The RtMidi Tutorial

Introduction

RtMidi is a set of C++ classes (**RtMidiIn**, **RtMidiOut** and API-specific classes) that provides a common API (Application Programming Interface) for realtime MIDI input/output across Linux (ALSA & JACK), Macintosh OS X (CoreMIDI & JACK), Windows (Multimedia Library & UWP), Web MIDI, iOS and Android systems. **RtMidi** significantly simplifies the process of interacting with computer MIDI hardware and software. It was designed with the following goals:

- object oriented C++ design
- simple, common API across all supported platforms
- only one header and one source file for easy inclusion in programming projects
- MIDI device enumeration

Where applicable, multiple API support can be compiled and a particular API specified when creating an RtAudio instance.

MIDI input and output functionality are separated into two classes, **RtMidiIn** and **RtMidiOut**. Each class instance supports only a single MIDI connection. **RtMidi** does not provide timing functionality (i.e., output messages are sent immediately). Input messages are timestamped with delta times in seconds (via a `double` floating point type). MIDI data is passed to the user as raw bytes using an std::vector<unsigned char>.

What's New (Version 6.0.0)

The version number has been bumped to 6.0.0 because new APIs (Android and Windows UWP) were added. Changes in this release include:

- run "git log 5.0.0..HEAD" to see commits since last release
- new Android API (thanks to YellowLabrador!)
- new Windows UWP API support (thanks to Masamichi Hosoda!)
- various build system updates and code efficiencies

Download

Latest Release (3 August 2023): Version 6.0.0

Getting Started

The first thing that must be done when using **RtMidi** is to create an instance of the **RtMidiIn** or **RtMidiOut** subclasses. **RtMidi** is an abstract base class, which itself cannot be instantiated. Each default constructor attempts to establish any necessary "connections" with the underlying MIDI system. **RtMidi** uses C++ exceptions to report errors, necessitating try/catch blocks around many member functions. An **RtMidiError** can be thrown during instantiation in some circumstances. A warning message may also be reported if no MIDI devices are found during instantiation. The **RtMidi** classes have been designed to work with "hot pluggable" or virtual (software) MIDI devices, making it possible to connect to MIDI devices that may not have been present when the classes were instantiated. The following code example demonstrates default object construction and destruction:

```
#include "RtMidi.h"

int main() {
  try {
    RtMidiIn midiin;
  } catch (RtMidiError &error) {
    // Handle the exception here
    error.printMessage();
  }
  return 0;
}
```

Obviously, this example doesn't demonstrate any of the real functionality of **RtMidi**. However, all uses of **RtMidi** must begin with construction and must end with class destruction. Further, it is necessary that all class methods that can throw a C++ exception be called within a try/catch block.

Error Handling

RtMidi uses a C++ exception handler called **RtMidiError**, which is declared and defined in **RtMidi.h**. The **RtMidiError** class is quite simple but it does allow errors to be "caught" by **RtMidiError::Type**. Many **RtMidi** methods can "throw" an **RtMidiError**, most typically if a driver error occurs or an invalid function argument is specified. There are a number of cases within **RtMidi** where warning messages may be displayed but an exception is not thrown. A client error callback function can be specified (via the **RtMidi::setErrorCallback** function) that is invoked when an error occurs. By default, error messages are not automatically displayed in **RtMidi** unless the preprocessor definition **RTMIDI_DEBUG** is defined during compilation. Messages associated with caught exceptions can be displayed with, for example, the **RtMidiError::printMessage()** function.

Probing Ports / Devices

A client generally must query the available MIDI ports before deciding which to use. The following example outlines how this can be done.

```
// midiprobe.cpp

#include <iostream>
#include <cstdlib>
#include "RtMidi.h"

int main()
{
  RtMidiIn  *midiin = 0;
  RtMidiOut *midiout = 0;
```

```
  // RtMidiIn constructor
  try {
    midiin = new RtMidiIn();
  }
  catch ( RtMidiError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  // Check inputs.
  unsigned int nPorts = midiin->getPortCount();
  std::cout << "\nThere are " << nPorts << " MIDI input sources available.\n";
  std::string portName;
  for ( unsigned int i=0; i<nPorts; i++ ) {
    try {
      portName = midiin->getPortName(i);
    }
    catch ( RtMidiError &error ) {
      error.printMessage();
      goto cleanup;
    }
    std::cout << "  Input Port #" << i+1 << ": " << portName << '\n';
  }

  // RtMidiOut constructor
  try {
    midiout = new RtMidiOut();
  }
  catch ( RtMidiError &error ) {
    error.printMessage();
    exit( EXIT_FAILURE );
  }

  // Check outputs.
  nPorts = midiout->getPortCount();
  std::cout << "\nThere are " << nPorts << " MIDI output ports available.\n";
  for ( unsigned int i=0; i<nPorts; i++ ) {
    try {
      portName = midiout->getPortName(i);
    }
    catch (RtMidiError &error) {
      error.printMessage();
      goto cleanup;
    }
    std::cout << "  Output Port #" << i+1 << ": " << portName << '\n';
  }
  std::cout << '\n';

  // Clean up
 cleanup:
  delete midiin;
  delete midiout;

  return 0;
}
```

Note that the port enumeration is system specific and will change if any devices are unplugged or plugged (or a new virtual port opened or closed) by the user. Thus, the port numbers should be verified immediately before opening a port. As well, if a user unplugs a device (or closes a virtual port) while a port connection exists to that

device/port, a MIDI system error will be generated.

MIDI Output

The **RtMidiOut** class provides simple functionality to immediately send messages over a MIDI connection. No timing functionality is provided. Note that there is an overloaded **RtMidiOut::sendMessage()** function that does not use std::vectors.

In the following example, we omit necessary error checking and details regarding OS-dependent sleep functions. For a complete example, see the `midiout.cpp` program in the `tests` directory.

```
// midiout.cpp

#include <iostream>
#include <cstdlib>
#include "RtMidi.h"

int main()
{
  RtMidiOut *midiout = new RtMidiOut();
  std::vector<unsigned char> message;

  // Check available ports.
  unsigned int nPorts = midiout->getPortCount();
  if ( nPorts == 0 ) {
    std::cout << "No ports available!\n";
    goto cleanup;
  }

  // Open first available port.
  midiout->openPort( 0 );

  // Send out a series of MIDI messages.

  // Program change: 192, 5
  message.push_back( 192 );
  message.push_back( 5 );
  midiout->sendMessage( &message );

  // Control Change: 176, 7, 100 (volume)
  message[0] = 176;
  message[1] = 7;
  message.push_back( 100 );
  midiout->sendMessage( &message );

  // Note On: 144, 64, 90
  message[0] = 144;
  message[1] = 64;
  message[2] = 90;
  midiout->sendMessage( &message );

  SLEEP( 500 ); // Platform-dependent ... see example in tests directory.

  // Note Off: 128, 64, 40
  message[0] = 128;
  message[1] = 64;
  message[2] = 40;
  midiout->sendMessage( &message );
```

```
  // Clean up
 cleanup:
  delete midiout;

  return 0;
}
```

MIDI Input

The **RtMidiIn** class uses an internal callback function or thread to receive incoming MIDI messages from a port or device. These messages are then either queued and read by the user via calls to the **RtMidiIn::getMessage()** function or immediately passed to a user-specified callback function (which must be "registered" using the **RtMidiIn::setCallback()** function). Note that if you have multiple instances of **RtMidiIn**, each may have its own thread. We'll provide examples of both usages.

The **RtMidiIn** class provides the **RtMidiIn::ignoreTypes()** function to specify that certain MIDI message types be ignored. By default, system exclusive, timing, and active sensing messages are ignored.

Queued MIDI Input

The **RtMidiIn::getMessage()** function does not block. If a MIDI message is available in the queue, it is copied to the user-provided `std::vector<unsigned char>` container. When no MIDI message is available, the function returns an empty container. The default maximum MIDI queue size is 1024 messages. This value may be modified with the RtMidiIn::setQueueSizeLimit() function. If the maximum queue size limit is reached, subsequent incoming MIDI messages are discarded until the queue size is reduced.

In the following example, we omit some necessary error checking and details regarding OS-dependent sleep functions. For a more complete example, see the `qmidiin.cpp` program in the `tests` directory.

```
// qmidiin.cpp

#include <iostream>
#include <cstdlib>
#include <signal.h>
#include "RtMidi.h"

bool done;
static void finish(int ignore){ done = true; }

int main()
{
  RtMidiIn *midiin = new RtMidiIn();
  std::vector<unsigned char> message;
  int nBytes, i;
  double stamp;

  // Check available ports.
  unsigned int nPorts = midiin->getPortCount();
  if ( nPorts == 0 ) {
    std::cout << "No ports available!\n";
    goto cleanup;
  }
  midiin->openPort( 0 );
```

```
  // Don't ignore sysex, timing, or active sensing messages.
  midiin->ignoreTypes( false, false, false );

  // Install an interrupt handler function.
  done = false;
  (void) signal(SIGINT, finish);

  // Periodically check input queue.
  std::cout << "Reading MIDI from port ... quit with Ctrl-C.\n";
  while ( !done ) {
    stamp = midiin->getMessage( &message );
    nBytes = message.size();
    for ( i=0; i<nBytes; i++ )
      std::cout << "Byte " << i << " = " << (int)message[i] << ", ";
    if ( nBytes > 0 )
      std::cout << "stamp = " << stamp << std::endl;

    // Sleep for 10 milliseconds ... platform-dependent.
    SLEEP( 10 );
  }

  // Clean up
 cleanup:
  delete midiin;

  return 0;
}
```

MIDI Input with User Callback

When set, a user-provided callback function will be invoked after the input of a complete MIDI message. It is possible to provide a pointer to user data that can be accessed in the callback function (not shown here). It is necessary to set the callback function immediately after opening the port to avoid having incoming messages written to the queue (which is not emptied when a callback function is set). If you are worried about this happening, you can check the queue using the RtMidi::getMessage() function to verify it is empty (after the callback function is set).

In the following example, we omit some necessary error checking. For a more complete example, see the `cmidiin.cpp` program in the `tests` directory.

```
// cmidiin.cpp

#include <iostream>
#include <cstdlib>
#include "RtMidi.h"

void mycallback( double deltatime, std::vector< unsigned char > *message, void *userData )
{
  unsigned int nBytes = message->size();
  for ( unsigned int i=0; i<nBytes; i++ )
    std::cout << "Byte " << i << " = " << (int)message->at(i) << ", ";
  if ( nBytes > 0 )
    std::cout << "stamp = " << deltatime << std::endl;
}

int main()
```

```
{
  RtMidiIn *midiin = new RtMidiIn();

  // Check available ports.
  unsigned int nPorts = midiin->getPortCount();
  if ( nPorts == 0 ) {
    std::cout << "No ports available!\n";
    goto cleanup;
  }

  midiin->openPort( 0 );

  // Set our callback function.  This should be done immediately after
  // opening the port to avoid having incoming messages written to the
  // queue.
  midiin->setCallback( &mycallback );

  // Don't ignore sysex, timing, or active sensing messages.
  midiin->ignoreTypes( false, false, false );

  std::cout << "\nReading MIDI input ... press <enter> to quit.\n";
  char input;
  std::cin.get(input);

  // Clean up
 cleanup:
  delete midiin;

  return 0;
}
```

Virtual Ports

The Linux ALSA, Macintosh CoreMIDI and JACK APIs allow for the establishment of virtual input and output MIDI ports to which other software clients can connect. **RtMidi** incorporates this functionality with the **RtMidiIn::openVirtualPort()** and **RtMidiOut::openVirtualPort()** functions. Any messages sent with the **RtMidiOut::sendMessage()** function will also be transmitted through an open virtual output port. If a virtual input port is open and a user callback function is set, the callback function will be invoked when messages arrive via that port. If a callback function is not set, the user must poll the input queue to check whether messages have arrived. No notification is provided for the establishment of a client connection via a virtual port. The **RtMidi::isPortOpen()** function does not report the status of ports created with the **RtMidi::openVirtualPort()** function.

Compiling

In order to compile **RtMidi** for a specific OS and API, it is necessary to supply the appropriate preprocessor definition and library within the compiler statement:

OS:	MIDI API:	Preprocessor Definition:	Library or Framework:	Example Compiler Statement:

Linux	ALSA Sequencer	**LINUX_ALSA**	`asound, pthread`	`g++ -Wall -D__LINUX_ALSA__ -o midiprobe midiprobe.cpp RtMidi.cpp -lasound -lpthread`
Linux or Mac	JACK MIDI	**UNIX_JACK**	`jack`	`g++ -Wall -D__UNIX_JACK__ -o midiprobe midiprobe.cpp RtMidi.cpp -ljack`
Macintosh OS X	CoreMIDI	**MACOSX_CORE**	`CoreMIDI, CoreAudio, CoreFoundation`	`g++ -Wall -D__MACOSX_CORE__ -o midiprobe midiprobe.cpp RtMidi.cpp -framework CoreMIDI -framework CoreAudio -framework CoreFoundation`
Windows	Multimedia Library	**WINDOWS_MM**	`winmm.lib, multithreaded`	*compiler specific*

The example compiler statements above could be used to compile the `midiprobe.cpp` example file, assuming that `midiprobe.cpp`, **`RtMidi.h`** and `RtMidi.cpp` all exist in the same directory.

Debugging

If you are having problems getting **RtMidi** to run on your system, try passing the preprocessor definition **`RTMIDI_DEBUG`** to the compiler (or define it in **RtMidi.h**). A variety of warning messages will be displayed that may help in determining the problem. Also try using the programs included in the `tests` directory. The program `midiprobe` displays the queried capabilities of all MIDI ports found.

Using Simultaneous Multiple APIs

Support for each MIDI API is encapsulated in specific **MidiInApi** or **MidiOutApi** subclasses, making it possible to compile and instantiate multiple API-specific subclasses on a given operating system. For example, one can compile both CoreMIDI and JACK support on the OS-X operating system by providing the appropriate preprocessor definitions for each. In a run-time situation, one might first attempt to determine whether any JACK ports are available. This can be done by specifying the api argument **RtMidi::UNIX_JACK** when attempting to create an instance of **RtMidiIn** or **RtMidiOut**. If no available ports are found, then an instance of **RtMidi** with the api argument **RtMidi::MACOSX_CORE** can be created. Alternately, if no api argument is specified, **RtMidi** will first look for JACK ports and if none are found, then CoreMIDI ports (in linux, the search order is JACK and then ALSA. In theory, it should also be possible to have separate instances of **RtMidi** open at the same time with different underlying API support, though this has not been tested.

The static function **RtMidi::getCompiledApi()** is provided to determine the available compiled API support. The function RtMidi::getCurrentApi() indicates the API selected for a given **RtMidi** instance.

API Notes

RtMidi is designed to provide a common API across the various supported operating systems and audio libraries. Despite that, some issues should be mentioned with regard to each.

Linux:

RtMidi for Linux was developed using the Fedora distribution. Two different MIDI APIs are supported on Linux platforms: ALSA and JACK. A decision was made to not include support for the OSS API because the OSS API provides very limited functionality and because ALSA support is now incorporated in the Linux kernel. The ALSA sequencer and JACK APIs allows for virtual software input and output ports.

Macintosh OS X (CoreAudio):

The Apple CoreMIDI API allows for the establishment of virtual input and output ports to which other software applications can connect.

The **RtMidi** JACK support can be compiled on Macintosh OS-X systems, as well as in Linux.

Windows (Multimedia Library):

The `configure` script provides support for the MinGW compiler.

The Windows Multimedia library MIDI calls used in **RtMidi** do not make use of streaming functionality. Incoming system exclusive messages read by **RtMidiIn** are limited to a length as defined by the preprocessor definition RT_SYSEX_BUFFER_SIZE (set in RtMidi.cpp). The default value is 1024. There is no such limit for outgoing sysex messages via **RtMidiOut**.

RtMidi was originally developed with Visual C++ version 6.0 but has been tested with Virtual Studio 2010.

Development & Acknowledgements

RtMidi is on github (https://github.com/thestk/rtmidi). Many thanks to the developers that are helping to maintain and improve **RtMidi**.

In years past, the following people provided bug fixes and improvements:

- Stephen Sinclair (Git repo, code and build system)
- amosonn
- Christopher Arndt
- Atsushi Eno (C API)
- Sebastien Alaiwan (JACK memory leaks, Windows kernel streaming)
- Jean-Baptiste Berruchon (Windows sysex code)
- Pedro Lopez-Cabanillas (ALSA sequencer API, client naming)
- Jason Champion (MSW project file for library build)
- Chris Chronopoulos
- JP Cimalando
- Eduardo Coutinho (Windows device names)
- Mattes D
- Michael Dahl
- Paul Dean (increment optimization)
- Francisco Demartino
- Luc Deschenaux (sysex issues)

- John Dey (OS-X timestamps)
- Christoph Eckert (ALSA sysex fixes)
- Thiago Goulart
- Ashley Hedges
- Sam Hocevar
- Rorey Jaffe
- jgvictores
- Martin Koegler (various fixes)
- Immanuel Litzroth (OS-X sysex fix)
- Bartek Lukawski
- Andi McClure
- Jon McCormack (Snow Leopard updates)
- Phildo
- Lane Spangler
- Axel Schmidt (client naming)
- Ryan Schmidt
- Saga Musix
- Bart Spaans
- Alexander Svetalkin (JACK MIDI)
- Ben Swift
- Casey Tucker (OS-X driver information, sysex sending)
- Bastiaan Verreijt (Windows sysex multi-buffer code)
- Dan Wilcox
- Yuri
- Serge Zaitsev
- Iohannes Zmölnig

License

```
however, not a binding provision of this license.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Maintained by Gary P. Scavone, gary at music.mcgill.ca