RtAudio Reference Manual

May 2002

Contents

Chapter 1

The RtAudio (p. 23) Tutorial

1.1 Introduction

RtAudio (p. 23) is a C++ class which provides a common API (Application Programming Interface) for realtime audio input/output across Linux (native ALSA and OSS), SGI, and Windows operating systems. **RtAudio** (p. 23) significantly simplifies the process of interacting with computer audio hardware. It was designed with the following goals:

- object oriented C++ design
- simple, common API across all supported platforms
- single independent header and source file for easy inclusion in programming projects (no libraries!)

- blocking functionality
- callback functionality
- extensive audio device parameter control
- audio device capability probing
- automatic internal conversion for data format, channel number compensation, de-interleaving, and byte-swapping
- control over multiple audio streams and devices with a single instance

RtAudio (p. 23) incorporates the concept of audio streams, which represent audio output (playback) and/or input (recording). Available audio devices and their capabilities can be enumerated and then specified when opening a stream. Multiple streams can run at the same time and, when allowed by the underlying audio API, a single device can serve multiple streams.

The **RtAudio** (p. 23) API provides both blocking (synchronous) and callback (asyncronous) functionality. Callbacks are typically used in conjunction with graphical user interfaces (GUI). Blocking functionality is often necessary for explicit control of multiple input/output stream synchronization or when audio must be synchronized with other system events.

1.2 Getting Started

The first thing that must be done when using **RtAudio** (p. 23) is to create an instance of the class. The default constructor **RtAudio::RtAudio**() (p. 25) scans the underlying audio system to verify that at least one device is available. **RtAudio** (p. 23) often uses C++ exceptions to report errors, necessitating try/catch blocks around most member functions. The following code example demonstrates default object construction and destruction:

```
#include "RtAudio.h"

int main()
{
  RtAudio *audio;

  // Default RtAudio constructor
  try {
    audio = new RtAudio();
  }
  catch (RtError &error) {
    // Handle the exception here
  }

  // Clean up
  delete audio;
}
```

Obviously, this example doesn't demonstrate any of the real functionality of **RtAudio** (p. 23). However, all uses of **RtAudio** (p. 23) must begin with a constructor (either default or overloaded varieties) and must end with class destruction. Further, it is necessary that all class methods which can throw a C++ exception be called within a try/catch block.

1.3 Error Handling

RtAudio (p. 23) uses a C++ exception handler called **RtError** (p. 33), which is declared and defined within the **RtAudio** (p. 23) class files. The **RtError** (p. 33) class is quite simple but it does allow errors to be "caught" by **RtError::TYPE** (p. 33). Almost all **RtAudio** (p. 23) methods can "throw" an **RtError** (p. 33), most typically if an invalid stream identifier is supplied to a method or a driver error occurs. There are a number of cases within **RtAudio** (p. 23) where warning messages may be displayed but an exception is not thrown. There is a private **RtAudio** (p. 23) method, error(), which can be modified to globally control how these messages are handled and reported.

1.4 Probing Device Capabilities

A programmer may wish to query the available audio device capabilities before deciding which to use. The following example outlines how this can be done.

```
// probe.cpp

#include <iostream>
#include "RtAudio.h"

int main()
{
  RtAudio *audio;

  // Default RtAudio constructor
  try {
    audio = new RtAudio();
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  // Determine the number of devices available
  int devices = audio->getDeviceCount();

  // Scan through devices for various capabilities
  RtAudio::RTAUDIO_DEVICE info;
  for (int i=0; i<devices; i++) {
    try {
      audio->getDeviceInfo(i, &info);
    }
    catch (RtError &error) {
      error.printMessage();
      break;
    }

    // Print, for example, the maximum number of output channels for each device
    cout << "device = " << i;
    cout << ": maximum output channels = " << info.maxOutputChannels << "\n";
  }

  // Clean up
  delete audio;

  return 0;
}
```

The RTAUDIO_DEVICE structure is defined in **RtAudio.h** and provides a variety of information useful in assessing the capabilities of a device:

```
  typedef struct {
    char name[128];
    DEVICE_ID id[2];                      // No value reported by getDeviceInfo().
    bool probed;                          // true if the device probe was successful.
    int maxOutputChannels;
    int maxInputChannels;
    int maxDuplexChannels;
    int minOutputChannels;
    int minInputChannels;
    int minDuplexChannels;
    bool hasDuplexSupport;                // true if duplex supported
    int nSampleRates;                     // Number of discrete rates, or -1 if range supported.
    double sampleRates[MAX_SAMPLE_RATES]; // Supported sample rates, or {min, max} if range.
    RTAUDIO_FORMAT nativeFormats;
  } RTAUDIO_DEVICE;
```

The following data formats are defined and fully supported by **RtAudio** (p. 23):

```
  typedef unsigned long RTAUDIO_FORMAT;
  static const RTAUDIO_FORMAT  RTAUDIO_SINT8;   // Signed 8-bit integer
  static const RTAUDIO_FORMAT  RTAUDIO_SINT16;  // Signed 16-bit integer
  static const RTAUDIO_FORMAT  RTAUDIO_SINT24;  // Signed 24-bit integer
  static const RTAUDIO_FORMAT  RTAUDIO_SINT32;  // Signed 32-bit integer
  static const RTAUDIO_FORMAT  RTAUDIO_FLOAT32; // 32-bit float
  static const RTAUDIO_FORMAT  RTAUDIO_FLOAT64; // 64-bit double
```

The *nativeFormats* member of the **RtAudio::RTAUDIO_DEVICE** (p. 31) structure is a bit mask of the above formats which are natively supported by the device. However, **RtAudio** (p. 23) will automatically provide format conversion if a particular format is not natively supported. When the *probed* member of the RTAUDIO_DEVICE structure is false, the remaining structure members are likely unknown and the device is probably unuseable.

In general, the user need not be concerned with the minimum channel values reported in the RTAUDIO_DEVICE structure. While some audio devices may require a minimum channel value > 1, **RtAudio** (p. 23) will provide automatic channel number compensation when the number of channels set by the user is less than that required by the device. Channel compensation is *NOT* possible when the number of channels set by the user is greater than that supported by the device.

It should be noted that the capabilities reported by a device driver or underlying audio API are not always accurate and/or may be dependent on a combination of device settings.

1.5 Device Settings

The next step in using **RtAudio** (p. 23) is to open a stream with a particular set of device settings.

```
#include "RtAudio.h"

int main()
{
  int channels = 2;
  int sample_rate = 44100;
  int buffer_size = 256;  // 256 sample frames
  int n_buffers = 4;      // number of internal buffers used by device
  int device = 0;         // 0 indicates the default or first available device
  int stream;             // our stream identifier
  RtAudio *audio;

  // Instantiate RtAudio and open a stream within a try/catch block
  try {
    audio = new RtAudio();
    stream = audio->openStream(device, channels, 0, 0, RtAudio::RTAUDIO_FLOAT32,
                               sample_rate, &buffer_size, n_buffers);
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  // Clean up
  delete audio;

  return 0;
}
```

The **RtAudio::openStream**() (p. 26) method attempts to open a stream with a specified set of parameter values. When successful, a stream identifier is returned. In this case, we attempt to open a playback stream on device 0 with two channels, 32-bit floating point data, a sample rate of 44100 Hz, a frame rate of 256 sample frames per read/write, and 4 internal device buffers. When device = 0, **RtAudio** (p. 23) first attempts to open the default audio device with the given parameters. If that attempt fails, an attempt is made to find a device or set of devices which will meet the given parameters. If all attempts are unsuccessful, an **RtError** (p. 33) is thrown. When a non-zero device value is specified, an attempt is made to open that device only.

RtAudio (p. 23) provides four signed integer and two floating point data formats which can be specified using the **RtAudio::RTAUDIO_FORMAT** (p. 25) parameter values mentioned earlier. If the opened device does not natively support the given format, **RtAudio** (p. 23) will automatically perform the necessary data format conversion.

The *bufferSize* parameter specifies the desired number of sample frames which will be written to and/or read from a device per write/read operation. The *nBuffers* parameter is used in setting the underlying device buffer parameters. Both the *bufferSize* and *nBuffers* parameters can be used to control stream latency though there is no guarantee that the passed values will be those used by a device. In general, lower values for both parameters will produce less latency but perhaps less robust performance. Both parameters can be specified with values of zero, in which case the smallest allowable values will be used. The *bufferSize* parameter is passed as a pointer and the actual value used by the stream is set during the device setup procedure. *bufferSize* values should be a power of two. Optimal and allowable buffer values tend to vary between systems and devices. Check the **OS Notes** (p. 16) section for general guidelines.

As noted earlier, the device capabilities reported by a driver or underlying audio API are not always accurate and/or may be dependent on a combination of device settings. Because of this, **RtAudio** (p. 23) does not attempt to query a device's capabilities or use previously reported values when opening a device. Instead, **RtAudio** (p. 23) simply attempts to set the given parameters on a specified device and then checks whether the setup is successful or not.

1.6 Playback (blocking functionality)

Once the device is open for playback, there are only a few final steps necessary for realtime audio output. We'll first provide an example (blocking functionality) and then discuss the details.

```
// playback.cpp

#include "RtAudio.h"
```

```
int main()
{
  int count;
  int channels = 2;
  int sample_rate = 44100;
  int buffer_size = 256;  // 256 sample frames
  int n_buffers = 4;      // number of internal buffers used by device
  float *buffer;
  int device = 0;         // 0 indicates the default or first available device
  int stream;             // our stream identifier
  RtAudio *audio;

  // Open a stream during RtAudio instantiation
  try {
    audio = new RtAudio(&stream, device, channels, 0, 0, RtAudio::RTAUDIO_FLOAT32,
                        sample_rate, &buffer_size, n_buffers);
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  try {
    // Get a pointer to the stream buffer
    buffer = (float *) audio->getStreamBuffer(stream);

    // Start the stream
    audio->startStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
    goto cleanup;
  }

  // An example loop which runs for about 40000 sample frames
  count = 0;
  while (count < 40000) {
    // Generate your samples and fill the buffer with buffer_size sample frames of data
    ...

    // Trigger the output of the data buffer
    try {
      audio->tickStream(stream);
    }
    catch (RtError &error) {
      error.printMessage();
      goto cleanup;
    }

    count += buffer_size;
  }

  try {
    // Stop and close the stream
    audio->stopStream(stream);
    audio->closeStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
  }

 cleanup:
  delete audio;

  return 0;
}
```

The first thing to notice in this example is that we attempt to open a stream during class instantiation with an overloaded constructor. This constructor simply combines the functionality of the default constructor, used earlier, and the **RtAudio::openStream**() (p. 26) method. Again, we have specified a device value of 0, indicating that the default or first available device meeting the given parameters should be used. The integer identifier of the opened stream is returned via the *stream* pointer value. An attempt is made to open the stream with the specified *bufferSize* value. However, it is possible that the device will not accept this value, in which case the closest allowable size is used and returned via the pointer value. The constructor can fail if no available devices are found, or a memory allocation or device driver error occurs. Note that you should not call the **RtAudio** (p. 23) destructor if an exception is thrown during instantiation.

Because **RtAudio** (p. 23) can be used to simultaneously control more than a single stream, it is necessary that the stream identifier be provided to nearly all public methods. Assuming the constructor is successful, it is necessary to get a pointer to the buffer, provided by **RtAudio** (p. 23), for use in feeding data to/from the opened stream. Note that the user should *NOT* attempt to deallocate the stream buffer memory ... memory management for the stream buffer will be automatically controlled by **RtAudio** (p. 23). After starting the stream with **RtAudio::startStream**() (p. 28), one simply fills that buffer, which is of length equal to the returned *bufferSize* value, with interleaved audio data (in the specified format) for playback. Finally, a call to the **RtAudio::tickStream**() (p. 28) routine triggers a blocking write call for the stream.

In general, one should call the **RtAudio::stopStream**() (p. 29) and **RtAudio::closeStream**() (p. 28) methods after finishing with a stream. However, both methods will implicitly be called during object destruction if necessary.

1.7 Playback (callback functionality)

The primary difference in using **RtAudio** (p. 23) with callback functionality involves the creation of a user-defined callback function. Here is an example which produces a sawtooth waveform for playback.

```
#include <iostream>
#include "RtAudio.h"

// Two-channel sawtooth wave generator.
int sawtooth(char *buffer, int buffer_size, void *data)
{
  int i, j;
  double *my_buffer = (double *) buffer;
  double *my_data = (double *) data;

  // Write interleaved audio data.
  for (i=0; i<buffer_size; i++) {
    for (j=0; j<2; j++) {
      *my_buffer++ = my_data[j];

      my_data[j] += 0.005 * (j+1+(j*0.1));
      if (my_data[j] >= 1.0) my_data[j] -= 2.0;
    }
  }

  return 0;
}

int main()
{
  int channels = 2;
  int sample_rate = 44100;
  int buffer_size = 256;  // 256 sample frames
  int n_buffers = 4;      // number of internal buffers used by device
  int device = 0;         // 0 indicates the default or first available device
  int stream;             // our stream identifier
  double data[2];
  char input;
  RtAudio *audio;

  // Open a stream during RtAudio instantiation
  try {
    audio = new RtAudio(&stream, device, channels, 0, 0, RtAudio::RTAUDIO_FLOAT64,
                        sample_rate, &buffer_size, n_buffers);
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  try {
    // Set the stream callback function
    audio->setStreamCallback(stream, &sawtooth, (void *)data);

    // Start the stream
    audio->startStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
    goto cleanup;
  }

  cout << "\nPlaying ... press <enter> to quit.\n";
  cin.get(input);

  try {
    // Stop and close the stream
    audio->stopStream(stream);
    audio->closeStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
  }

 cleanup:
  delete audio;

  return 0;
}
```

After opening the device in exactly the same way as the previous example (except with a data format change), we must set our callback function for the stream using **RtAudio::setStreamCallback**() (p. 27). This method will spawn a new process (or thread) which automatically calls the callback function when more data is needed. Note that the callback function is called only when the stream is "running" (between calls to the **RtAudio::startStream**() (p. 28) and **RtAudio::stopStream**() (p. 29) methods). The last argument to **RtAudio::setStreamCallback**() (p. 27) is a pointer to arbitrary data that you wish to access from within your callback function.

In this example, we stop the stream with an explicit call to **RtAudio::stopStream**() (p. 29). When using callback functionality, it is also possible to stop a stream by returning a non-zero value from the callback function.

Once set with **RtAudio::setStreamCallback** (p. 27), the callback process will continue to run for the life of the stream (until the stream is closed with **RtAudio::closeStream**() (p. 28) or the **RtAudio** (p. 23) instance is deleted). It is possible to disassociate a callback function and cancel its process for an open stream using the **RtAudio::cancelStreamCallback**() (p. 27) method. The stream can then be used with blocking functionality or a new callback can be associated with it.

1.8 Recording

Using **RtAudio** (p. 23) for audio input is almost identical to the way it is used for playback. Here's the blocking playback example rewritten for recording:

```
// record.cpp

#include "RtAudio.h"

int main()
{
  int count;
  int channels = 2;
  int sample_rate = 44100;
  int buffer_size = 256;  // 256 sample frames
  int n_buffers = 4;      // number of internal buffers used by device
  float *buffer;
  int device = 0;         // 0 indicates the default or first available device
  int stream;             // our stream identifier
  RtAudio *audio;

  // Instantiate RtAudio and open a stream.
  try {
    audio = new RtAudio(&stream, 0, 0, device, channels,
                        RtAudio::RTAUDIO_FLOAT32, sample_rate, &buffer_size, n_buffers);
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  try {
    // Get a pointer to the stream buffer
    buffer = (float *) audio->getStreamBuffer(stream);

    // Start the stream
    audio->startStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
    goto cleanup;
  }

  // An example loop which runs for about 40000 sample frames
  count = 0;
  while (count < 40000) {

    // Read a buffer of data
    try {
      audio->tickStream(stream);
    }
    catch (RtError &error) {
      error.printMessage();
      goto cleanup;
    }

    // Process the input samples (buffer_size sample frames) that were read
    ...

    count += buffer_size;
  }

  try {
    // Stop the stream
    audio->stopStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
  }

 cleanup:
  delete audio;

  return 0;
}
```

In this example, the stream was opened for recording with a non-zero *inputChannels* value. The only other difference between this example and that for playback involves the order of data processing in the loop, where it is necessary to first read a buffer of input data before manipulating it.

1.9 Duplex Mode

Finally, it is easy to use **RtAudio** (p. 23) for simultaneous audio input/output, or duplex operation. In this example, we use a callback function and pass our recorded data directly through for playback.

```
// duplex.cpp

#include <iostream>
#include "RtAudio.h"

// Pass-through function.
int pass(char *buffer, int buffer_size, void *)
{
  // Surprise!!  We do nothing to pass the data through.
  return 0;
}

int main()
{
  int channels = 2;
  int sample_rate = 44100;
  int buffer_size = 256;  // 256 sample frames
  int n_buffers = 4;      // number of internal buffers used by device
  int device = 0;         // 0 indicates the default or first available device
  int stream;             // our stream identifier
  double data[2];
  char input;
  RtAudio *audio;
  // Open a stream during RtAudio instantiation
  try {
    audio = new RtAudio(&stream, device, channels, device, channels, RtAudio::RTAUDIO_FLOAT64,
                        sample_rate, &buffer_size, n_buffers);
  }
  catch (RtError &error) {
    error.printMessage();
    exit(EXIT_FAILURE);
  }

  try {
    // Set the stream callback function
    audio->setStreamCallback(stream, &pass, NULL);

    // Start the stream
    audio->startStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
    goto cleanup;
  }

  cout << "\nRunning duplex ... press <enter> to quit.\n";
  cin.get(input);

  try {
    // Stop and close the stream
    audio->stopStream(stream);
    audio->closeStream(stream);
  }
  catch (RtError &error) {
    error.printMessage();
  }

 cleanup:
  delete audio;

  return 0;
}
```

When an **RtAudio** (p. 23) stream is running in duplex mode (nonzero input *AND* output channels), the audio write (playback) operation always occurs before the audio read (record) operation. This sequence allows the use of a single buffer to store both output and input data.

As we see with this example, the write-read sequence of operations does not preclude the use of **RtAudio** (p. 23) in situations where input data is first processed and then output through a duplex stream. When the stream buffer is first allocated, it is initialized with zeros, which produces no audible result when output to the device. In this example, anything recorded by the audio stream input will be played out during the next round of audio processing.

Note that duplex operation can also be achieved by opening one output stream and one input stream using the same or different devices. However, there may be timing problems when attempting to use two different devices, due to possible device clock variations. This becomes even more difficult to achieve using two separate callback streams because it is not possible to explicitly control the calling order of the callback functions.

1.10 Summary of Methods

The following is short summary of public methods (not including constructors and the destructor) provided by **RtAudio** (p. 23):

- **RtAudio::openStream**() (p. 26): opens a stream with the specified parameters.
- **RtAudio::setStreamCallback**() (p. 27): sets a user-defined callback function for a given stream.
- **RtAudio::cancelStreamCallback**() (p. 27): cancels a callback process and function for a given stream.
- **RtAudio::getDeviceCount**() (p. 27): returns the number of audio devices available.
- **RtAudio::getDeviceInfo**() (p. 28): fills a user-supplied RTAUDIO_DEVICE structure for a specified device.
- **RtAudio::getStreamBuffer**() (p. 28): returns a pointer to the stream buffer.
- **RtAudio::tickStream**() (p. 28): triggers processing of input/output data for a stream (blocking).
- **RtAudio::closeStream**() (p. 28): closes the specified stream (implicitly called during object destruction). Once a stream is closed, the stream identifier is invalid and should not be used in calling any other **RtAudio** (p. 23) methods.
- **RtAudio::startStream**() (p. 28): (re)starts the specified stream, typically after it has been stopped with either stopStream() or abortStream() or after first opening the stream.
- **RtAudio::stopStream**() (p. 29): stops the specified stream, allowing any remaining samples in the queue to be played out and/or read in. This does not implicitly call **RtAudio::closeStream**() (p. 28).
- **RtAudio::abortStream**() (p. 29): stops the specified stream, discarding any remaining samples in the queue. This does not implicitly call closeStream().
- **RtAudio::streamWillBlock**() (p. 29): queries a stream to determine whether a call to the *tickStream()* method will block. A return value of 0 indicates that the stream will NOT block. A positive return value indicates the number of sample frames that cannot yet be processed without blocking.

1.11 Compiling

In order to compile **RtAudio** (p. 23) for a specific OS and audio API, it is necessary to supply the appropriate preprocessor definition and library within the compiler statement:

OS:	**Audio API:**	**Preprocessor Definition:**	**Library:**	**Example Compiler Statement:**
Linux	ALSA	__LINUX_ALSA__	`libasound, libpthread`	`g++ -Wall -D__LINUX_ALSA__ -o probe probe.cpp RtAudio.cpp -lasound -lpthread`
Linux	OSS	__LINUX_OSS__	`libpthread`	`g++ -Wall -D__LINUX_OSS__ -o probe probe.cpp RtAudio.cpp -lpthread`
Irix	AL	__IRIX_AL__	`libaudio, libpthread`	`CC -Wall -D__IRIX_AL__ -o probe probe.cpp RtAudio.cpp -laudio -lpthread`
Windows	Direct Sound	__WINDOWS_DS__	`dsound.lib (ver. 5.0 or higher), multithreaded`	*compiler specific*

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

1.12 OS Notes

RtAudio (p. 23) is designed to provide a common API across the various supported operating systems and audio libraries. Despite that, however, some issues need to be mentioned with regard to each.

1.12.1 Linux:

RtAudio (p. 23) for Linux was developed under Redhat distributions 7.0 - 7.2. Two different audio APIs are supported on Linux platforms: OSS and `ALSA`. The OSS API has existed for at least 6 years and the Linux kernel is distributed with free versions of OSS audio drivers. Therefore, a generic Linux system is most likely to have OSS support. The ALSA API, although relatively new, is now part of the Linux development kernel and offers significantly better functionality than the OSS API. **RtAudio** (p. 23) provides support for the 0.9 and higher versions of ALSA. Input/output latency on the order of 15 milliseconds can typically be achieved under both OSS or ALSA by fine-tuning the **RtAudio** (p. 23) buffer parameters (without kernel modifications). Latencies on the order of 5 milliseconds or less can be achieved using a low-latency kernel patch and increasing FIFO scheduling priority. The pthread library, which is used for callback functionality, is a standard component of all Linux distributions.

The ALSA library includes OSS emulation support. That means that you can run programs compiled for the OSS API even when using the ALSA drivers and library. It should be noted however that OSS emulation under ALSA is not perfect. Specifically, channel number queries seem to consistently produce invalid results. While OSS emulation is successful for the majority of **RtAudio** (p. 23) tests, it is recommended that the native ALSA implementation of **RtAudio** (p. 23) be used on systems which have ALSA drivers installed.

The ALSA implementation of **RtAudio** (p. 23) makes no use of the ALSA "plug" interface. All necessary data format conversions, channel compensation, deinterleaving, and byte-swapping is handled by internal **RtAudio** (p. 23) routines.

1.12.2 Irix (SGI):

The Irix version of **RtAudio** (p. 23) was written and tested on an SGI Indy running Irix version 6.5.4 and the newer "al" audio library. **RtAudio** (p. 23) does not compile under Irix version 6.3, mainly because the C++ compiler is too old. Despite the relatively slow speed of the Indy, **RtAudio** (p. 23) was found to behave quite well and input/output latency was very good. No problems were found with respect to using the pthread library.

1.12.3 Windows:

RtAudio (p. 23) under Windows is written using the DirectSound API. In order to compile **RtAudio** (p. 23) under Windows, you must have the header and source files for DirectSound version 5.0 or higher. As far as I know, there is no DirectSoundCapture support for Windows NT (in which case, you cannot use **RtAudio** (p. 23)). Audio output latency with DirectSound can be reasonably good (on the order of 20 milliseconds). On the other hand, input audio latency tends to be terrible (100 milliseconds or more). Further, DirectSound drivers tend to crash easily when experimenting with buffer parameters. On my system, I found it necessary to use values around nBuffers = 8 and bufferSize = 512 to avoid crashing my system. **RtAudio** (p. 23) was developed with Visual C++ version 6.0. I was forced in several instances to modify code in order to get it to compile under the non-standard version of C++ that Microsoft so unprofessionally implemented. We can only hope that the developers of Visual C++ 7.0 will have time to read the C++ standard.

1.13 Acknowledgements

The **RtAudio** (p. 23) API incorporates many of the concepts developed in the `PortAudio` project by Phil Burk and Ross Bencina. Early development also incorporated ideas from Bill Schottstaedt's `sndlib`. The CCRMA `SoundWire group` provided valuable feedback during the API proposal stages.

RtAudio (p. 23) was slowly developed over the course of many months while in residence at the `Institut Universitari de L'Audiovisual (IUA)` in Barcelona, Spain, the `Laboratory of Acoustics and Audio Signal Processing` at the Helsinki University of Technology, Finland, and the `Center for Computer Research in Music and Acoustics (CCRMA)` at `Stanford University`. This work was supported in part by the United States Air Force Office of Scientific Research (grant #F49620-99-1-0293).

Chapter 2

RtAudio Compound Index

2.1 RtAudio Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

RtAudio File Index

3.1 RtAudio File List

Here is a list of all files with brief descriptions:

Chapter 4

RtAudio Class Documentation

4.1 RtAudio Class Reference

`#include <RtAudio.h>`

Public Types

- typedef unsigned long **RTAUDIO_FORMAT**
- typedef int (∗ **RTAUDIO_CALLBACK**)(char ∗buffer, int bufferSize, void ∗userData)
- enum { **MAX_SAMPLE_RATES** = 14 }

Public Methods

- **RtAudio** ()

  The default constructor.

- **RtAudio** (int ∗streamId, int outputDevice, int outputChannels, int inputDevice, int inputChannels, **RTAUDIO_FORMAT** format, int sampleRate, int ∗bufferSize, int numberOfBuffers)

  A constructor which can be used to open a stream during instantiation.

- ∼**RtAudio** ()

  The destructor.

- int **openStream** (int outputDevice, int outputChannels, int inputDevice, int inputChannels, **RTAUDIO_FORMAT** format, int sampleRate, int ∗bufferSize, int numberOfBuffers)

  A public method for opening a stream with the specified parameters.

- void **setStreamCallback** (int streamId, **RTAUDIO_CALLBACK** callback, void ∗userData)

  A public method which sets a user-defined callback function for a given stream.

- void **cancelStreamCallback** (int streamId)

  A public method which cancels a callback process and function for a given stream.

- int **getDeviceCount** (void)

  A public method which returns the number of audio devices found.

- void **getDeviceInfo** (int device, **RTAUDIO_DEVICE** ∗info)

  Fill a user-supplied **RTAUDIO_DEVICE** (p. 31) *structure for a specified device.*

- char∗ const **getStreamBuffer** (int streamId)

  A public method which returns a pointer to the buffer for an open stream.

- void **tickStream** (int streamId)

  Public method used to trigger processing of input/output data for a stream.

- void **closeStream** (int streamId)

  Public method which closes a stream and frees any associated buffers.

- void **startStream** (int streamId)

  Public method which starts a stream.

- void **stopStream** (int streamId)

  Stop a stream, allowing any samples remaining in the queue to be played out and/or read in.

- void **abortStream** (int streamId)

  Stop a stream, discarding any samples remaining in the input/output queue.

- int **streamWillBlock** (int streamId)

  Queries a stream to determine whether a call to the **tickStream**() (p. 28) *method will block.*

Static Public Attributes

- const **RTAUDIO_FORMAT RTAUDIO_SINT8**
- const **RTAUDIO_FORMAT RTAUDIO_SINT16**
- const **RTAUDIO_FORMAT RTAUDIO_SINT24**
- const **RTAUDIO_FORMAT RTAUDIO_SINT32**
- const **RTAUDIO_FORMAT RTAUDIO_FLOAT32**
- const **RTAUDIO_FORMAT RTAUDIO_FLOAT64**

4.1.1 Member Typedef Documentation

4.1.1.1 typedef unsigned long RtAudio::RTAUDIO_FORMAT

4.1.1.2 typedef int(∗ RtAudio::RTAUDIO_CALLBACK)(char ∗buffer, int bufferSize, void ∗userData)

4.1.2 Member Enumeration Documentation

4.1.2.1 anonymous enum

Enumeration values:
MAX_SAMPLE_RATES

4.1.3 Constructor & Destructor Documentation

4.1.3.1 RtAudio::RtAudio ()

The default constructor.

Probes the system to make sure at least one audio input/output device is available and determines the api-specific identifier for each device found. An **RtError** (p. 33) error can be thrown if no devices are found or if a memory allocation error occurs.

4.1.3.2 RtAudio::RtAudio (int ∗ *streamId*, int *outputDevice*, int *outputChannels*, int *inputDevice*, int *inputChannels*, RTAUDIO_FORMAT *format*, int *sampleRate*, int ∗ *bufferSize*, int *numberOfBuffers*)

A constructor which can be used to open a stream during instantiation.

The specified output and/or input device identifiers correspond to those enumerated via the **getDeviceInfo**() (p. 28) method. If device = 0, the default or first available devices meeting the given parameters is selected. If an output or input channel value is zero, the corresponding device value is ignored. When a stream is successfully opened, its identifier is returned via the "streamId" pointer. An **RtError** (p. 33) can be thrown if no devices are found for the given parameters, if a memory allocation error occurs, or if a driver error occurs.

See also:
openStream() (p. 26)

4.1.3.3 RtAudio::~RtAudio ()

The destructor.

Stops and closes any open streams and devices and deallocates buffer and structure memory.

4.1.4 Member Function Documentation

4.1.4.1 int RtAudio::openStream (int *outputDevice*, int *outputChannels*, int *inputDevice*, int *inputChannels*, RTAUDIO_FORMAT *format*, int *sampleRate*, int * *bufferSize*, int *numberOfBuffers*)

A public method for opening a stream with the specified parameters.

If successful, the opened stream ID is returned. Otherwise, an **RtError** (p. 33) is thrown.

Parameters:

outputDevice: If equal to 0, the default or first device found meeting the given parameters is opened. Otherwise, the device number should correspond to one of those enumerated via the **getDeviceInfo**() (p. 28) method.

outputChannels: The desired number of output channels. If equal to zero, the outputDevice identifier is ignored.

inputDevice: If equal to 0, the default or first device found meeting the given parameters is opened. Otherwise, the device number should correspond to one of those enumerated via the **getDeviceInfo**() (p. 28) method.

inputChannels: The desired number of input channels. If equal to zero, the inputDevice identifier is ignored.

format: An RTAUDIO_FORMAT specifying the desired sample data format.

sampleRate: The desired sample rate (sample frames per second).

bufferSize: A pointer value indicating the desired internal buffer size in sample frames. The actual value used by the device is returned via the same pointer. A value of zero can be specified, in which case the lowest allowable value is determined.

numberOfBuffers: A value which can be used to help control device latency. More buffers typically result in more robust performance, though at a cost of greater latency. A value of zero can be specified, in which case the lowest allowable value is used.

4.1.4.2 void RtAudio::setStreamCallback (int *streamId*, RTAUDIO_CALLBACK *callback*, void ∗ *userData*)

A public method which sets a user-defined callback function for a given stream.

This method assigns a callback function to a specific, previously opened stream for non-blocking stream functionality. A separate process is initiated, though the user function is called only when the stream is "running" (between calls to the **startStream**() (p. 28) and **stopStream**() (p. 29) methods, respectively). The callback process remains active for the duration of the stream and is automatically shutdown when the stream is closed (via the **closeStream**() (p. 28) method or by object destruction). The callback process can also be shutdown and the user function de-referenced through an explicit call to the **cancelStreamCallback**() (p. 27) method. Note that a single stream can use only blocking or callback functionality at the same time, though it is possible to alternate modes on the same stream through the use of the **setStreamCallback**() (p. 27) and **cancelStreamCallback**() (p. 27) methods (the blocking **tickStream**() (p. 28) method can be used before a callback is set and/or after a callback is cancelled). An **RtError** (p. 33) will be thrown for an invalid device argument.

4.1.4.3 void RtAudio::cancelStreamCallback (int *streamId*)

A public method which cancels a callback process and function for a given stream.

This method shuts down a callback process and de-references the user function for a specific stream. Callback functionality can subsequently be restarted on the stream via the **setStreamCallback**() (p. 27) method. An **RtError** (p. 33) will be thrown for an invalid device argument.

4.1.4.4 int RtAudio::getDeviceCount (void)

A public method which returns the number of audio devices found.

4.1.4.5 void RtAudio::getDeviceInfo (int *device*, RTAUDIO_DEVICE ∗ *info*)

Fill a user-supplied **RTAUDIO_DEVICE** (p. 31) structure for a specified device.

Any device between 0 and **getDeviceCount**() (p. 27)-1 is valid. If a device is busy or otherwise unavailable, the structure member "probed" has a value of "false". The system default input and output devices are referenced by device identifier = 0. On systems which allow dynamic default device settings, the default devices are not identified by name (specific device enumerations are assigned device identifiers > 0). An **RtError** (p. 33) will be thrown for an invalid device argument.

4.1.4.6 char ∗const RtAudio::getStreamBuffer (int *streamId*)

A public method which returns a pointer to the buffer for an open stream.

The user should fill and/or read the buffer data in interleaved format and then call the **tickStream**() (p. 28) method. An **RtError** (p. 33) will be thrown for an invalid stream identifier.

4.1.4.7 void RtAudio::tickStream (int *streamId*)

Public method used to trigger processing of input/output data for a stream.

This method blocks until all buffer data is read/written. An **RtError** (p. 33) will be thrown for an invalid stream identifier or if a driver error occurs.

4.1.4.8 void RtAudio::closeStream (int *streamId*)

Public method which closes a stream and frees any associated buffers.

If an invalid stream identifier is specified, this method issues a warning and returns (an **RtError** (p. 33) is not thrown).

4.1.4.9 void RtAudio::startStream (int *streamId*)

Public method which starts a stream.

An **RtError** (p. 33) will be thrown for an invalid stream identifier or if a driver error occurs.

4.1.4.10 void RtAudio::stopStream (int *streamId*)

Stop a stream, allowing any samples remaining in the queue to be played out and/or read in.

An **RtError** (p. 33) will be thrown for an invalid stream identifier or if a driver error occurs.

4.1.4.11 void RtAudio::abortStream (int *streamId*)

Stop a stream, discarding any samples remaining in the input/output queue.

An **RtError** (p. 33) will be thrown for an invalid stream identifier or if a driver error occurs.

4.1.4.12 int RtAudio::streamWillBlock (int *streamId*)

Queries a stream to determine whether a call to the **tickStream**() (p. 28) method will block.

A return value of 0 indicates that the stream will NOT block. A positive return value indicates the number of sample frames that cannot yet be processed without blocking.

4.1.5 Member Data Documentation

4.1.5.1 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT8 `[static]`

4.1.5.2 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT16 `[static]`

4.1.5.3 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT24 `[static]`

Upper 3 bytes of 32-bit integer.

4.1.5.4 const RTAUDIO_FORMAT RtAudio::RTAUDIO_SINT32 `[static]`

4.1.5.5 const RTAUDIO_FORMAT RtAudio::RTAUDIO_FLOAT32 `[static]`

Normalized between plus/minus 1.0.

4.1.5.6 const RTAUDIO_FORMAT RtAudio::RTAUDIO_FLOAT64 `[static]`

Normalized between plus/minus 1.0.

The documentation for this class was generated from the following file:

- **RtAudio.h**

4.2 RtAudio::RTAUDIO_DEVICE Struct Reference

`#include <RtAudio.h>`

Public Attributes

- char **name** [128]
- DEVICE_ID **id** [2]
- bool **probed**
- int **maxOutputChannels**
- int **maxInputChannels**
- int **maxDuplexChannels**
- int **minOutputChannels**
- int **minInputChannels**
- int **minDuplexChannels**
- bool **hasDuplexSupport**
- int **nSampleRates**
- int **sampleRates** [MAX_SAMPLE_RATES]
- **RTAUDIO_FORMAT nativeFormats**

4.2.1 Member Data Documentation

4.2.1.1 char RtAudio::RTAUDIO_DEVICE::name

4.2.1.2 DEVICE_ID RtAudio::RTAUDIO_DEVICE::id

No value reported by **getDeviceInfo**() (p. 28).

4.2.1.3 bool RtAudio::RTAUDIO_DEVICE::probed

true if the device capabilities were successfully probed.

4.2.1.4 int RtAudio::RTAUDIO_DEVICE::maxOutputChannels

4.2.1.5 int RtAudio::RTAUDIO_DEVICE::maxInputChannels

4.2.1.6 int RtAudio::RTAUDIO_DEVICE::maxDuplexChannels

4.2.1.7 int RtAudio::RTAUDIO_DEVICE::minOutputChannels

4.2.1.8 int RtAudio::RTAUDIO_DEVICE::minInputChannels

4.2.1.9 int RtAudio::RTAUDIO_DEVICE::minDuplexChannels

4.2.1.10 bool RtAudio::RTAUDIO_DEVICE::hasDuplexSupport

true if device supports duplex mode.

4.2.1.11 int RtAudio::RTAUDIO_DEVICE::nSampleRates

Number of discrete rates or -1 if range supported.

4.2.1.12 int RtAudio::RTAUDIO_DEVICE::sampleRates

Supported rates or (min, max) if range.

4.2.1.13 RTAUDIO_FORMAT RtAudio::RTAUDIO_-DEVICE::nativeFormats

Bit mask of supported data formats.

The documentation for this struct was generated from the following file:

- **RtAudio.h**

4.3 RtError Class Reference

`#include <RtAudio.h>`

Public Types

- enum **TYPE** { **WARNING**, **DEBUG_WARNING**, **UNSPECIFIED**, **NO_DEVICES_FOUND**, **INVALID_DEVICE**, **INVALID_STREAM**, **MEMORY_ERROR**, **INVALID_PARAMETER**, **DRIVER_ERROR**, **SYSTEM_ERROR**, **THREAD_ERROR** }

Public Methods

- **RtError** (const char ∗p, **TYPE** tipe=RtError::UNSPECIFIED)

  The constructor.

- virtual ∼**RtError** (void)

  The destructor.

- virtual void **printMessage** (void)

  Prints "thrown" error message to stdout.

- virtual const **TYPE**& **getType** (void)

  Returns the "thrown" error message TYPE.

- virtual const char∗ **getMessage** (void)

  Returns the "thrown" error message string.

Protected Attributes

- char **error_message** [256]
- **TYPE type**

4.3.1 Member Enumeration Documentation

4.3.1.1 enum RtError::TYPE

Enumeration values:

WARNING

DEBUG_WARNING

UNSPECIFIED

NO_DEVICES_FOUND

INVALID_DEVICE

INVALID_STREAM

MEMORY_ERROR

INVALID_PARAMETER

DRIVER_ERROR

SYSTEM_ERROR

THREAD_ERROR

4.3.2 Constructor & Destructor Documentation

4.3.2.1 RtError::RtError (const char * *p*, TYPE *tipe* = RtError::UNSPECIFIED)

The constructor.

4.3.2.2 RtError::∼RtError (void) `[virtual]`

The destructor.

4.3.3 Member Function Documentation

4.3.3.1 void RtError::printMessage (void) `[virtual]`

Prints ”thrown” error message to stdout.

4.3.3.2 const TYPE & RtError::getType (void) `[inline, virtual]`

Returns the ”thrown” error message TYPE.

4.3.3.3 const char * RtError::getMessage (void) `[inline, virtual]`

Returns the ”thrown” error message string.

4.3.4 Member Data Documentation

4.3.4.1 char RtError::error_message `[protected]`

4.3.4.2 TYPE RtError::type `[protected]`

The documentation for this class was generated from the following file:

- **RtAudio.h**

Chapter 5

RtAudio File Documentation

5.1 RtAudio.h File Reference

`#include <map>`

Compounds

- class **RtError**
- class **RtAudio**
- struct **RtAudio::RTAUDIO_DEVICE**
- struct **RtAudio::RTAUDIO_STREAM**

Defines

- #define **__RTAUDIO_H**

5.1.1 Define Documentation

5.1.1.1 #define __RTAUDIO_H

Value:

5.2 tutorial.txt File Reference

Index