RTMIDI, RTAUDIO, AND A SYNTHESIS TOOLKIT (STK) UPDATE

Gary P. Scavone
`gary@music.mcgill.ca`
Music Technology, Faculty of Music
McGill University

Perry R. Cook
`prc@cs.princeton.edu`
Departments of Computer Science & Music
Princeton University

ABSTRACT

This paper presents new and ongoing development efforts directed toward open-source, cross-platform C++ "tools" for music and audio programming. `RtMidi` provides a common application programming interface (API) for realtime MIDI input and output on Linux, Windows, Macintosh, and SGI computer systems. `RtAudio` provides complementary functionality for realtime audio input and output streaming. The Synthesis ToolKit in C++ (STK) is a set of audio signal processing and algorithmic synthesis classes designed to facilitate rapid development of music synthesis and audio processing software.

1. INTRODUCTION

The development of cross-platform computer applications for music and audio has long been limited by the lack of universal programming interfaces to communicate with audio and MIDI hardware. The multitude of APIs in existence offer vastly different levels of control and often present fundamentally different programming paradigms. In several instances, multiple APIs exist for a single operating system (OS). `RtAudio` is a set of C++ classes that provides a uniform, flexible, and easy to use interface for realtime audio input/output across the most common audio APIs [8]. `RtMidi`, a set of companion classes for realtime MIDI input/output, was more recently released in September 2004.

The Synthesis ToolKit in C++ (STK) has been available and in use for nearly ten years [7]. STK provides a flexible set of objects that can be used for rapid prototyping of music and audio algorithms, embedded in a computer program to provide audio signal processing functionality, or used to teach practical aspects of computer music programming, for example. In this paper, we discuss recent updates that have been made in a continuing effort to serve the STK user community of over 200 developers and thousands of users worldwide.

2. RTAUDIO

The `RtAudio` C++ class was originally developed to provide realtime audio input/output support for the Synthesis ToolKit. A completely redesigned, standalone version of the class was released by the first author in 2002 [8]. This version included support for Linux (OSS and ALSA), Windows (DirectSound), and Irix platforms. Since then, support has expanded to also include the CoreAudio (Macintosh OS X), ASIO (Windows), and Jack (Linux) APIs.

2.1. The `RtAudio` API

Given space limitations, it is not possible to provide a complete programming example of `RtAudio` here. However, an extensive tutorial and class documentation are available from the `RtAudio` website[1]. The following pseudo-code outlines a typical `RtAudio` "session".

```
// Write a callback function to process
// audio data to/from buffer.
int callback(char *buffer, int frames,
             void *userData) {}

// Open an RtAudio stream
RtAudio audio(oDevice, oChannels,
              iDevice, iChannels,
              dataFormat, sampleRate,
              &bufferFrames, nBuffers);

// Set callback and start the stream
audio.setStreamCallback(&callback, NULL);
audio.startStream();

// Running ... until done

// Stop and close the stream
audio.stopStream();
audio.closeStream();
```

2.2. Simultaneous Multi-API Support

Since release 3.0, `RtAudio` has provided support for compilation of multiple simultaneous APIs within a given operating system. For example, it is now possible to compile simultaneous support for the OSS, ALSA, and Jack APIs in Linux or both the DirectSound and ASIO APIs in Windows. This was accomplished by subclassing each API and redesigning `RtAudio` to choose an appropriate subclass for instantiation. The user can specify a particular API during object construction or the `RtAudio` controller will automatically search for an available API support in a prioritized order (for example, Jack, ALSA, and

[1] `http://music.mcgill.ca/~gary/rtaudio/`

OSS in Linux). It should be noted that this change required no modification to the `RtAudio` API aside from the addition of an optional parameter in the constructor.

2.3. Other Changes

One of the original design objectives for `RtAudio` was concurrent support for multiple audio devices. This functionality was dropped with version 3.0 to simplify the interface. Further, such support was deemed unnecessary and unreliable given the difficulties of synchronizing multiple devices with independent clocks. It is still possible with most APIs to simultaneously interface with multiple soundcards by creating several instances of `RtAudio`.

2.4. Ongoing Support

A number of items remain on the `RtAudio` "to-do" list:

- Currently, the API provides no method for selecting specific channels of a soundcard. For example, if an audio device supports 8 channels and the user wishes to send data out channels 7-8 only, it is necessary to open all 8 channels and write the two channels of output data to the correct positions in each audio frame of an interleaved data buffer.
- While `RtAudio` does support devices and APIs that pass non-interleaved buffers, it does not expose a mechanism for the user to read or write data that is non-interleaved. That is, the data buffer passed to the user must always be read or filled with interleaved data.
- There currently does not exist a mechanism to allow the user to "pre-fill" audio output buffers in order to allow precise measurement of an acoustic response.
- Support for multi-channel (> 2) audio input/output under the Windows OS is currently available only via the ASIO interface (in large part due to limitations with the DirectSound API). Other options for multi-channel support, perhaps using the WinMM API, should be investigated.

In an effort to have these issues addressed as fast as possible, it is likely that `RtAudio` will be made available for community development through a CVS server.

3. RTMIDI

Inspired by `RtAudio`, `RtMidi` is a set of C++ classes (`RtMidiIn` and `RtMidiOut`) that provides a common programming interface for realtime MIDI input and output across multiple computer operating systems. RtMidi significantly simplifies the process of interacting with computer MIDI hardware and software, with support for the Linux ALSA, Macintosh CoreMIDI, SGI md, and Windows Multimedia Library APIs.

Within the `RtMidi` framework, MIDI input and output functionality are separated into the `RtMidiIn` and `RtMidiOut` subclasses. Whereas simultaneous audio input/output operations must be handled together for proper synchronization, MIDI input and output operations are asynchronous. By separating the two functionalities, a simpler and more robust interface was possible.

In the following examples, we ignore error checking for the sake of clarity. `RtMidi` uses the same C++ exception handling class (`RtError`) as `RtAudio`. The error reporting functionality in `RtMidi` is encapsulated in a single function and can be easily modified to suit a given error handling scheme. Complete documentation is available from the `RtMidi` website[2] .

3.1. MIDI Output

The `RtMidiOut` class provides simple functionality to immediately send messages over a MIDI connection. No timing functionality is provided.

```
RtMidiOut midiout;
std::vector<unsigned char> message(3);

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

// Compose a Note On message.
message[0] = 144;
message[1] = 64;
message[2] = 90;

// Send the message immediately.
midiout.sendMessage( &message );
```

3.2. MIDI Input

The `RtMidiIn` class uses an internal callback function or thread to receive incoming messages from a MIDI port. These messages are then either queued and read by the user via the `getMessage()` function or immediately passed to a user-specified callback function (which must be "registered" using the `setCallback()` function). The following example uses the polled queue approach.

```
RtMidiIn midiin;
std::vector<unsigned char> message;
int nBytes;
double stamp;

// Open first available port.
midiin.openPort( 0 );

// Periodically check input queue.
while ( !done ) {
  stamp = midiin.getMessage( &message );
  nBytes = message.size();
  if ( nBytes > 0 ) {
    // Do something with MIDI data.
  }

  // Sleep for 10 milliseconds.
  SLEEP( 10 );
}
```

[2] `http://music.mcgill.ca/~gary/rtmidi/`

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

The `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 `setQueueSizeLimit()` function. If the maximum queue size limit is reached, subsequent incoming MIDI messages are discarded until the queue size is reduced.

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.

3.3. MIDI Ports

The `RtMidiIn` and `RtMidiOut` classes both provide `getPortCount()` and `getPortName()` functions to query the available MIDI ports on a system. A MIDI port represents a single MIDI input source or output destination.

The Linux ALSA and Macintosh CoreMIDI APIs allow for the establishment of virtual, or software-based, input and output MIDI ports to which other software clients can connect. `RtMidi` incorporates this functionality with the `openVirtualPort()` function. Any messages sent with the `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.

4. AN STK UPDATE

The Synthesis ToolKit in C++ has been publicly available since 1996. Throughout that time, the fundamental design goals have remained the same:

- Cross-platform functionality
- Programming flexibility
- Ease of use (educational example code)
- User extensibility
- Real-time synthesis and control
- Open source C and C++ code

STK currently runs with "realtime" support (provided by `RtAudio` and `RtMidi`) on Linux, Macintosh OS X, Windows, and SGI (Irix) computer platforms. Generic, non-realtime, scorefile-based support is available for all computer systems with a standard C++ compiler. There are currently over 90 classes provided in the ToolKit, 11 of which contain operating system dependencies related to audio and MIDI input/output, threading support, and socket-based, network communication.

STK provides an assortment of "unit generator" classes such as envelopes, filters, noise generators, looping wavetables, and input/output handlers. A large number of sound synthesis and audio effects algorithms are provided as well, including additive (Fourier) synthesis, subtractive synthesis, frequency modulation synthesis of various topologies, modal (resonant filter) synthesis, a variety of physical models including stringed and wind instruments, and physically inspired stochastic event models for the synthesis of particle sounds. Detailed usage and class information is available from the STK website [3].

4.1. `StkFloat`

All audio and control signals in STK are handled using the `StkFloat` floating-point data type. The `StkFloat` type is defined in the `Stk` abstract base class and can be modified to achieve a desired precision. By default, the `StkFloat` type is defined to be a 64-bit, double-precision floating-point value. Operations with ToolKit instruments and unit generators assume input and output values in the range $\pm$ 1.0. Note that this data type was previously defined as `MY_FLOAT` in STK versions before 4.2.0 (October 2004).

4.2. "Ticking"

STK was initially developed around a single-sample computational framework. All audio sample-based unit generators in STK implement a `tick()` method in which their fundamental sample calculations are performed. Some unit generators are only sample sources, such as the linear envelope generator `Envelope`, the noise generator `Noise`, or the soundfile input class `WvIn`. These source-only objects return `StkFloat` values, but take no input arguments via their `tick()` function. Consumer-only objects, like the soundfile output class `WvOut`, take an `StkFloat` argument and but return nothing. Objects like filters take and yield `StkFloat` values via their `tick()` function. In addition, all objects which are sources of audio samples implement a method `lastOut()` that returns the last computed sample. This allows a single source to feed multiple sample consuming objects without necessitating an interim storage variable.

4.3. Multi-Channel Data & `StkFrames`

New with STK version 4.2.0, the class `StkFrames` is provided to explicitly support vectorized computations and multi-channel audio data. Built using the efficient C++ Standard Library `valarray` class, `StkFrames` supports both interleaved and non-interleaved data, as well as an arbitrary number of channels per audio frame [9]. While functionality for vectorized computations using arrays of `StkFloat` values is still available, the `StkFrames` class

[3] `http://ccrma.stanford.edu/software/stk/`

provides a safer, robust, and more flexible mechanism for passing data arrays between STK unit generators. All unit generators have been modified to provide overloaded `tick()` functions that accept and/or return references to `StkFrames` objects. For classes that inherently operate on single-channel data, these overloaded `tick()` functions accept an input parameter specifying a particular channel for processing.

The following example demonstrates the use of an `StkFrames` instance for vectorized computations:

```
// Create 20 two-channel frames
StkFrames frames(20, 2);
Noise noise;

// Perform vectorized computation on
// channel two only.
noise.tick( frames, 2 );
```

Within the ToolKit, multi- and single-channel data processing are distinguished via the use of `tickFrame()` functions, as shown in the following example:

```
// Create 20 two-channel frames
StkFrames frames(20, 2);
Noise noise;

// Fill both channels with noise.
unsigned int i;
for ( i=0; i<frames.channels; i++ )
  noise.tick( frames, i );

// Filter each channel separately.
BiQuad filter1, filter2;
filter1.setResonance(440.0, 0.99, true);
filter2.setResonance(660.0, 0.99, true);
filter1.tick( frames, 1 );
filter2.tick( frames, 2 );

// Output the result to 16-bit WAV-file.
WvOut output( "icmc.wav", 2,
              WvOut::WVOUT_WAV,
              Stk::STK_SINT16 );

output.tickFrame( frames );
```

4.4. More Recent Changes

Several new classes were introduced with version 4.2.0 of the ToolKit, including an exponential envelope generator (`Asymp`) and a MIDI file reader (`MidiFileIn`). A variety of less visible modifications were made to the underlying structure of the ToolKit in an effort to improve performance, functionality, and better conform to standard C++ programming practices. Most of the classes were organized by functionality under the following new base class groupings: `Effect`, `Filter`, `Function`, `Generator`, and `Instrmnt`. To provide more consistent realtime performance across all supported platforms, most of the STK example programs were rewritten to use an audio callback paradigm.

Since STK version 4.0.0 (April 2002), complete class documentation and a tutorial have been provided in the distribution and from the STK website. This was accomplished using *Doxygen* by Dimitri van Heesch [10].

5. OTHER LIBRARIES & TOOLKITS

A variety of other open-source audio processing libraries exist, such as CLAM, SPKit, Sig++, Csound, and SndObj [1, 4, 2, 6, 3]. Of these, only CLAM, Csound, and SndObj appear to be actively supported. There are many common elements among these libraries, though STK is unique in its emphasis on algorithmic synthesis examples, including an extensive set of physical models.

`RtAudio` and `RtMidi` share many of the same goals as the PortAudio and PortMidi projects [5]. At the present time, however, PortMidi does not provide support for the Irix OS and version 19 of PortAudio has still not been completed after more than 3 years of development. In comparison, `RtAudio` and `RtMidi` provide a more simple interface for audio and MIDI control that makes for easier, and perhaps more robust, cross-platform support.

6. REFERENCES

[1] C++ Library for Audio and Music (CLAM). http://www.iua.upf.es/mtg/clam/.

[2] Sig++: Musical Signal Processing in C++. http://ccrma.stanford.edu/˜craig/sig/.

[3] The Sound Object Library (SndObj). http://www.nuim.ie/academic/music/musictec/-SndObj/.

[4] The Sound Processing Kit (SPKit). http://www.music.helsinki.fi/research/spkit/.

[5] R. Bencina and P. Burk. PortAudio - an Open Source Cross Platform Audio API. In *Proc. 2001 Int. Computer Music Conf.*, pages 263–266, Havana, Cuba, 2001. Comp. Music Assoc.

[6] R. Boulanger, editor. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing and Programming*. MIT Press, Boston, MA, 2000.

[7] P. R. Cook and G. P. Scavone. *Audio Anecdotes: A Cookbook of Audio Algorithms and Techniques*, chapter The Synthesis ToolKit (STK) in C++. A.K. Peters, Natick, MA, 2004.

[8] G. P. Scavone. RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output. In *Proc. 2002 Int. Computer Music Conf.*, pages 196–199, Göteborg, Sweden, 2002. Comp. Music Assoc.

[9] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 2000.

[10] D. van Heesch. `http://www.doxygen.org/`.