libremidi: a cross-platform library for real-time MIDI 1 and 2

Jean-Michaël Celerier[1]

[1]**Société des Arts Technologiques**, Montréal, QC, Canada

ABSTRACT

MIDI (Musical Instrument Digital Interface) has been a fundamental backbone for communication between digital music devices in modern music production. With the recent release of the MIDI 2 standard, available on Linux and macOS and soon on Windows, there is a now need for a robust, efficient, and easy-to-use cross-platform MIDI 2 library. This paper introduces libremidi, a cross-platform C++ library stemming from RtMidi and ModernMidi, rewritten from the ground up for real-time MIDI 2 communication. libremidi provides a simple and consistent API for managing MIDI ports, including hot-plug support, and handling MIDI events. It supports multiple platforms, including Linux (through ALSA RawMidi, ALSA Sequencer, and PipeWire), Windows, macOS, iOS, FreeBSD, and JACK on all platforms, and abstracts the underlying platform-specific MIDI APIs into a unified interface while enabling the end-user to have precise control over the back-ends. Designed for applications that require real-time MIDI communication, such as music production software, digital audio workstations, and interactive installations, libremidi's efficient and low-level API allows developers to build responsive and high-performance applications that can handle multiple MIDI inputs and outputs simultaneously. Work has also been done to approximate real-time guarantees by avoiding memory allocations altogether during input and output. This paper will provide an overview of libremidi's architecture, API, features and improvements over the current cross-platform MIDI state of the art.

1. INTRODUCTION

Communication between Digital Music Instruments (DMIs) and synthesizers, audio workstations, and more generally any kind of music-oriented tool, has been a long-standing field of work in the sound and music computing community: our tools require protocols for communication and interoperation. Published in 1983, the initial MIDI specification has been a game changer for the music and media industry, with it quickly becoming the standard data exchange protocol between DMIs[4]. While originally targeting synthesizers, computers have quickly incorporated real-time MIDI input-output features: in 1985, the Atari ST computer provided built-in MIDI ports. It sold to 2.1 million units.

Every non-trivial computing platform since then has incorporated MIDI: in particular, all the major desktop operating systems and software platform have some form of MIDI implementation. This has led to the creation of multiple cross-platform abstractions for MIDI, to enable music software authors to write software that would be able to input and output MIDI no matter the operating system. Among those, the C and C++ libraries JUCE [6], PortMidi [1] and RtMidi [7] are widely used in the media arts field and serve as base layer of many cross-platform applications and libraries in higher-level languages such as Python, LISP, SuperCollider, etc. or within frameworks such as OpenFrameworks.

As part of the ossia [2] project, a MIDI library was necessary. Originally based on RtMidi, this library has evolved over time in many ways: first by integrating ModernMidi (`github.com/ddiakopoulos/modern-midi`) a MIDI file reading and writing library and sharing most of the common code between those, for instance the in-memory representation of MIDI messages, and then refactoring and improving the library for the needs of ossia, along with updating it to contemporary practices in the C++ community and leveraging modern C++ standards where they allows tangible improvements. In particular, many safety improvements were implemented to the API, more back-ends, more configurability option, and more recently, MIDI 2 support.

This paper gives an overview of the current state of libremidi, including the ways it improves upon the current open-source cross-platform C++ libraries for MIDI communication and solves common requirements in MIDI-compatible software, usually bereft of features such as hot-plugging support. The library documentation is available at the following link: `celtera.github.io/libremidi/foreword.html`. The aim is to enable software authors to trivially capture the benefits of the modern MIDI standards, and improve the stability and reliability of their MIDI-based software.

2. CORE IMPROVEMENTS

2.1 MIDI 2 support

The library has been overhauled to support the new MIDI 2 standard by directly communicating with the relevant operating system APIs and giving to the end-user access to UMP (Universal MIDI Packets) for sending and receiving messages. Note that operating system support is barely maturing: MIDI 2 is supported by macOS since macOS 11 (November 2020), Linux since kernel 6.5 (September 2023), and still not officially supported by Windows but an experimental driver is available (`github.com/microsoft/`

MIDI) and will be distributed with Windows 11 when finished. The MIDI 2 support provided by libremidi only currently covers real-time I/O and not Standard MIDI Files 2 reading and writing.

2.2 New back-ends

A major improvement to the library is the variety of system back-ends supported. Table 1 shows the supported back-ends per-platform. As far as we know, libremidi is the only cross-platform MIDI library to support back-ends such as PipeWire and Windows MIDI Services. The only remaining major back-end to support is Android MIDI. In particular, the PipeWire back-end enables direct support for BLE MIDI on Linux.

Platform	Available backends
Linux, BSD	ALSA RawMidi
	ALSA RawMidi UMP
	ALSA Sequencer
	ALSA Sequencer UMP
	JACK
	PipeWire
macOS	CoreMIDI
	CoreMIDI UMP
	JACK
iOS	CoreMIDI
	CoreMIDI UMP
Windows	WinMM
	WinUWP
	Windows MIDI Services
	JACK
Web	WebMIDI through Emscripten

Table 1. Backends available in libremidi. The MIDI 2 backends are indicated by **bold** text, the other backends are MIDI 1. All backends support input, output and hot-plug.

2.3 Hot-plug support

A major improvement compared to PortMidi and RtMidi is support for hot-plug detection: an `observer` API allows the user to plug-in callback functions that will be called whenever a MIDI device is connected or disconnected from the computer. It allows to filter hardware and / or software devices, for APIs that provide access to cross-process communication such as JACK and PipeWire. For back-ends which do not have any hot-plugging mechanism built-in, detection is simulated with a background thread updating the list of connected MIDI devices regularly.

3. API IMPROVEMENTS

3.1 Improved reliability

Focus has been done on improving the reliability of the original RtMidi implementation, which ended up overtime into almost an entire rewrite due to fundamental reliability issues with the API, due to reliability issues with some back-ends, in particular the Windows Multimedia one.

Among other improvements:

- The entire library goes through testing with the thread, address and undefined behaviour sanitizers [8, 9].
- Callback functions are passed to the constructor to make sure that they are set before the MIDI port can be opened and streaming can start, which was an unsafe possibility and cause of data races in the original library. This also means that the callbacks will by design never miss any message as they will be set before it is possible for a MIDI port to be opened.
- Replace the handling of ports through a numerical index, by a handle object that contains all the necessary information to recover the actual port. This is in particular important in case of unplugging / re-plugging a MIDI device: in the previous design the index of the port could change after the user listed the ports and selected one. Now, the back-ends will leverage more information to try to recover which device to connect to, such as its name, or any extended unique identifier provided by the host API. The reliability of the library in case of for instance semi-faulty USB cable or socket is thus greatly improved.

3.2 Leveraging modern C++ for fun and profit

Some standard C++ types are used to harmonize the library with current C++ ecosystem norms:

- Callbacks are passed through `std::function` instead of separate (data pointer, function pointer) pairs. This allows for instance to simply use C++ lambda functions or function objects for MIDI callbacks.
- Instead of `std::vector` being used as data type for sending messages, `std::span` is used. This required setting the minimum C++ standard version to C++20, which is available on every contemporary system in 2024: continuous integration continually tests on Debian Bullseye, Bookworm, Trixie, macOS, Windows with the three mainstream C++ compilers: clang, GCC and MSVC. This allows the library to be agnostic to the data type used by its user for sending MIDI 1 messages: `std::vector`, `std::array`, raw C arrays, `boost::container::small_vector`, etc. – the only requirement is for the type to model a contiguous sequence of bytes. This removes the need for memory allocations if the end-user was using a different type than `std::vector` for internal MIDI byte storage.
- Back-ends use `std::semaphore` for cross-platform semaphore support. That is especially important for the JACK backend, now supported on macOS and Windows in addition to Linux with a single implementation.

4. CONTROL TO THE USER

A driving force in the libremidi design is to give as much control to the user as possible while still maintaining a cross-platform API for input and output management. This has been done at multiple levels in our proposed evolution of the library.

4.1 Custom configuration mechanism

A configuration mechanism for construction of MIDI input, output and observer objects was introduced. The configuration contains both a generic part common to all back-ends and an API-specific part.

Listing 1 showcases an example: how the end-user can create a CoreMIDI client with specific settings. This solves long-standing issues with multiple MIDIClient opening and closing in the same process [1].

Listing 1. Creating a MIDI device with a CoreMIDI-specific configuration, allowing to plug-in a global CoreMIDI MIDIClientRef object:

```
using namespace libremidi;

// Generic configuration
input_configuration in_config;
in_config.on_message = [] (auto msg) {
  // Process the message
};

// Specific configuration
coremidi_input_configuration in_api_config;
in_api_config.client_name = "My client";
in_api_config.context = getMyAppMIDIClientRef();

midi_in in{in_config, in_api_config};
in.open_virtual_port();
```

4.2 Removal of the implicit queue

The original RtMidi implementation allowed retrieving MIDI input messages from within the main thread, by implementing a queue (whose implementation is currently not thread-safe due to concurrent access to non-atomic variables). This feature was removed in libremidi for two reasons: first, the research field of thread-safe lock-free ring-buffers and queues is in constant evolution, with new solutions appearing in the recent years pushing the boundaries of efficiency [3, 5]. Second, we argue most professional audio and MIDI applications already carry preexisting implementations of a lock-free queue or ring-buffer, which can be trivially fed in the callback provided by the user. Removing this simplified the code logic in all back-ends. Sample code files showcasing the integration with both a custom message queue and a C++20 coroutine-based system are provided with the library.

4.3 Logging

Logging in PortMidi is done through `printf` calls, and in RtMidi through `std::cout` and `std::cerr` calls. This makes precise control of the logging in host applications impossible as one would not be able to redirect such calls done in the threaded loop of the library with full reliability. libremidi allows the end-user to provide explicit logging callbacks passed to the object constructors so that error messages can be redirected wherever fit.

[1] https://github.com/thestk/rtmidi/issues/155

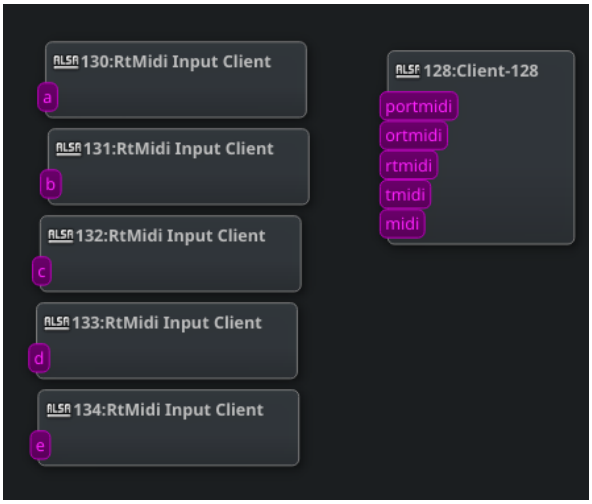

Figure 1. A patchbay with virtual MIDI input ports opened by RtMidi (left) and PortMidi (right)

4.4 Context sharing

To give context on the existing state of the art, RtMidi did not allow any sharing: each MIDI object would create a new client with a single port. Conversely, PortMidi would create all the ports in a single client. This can be seen in Fig. 1, which shows the state of the MIDI graph on a Linux system when creating five virtual ports in both these libraries. For improved flexibility, libremidi's design allows the user to pass an existing API-specific client object to enable creating ports in a preexisting, possibly shared, context: a single or multiple ALSA, JACK client, PipeWire filter, etc. Thus, the end-user gets maximum control over the context of their MIDI ports.

4.5 Poll and thread-loop management

Every API handles input and output differently. They can be split in three major categories for input: poll-driven, message-callback-driven and synchronous-callback-driven. Poll-driven (ALSA) means that the API provides a file descriptor to be polled with the UNIX `poll` API. Readiness of the file descriptor will indicate that new messages can be read. Message-callback-driven (Windows APIs, CoreMIDI) means that an operating-system-managed thread will call a user-provided callback on every new incoming MIDI message. Synchronous-callback-driven (JACK, PipeWire) means that the API calls a user-provided callback at regular interval, generally driven by the audio rate / audio buffer size.

For message-callback-driven APIs, there is no meaningful control possible: the OS handles the threading associated with the library internally. For other APIs, it may be beneficial to share the threading and polling loop across instances, or even with other operations that MIDI I/O. For

instance, a simple visualization app that does not want to bother with the risks of multi-threading may want to run the ALSA sequencer polling in an existing poll-driven application main-loop to have everything happen in a single thread. Another important capability missing from PortMidi and RtMidi is the ability to combine audio and MIDI processing in a single callback operation for instance for DAWs using APIs that support both audio and MIDI ports from the same client, such as JACK and PipeWire, and for which we want to make sure that the flow of operation on each audio scheduler tick is: *Read MIDI input → Process audio → Write MIDI output*.

libremidi's design allows for the relevant back-end to plug callback functions that when present will make the implementation refrain from creating its own thread or start its own processing through JACK / PipeWire: instead, the user will be provided with functions that they can themselves add in their audio callback processing loop, to guarantee orderly, frame-accurate synchronous operation. Examples are provided for each back-end where this feature is relevant.

4.6 Error handling

Error handling in RtMidi was done through exceptions. While in line with the C++ RAII-based object model, this can prove challenging in the audio field where it is common to require software to be compiled without exception support. Error handling across the library was thus redeveloped by leveraging the latest C++ developments on error-code-based error handling: `std::error` as proposed by Herb Sutter in [10]. We use for this an open-source prototype implementation (`github.com/charles-salvia/std_error`) that should be close to the final version expected for C++26. In particular, this implementation enables the user to react both to a given platform-specific error, for instance by retrieving an `OSStatus` on macOS or `MMERR` on Windows, viewing it as a more generic cross-platform `std::errc` POSIX error code through an equivalence class system, and getting a text string, all the while not requiring any memory allocation and being fully supported in header-only libraries and across DLLs, unlike the C++11 solution (`std::error_code`).

4.7 Advanced time-stamping

The library provides an extensive array of time-stamping options: the user can choose at run-time between:

- No time-stamping.
- Relative timestamps between successive messages. The original RtMidi behaviour.
- Absolute timestamp as provided by the OS API. A utility function has been added to enable the end-user to query the API for its current time – very often, this is the system's monotonic clock.
- Absolute timestamp as provided by the OS-provided monotonic clock. This allows to very easily have coherent time-stamping across all the parts of a media application but may lose precision compared to the API timestamp which may be able to leverage hardware MIDI time-stamping.
- Audio frame timestamp. This will for the JACK and PipeWire cases use the offset in samples as timestamp, to enable sample-accurate operation without requiring calculations from the end user.
- Finally, the user can provide a custom time-stamping callback, to leverage for instance an external clock source.

In addition, the timestamp format has been changed from double-precision to 64-bit integer with nanosecond precision to avoid subtle precision issues (except in the audio frame case where the unit of time is then the audio frame index).

4.8 Compilation options

The library has been changed so that it could optionally be used *header-only*: that is, one can compile a MIDI example with a simple invocation:

```
$ g++ \
  examples/midi2_echo.cpp \
  -Iinclude \
  -DLIBREMIDI_HEADER_ONLY=1 \
  -DLIBREMIDI_ALSA=1 \
  -std=c++20
```

The library also enables using different types than a heap-based dynamic vector, `std::vector`, for storing of MIDI 1 messages: for instance, it is possible to leverage `boost::small_vector` or even `boost::static_vector` to reduce heap allocations further, by toggling CMake flags. These data types implement respectively small-vector optimization (up to 20 bytes can be stored in the object without requiring memory allocation) or use of fixed storage to prevent dynamic memory allocations altogether. Note that for MIDI 2 support, no dynamic allocation is used in any case.

4.9 Dynamic loading of dependencies

Unlike most libraries, libremidi loads all its dependencies on Linux through `dlopen` / `dlsym`. This allows to provide a build of the library that supports all back-ends (JACK, PipeWire, ALSA) but will still work if the user's system does not provide the required library, unlike traditional dynamic linking which would fail on startup if the end-user's machine does not have all the necessary libraries. This is especially necessary for JACK and PipeWire which not everyone may be running: we want the library to be usable in the context of AppImages which can run on a wide variety of Linux distributions.

5. FROM MIDI 1 TO MIDI 2

The major advance in the library is support for the new MIDI 2 standard. We base ourselves on the June 2023 MIDI 2 standard documents [2]. This section gives an overview of the MIDI 2 standard, and details how it is implemented in libremidi.

[2] `https://midi.org/details-about-midi-2-0-midi-ci-profiles-and-property-exchange-updated-june-2023`

5.1 A MIDI 2 primer

MIDI 2 is a major advance in the MIDI protocol. It is currently still evolving, with the latest changes to the standard documents being dated from June 2023.

Multiple specifications define what MIDI 2 exactly is:

> MIDI 2.0 builds on the core principles, architecture, and semantics of MIDI 1.0. Primary features are auto-configuration, enabled by bidirectional connections enabling devices to discover details about other connected devices, and an expanded data format for higher resolution with extensibility to define many new messages in the future.

The core elements of MIDI 2 are: MIDI Capability Inquiry (MIDI-CI) discovery and the UMP message format. To be a compliant MIDI 2 device, one has to implement at least one of these two. In particular, MIDI-CI works over a MIDI 1 transport through SysEx messages and does not require UMP support.

5.2 Core differences with MIDI 1

For artists, the most immediate benefits will be, in short, "more data": pitch bend, control change and polyphonic aftertouch now carry 32 bits of information instead of MIDI 1's 7 bits ; note messages 16 bits of velocity and 6 bits of custom attribute data ; instead of only 16 channels, a MIDI 2 link can carry 16 groups of 16 channels each ; new controllers are introduced ; SysEx messages can now use 8-bit bytes instead of 7-bit in MIDI 1. For implementers, the low-level protocol is much more complex with multiple round-trips to define the capabilities of MIDI 2 devices, which are supposed to be retro-compatible with MIDI 1.

For library users such as the target audience of this paper, many design choices in MIDI 2 also greatly simplify usage of MIDI 2: we recommend that over time every desktop computer user migrates to MIDI 2 as an end-user API. In particular, the three major desktop operating systems, Linux, macOS, and Windows, provide automatic translation of MIDI 1 devices to the MIDI 2 API.

In particular, MIDI 2 messages are now in a new, fixed-size format, unlike MIDI 1 messages which could be of arbitrary-length due to the SysEx format. The Universal Midi Packet (UMP) can be 32, 64, 96 or 128 bits, generally organized as four unsigned 32-bit values. This obviates the need for dynamic memory allocation which was often necessary with SysEx support in MIDI 1 APIs, and is a better fit to the current CPUs, with generally 64-bit registers and 64-byte L1 cache lines.

Another feature, necessary in the MIDI 1 times and made irrelevant with current hardware performance, is the running status which is not part of MIDI 2 anymore: every message now carries the entire information needed to interpret it. Finally, this UMP message format carries more space for future extensions to the protocol: many are already in the works by the MIDI association.

Besides the improved resolution, MIDI 2 also provides a complete protocol for bidirectional information exchange between devices, MIDI-CI. This allows two MIDI-CI-compatible devices to interoperate more closely: a synthesizer can now for instance report its controls to a MIDI controller. MIDI-CI communication follows rules established in the MIDI standard documents.

6. RESULTS AND DISCUSSION

The library has already seen usage in the media ecosystem: it has been validated by years of use in the ossia score software sequencer and has also been integrated in multiple open-source projects, for instance OBS Studio's MIDI plugin.

6.1 Porting to libremidi

Multiple examples are provided by the library, showing different degrees of integration. As a starting point, Listing 2 shows how one may implement a basic MIDI 2 echo between the first two detected devices.

Listing 2. Minimal MIDI 2 echo example

```
#include <libremidi/libremidi.hpp>
#include <iostream>

int main() try {
  using namespace libremidi;
  namespace lm2 = libremidi::midi2;

  /// The observer object enumerates available
  /// inputs and outputs:
  observer obs{{}
    , lm2::observer_default_configuration()};
  auto pi = obs.get_input_ports();
  auto po = obs.get_output_ports();
  if (pi.empty() || po.empty())
    throw std::runtime_error("No MIDI in / out
        available");

  /// Create a MIDI 2 out object:
  midi_out midiout{{}
    , lm2::out_default_configuration()};

  // Open the midi output object with a given
  // handle obtained from the observer
  if (auto err = midiout.open_port(po[0]);
      err != stdx::error{}) {
     // error-handling decisions are deferred
     // to the user of the library.
     // Here for simplicity we throw.
    err.throw_exception();
  }

  /// Create a MIDI 2 input object:
  // An UMP-friendly callback
  auto on_ump =
    [&](const libremidi::ump& message) {
    // Note: one may want to lock the
    // midiout object with a mutex if
    // it was to be used at the same time
    if (midiout.is_port_connected())
      midiout.send_ump(message);
  };
  midi_in midiin{{.on_message = on_ump}
    , lm2::in_default_configuration()};

  // pi[0] and po[0] are distinct types:
  // one cannot mistakenly open an input
  // with an output handle
  if (auto err = midiin.open_port(pi[0]);
      err != stdx::error{})
    err.throw_exception();

  // Wait until we exit
  char input; std::cin.get(input);
}
catch (const std::exception& error) {
        std::cerr << error.what() << std::endl;
        return EXIT_FAILURE;
}
```

6.2 Performance results

Summary performance benchmarking of a MIDI echo implementation between libremidi and RtMidi has been done. A core point though is that RtMidi does not allow accurate benchmarking: the RtMIDI output does not inform the user of an error if a message could not be written (for instance if the operating system's internal MIDI buffer is full, its `sendMessage` function can fail silently, e.g. in the JACK back-end). In contrast, libremidi allows to take action and count the number of actually successfully transmitted messages. With this taken into account, our sample benchmark run (sending 1 000 000 messages as fast as possible in a round-trip fashion on Linux with the ALSA Sequencer back-end on a Core i7-10750H CPU, five runs, `-Ofast -flto`) gives an average of 720ms (best: 700.3) with libremidi, and 816 (best: 787.9) with RtMidi. That said, most of the MIDI performance comes from the underlying back-end implementation and both libraries have very little overhead: 88.5 percent of the execution time of our `send_message` implementation is spent inside the ALSA API.

6.3 MIDI-CI support ?

In short, from our prolonged study of the MIDI 2 standard and implementation experience since the first MIDI 2 document release in 2020, our conclusion is that a library such as libremidi does not operate at the abstraction level at which MIDI-CI is useful, which is specific to each application. On the other hand, providing UMP support is the right fit: libraries defining a specific MIDI 2 state machine can then just use libremidi for the cross-platform input-output aspect and feed MIDI-CI messages to an appropriate MIDI 2 data processing library. Multiple libraries implement C and C++ API support for high-level MIDI 2 messages, such as creating UMP packets matching a specific step of a MIDI-CI communication from simple function calls. For the sake of ease-of-use, libremidi has been made compatible with the two major implementations: `ni-midi2` [3] developed by Native Instruments and `cmidi2` [4] developed by Atsushi Eno. In particular, one can directly send `ni-midi2` data types to libremidi outputs, and the libremidi message type can be converted to the `ni-midi2` UMP data type automatically. Examples in the codebase show how to initiate a basic MIDI-CI communication by combining libremidi and `ni-midi2`, only tested so far with the MIDI 2 Workbench software provided by the MIDI association due to the lack of MIDI 2 hardware available.

6.4 UMP, a friendly format

Every desktop platform provide automatic MIDI 1 support with their UMP APIs: that is, one can just use the MIDI 2 UMP APIs, forget about MIDI 1 entirely and get access to the entirety of the MIDI 1 and MIDI 2 devices, at the cost of restricting the applications to recent operating system versions (macOS 11+, Linux 6.5+, and yet unreleased Windows versions). Due to the major advantages of the UMP format, we heartily recommend a complete migration of MIDI support in media applications to the UMP format. The only drawback is that some back-ends do not provide MIDI 2 support yet: JACK, PipeWire, WebMIDI. Both `cmidi2` and `ni-midi2` provide efficient implementations of the conversion from MIDI 1 messages to MIDI 2 UMPs. libremidi leverages this for the MIDI output: all the back-ends supported by the library can receive UMPs including the MIDI 1 ones. Conversion of MIDI 1 input to UMP is a work-in-progress.

[3] `github.com/midi2-dev/ni-midi2`

[4] `github.com/atsushieno/cmidi2`

7. CONCLUSIONS

We introduce libremidi, a modern C++ take on the perennial problem of cross-platform MIDI I/O library. Based on preexisting code, it evolved to enable support of the more recent MIDI 2 standard, and provides improved safety and reliability over existing C & C++ MIDI libraries. Future directions will mainly be about providing more back-ends, for instance for Android and BSDs, and ensuring stability of the library now that the necessary features are there.

8. REFERENCES

[1] R Bencina, P Burk, and R Dannenberg. *portmidi-Platform Independent Library for MIDI*. 2007. URL: `https : / / github . com / PortMidi / portmidi`.

[2] Jean-Michaël Celerier et al. "OSSIA: Towards a Unified Interface for Scoring Time and Interaction". In: *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Paris, France, 2015.

[3] Reginald Austin Frank. "Designing a High Throughput Bounded Multi-Producer, Multi-Consumer Queue". PhD thesis. 2021.

[4] Gareth Loy. "Musicians make a standard: The MIDI phenomenon". In: *Computer Music Journal* 9.4 (1985), pp. 8–26.

[5] Gal Milman-Sela et al. "BQ: a lock-free queue with batching". In: *ACM Transactions on Parallel Computing* 9.1 (2022), pp. 1–49.

[6] Martin Robinson. *Getting started with JUCE*. Packt Publishing Ltd, 2013. DOI: `10 . 4324 / 9780429455971-4`.

[7] Gary P. Scavone and Perry R. Cook. "RtMidi, RtAudio, and a synthesis toolkit (STK) update". In: *Synthesis* (2004).

[8] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.

[9] Konstantin Serebryany et al. "AddressSanitizer: A fast address sanity checker". In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[10] Herb Sutter. "Zero-overhead deterministic exceptions: Throwing values". In: *C++ open-std proposal P0709* 2 (2019), p. 10.