Sound Programming for Windows in C++

Audio and MIDI

By
Ismael Mosquera Rivera

To my nephew and friend Sergi, with my best wishes.

Abstract

This paper describes a C++ implementation using the low level API offered by MSWindows. This development covers the following concepts:

Audio: Storage, recording and playback.
Audio file formats currently supported: Wave (*.wav)

MIDI: Storage and playback.

The idea is to add more features, since this is the first attempt of implementing such a library. We expect, for instance, adding support for MP3 storage and for other audio file formats. We would like to manage MIDI input too, since at this moment there are just support for MIDI output; well, we can say that is some support for MIDI input in storage by loading SMF (Standard MIDI File).

The language that we decided to use is C++, because it is an excellent object oriented programming language, and enables scalability to our project. With this implementation, several OOP concepts are demonstrated: Inheritance, Encapsulation, Polymorphism... inmutable objects and mor.

The audience for this work that we expect are people interested in how to program low level sound applications for Windows, but the skills and C++ concepts are independent of any platform so, we think that they can be adapted in other platforms like Linux.

If the reader finds this work interesting, we encourage you to join it and share ideas to improve it by refactoring and add extra features.

Contents

1 Introduction

This low level implementation for Audio and MIDI for MSWindows was built from the scratch; that is, we just use headers and libraries which come with any C/C++ compiler for Windows. In our case, we use the MinGW 32 bit compiler; the only added package was pthreads-w32 in order to launch the players and recorder as an independent thread.

If you already have the MinGW (Minimalist GNU for Windows) installed, but not the pthreads-w32 package, type and execute the following command:

```
Mingw-get install pthreads-w32
```

And the posix threads package will be downloaded and installed in your system. You can download the mingw-get command line installer from:

https://github.com/ismaelmosquera/cwmidi

In that repository, there is a pdf where the compiler installation is explained.

In this first implementation for the project, we have the following features:

Store, play and record audio (currently just Wave file format supported).
Storage and playback for MIDI (Musical Instrument Digital Interface).

Since this is a first attempt to build such a library, we designed and structured the code to be scalable, but maybe some little refactoring can be needed, overall for the Audio class in order to make it more flexible adding support for more audio file formats like MP3. Well, if refactoring supposes improvement, it is welcome evrywhere.

To test the functionality offered by this namespace (IMR), there are 2 examples: player and recorder. In the player example, you can play *.wav and *.mid files and polymorphism is demostrated by asign the proper kind of player in any case. In the recorder example, you can record, play and store the resulting record in a Wave audio file.

In this namespace, you can get information about all Audio and MIDI I/O devices in your system retrieving that info through inmutable objects.

The FileUtil class composed by static methods, allows us to read/write audio file headers and read/write entire SMF (Standard MIDI Files).

In the next chapter, we present briefly the tools used in this project; afterwards, we discuss the implementation; then a chapter with some conclusions and, finally, there is an appendix with some notes to build the library and a comment about the author.

We expect that this work could be of interest and also useful for somebody.

2 Tools

To accomplished this project, we used the following tools:

A programming language.
A text editor.
A compiler.
A tool to automate compilation.

The choosed programming language was C++; apart that it is a very efficient programming language, we can call C code from a C++ compiler; the inverse is not true, that is, we cannot call C++ code from a C compiler. So, we could thingk that C is a subset of C++; well, in fact, that can be true but really it is not. The C++ programming language, developed by Bjarne Stroustrup, has a different semantics than C; first of all, C++ is an object oriented programming language but C is just a procedural language; the C programming language does not support object oriented technology.

On the other hand, using C++ we can structure the code in an elegant fashion and we can also apply software patterns to do some tasks in a compact way. Since this is a project which we expect to continue adding extra features, with C++ we can achieve a higher level of scalability by adding functionality without need to change existing code.

The C++ programming language has grown since his first implementation; apart from parametric classes (templates), lambda expressions and even support for multithreading has been added as built-in features of the language; anyway, as the thread technology is concern, we decided to use the posix pthreads package in this project when needed.

The minimum that an object oriented programming language must support are: Inheritance, Encapsulation and Polymorphism; from this, we can add support for software patterns, enable easy scalability, inmutable objects, namespaces to avoid naming conflicts and much more, with the valuable fact offered by use C code inside C++ implementations.

To type the code, a simple text editor was enough, we did not use an IDE (Integrated Development Environment), but a nice text editor capable to work with multiple archives opened. You can use your prefered one, but a simple editor without format like ‘notepad’ could be ok.

The compiler choosed was MinGW (Minimalist GNU for Windows), which we think is an excellent and free tool where you can have a ‘little Linux’ in your MSWindows system. Finally, to build the code, the make utility was employed; this tool was initially thought to work with C/C++ code, but you can adapt it to work fine with any language compiler running from the command line.

Following, we recommend some readings and a link to the MinGW web site.

To learn about C++, you can consult the book:
Bjarne Stroustrup**: The C++ Programming Language 4th ed, Addison-Wesley**

You can download the MinGW compiler from:
https://osdn.net/projects/mingw/

To know more of the GNU make utility, a good book is:
Robert Mecklenburg: Managing Projects with GNU Make 3rd ed, O'Reilly

And also you can look at the following link:
http://savannah.gnu.org/projects/make/

3 Implementation

This chapter is devoted to explain how this implementation in C++ for some aspects of digital audio and MIDI was done. As we said before, this is a first attempt to implement this namespace which offers a littel audio and MIDI features ready to work. On the other hand, this code works only under MSWindows systems since it uses the low level API offered by that OS. Actually, we wanted to do cross platform, but because the author got blind, we did under Windows for having a complete functional screen reader in order to have a more integral access to the concrete computer system.

Now that we clarified this issue, let us go strict to the point. Since we did it from the scratch, a number of classes were define to achieve the first goal: Storage for digital audio and MIDI, record and play digital audio an MIDI playback. We make an effort to avoid code redundancy and also code reusability and open scalability. In this first version,we did it as well as we can, but we recognize that to add some extra features we must do some refactoring.

Currently, we first design an Audio class encapsulating raw adio and metadata about its format; in fact, to support wave audio files (*.wav). So, an Audio object has a wave audio header; that enable us to load and store wave audio files, another useful class: FileUtil; this class allows us to read/write audio file headers, and also read/write entire SMF (Standard MIDI Files); from this, we could add support for other audio file formats improving the capabilities for this namespace.

Another useful feature already implemented is that we can retrieve information about all Audio/MIDI I/O devices in your system, so, you can select the one prefered depending on your needs.

In this chapter, we are not going to explain in detail all the aspects of the implementation, but some about the abstraction offered by this set of classes working together to achieve the goal. Anyway, you have all the code availabel to pick up inside it whatever you want.

The general idea for this implementation is not too complex, in fact, the usage is so easy. For instance, look at the interface to load and store audio files:

```
#ifndef ___AUDIOFILE_H___
#define ___AUDIOFILE_H___

#include <stdio.h>
#include "Audio.h"
#include "AudioFileHeader.h"

namespace IMR
{
/**
* Class AudioFile.
*
* This is an abstract class which acts as an interface to load and store audio files.
*
* Concrete audio file formats must inherit from this class and implement its pure virtual
methods.
*
* @see WaveAudioFile
*/
class AudioFile
{

public:

/** constructor */
AudioFile() : _fpread(0), _fpwrite(0)
{}

/** virtual destructor */
virtual ~AudioFile()
{
        if(_fpread) fclose(_fpread);
        if(_fpwrite) fclose(_fpwrite);
        }

/**
* Virtual methods to be implemented for the concrete subclasses.
*
*/

virtual bool Load(Audio&, const char*) = 0;
virtual bool Store(const Audio&, const char*) = 0;

protected:
FILE* _fpread;
FILE* _fpwrite;
AudioFileHeader _header;

virtual bool ReadHeader() = 0;
virtual void WriteHeader() = 0;

};

}

#endif
```

At this moment, we just subclassed this one to implement support to load/store wave audio files, as we said before, but you can use this interface to enable support for storage to other audio file formats.

Now that we can load a wave audio file, one think that we can do is playing it; since we want to implement a MIDI player too, we should define another abstract class for a generic player implementation:

```
#ifndef ___PLAYER_H___
#define ___PLAYER_H___

namespace IMR
{

/**
* Class Player.
*
* Player abstract class.
* This class acts like an interface from any player can be subclassed.
* It has an already implemented method to set the concrete device ID for a player.
* The virtual methods must be implemented by the subclasses inherited by this class.
*
* @see AudioPlayer
* @ see MIDIPlayer
*/
class Player
{

public:

/** constructor */
Player() : _devID(0) {}

/** virtual destructor */
virtual ~Player() {}

/**
* Sets the device id for this player.
*
* @param int id  The device ID
*
*/
void SetDeviceID(int id)
{
_devID = id;
}

/**
* These methods must be implemented by subclasses inherited from this one.
*/
virtual bool Load(const char*) = 0;
virtual int Play() = 0;
virtual void Stop() = 0;

protected:
int _devID;
};

}

#endif
```

Subclassing from this abstract class, we can implement an Audio player and also a MIDI player, which enables polymorphism, so, if we wish, for example, make an implementation for a player capable to reproduce digital audio and MIDI, we just inherit each concrete player from this one and asign the correct one in order to do the task:

```
IMR::Player* player = 0;
IMR::Player* audio_player = new IMR::AudioPlayer();
IMR::Player* midi_player = new IMR::MIDIPlayer();
< request for a sound file >
```

```
If ext equals “mid”
Player = midi_player;
Else
Player = audio_player;
Player->Load(“filename”);
Player->Play();
...
player->Stop();
```

To record digital audio we defined the class ‘AudioRecorder’ which has the following interface:

```
#ifndef ___AUDIORECORDER_H___
#define ___AUDIORECORDER_H___

#include <pthread.h>

#include "AudioFileHeader.h"
#include "Audio.h"

namespace IMR
{

/**
* Class AudioRecorder.
*
* This class implements an audio recorder.
*
* Recording is done by the followind features:
* - Sample Rate = 11025
* - Byte Rate = 22050
* - Number of Channels = 1
* - Bits per sample = 16
*
* After recording, you can retrieve the recorded Audio object and store it in a audio file.
*
* @see AudioFile
*/
class AudioRecorder
{

public:
/** Constructor */
AudioRecorder();
/** Destructor */
~AudioRecorder();

/**
* Sets the device ID for this audio recorder object.
*
* @param int Device ID
*
* @see AudioDevice
*/
void SetDeviceID(int);

/**
* Gets the recorded Audio.
*
* @return the recorded Audio object
*/
const Audio& GetAudio() const;

/**
* Stars recording.
*
* @return 0 if success, 1 otherwise
*/
int Rec();

/**
* Stops recording.
*/
void Stop();

private:
int _devID;
Audio _audio;
AudioFileHeader _fileHeader;
void FillRecordedAudio();

pthread_t _pt;
static void* LaunchThread(void* ptr);
void thread_code();
};

}

#endif
```

To make thinks easier, we coded an AudioSystem class to get the most relevant objects related to audio; the interface is as follows:

```
#ifndef ___AUDIOSYSTEM_H___
#define ___AUDIOSYSTEM_H___

#include <list>

namespace IMR
{

class AudioDevice;
class Player;
class AudioRecorder;
class AudioFile;

/**
* Class AudioSystem.
*
```

```
* This class acts like a facade to get the most relevant objects for the IMR audio
system.
* It has static methods to do the task.
*
* @see AudioDevice
* @ see AudioInDevice
* @see AudioOutDevice
* @see Player
* @see AudioPlayer
* @see AudioRecorder
* @see AudioFile
* @see WaveAudioFile
*/
class AudioSystem
{

public:

/**
* Gets a list containing all the audio input devices in your system.
*
* @return The list of audio input devices available in your system.
*/
static std::list<AudioDevice*> GetInputDeviceList();

/**
* Gets a list with all the audio output devices available in your system
*
* @return The list of audio output devices available in your system
*/
static std::list<AudioDevice*> GetOutputDeviceList();

/**
* Gets an audio player.
*
* Retrieves a pointer to an audio player suitable to load, play and stop audio.
*
* @ see Player
* @see AudioPlayer
*
* @return An audio player object.
*/
static Player* GetPlayer();

/**
* Gets an audio recorder from your audio system.
*
* @see AudioRecorder
* @return An audio recorder object
*/
static AudioRecorder* GetRecorder();

/**
* Gets a WaveAudioFile object from your audio system.
*
* @see AudioFile
* @see WaveAudioFile
* @return a WaveAudioFileObject
*/
static AudioFile* GetWaveFile();

};

}

#endif
```

We already presented the digital audio capabilities offered by this first version of this project; the interface is so intuitive and easy to use, and all the header files (*.h) are fully documented. Now it is time to explain the support for MIDI, which interface follows the same style as the digital audio one. We can look, for example to the interface for storage:

```
#ifndef ___MIDIFILE_H___
#define ___MIDIFILE_H___

#include "MIDISong.h"

namespace IMR
{

/**
* Class MIDIFile.
*
* This class has methods to load an store SMF ( Standard MIDI File ).
* The information is stored in a MIDISong object.
*
* @see MIDISong
*/
class MIDIFile
{

public:

/** constructor */
MIDIFile();

/** destructor */
~MIDIFile();

/**
* This method loads a MIDI file and stores its information in a MIDISong object.
*
* @param const MIDISong& A MIDISong object to store the information extracted from
a MIDI file
* @param const char* Filename
*
* @see MIDISong
*
* @return true if success or false otherwise
*/
bool Load(MIDISong&, const char*);

/**
* This method stores the information in a MIDISong object to a MIDI file.
*
* @param const MIDISong& A MIDISong object with proper information
* @param const char* Filename
*
* @see MIDISong
*
* @return true if success or false otherwise
*/
bool Store(const MIDISong&, const char*);

};

}

#endif
```

As we can see, the interface is so intuitive and easy to use as the one for digital audio storage. When a MIDI file is loaded, the information extracted is stored in a MIDISong object; other classes involved in that task are MIDITrack, MIDIEvent and MIDIMessage. As in the case of digital audio, one think that we can do after loading a SMF is playing it. There is a MIDIUtil class which offers one static method to get the tempo in microseconds from a MIDI tempo event, and another one to convert ticks (timestamp for MIDI events in a MIDI file) to milliseconds, which we use to launch MIDI events to be played at the correct time.

We also coded a MIDISystem class to easy get the most relevant components related to MIDI through static methods:

```
#ifndef ___MIDISYSTEM_H___
#define ___MIDISYSTEM_H___

#include <list>

namespace IMR
{

class MIDIDevice;
class Player;
class MIDIFile;

/**
* Class MIDISystem.
*
* This class acts like a facade to get the most relevant objects for the IMR MIDI system.
* It has static methods to do the task.
*
* @see MIDIDevice
* @ see MIDIInDevice
* @see MIDIOutDevice
* @see Player
* @see MIDIPlayer
* @see MIDIFile
*/
class MIDISystem
{

public:

/**
* Gets a list containing all the MIDI input devices in your system.
*
* @return The list of MIDI input devices available in your system.
*/
static std::list<MIDIDevice*> GetInputDeviceList();

/**
* Gets a list with all the MIDI output devices available in your system
*
* @return The list of MIDI output devices available in your system
*/
static std::list<MIDIDevice*> GetOutputDeviceList();

/**
* Gets a MIDI player.
*
* Retrieves a pointer to a MIDI player suitable to load, play and stop MIDI.
*
* @ see Player
* @see MIDIPlayer
*
* @return A MIDI player object.
*/
static Player* GetPlayer();

/**
* Gets a MIDIFile object from your MIDI system.
*
* @see MIDIFile
* @return a MIDIFileObject
*/
static MIDIFile* GetMIDIFile();

};

}

#endif
```

As we can see, the MIDISystem class allows us to get easily all what we need to know about the MIDI I/O devices in our system and the objects needed to storage and playback MIDI data.

By the way, related to wave audio file headers, we have to say that the quantities are store as big-endian, so, we need to code methods to solve this issue. Our solution was to implement another class with static methods to solve the problem; we have to flip the bytes to change from big-endian to little-endian and to pass little-endian to big-endian; our class has 2 methods, one to flip a 16 bit quantities and another to flip 32 bit; the same method allow us to do the two tasks; that is, if we flip a 16 bit value to change from big-endian to little-endian, the same method will change the resulting quantity from little-endian to big-endian; those methods are implemented in the EndianUtil class.

4 Conclusions

We did a first implementation of a project suitable to manage some basics about digital audio and MIDI. The idea is to add extra features related to this topic as can be support for several audio file formats, MIDI input, and whatever we could do in the digital audio and MIDI ambit.

On the other hand, we expect that this work could be useful for everyone interested in this subject in order to get knowledge about a possible way to make a C++ sound programming implementation under MSWindows.

As you already know, any contrubution is welcome.

6 Appendix

In this brief appendix, we explain how is the code structured and how to build the shared library and the examples for this namespace.

The root folder is <imrsnd>; inside it we can find the following subfolders and files:

<examples>
<include>
<lib>
<src>
Makefile
Readme.txt

Some notes about building the library are in the 'readme.txt' file. The compiler used was MinGW (Minimalist GNU for Windows) which is the only tool that you need, since the compiler, make utility and the posix threads package are integrated in it. We now remember you the command line you have to type in order to install whatever you need using the mingw-get command line installer which you can download from:

https://github.com/ismaelmosquera/cwmidi/

to install all the components that you need type:

```
mingw-get install gcc g++ gdb pthreads-w32 mingw32-make msys-base
```

Inside the examples folder, you can find an example for playing and another one for recording; in both folders, there are 'readme.txt' files explaining how to build and execute them.

We hope that this work could be interesting and useful to somebody.

About The Author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org (C++ Library for Audio and Music); unfortunately, he is affected by a severe eye disease (Retinitis Pigmenthosa), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this little package encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.