrl-C is pressed;
        // Look up "output_handling.h"
        keep_process_running = 1;
    }

    // Send out a series of MIDI messages.
    send_midi_message(data, MIDI_PROGRAM_CHANGE_MSG, 2);
    send_midi_message(data, MIDI_CONTROL_CHANGE_MSG, 3);

    // Add a SIGINT handler to set keep_process_running to 0
    // so the program can exit
    signal(SIGINT, sigint_handler);

```

(continues on next page)

(continued from previous page)

```
    // Run until SIGINT is received
    while (keep_process_running) {
        if (millis() - timer_msec_last > 100.) {
            timer_msec_last = millis();
            // Send a Note On message
            if (msg_mode) send_midi_message(data, MIDI_NOTE_ON_MSG, 3);
            // Send a Note Off message
            else          send_midi_message(data, MIDI_NOTE_OFF_MSG, 3);
            printf("mode: %d\n", msg_mode);
            msg_mode = !msg_mode;
            fflush(stdout);
            usleep(10);
        }
    }

    // Destroy a MIDI output port:
    // close a port connection and perform a cleanup.
    if (destroy_midi_output(data, NULL) != 0) slog("destructor", "destructor error");

    // Destroy a port configuration
    destroy_port_config(port_config);

    // Exit without an error
    return 0;
}
```

1.3 Terminology

1.3.1 PPQ / PPQN / TPQN value

Ticks are regular units of time a computer or hardware MIDI device uses for measuring time steps.

Pulses per quarter note, ticks per quarter note or PPQ defines **the base resolution of the ticks**, and it indicates **the number of divisions a quarter note has been split into** (according to Sweetwater's "What is PPQN" page). Thus, PPQ value of zero is invalid.

It is the smallest unit of time used for sequencing note and automation events. PPQ is one of the two values Alsa uses to specify the tempo (another one is MIDI tempo). PPQ cannot be changed while the Alsa queue is running. It must be set before the queue is started. (For this library, it means that PPQ value stays while a port is open.)

According to Wikipedia, "modern computer-based MIDI sequencers designed to capture more nuance may use 960 PPQN and beyond".

Typical PPQ values: 24, 48, 72, 96, 120, 144, 168, 192, 216, 240, 288, 360, 384, 436, 480, 768, 960, 972, 1024.

1.3.2 MIDI Tempo and BPM

We often hear about "beats per minute" or "BPM" in computer music.

BPM itself means "the amount of quarter notes in every minute". Sometimes, BPM values are not strict enough and we might need more control.

For that reason, Alsa and RMR do not rely on "BPM" itself. Instead, Alsa is using so-called "MIDI tempo" and RMR provides an interface to configure it. (MIDI tempo is one of the two values Alsa uses to specify the tempo, another one is PPQ.)

MIDI tempo is not "the amount of quarter notes in every minute", but "time in microseconds per quarter note", so it defines the beat tempo in microseconds. Increasing MIDI tempo will increase the length of a tick, so it will make playback slower.

Name	Description	Example value
T	• MIDI tempo • a value we are trying to find • time in microseconds per quarter note	• unknown • calculated to 500000
BPM	• Beats per minute • the amount of quarter notes in every minute	120 bpm
min_{ms}	A minute in microseconds	60 s * 1000000 ms

We calculate a MIDI tempo by dividing a minute in microseconds by an amount of BPM:

$T = \frac{min_{ms}}{BPM}$

Let us assume our BPM value is 120 and substitute a minute in microseconds to calculate an exact value.

$T = \frac{min_{ms}}{120} = \frac{60 \cdot 1000000}{120} = 500000$

- ALSA project - the C library reference - Sequencer interface - setting queue tempo is the most interesting
- MIDI Technical Fanatic's Brainwashing Center — MIDI file format: tempo and timebase

1.3.3 Tempo in MIDI files

MIDI files use both MIDI tempo and PPQ value to specify the tempo.

1.4 Architecture

1.4.1 Approach to data

RMR is made to work well with in realtime, so:

- It doesn't use OOP features and is written in C
- It is oriented on data (structs in particular) and operations possible with those

1.4.2 Queue use

Both **input** and **virtual input** modes are using thread that communicates with the main code using GLib's Asynchronous Queue mechanism for both messages and errors.

- *MIDI_in_data.midi_async_queue*
- *MIDI_in_data.error_async_queue*

Messages are contained in a structure, *MIDI_message*.

MIDI_message.buf contains message bytes and *MIDI_message.count* contains length of this byte array.

Each *MIDI_message* also contains a *MIDI_message.timestamp* member: a double value, based on snd_seq_real_time_t contents and previous time values.

1.4.3 Double pointers

Often enough, you will be seeing pointers to pointers in code. It is done for allocating memory in a function or something close to that, usually.

1.5 API documentation

1.5.1 Core

A main include file

NANOSECONDS_IN_SECOND
An amount of nanoseconds in a second

QUEUE_TEMPO
A constant that defines the beat tempo in microseconds

QUEUE_STATUS_PPQ
A constant that defines the base resolution of the ticks (pulses per quarter note)

void **init_midi_port** (*MIDI_port* ***current_midi_port*)
Allocates memory for a *MIDI_port* instance

Parameters

- **current_midi_port** – a double pointer used to allocate memory for a *MIDI_port* instance

Since v0.1

void **print_midi_port** (*MIDI_port* **current_midi_port*)
Prints a name of `snd_seq_client_info_t` and `snd_seq_port_info_t` containers

Parameters

- **current_midi_port** – a *MIDI_port* instance to display

Since v0.1

void **free_midi_message** (*MIDI_message* **msg*)
Deallocates memory for *MIDI_message* instance

Parameters

- **msg** – a *MIDI_message* instance

Since v0.1

void **init_amidi_data_instance** (*Alsa_MIDI_data* ***amidi_data*)
Allocates memory for an instance of `Alsa_MIDI_data` or throws an error

Parameters

- **amidi_data** – a double pointer used to allocate memory for a `Alsa_MIDI_data` instance

Since v0.1

void **assign_midi_queue** (*MIDI_in_data* **input_data*)
Creates an assigns a MIDI message queue for a `MIDI_in_data` instance

Parameters

- **input_data** – a pointer containing a `MIDI_in_data` instance for creating a `GAsyncQueue` instance.

Since v0.1

void **assign_midi_data** (*MIDI_in_data* **input_data*, *Alsa_MIDI_data* **amidi_data*)
Assign `Alsa_MIDI_data` instance to `MIDI_in_data` instance

Parameters

- **input_data** – `MIDI_in_data` instance
- **amidi_data** – `Alsa_MIDI_data` instance

Since v0.1

void **set_MIDI_in_callback** (*MIDI_in_data* **input_data*, *MIDI_callback callback*, void **user_data*)
Sets a callback for MIDI input events.

Parameters

- **input_data** – `MIDI_in_data` instance
- **callback** – `MIDI_callback` instance
- **user_data** – an optional pointer to additional data that is passed to the callback function whenever it is called.

Since v0.1

void **enqueue_error** (*MIDI_in_data* **input_data*, char **etype*, char **msg*)
Add an `error_message` instance to `MIDI_in_data` instance

Parameters

- **input_data** – a `MIDI_in_data` instance containing a queue to send an error to
- **etype** – error type string, for example, “V0001”, “S0001”
- **msg** – error message

Since v0.1

void **assign_error_queue** (*MIDI_in_data* **input_data*)
Create an error queue, add to `MIDI_in_data` instance

Parameters

- **input_data** – a `MIDI_in_data` instance to add an error queue to

Since v0.1

int **init_amidi_data**(*Alsa_MIDI_data* *amidi_data, *mp_type_t* port_type)
Fill *Alsa_MIDI_data* struct instance

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance to initialize
- **port_type** – a port type to use, supports all values for *mp_type_t*

Returns **0** on success, **-1** when *port_type* is not *mp_type_t.MP_IN*, *mp_type_t.MP_VIRTUAL_IN*, *mp_type_t.MP_OUT*, *mp_type_t.MP_VIRTUAL_OUT*.

Since v0.1

int **init_seq**(*Alsa_MIDI_data* *amidi_data, **const** char *client_name, *mp_type_t* port_type)
Creates a seq instance, assigns it to *Alsa_MIDI_data*

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **client_name** – client name string
- **port_type** – a port type for a sequencer, supports all values for *mp_type_t*

Returns **0** on success, **-1** on error.

Since v0.1

int **start_input_seq**(*Alsa_MIDI_data* *amidi_data, **const** char *queue_name, *RMR_Port_config* *port_config)
Creates a named input queue, sets its tempo and other parameters.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **queue_name** – a name for a new named queue
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** or **-1** when an error happens

Since v0.1

int **prepare_output**(int *is_virtual*, *Alsa_MIDI_data* *amidi_data, **const** char *port_name)
Creates a MIDI event parser, a virtual port for virtual mode and allocates a buffer for normal mode.

TODO pick a new name, considering calls to snd_midi_event_new, snd_midi_event_init, snd_seq_create_simple_port (virtual mode) and malloc for amidi_data->buffer (normal mode).

Parameters

- **is_virtual** – should the function create normal output port or a virtual one?
- **amidi_data** – *Alsa_MIDI_data* instance
- **port_name** – a name to set for a new virtual output port

Returns **0** on success, **-1** on an error

Since v0.1

unsigned int **port_info**(int *seq, int *pinfo, unsigned int *type*, int *port_number*)
This function is used to count or get the pinfo structure for a given port number.

Parameters

- **snd_seq_t** – Alsa's `snd_seq_t` instance

- **pinfo** – Alsa's `snd_seq_port_info_t` instance
- **type** – Alsa MIDI port capabilities `SND_SEQ_PORT_CAP_READ | SND_SEQ_PORT_CAP_SUBS_READ` or `SND_SEQ_PORT_CAP_WRITE | SND_SEQ_PORT_CAP_SUBS_WRITE`
- **port_number** – use -1 for counting ports and a port number to get port info (in that case a function returns 1).

TODO do we query a client number, not a port number? We are using snd_seq_query_next_client.

Returns port count (or the amount of ports) when a negative port_number value is provided, 1 when a port ID is provided and the port is found, 0 when a port ID is not found

unsigned int **get_midi_port_count**(*Alsa_MIDI_data* *amidi_data, unsigned int *type*)

Counts midi ports for input and output types

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **type** – Alsa MIDI port capabilities, like "`SND_SEQ_PORT_CAP_READ | SND_SEQ_PORT_CAP_SUBS_READ`" or "`SND_SEQ_PORT_CAP_WRITE | SND_SEQ_PORT_CAP_SUBS_WRITE`"

Returns MIDI port count

Since v0.1

void **get_port_descriptor_by_id**(*MIDI_port* *port, *Alsa_MIDI_data* *amidi_data, unsigned int *port_number*, unsigned int *type*)

Updates a **port** pointer to *MIDI_port* instance from a selected *port_number*.

Parameters

- **port** – a pointer to update
- **amidi_data** – *Alsa_MIDI_data* instance
- **port_number** – a port number
- **type** – Alsa MIDI port capabilities `SND_SEQ_PORT_CAP_READ | SND_SEQ_PORT_CAP_SUBS_READ` or `SND_SEQ_PORT_CAP_WRITE | SND_SEQ_PORT_CAP_SUBS_WRITE`

Since v0.1

void **deallocate_input_thread**(**struct** *MIDI_in_data* *input_data)

Free an Alsa MIDI event parser, reset its value in *MIDI_in_data* instance, set current *MIDI_in_data* thread to **dummy_thread_id**

Parameters

- **input_data** – a *MIDI_in_data* instance

Since v0.1

void ***alsa_MIDI_handler**(void *ptr)

A start routine for *alsa_MIDI_handler*.

Parameters

- **ptr** – a void-pointer to *MIDI_in_data*.

Since v0.1

int **open_virtual_port** (*Alsa_MIDI_data* *amidi_data, **const** char *port_name, *MIDI_in_data* *input_data)

Opens a MIDI port with a given name, creates a thread and queues for it.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **port_name** – a char array pointer pointing to a port name string
- **input_data** – a *MIDI_in_data* instance

Returns 0 on success

Since v0.1

int **send_midi_message** (*Alsa_MIDI_data* *amidi_data, **const** unsigned char *message, int size)

Sends a MIDI message using a provided *Alsa_MIDI_data* instance

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **message** – a MIDI message to be sent
- **size** – a size of MIDI message in bytes

Returns **0** on success, **-1** on an error

Since v0.1

int **find_midi_port** (*Alsa_MIDI_data* *amidi_data, *MIDI_port* *port, *mp_type_t* port_type, **const** char *substr)

Finds a MIDI port (port) by a given substring. Matches a substring in *MIDI_port.client_info_name* attribute.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **port** – *MIDI_port* instance
- **port_type** – a port type for a sequencer, supports all values for *mp_type_t*, distinguishes only between "input" and "output", so a virtual input will still be "an input"
- **substr** – a port substring

Returns **1** when a port was found, **-1** on an error, **-2** when an invalid port type was provided

Since v0.1

int **open_port** (*mp_type_t* port_type, unsigned int port_number, **const** char *port_name, *Alsa_MIDI_data* *amidi_data, *MIDI_in_data* *input_data)

Opens a MIDI port by its number. Converted from two RtMIDI methods, initially accepted boolean pointing if it's input.

TODO read all calls in a function and decide if it really makes sense for port_type not to be inverted.

Parameters

- **port_type** – a port type for a opening, supports all values for *mp_type_t*, currently expects MP_IN or MP_VIRTUAL_IN in input context
- **port_number** – number of a port to look for
- **port_name** – a name of a port to set using `snd_seq_port_info_set_name()`
- **amidi_data** – *Alsa_MIDI_data* instance

- **input_data** – a *MIDI_in_data* instance

Returns **0** on success, **-1** on an error

Since v0.1

void **close_port** (*Alsa_MIDI_data* *amidi_data, *MIDI_in_data* *input_data, int *mode*)
Closes input or output port, works for both virtual and not ports

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **input_data** – *MIDI_in_data* instance
- **mode** – accepts `SND_SEQ_OPEN_INPUT` or `SND_SEQ_OPEN_OUTPUT`

Since v0.1

int **destroy_midi_output** (*Alsa_MIDI_data* *amidi_data, *MIDI_in_data* *input_data)
Destroys a MIDI output port: closes a port connection and performs a cleanup.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **input_data** – *MIDI_in_data* instance

Returns **0** on success

Since v0.1

int **destroy_midi_input** (*Alsa_MIDI_data* *amidi_data, *MIDI_in_data* *input_data)
Destroys a MIDI input port: closes a port connection, shuts the input thread down, performs cleanup / deallocations.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **input_data** – *MIDI_in_data* instance

Returns **0** on success

Since v0.1

void **set_port_name** (*Alsa_MIDI_data* *amidi_data, **const** char *port_name)
Set the name of a port_info container in Alsa SEQ interface port information container, update a port info value for an `amidi_data.vport` value.

Parameters

- **amidi_data** – Alsa_MIDI_data instance
- **port_name** – a new name for Alsa seq port

Since v0.1

void **set_client_name** (*Alsa_MIDI_data* *amidi_data, **const** char *client_name)
Set name for a `amidi_data.seq` `snd_seq_t` instance.

Parameters

- **amidi_data** – *Alsa_MIDI_data* instance
- **client_name** – a new name for Alsa seq client

Since v0.1

int **prepare_input_data_with_queues**(*MIDI_in_data* ***input_data*)
Allocates memory for *MIDI_in_data* instance. Assigns two queues: one for MIDI messages and one for errors.

Parameters

- **input_data** – a double pointer used to allocate memory for a *MIDI_in_data* instance

Returns **0** on success

Since v0.1

int **start_virtual_output_port**(*Alsa_MIDI_data* ***amidi_data*, *RMR_Port_config* **port_config*)
A wrapper function for a initializing a virtual output MIDI port.

Parameters

- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** on success

Since v0.1

int **start_output_port**(*Alsa_MIDI_data* ***amidi_data*, *RMR_Port_config* **port_config*)
A wrapper function for starting a non-virtual output port.

Parameters

- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** on success

Since v0.1

int **start_virtual_input_port**(*Alsa_MIDI_data* ***amidi_data*, *RMR_Port_config* **port_config*)
A wrapper function for opening a virtual input port

Parameters

- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** on success

Since v0.1

int **start_input_port**(*Alsa_MIDI_data* ***amidi_data*, *RMR_Port_config* **port_config*)
A wrapper function for starting a non-virtual input port.

Parameters

- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** on success

Since v0.1

int **reset_port_config**(*RMR_Port_config* **port_config*, *mp_type_t port_type*)
Fills *RMR_Port_config* attributes.

A port type is set first, then a queue tempo and ppq you can change. "client_name", "port_name" and "queue_name" are set as "N/A" and then changed to default values needed for a certain port type.

Parameters

- **port_config** – an instance of port configuration: *RMR_Port_config*
- **port_type** – a port type for a sequencer, supports all values for *mp_type_t*

Returns **0** on success

Since v0.1.3

int **setup_port_config**(*RMR_Port_config* ***port_config*, *mp_type_t port_type*)

Allocates memory for an instance of *RMR_Port_config*, sets default values for it

Parameters

- **port_config** – an instance of port configuration: *RMR_Port_config*
- **port_type** – a port type for a sequencer, supports all values for *mp_type_t*

Returns **0** on success

Since v0.1.3

int **destroy_port_config**(*RMR_Port_config* **port_config*)

Deallocates an instance of *RMR_Port_config*

Parameters

- **port_config** – an instance of port configuration (*RMR_Port_config*)

Returns **0** on success

Since v0.1.3

int **start_port**(*Alsa_MIDI_data* ***amidi_data*, *RMR_Port_config* **port_config*)

A wrapper for in, out, virtual in and virtual out MIDI port initialization. TODO add Port_config input.

Parameters

- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance
- **port_config** – an instance of port configuration: *RMR_Port_config*

Returns **0** on success

Since v0.1

int **get_full_port_name**(char **port_name*, unsigned int *port_number*, *mp_type_t port_type*, *Alsa_MIDI_data* **amidi_data*)

Finds a complete port name, including both **client info** and **port info**. A rewrite of both RtMIDI's getPortName functions.

Parameters

- **port_name** – a const char pointer pointing to a string to be filled
- **port_number** – a number of a port to look for
- **port_type** – supports all values for *mp_type_t*, used to select `SND_SEQ_PORT_CAP_READ | SND_SEQ_PORT_CAP_SUBS_READ` or `SND_SEQ_PORT_CAP_WRITE | SND_SEQ_PORT_CAP_SUBS_WRITE` port capabilities
- **amidi_data** – a double pointer to *Alsa_MIDI_data* instance

Returns **0** on success, **-1** when port_name is a null pointer, **-2** when incorrect value was passed for port_type argument, **-3** when port wasn't found.

Since v0.1

1.5.2 MIDI-related structs

MAX_PORT_NAME_LEN
A constant that defines a maximum port name length

struct MIDI_port
A structure that contains names and numbers related to a certain MIDI port in Alsa.

unsigned int **id**
MIDI port number; related to iteration in port_info function and its port_number argument

char **client_info_name**[128]
A name of ALSA's `snd_seq_client_info_t` container. Can be redefined as char pointer later, but MIDI_port will need a destructor.

char **port_info_name**[128]
A name of ALSA's `snd_seq_port_info_t` container. Can be redefined as char pointer later, but MIDI_port will need a destructor.

int **port_info_client_id**
A client id of ALSA's `snd_seq_port_info_t` container

int **port_info_id**
A port id of ALSA's `snd_seq_port_info_t` container

struct RMR_Port_config
A structure that initializes a port. It contains a port type, names for seq interface, queue tempo and ppq for input queues.

struct MIDI_message
A struct to store a single MIDI message in a form of a buffer, its length and a timestamp.

unsigned char ***buf**
Same as bytes, but just an allocated array

long **count**
Length of buf

double **timestamp**
Time in seconds elapsed since the previous message

struct Alsa_MIDI_data
A structure to hold variables related to the ALSA API implementation.

int ***seq**
A pointer to Alsa MIDI sequencer handle

unsigned int **port_num**
Seems to be a MIDI port number that is only set to "-1" now.

int **vport**
A receiver or sender port number, generated by `snd_seq_create_simple_port()`

int ***subscription**
A pointer to ALSA port subscription container instance

int ***coder**
A pointer to Alsa MIDI sequencer event parser

unsigned int **buffer_size**
A **bufsize** value for MIDI event parser (`snd_midi_event_new()`).

unsigned char ***buffer**
A MIDI message buffer.

pthread_t **thread**
An input thread instance

pthread_t **dummy_thread_id**
The ID of the calling thread generated by a `pthread_self()` function

int **last_time**
`snd_seq_real_time_t` instance decoded from Alsa MIDI event, contains nanosecond and second values

int **queue_id**
An input queue is needed to get timestamped events

int **trigger_fds**[2]
File descriptors set by a pipe call in "start_input_seq" function.

int **port_connected**
Tells if a MIDI port is connected, set by *open_port()*

struct MIDI_in_data
A struct to be passed to a `pthread_create()` call.

int ***midi_async_queue**
A GLib asynchronous que to store MIDI messages

int ***error_async_queue**
A GLib asynchronous que to store errors

MIDI_message **message**
A *MIDI_message* instance

unsigned char **ignore_flags**
?

int **do_input**
Marks if data input thread was started

int **first_message**
?

int **using_callback**
Marks if a callback is used; set by *set_MIDI_in_callback()*

MIDI_callback **user_callback**
Current MIDI callback pointer to be called on a message

void ***user_data**
Additional data passed to a callback, seems to always be a void pointer

int **continue_sysex**
Determines if previous message should be extended or a new array should be created

Alsa_MIDI_data ***amidi_data**
Alsa_MIDI_data instance

1.5.3 Typedefs

MIDI-related type definitions

type MIDI_callback
A function definition for processing MIDI callbacks

enum mp_type_t
MIDI port type

enumerator MP_VIRTUAL_IN
Virtual input mode

enumerator MP_VIRTUAL_OUT
Virtual output mode

enumerator MP_IN
Input mode

enumerator MP_OUT
Output mode

1.5.4 Generic helpers

Helper functions

unsigned char **get_last_bytearray_byte**(int **bytearray*)
Retrieve a last byte in a bytearray

Parameters

- **bytearray** – a GLib `GArray` instance to extract the last byte from

Returns a last character of GArray

Since v0.1

1.5.5 Error handling

ERROR_MSG_ETYPE_SIZE
Error handling

struct error_message
A struct to hold an error message.

Since v0.1

char **error_type**[5]
Integer value to differentiate between possible error types

char **message**[255]
Current error message string

void **free_error_message**(*error_message *msg*)
Free the memory used by an error message

Parameters

- **msg** – a message to deallocate

Since v0.1

void **serr**(**const** char *_err_id_, **const** char *_section_, **const** char *_message_)
A function to display error messages. TODO: integrate error classes support, currently the function doesn't differentiate between system and value errors.

Parameters

- **err_id** – an ID of an error to classify it by
- **section** – where error happened
- **message** – error message to send

Since v0.1

1.5.6 Logging

Logging-related functions

void **slog**(**const** char *_section_, **const** char *_message_)
A function to display generic status messages

Parameters

- **section** – a section of a status message to display
- **message** – a message to be displayed

Since v0.1

1.5.7 Future ideas

Getting a port name

Idea: rewrite a **get_port_name** part:

- MIDI in
- MIDI out

1.6 Errors

All errors, thrown by a library.

1.6.1 Error classes

- *System*
- *Value*

1.6.2 Errors

Error name	Error description	Details
V0001	Event parsing error or not a MIDI event	Thrown if `continue_sysex` is not set and `do_decode` is. It is generated by *`alsa_MIDI_handler`* function.
S0001	Error initializing MIDI event parser	Thrown if there was an error in `snd_midi_event_new()`. It is generated by *`alsa_MIDI_handler`* function.

1.7 RtMIDI feature translation

1.7.1 Queue messages as bytes

RtMIDI translates all ALSA messages to bytes.[1]

It is left the same way, although data types might be changed to ensure bytes are always 8-bit (something like `uint8_t`).

1.7.2 Rejoining message chunks

The ALSA sequencer has a maximum buffer size for MIDI sysex events of 256 bytes.

If a device sends sysex messages larger than this, they are segmented into 256 byte chunks.

RtMIDI rejoins 256-byte SysEx message chunks to a single bytearray.

1.7.3 Adding timestamps to MIDI messages

The initial source contains two ways to do that.

First one uses the system time.

```
(void)gettimeofday(&tv, (struct timezone *)NULL);
time = (tv.tv_sec * 1000000) + tv.tv_usec;
```

Second one uses the ALSA sequencer event time data and was implemented by Pedro Lopez-Cabanillas.[2]

The second one was commented quite a bit and it can be found in a current source code.

1.7.4 init_seq client name setting

Uses an internal function and different ordering, but it doesn't seem to influence anything and this doc section is about to be removed.

Before:

```
snd_seq_set_client_name(seq, client_name);
amidi_data->seq = seq;
```

Now:

[1] *I'm not sure if it helps to rejoin SysEx messages or it is done as a way to unify output for Alsa, Jack, etc.*
[2] LibC manual on getting elapsed time.

```
amidi_data->seq = seq;
set_client_name(amidi_data, client_name);
```

1.8 Changes done while rewriting

- Some classes were changed to structs
- There are changes from camelcase to underscore, too

1.8.1 RtMIDI's MidiMessage

RtMIDI	RMR
`MidiMessage`	`MIDI_message` (typedef struct)
`std::vector<unsigned char> bytes`	`unsigned char bytes[MSG_SIZE]`, then `GArray * bytes`
`double timeStamp`	`double timestamp`

1.8.2 AlsaMidiData

RtMIDI	RMR
`seq`	`seq`
`AlsaMidiData`	`Alsa_MIDI_data`, typedef added
`portNum`	renamed to `port_num`, **TODO**: seems to be unused
`vport`	`vport`
`subscription`	`subscription`
`coder`	`coder`
`bufferSize`	`buffer_size`
`buffer`	`buffer`
`thread`	`thread`
`dummy_thread_id`	`dummy_thread_id`
`lastTime`	`last_time`
`queue_id`	`queue_id`
`trigger_fds[2]`	`trigger_fds[2]`
`connected_`	`port_connected`

1.8.3 RtMidiInData

RtMIDI	RMR
`RtMidiInData`	`MIDI_in_data`
`queue`	`midi_async_queue`
`message`	`message`
`ignoreFlags`	`ignore_flags`
`doInput`	`do_input`
`firstMessage`	`first_message`
`apiData`	`amidi_data`
`usingCallback`	`using_callback`
`userCallback`	`user_callback`
`userData`	`user_data`
`continueSysex`	`continue_sysex`

1.8.4 RtMidiCallback function

RtMIDI	RMR
`RtMidiCallback`	`MIDI_callback`

1.8.5 MIDI input opening

RtMIDI	RMR
`SND_SEQ_OPEN_DUPLEX`	`SND_SEQ_OPEN_INPUT`

1.9 Plans

There are things I want to change, but issues are already full with random stuff I'll need to track. This document exists to handle things gradually, not create issues when I solve other issues. (It seems I will always create more than I solve.)

P.S.: this doesn't seem like a good way to continue, because now there's a "clutter 1" and "clutter 2", but I'll let time to show how it goes.

1.9.1 Terminology

Terms I want to change

Check if "port_config->virtual_seq_name" is a term that makes sense, rename if it doesn't.

1.9.2 Callback and queues

Error queue in callback examples

When I finished a callback example, closing issue #14, I noticed there's still a queue for errors that happened in input thread. I am thinking about a different method of error propagation. There are also disconnections, which aren't exactly errors. I think I can replace error queue with callback calls easily. In which thread the callback runs, though? I also feel like I'll have to redesign "serr" function to work well with a new "error" or "status callback" mechanism.

1.9.3 Implementation details

Port capabilities while retrieving a port name

RtMIDI has two Alsa *getPortName* methods, I have one *get_full_port_name* function.

Port capabilities in RtMIDI might've been inverted, and I think that I have inverted them back, properly. I am probably wrong, but I debugged it and it works for some reason.

```
if (in) port_mode = SND_SEQ_PORT_CAP_WRITE | SND_SEQ_PORT_CAP_SUBS_WRITE;
else    port_mode = SND_SEQ_PORT_CAP_READ | SND_SEQ_PORT_CAP_SUBS_READ;
```

"SND_SEQ_PORT_CAP_WRITE" is defined as "writable to this port", too.

UPD: I have changed "in" variable type from bool to a "mp_type_t". I renamed it to "port_type", too. And it did not work with input until I changed arguments back. Now it all makes even less sense. It finds both "virtual input" and "virtual output" values.

It's strange to think about, but maybe I understood boolean function arguments incorrectly?

UPD 2: I was wrong about the arguments, I changed them to less nonsensical ones, it all works better now. I think.

1.9.4 Architectural changes

Default amount of channels?

snd_seq_port_info_set_midi_channels accepts 16 channels. It sounds like a magic constant I'll have to fix, but I am not sure about that. If that is correct, a define seems enough, but moving amount of channels into a port configurator sounds even better.

MIDI2 handling

MIDI2 support doesn't seem to be globally implemented. I receive mixed news: some claim to have finished MIDI2 compatible synths, some claim MIDI2 is not supported by Windows, etc. For now I'll collect information and will try to prepare.

- https://github.com/jackaudio/jack2/issues/535
- https://github.com/atsushieno/cmidi2

1.9.5 Memory management

Check all “malloc”, “calloc”, “alloca” calls are cleaned up after.

1.9.6 Self-reviews

Read the header code through at least 4 times. I’m sure I missed something, be it apidoc or other things.

Current times: 1.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

N

O

P

Q

R

S