RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output

Gary P. Scavone
gary@ccrma.stanford.edu

Center for Computer Research in Music and Acoustics
Department of Music, Stanford University
Stanford, California 94305-8180 USA

Abstract

This paper presents a cross-platform C++ class for realtime audio input and output streaming. RtAudio provides a flexible, easy to use application programming interface (API) which allows complete audio system control, including device capability querying, multiple concurrent streams, blocking and callback functionality. RtAudio is currently supported on Windows platforms using the DirectSound API, Linux platforms using both the OSS and ALSA APIs, and on Irix platforms. Support for OS-X and Steinberg ASIO drivers is planned for Spring 2002.

1 Introduction

While programming languages have gained standardized support across the myriad of computer platforms and operating systems in existence, a commonly supported API for audio programming is far from a reality. As a result, an attempt to provide multi-platform support for an audio application can prove difficult at best. To further complicate matters, multiple audio driver interfaces often exist for a single operating system. For example, Windows platforms have DirectSound, Windows Multimedia Library, and ASIO (Steinberg) driver options, Linux platforms have Open Sound System (OSS) and Advanced Linux Sound Architecture (ALSA) drivers, and Macintosh platforms have Sound Manager, ASIO and Core Audio drivers. *RtAudio* was designed to provide a common interface across a variety of these APIs in as flexible, yet simple, manner as possible.

RtAudio was originally developed to provide audio input/output support for the Synthesis ToolKit in C++ (STK) (Cook and Scavone, 1999). However, the latest release of *RtAudio* (version 2.0, January 2002) was designed to function independently from STK, as well as any libraries other than those necessitated by the underlying platform-specific audio interfaces.

2 Features & Design Goals

RtAudio is a C++ class which provides a common API for realtime audio input/output across Linux, Irix, and Windows operating systems. *RtAudio* significantly simplifies the process of interfacing with computer audio hardware. It was designed with the following goals:

- object-oriented C++ structure
- single independent header and source file for easy inclusion in programming projects
- blocking and callback functionality
- flexible, easy to use, audio device parameter control
- automatic internal conversion for data format, channel number compensation, deinterleaving, and byte-swapping
- control over multiple audio streams and devices with a single class instance
- audio device capability probing

RtAudio incorporates the concept of audio streams, which represent independent audio output (playback) and/or input (recording) "connections" to audio devices. 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* API provides both blocking (synchronous) and callback (asynchronous) functionality. Callbacks offer a simple means for achieving non-blocking audio input/output. Blocking functionality is often necessary for explicit control of multiple input/output stream synchronization or when audio must be synchronized with other system events. All public *RtAudio* functions are thread-safe. This allows users to safely

embed blocking *RtAudio* functions within a multi-threaded programming structure of their own design.

RtAudio offers uniform support for 8-bit, 16-bit, 24-bit, and 32-bit signed integer data formats, as well as 32-bit and 64-bit floating point formats. When an audio device does not natively support a requested user format, *RtAudio* provides automatic format conversion. In addition, internal routines will automatically perform any byte-swapping, channel number compensation, and channel de-interleaving required by the underlying audio driver or hardware.

On Linux platforms, both native ALSA and OSS audio APIs are supported. Portability to other OSS supported systems, such as Solaris and HP-UX, is untested but most likely easily achieved. The ALSA driver model was recently incorporated into the Linux development kernel and will likely gain wide acceptance in the near future. The ALSA API provides a more developed level of support for professional quality audio devices than OSS. On Windows platforms, only the DirectSound API is currently supported. On SGI platforms, the newer "al" API is supported.

The *RtAudio* API incorporates many of the concepts developed in the PortAudio project (Bencina and Burk, 2001). *RtAudio* distinguishes itself from PortAudio in its object-oriented, C++ framework, single-file encapsulation, native blocking support, ALSA support, thread-safe routines, and slightly less ambitious API (which makes *RtAudio* less prone to bugs and easier to maintain and extend).

All source code for *RtAudio* is made freely available, allowing full user extensibility and customization. *RtAudio* is distributed with a tutorial and complete API documentation in HTML, PDF, and RTF formats.

3 The *RtAudio* API

All uses of *RtAudio* must begin with object instantiation. The default constructor `RtAudio()` scans the underlying audio system to verify that at least one audio input/output device is available. *RtAudio* uses C++ exceptions to handle critical errors, necessitating try/catch blocks around most member functions as well as constructors. Likewise, all uses of *RtAudio* must end with class destruction.

RtAudio uses a C++ exception handler called RtError, which is declared and defined within the *RtAudio* class files. An RtError can be caught by type, providing a means for error correction or at a minimum, more detailed error reporting. Almost all *RtAudio* methods can "throw" an RtError, 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* where warning messages may be displayed but an exception is not thrown.

3.1 Device Capabilities

RtAudio provides the following functions for use in probing the number and capabilities of available audio devices:

```
int getDeviceCount (void);

void getDeviceInfo (int device,
                    RTAUDIO_DEVICE *info);
```

The RTAUDIO_DEVICE structure contains information commonly required in assessing the capabilities of an audio device, including its name, minimum and maximum number of input, output, and duplex channels, supported sample rates, and native data formats.

3.2 Stream Creation & Parameters

In addition to the default constructor, *RtAudio* provides an overloaded constructor which allows a stream to be immediately opened with a given set of device parameters. Alternately, a stream can be opened after instantiation in much the same way.

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

int openStream (int outputDevice,
                int outputChannels,
                int inputDevice,
                int inputChannels,
                RTAUDIO_FORMAT format,
                int sampleRate,
                int *bufferSize,
                int numberOfBuffers);
```

A stream is opened with specified output and input devices, output and input channels, data format, sample rate, and buffer parameters. When successful, a stream identifier is returned which must be used for subsequent function calls on the stream. Audio devices are identified by integer values of one and higher, as enumerated by the `getDeviceInfo()` function. In addition, the system default input/output devices are identified by a zero value. When a device identifier of zero is

specified during stream creation, *RtAudio* first attempts to open the default audio device(s) with the given parameters. If that 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 is thrown. When a positive, non-zero device value is specified, no *additional* devices are probed. Example program code is provided in the appendix of this paper.

Because *RtAudio* can be used to simultaneously control more than a single stream, it is necessary that the returned stream identifier be provided to nearly all public methods.

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. Both the *bufferSize* and *numberOfBuffers* parameters can be used to control stream latency, though there is no guarantee that the passed values will be accepted 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.

3.3 Stream Control

An opened stream will not begin to input/output data until it is "started" using the `startStream()` function. Several other useful functions are listed below as well. A stream can be stopped and "restarted" as many times as necessary. Once the stream is closed, however, it ceases to exist. The `abortStream()` function stops a stream immediately, dropping any remaining audio samples in its queue. The `stopStream()` function plays out any remaining data in its queue before stopping.

```
void startStream (int streamId);
void stopStream (int streamId);
void abortStream (int streamId);
void closeStream (int streamId);
```

In general, the `stopStream()` and `closeStream()` methods should be called after finishing with a stream. However, both methods will implicitly be called during object destruction if necessary.

The remaining steps involved in audio playback or recording vary depending on whether blocking or callback functionality is used.

3.4 Blocking Input/Output

Blocking read/write functionality provides synchronous control of audio processing. In this mode, the user must first get a pointer to the stream buffer, provided by *RtAudio*, for use in feeding data to/from the opened stream. Memory management for the stream buffer is automatically controlled by *RtAudio*. The *bufferSize* value returned during stream creation defines the length, in sample frames, of the stream buffer. Multichannel data in the stream buffer must be in interleaved order.

```
char *const getStreamBuffer (int streamId);
void tickStream (int streamId);
int streamWillBlock (int streamId);
```

After starting the stream, the sequence of events then consists of filling or reading from the stream buffer between calls to `tickStream()`. The `tickStream()` function blocks until the data within the stream buffer can be completely processed by the audio device. The `streamWillBlock()` function is provided as a means for determining, *a priori*, whether the `tickStream()` function will block, returning the number of sample frames that cannot be processed without blocking.

3.5 Callback Functionality

Callback functionality provides non-blocking, asynchronous control of audio processing. In this mode, the user defines a global C function which is periodically called when the audio device is ready to receive/send a new buffer of audio data. The callback function fills or reads interleaved data from the stream buffer, interfacing with the user program in an application-dependent manner. Internally, callback functionality involves the creation of a separate process or thread which provides non-blocking access to an audio device.

```
void setStreamCallback (int streamId,
                        RTAUDIO_CALLBACK callback,
                        void *userData);

void cancelStreamCallback (int streamId);
```

The `cancelStreamCallback()` function disassociates a callback function from an open stream. The user can subsequently set a new callback function for the stream or even use blocking functions.

It should be noted that it is not possible to explicitly synchronize multiple simultaneous callback streams. When synchronous control is required in a non-blocking scheme, users should create their own thread in which they embed *RtAudio* blocking functions.

4 Summary

RtAudio provides a flexible and easy to use cross-platform API for realtime audio in-

put/output within an object-oriented C++ framework. This paper has briefly presented some features and uses of *RtAudio*. Within the confines of this space, it is impossible to address all the necessary issues of interest to audio application programmers. We recommend that interested parties download the *RtAudio* distribution (`http://www-ccrma.stanford.edu/~gary/-rtaudio/`) and read the extensive documentation provided.

References

Bencina, R. and Burk, P. (2001). PortAudio - an Open Source Cross Platform Audio API. In *Proc. 2001 Int. Computer Music Conf.*, pp. 263–266, Havana, Cuba. Computer Music Association.

Cook, P. R. and Scavone, G. P. (1999). The Synthesis ToolKit (STK). In *Proc. 1999 Int. Computer Music Conf.*, pp. 164–166, Beijing, China. Computer Music Association.

A Programming Examples

The following program example outlines the use of *RtAudio* in a simple, blocking playback situation. For the sake of clarity and space, error checking is omitted.

```
// playback.cpp
#include "RtAudio.h"

int main()
{
  int buffer_size = 256;  // sample frames
  int id;                 // the stream id
  RtAudio *out;

  // Open a 2 channel output stream during
  // class instantiation using the default
  // device, 32-bit floating point data,
  // and 44100 Hz sample rate. Suggest the
  // use of 4 internal device buffers of
  // 256 sample frames each.
  out = new RtAudio(&id, 0, 2, 0, 0,
                RtAudio::RTAUDIO_FLOAT32,
                44100, &buffer_size, 4);

  // Get a pointer to the stream buffer
  float *buf;
  buf = (float *)out->getStreamBuffer(id);

  // An example loop which runs for about
  // 40000 sample frames
  int count = 0;
  out->startStream(id);
  while (count < 40000) {
    // Generate samples and fill the buffer
    // with buffer_size sample frames.
    ...

    // Trigger the output of the data buffer
    out->tickStream(id);
    count += buffer_size;
  }
  out->stopStream(id);
  out->closeStream(id);
  delete out;  // Cleanup.
  return 0;
}
```

The last program example demonstrates callback functionality in a simple duplex, pass-through scenario. Again, error checking is omitted.

```
// duplex.cpp
#include <iostream.h>
#include "RtAudio.h"

// Pass-through callback function.
int pass(char *buffer, int size, void *)
{
  // Surprise!!  Nothing to do here.
  return 0;
}

int main()
{
  int buffer_size = 256;  // sample frames
  int stream;             // the stream id
  RtAudio *audio;

  // Open a 2 channel input/output stream
  // during class instantiation using the
  // default devices, 64-bit floating point
  // data, and 44100 Hz sample rate.
  // Suggest the use of 2 internal device
  // buffers of 256 sample frames each.
  audio = new RtAudio(&stream, 0, 2, 0, 2,
                RtAudio::RTAUDIO_FLOAT64,
                44100, &buffer_size, 2);

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

  audio->startStream(stream);
  cout << "Hit <enter> to quit." << endl;
  char input;
  cin.get(input);

  audio->stopStream(stream);
  audio->closeStream(stream);
  delete audio;  // Cleanup
  return 0;
}
```