MIDI Programming for Windows in C

SMF and Playback

By
Ismael Mosquera Rivera

To my mother and in memory of my father.

Abstract

This paper describes the design and implementation of a small MIDI library covering Standard MIDI Files and a player to reproduce MIDI sequences.

The code is completely written in the C language from the scratch. The library is for Windows, but you can easily adapt the code for Linux just changing the part related to MIDI out device.

The lack of literature about this subject as encouraged the author to write this paper in the hope that it would be useful for an interested reader who wants to learn something about the MIDI protocol so, it is primarily focused in what is really needed as this article is concerned.

In addition of the brief theory about MIDI, all the code of the library is also available. It was entirely developed in a very simple laptop running Windows XP 32 bits, and it runs smoothly; It is supposed to run well under other Windows system.

Contents

1 Introduction

As we said before, here it is covered only a small subset of what the MIDI protocol is capable to do. We do not talk about MIDI input, but just output; the main reason is because the author has available just an old computer which has only one MIDI output port, anyway, covering SMF (Standard MIDI Files) and the implementation of a player for reproduce MIDI sequences is enough to take a good understanding about MIDI.

Well, in some way, some part of input is done by reading MIDI files. All the structures in C and the algorithms used were developed from scratch. Although it can seem, and in fact is, just a few among what this protocol can perform, sometimes is best to know some basics at the beginning than get too many information so, this paper was thought just in that direction; learn step by step with a good introduction about MIDI programming in C.

Now, in this brief introductory chapter we are going to tell what the reader will find in this article. The next chapter is a MIDI overview which explains some aspects of MIDI always focused on what is covered here; messages, events, tracks... SMF, General MIDI and so on.

In the third chapter, the tools used in this project are presented. Not too much was needed for that, just a compiler and a few more. The subject of the four chapter is the design and implementation of the small developed library; structures, concepts and explanations, always accompanied by the sources which can be consulted at any moment. In the fifth chapter, we talk about the conclusions taken from this work and finally, there is an appendix explaining how to build the library and some notes about the author.

In the hope that the reader find this paper useful and can learn about MIDI or reinforce his or her knowledge, let us start. Enjoy it!

2 MIDI Overview

MIDI stands for Musical Instrument Digital Interface. It is a protocol widely accepted for authors to make their musical compositions. Its intent is to connect transmitters and receivers in order to achieve the goal; for example, a transmitter can be a MIDI keyboard, and a receiver a MIDI synthesizer; then, when a transmitter sends a message to a receiver the last one must generate the requested sound. The advantages of using MIDI instead digitalized audio are several; one of them is storage, since the space needed to sample and quantify 1 minute of digitalized audio is extremely large than the same one compared with MIDI; in the case of sampled audio we are talking of megabytes while in case of MIDI we need just a few bytes. So, it is clear that MIDI data is an excellent choice not only for musical compositions, but broadly used in other areas as can be multimedia applications or computer games. Another benefit of using MIDI to compose musical pieces is that the composer can edit easily her composition for instance, changing the tempo or making adjustments on any part of the song by using a sequencer.

Sound Synthesis

At the beginning, most synthesizers were so simple that they were only able to play just one sound each time; later, more sophisticated ones were developed. Nowadays, there are synthesizers capable of playing polyphonically and multitimbral. In a computer, the MIDI facilities are integrated in a sound card; depending of the type and quality of the sound card, it can have even several input and output ports; in our case, just a synthesizer with one MIDI out port is required.

There are, among others, two widely used techniques for sound synthesis; one of them, is Frequency Modulation, where a periodic signal (the modulator), modulates another signal (the carrier), so, for FM synthesis, at least two signal generators are needed. Using FM synthesis, maybe with appropriate envelopes, a significant change in the timbre of the produced sound can be achieved. Although early FM synthesizers were implemented in the analog domain, today they can perform digitally. FM synthesis has very good results creating new sounds but, if what we wish is to recreate an existing sound, for instance, the sound produced by a concrete instrument as can be a trumpet, a more sofisticated technique is available: Wavetable synthesis.

In Wavetable synthesis digital sample-based tecniques are employed; the idea is to store high quality of digital audio samples in a number of segments and apply different techniques such looping, pitch shifting, mathematical interpolation and digital filtering to get more accuracy when the sound is produced and also to save some amount of memory. In these synthesis systems, the sampled audio segments are stored in a tabular structure, so that they can be accessed rapidly. In the rest of this section, we are going to describe briefly this variety of tecniques.

One of the primary techniques used is called looping combined with envelope generation; in this technique, a short segment of samples is stored and loop repetitively during playback; so, a great amount of memory can be sabed. Normally, two sections of the sound are stored: attack and sustain; in the attack section, the spectral characteristics of the sound can change rapidly, while in the sustain section they remain fairly constant. The attack section is very short and it is played once through then, the sustain section is

played looping until the end of the sound. An envelope generation function is applied to the wave in order to model the timbre of each concrete instrument with the appropriate envelope; a commonly used kind of envelope is composed of four linear segments: ADSR (Attack, Decay, sustain, Release). When a Note On message is received, the attack section is played immediately until the maximum amplitude is reached, then a relatively short decay section followed by the sustain section then, when the Note Off message is received it is the time for the release section. There are some kind of sounds where this technique is not suitable to used; short duration sounds like drums, for instance; these sounds are called one-shot sounds; in this case, the sound is stored in a single segment and played once through without looping.

Another useful technique in wavetable synthesis is pitch shifting, also called pitch transposition. The idea is to access the samples at different rates while playback in order to increase or decrease the pitch for a musical note. For instance, if we take every second sample instead every one we shift up the pitch for an octave because we double the frequency; a mor general way is to increase the pitch in a fractional part; to implement this we access the samples using a pointer also called phase accumulator and to maintain frequency accuracy we use another pointer named phase increment. With an increment of ½, for example, we shift down an octave; an octave has 12 semitones and one semitone equals to an increment value of 1.05946 which is the 12^{th} root of 2; so, we can increment or decrement the pitch achieving the desired granularity; for example, if we have a sample segment to play middle C, with this technique we can process the samples in the segment to play a range of different musical notes.

To reduce distortion, we can apply mathematical interpolation. There are different algorithms for this technique; the simplest is linear interpolation where we have just to compute the weighted average between two samples, for instance, if we want to interpolate only 1 sample and we call s0 and s1 the related samples then the average is (s1 – s0) / 2; in general, if we want to interpolate n samples between s0 and s1 the distance between samples will be d = (s1 – s0) / (n + 1). There are more sophisticated interpolation algorithms, but they can be computationally expensive.

Low-pass filtering is used to eliminate aliasing noise, applying a low-pass filter with the adequate cutoff frequency aliasing noise can be eliminated.

We saw that when we use pitch shifting only a limited range of notes can be played naturally since the timbre of the instrument can change; to avoid that, we can have several splits, each one with a different sampled wave; that is, we distribute the entire range of pitchs for an instrument into splits in order to achieve the goal.

Actually, sound synthesis systems use not only one synthesis tecnique but a combination of them, the more the tecniques we will combine the better quality of the resulting synthesized sound, for example, we can also apply LFOs (Low Frequency Oscillators), to get some effects like vibrato and tremolo; for vibrato, we modulate the frequency and for tremolo we modulate the amplitude of the sound. There are more tecniques that can be applied; data compression for instance, where the data is decompressed during playback.

Probably, due to the excellent results for synthesis sound in wavetable synthesis, the trend in this area will continue in that direction.

Channel Voice Messages

MIDI is a protocol where the comunication is achieve through messages. A MIDI message can have 3 bytes as much; the first byte is named status byte and the

most significant bit is always set to binary 1, agterwards, there are 1 or two data bytes which the most significant bit is always set to binary 0. The code for each different message is placed in the most significant nibble of the status byte, since the most significant bit in that nibble is always set to binary 1, we have 3 bits to encode a particular message 1000 .. 1111 in binary. There are 7 different channel voice messages, the remaining bit, when used, is for another kind of messages like system exclusive messages or meta messages, but this subject will be covered in the next section. The less significant nibble in the status byte is used to select a concrete channel, because a nibble has 4 bits, we have 16 logical channels available to send MIDI messages to them. Since the most significant bit in the data bytes is always set to 0, the range of values is 00000000 .. 01111111 in binary; 0x00 .. 0x7F in hexadecimal; 0 .. 127 in decimal.

Following, we are going to describe this fundamental set of 7 messages ordered by status byte.

NOTE OFF:
status byte: 1000nnnn binary; 0x8N hexadecimal, data1: pitch, data2: velocity.

The nnnn in binary means 4 bits, the N in hexadecimal means 1 nibble; this explanation is valid for the rest of messages, remember that the less significant nibble in the status byte is used for channel selection.

The pitch value in the data1 byte determines the note to play; so we have 128 values where each increment of 1 equals to a semitone, because an octave has 12 semitones, then 128 / 12 = 10.6 which means a range with more than 10 octaves available; that is enough to cover the entire range of any existing musical instrument.

Velocity is the pressure applied to release the key; we will see later that a note off message is equivalent to a note on message with velocity = 0.

NOTE ON:
status: 1001nnnn binary; 0x9N hexadecimal, data1: pitch, data2: velocity.

The explanation for this message is equivalent to above. In this case the velocity value determines the loudness of the sound, that is, the more the pressure over the key the louder the produced sound. A note on message with velocity equals zero is equivalent to a note off message.

An optimization called Running Status is useful in some cases where the status byte of the next message equals to the previous one. For instance, a chain of note on and their note off messages can be transmitted with the same status byte, so we need to send only the data bytes; that is achieved sending note on messages with the velocity value set to 0 which is equivalent to a note off message.

POLY AFTER TOUCH:
status: 1010nnnn binary; 0xAN hexadecimal, data1: pitch, data2: pressure.

This message affects a single note, applying it to a concrete sound we can modify volume, vibrato, timbre, etc by moving an horizontal slider or by pressing down on the key after it “bottoms out”.

CONTROL CHANGE:
status: 1011nnnn binary; 0xBN hexadecimal, data1: control type, data2: value.

Normally, this message is sent by a device different than a keyboard, as can be a pedal, a slider, a wheel, etc. Currently, there are 120 control types 0 .. 119. They can also control sound effects like tremolo or chorus among others. Usually, the minimum value for a concrete control is set to 0 and the maximum is set to 127; for example,

volume control has type = 7 and it is applied to a particular channel with minimum = 0 and maximum = 127; a balance control has control type = 8 where a value of 0 corresponds to a maximum left, a value = 64 for center and a value = 127 for maximum right.

An important control type is used to bank selection; this control is sent before a program change message. There are a number of control types but we are not going to describe all of them here.

PROGRAM CHANGE:
status: 1100nnnn binary; 0xCN hexadecimal, data1: program.

Sending this message to a particular channel with a program value in the range 00000000 .. 01111111 binary; 0x00 .. 0x7F hexadecimal; 0 .. 127 decimal, causes a change in the selected musical instrument; that is, the following sounds in that concrete channel will be played according with the timbre of the instrument associated with that program number. Notice that this message has only one data byte so, there is a palette of 128 different instruments available.

CHANNEL AFTER TOUCH:
status: 1101nnnn binary; 0xDN hexadecimal, data1: pressure.

The description for this message is the same as for poly after touch, except that in this case the effect applies for the entire selected channel instead for just a single note.

PITCH BEND:
status: 1110nnnn binary; 0xEN hexadecimal, data1: MS byte, data2: LS byte.

The purpose of this message is to make smooth variations in the pitch with a limit of 2 semitones op and 2 semitones down. Usually, this effect is controlled by a wheel or similar. The values for data1 and data2 are as follows:

Maximum down: 0x00, 0x00 hexadecimal; 0, 0 decimal.
Center (no effect): 0x00, 0x40 hexadecimal; 0, 64 decimal.
Maximum up: 0x7F, 0x7F hexadecimal; 127, 127 decimal.

This message is the last one of this fundamental set of 7 MIDI messages.

Events

A MIDI event contains a MIDI message and a timestamp. The most frequent messages in a MIDI song belong to the above described set of fundamental messages, that is, channel voice messages. A MIDI event can be composed also by a meta message or a system exclusive message instead, in that case, a buffer for extra bytes is needed, where the number of bytes in the buffer can be different depending on the type of message.

A MIDI meta event has a status byte equals to 0xFF. There are several kind of meta events; for example, a meta event containing the lyrics of the song, a one with the name of a particular track, a tempo event is also a meta event; the extra buffer of bytes stores relevant information for a concrete meta event.

In the case of a system exclusive message, the manufacturer is responsible to publish an ID for a particular sysex event and also a specification for the event. A sysex event has status byte equals to 0xF0 and a buffer of bytes to store the

extra bytes; a sysex event must terminate with what is called an EOX (End Of Exclusive) byte which equals to 0xF7. A MIDI system exclusive event could have different formats, some manufacturers send sysex events through paquets.

The timestamp for a particular MIDI event tells the system when that concrete event must be launched.

Tracks & Sequences

A MIDI track is a list of MIDI events and also could have a name. A MIDI Sequence is a list of MIDI tracks; in the rest of this article, we will call MIDI song or MIDI sequence to reference the same thing.

A MIDI song has one track or multiple tracks, so a composer can work on each track independently and, finally, reproduce all the tracks at once to know the rsult and, eventually, make changes if necessary.

There is not too much to say about tracks and songs as the purpose of this article is concerned, anyway, in the next section related to standard MIDI files we will complete some concepts.

Standard MIDI Files

This standard was created to solve the isssue of sharing MIDI files among different file systems. A SMF (Standard MIDI File) is simply a stream of bytes with a well defined format. Basically, a SMF has two types of what we can call MIDI chumck types; They have a header chunck an one or more track chunck types. A MIDI chunck file header is placed at the beginning of the file as follows:

‘M’ ‘T’ ‘h’ ‘d’ < 32-bit value >
length in bytes < 32-bit value >
format < 16-bit value >
# of tracks < 16-bit value >
division < 16-bit value >

As we can see, the first are 4 ascii coded letters (MThd), following by the length in bytes as a 32-bit value, afterwards, there are 3 16-bit values: format, number of tracks and division. There are three types of formats: 0, 1, and 2; we take in account just formats 0 and 1. A SMF has format 0 if it has only one track and format 1 if it has multiple tracks; format 2 is for files containing more than one sequence but, normally, this format is not used; the division is the number of ticks per quarter note.

After the header chunck, it could be one multi channel track in case of format 0 and a number of tracks in case of format 1. The chunk type for a MIDI track is as follos:

‘M’ ‘T’ ‘r’ ‘k’ < 32-bit value >
track length in bytes < 32-bit value >
< track data >
<end of track >

The end of track is a meta message with the value 0xFF for status byte and 0x2F for meta type. So, an abstraction of the format for a SMF could have the following structure:

“MThd” < 32-bit value >
0x00000006 < 32-bit value >
< format > < 16-bit value >
< ntracks > < 16-bit value >
< division > < 16-bit value >

“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >
“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >
“MTrk” < 32-bit value >
< number of bytes in the track > < 32-bit value >
< track data >
< end of track >

...

The track data is the event list in the track, remember that a MIDI event has a message and a timestamp, in SMF the timestamp is messured in ticks; some numbers in SMF are represented as what is called a variable-lenght quantity; these numbers are represented 7 bits per byte, most significant bits first; all bytes except the last have bit 7 set, and the last byte has bit 7 clear; if the number is between 0 and 127, it is thus represented exactly as one byte.

The numbers represented as a variable-lenght quantity are those related to a timestamp, or to extra data in a meta or sysex message.

We identify the kind of message looking at its status byte, if it is in the range 0x8N .. 0xEN then, it is a channel voice message, if its status byte equals to 0xFF it is a meta message and the next byte indicates which type of meta message it is, and if the status byte is 0xF0, it is a system exclusive message which must terminate with 0xF7.

We will retrieve this subject in the fourth chapter, devoted to design and implementation, to give a little bit more information about it.

General MIDI

The main motivation for create this standard was the compatibility among synthesizers. Without it, the same program number could have a different instrument timbre depending on the manufacturer for each concrete synthesizer, what causes that a particular music composition can perform without achieve the intended sound.

To accomplish with GM (General MIDI), a system must have a minimum set of requirements; some of them are related to control messages but, maybe the most important are the program numbers for the sound presets. A synthesizer accomplishing with GM must have 128 sound presets for different instruments, each program number associated with the same instrument timbre, and 47 sound presets for drums; in addition, the channel 10 is reserved for drums; remember that there are 16 availabel channels indexed 0 to 15 in decimal 0000 .. 1111 in binary.

Following, we are going to list all of these sound presets ordering by their program number; take in account that the 128 sound presets are internaly indexed 0 .. 127 in a computer.

List of GM instruments

Piano
1 Acoustic Grand Piano
2 Bright Acoustic Piano
3 Electric grand Piano
4 Honky Tonk Piano
5 Electric Piano 1
6 Electric Piano 2
7 Harpsichord
8 Clavinet

Chromatic Percussion
9 Celesra
10 Glockenspiel
11 Music Box
12 Vibraphone
13 Marimba
14 Xylophone
15 Tubular bells
16 Dulcimer

Organ
17 Drawbar Organ
18 Percussive Organ
18 Rock Organ
20 Church Organ
21 Reed Organ
22 Accordion
23 Harmonica
24 Tango Accordion

Guitar
25 Nylon Accustic Guitar
26 Steel Acoustic Guitar
27 Jazz Electric Guitar
28 Cian Electric Guitar
29 Muted Electric Guitar
30 Overdrive Guitar
31 Distorted Guitar
32 Guitar Harmonics

Bass
33 Acoustic Bass
34 Electric Fingered Bass
35 Electric Picked Bass
36 Fretless Bass
37 Slap Bass 1
38 Slap Bass 2
39 Syn Bass 1
40 Syn Bass 2

Strings/Orchestra
41 Violin
42 Viola
43 Cello
44 Contrabass
45 Tremolo Strings
46 Pizzicato Strings
47 Orchestral Harp
48 Timpani

Ensemble
49 String Ensemble 1
50 String Ensemble 2 (Slow)
51 Syn Strings 1
52 Syn Strings 2
53 Choir Aahs
54 Voice Oohs
55 Syn Choir
56 Orchestral Hit

Brass
57 Trumpet
58 Trombone
59 Tuba
60 Muted Trumpet
61 French Horn
62 Brass Section
63 Syn Brass 1
64 Syn Brass 2

Reed
65 Soprano Sax
66 Alto Sax
67 Tenor Sax
68 Baritone Sax
69 Oboe
70 English Horn
71 Bassoon
72 Clarinet

Pipe
73 Piccolo
74 Flute
75 Recorder
76 Pan Flute

77 Bottle Blow
78 Shakuhachi
79 Whistle
80 Ocarina

Synth Lead
81 Syn Square Wave
82 Syn Sawtooth Wave
83 Syn Calliope
84 Syn Chiff
85 Syn Charang
86 Syn Voice
87 Syn Fifths Sawtooth Wave
88 Syn Brass & Lead

Synth Pad
89 New Age Syn Pad
90 Warm Syn Pad
91 Polysynth Syn Pad
92 Choir Syn Pad
93 Bowed Syn Pad
94 Metal Syn Pad
95 Halo Syn Pad
96 Sweep Syn Pad

Synth Effects
97 SFX Rain
98 SFX Soundtrack
99 SFX Crystal
100 SFX Atmosphere
101 SFX Brightness
102 SFX Goblins
103 SFX Echoes
104 SFX Sci-fi

Ethnic
105 Sitar
106 Banjo
107 Shamisen
108 Koto
109 Kalimba
110 Bag Pipe
111 Fiddle
112 Shanai

Percussive
113 Tinkle Bell
114 Agogo
115 Steel Drums
116 Woodblock
117 Taiko Drum
118 Melodic Tom
119 Syn Drum
120 Reverse Cymbal

Sound Effects
121 Guitar Fret Noise
122 Breath Noise
123 Seashore
124 Bird Tweet
125 Telephone Ring
126 Helicopter
127 Applause
128 Gun Shot

General MIDI Drums note number
(MIDI channel 10 is used for drums)

35 Acoustic Bass Drum
36 Bass Drum 1
37 Side Stick
38 Acoustic Snare
39 Hand Clap
40 ElectricSnare
41 Low Floor Tom
42 Closed Hi Hat
43 High Floor Tom
44 Pedal Hi Hat
45 Low Tom
46 Open Hi Hat
47 Low Mid Tom
48 High Mid Tom
49 Crash Cymbal 1
50 High Tom
51 Ride Cymbal 1
52 Chinese Cymbal
53 Ride Bell
54 Tambourine
55 Splash Cymbal
56 Cowbell
57 Crash Cymbal 2
58 Vibraslap
59 Ride Cymbal 2
60 HighBongo
61 Low Bongo
62 Mute High Conga
63 Open High Conga
64 Low Conga
65 High Timbale
66 Low Timbale
67 High Agogo
68 Low Agogo
69 Cabasa
70 Maracas
71 Short Whistle
72 Long Whistle
73 Short Guiro
74 Long Guiro
75 Claves

76 High Wood Block
77 Low Wood Block
78 Mute Cuica
79 Open Cuica
80 Mute Triangle
81 Open Triangle

A MIDI synthesizer could not be limited to GM having extra features; on the other hand, even accomplishing according to GM, different synthesizers can produce sounds with more or less accuracy and quality depending on the techniques employed and also on the hole sound system, since the digital samples are sent to a DAC (Digital to Analog Converter) and, eventually to the speakers.

If you want to get more in depth on MIDI, you should consult:

MIDI 1.0 Detailed Specification

Another excellent source of information is the book:

Curtis Roads: The Computer Music Tutorial, MIT Press

3 Tools

For software programming, in most cases we can make an implementation using different tools and, finally, achieve the same goal. In this concrete implementation we, basically, decided to use just the following ones:

- A programming language.
- A compiler.
- A tool to automatically build the code.

Programming Language

The C programming language was the one choosed. The main reason is that it is a really effective and efficient high level programming language, which we think is closer to the machine than others thanks to his compiler which is the machine code generator. This programming language and its compiler were created by Dennis Ritchie who rewrote the Unix operating system in C.

We make intensive use of pointers, structures, bitwise operators among other features that make this programming language powerful. On the other hand, it is a general purpose language useful for any task and, we also have to take in account that it is the preferred language for operating system implementations, device driver programming and applications which need a good performance an also a quick response.

Compiler

Since we decided to make this implementation in C, obviously, what we need is a C compiler. An open source compiler like MinGW (Minimalist GNU for Windows) seems a good choice.

If you do not have this compiler already installed in your system, you can download it for free from its web site; It comes with all the needed headers and libraries to write and generate native microsoft windows code. In addition, you can also install a command window shell and, from there, execute linux like commands and shell scripts.

As we already mention, the installer can be downloaded from the web; there are 2 types of installer: a graphical interface installer and a command line installer. As you may know, this is the GNU compilers suite (gcc, g++ ...); in our case, we only need the C compiler (gcc) but, we also installed the g++ compiler. For the installation, we used the command line installer (mingw-get), but you can used the graphical interface one if you wish.

Following, we are going to describe briefly the steps to install MinGW in your system using the command line installer.

First, go to the MinGW web site and download the compressed file containing the mingw-get installer (in our case it was mingw-get-0.6.3-mingw32-pre-20170905-1-bin.zip). Create a new folder in the 1st level of directories (c:\ ), and name it ‘MinGW’, put the downloaded zip file inside it and then, unzip the file in

that folder; execute the command ‘cd c:\MinGW\bin’; now you are in the folder where the mingw-get installer is.

To install the tools needed, execute the following command:

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

This will download and install the C and C++ GNU compilers and debugger, the posix threads package, the make utility to automate compilation and the MinGW Shell in your system.

In addition, you can download and install other compilers and tools; to know their names to get them, please look in the MinGW web site.

Finally, you must to add to your PATH environment variable the following:

```
;c:\MinGW\bin;c:\MinGW\msys\1.0\bin
```

That will make visible the executable commands from a shell, for testing it, open a command line window and type and execute: gcc –version and look at the output.

If it was a success, look inside the c:\MinGW folder; all the files neded should be there. Look in the msys\1.0 folder; there is an archive named ‘msys.bat’, you can create a direct link to this file in your desktop to rapidly open a linux like command shell.

Code Builder

Even in the case of small projects like this one, it is a good practice to use a tool to automatically manage code compilation. The GNU make utility is an excellent choice for our purpose. The most common language for make users is the C programming language, but it can be used with other languages having a compiler which can be run from a command shell.

To prepare the make utility you must write a file called Makefile, you can name this file GNUmakefile, makefile or Makefile; we recommend ‘Makefile’ which is the commonly used name; if you name it ‘GNUmakefile’ then, it will be valid just for GNU make but not for other make utilities.

In a Makefile, you can define variables and functions, call another makefile, etc.

A Makefile has rules, prerequisites and recipes; there are some automatic variables (built in variables), for example, if we have the followind rule and recipe:

```
$(PROG): $(OBJ)
	$(CC) $^ -o $@
```

Then $^ takes the value of the variable OBJ which are the prerequisites, and $@ takes the value of the target stored in the PROG variable. Another widely used built in variable is $< which points to the first prerequisite in a rule, for example:

```
bar.o: bar.c bar.h
	$(CC) -c $<
```

will compile the bar.c file to generate the bar.o object file; in this last example, the prerequisites of the rule are: bar.c bar.h and the recipe is $(CC) –c $<; to know what is a recipe, a tab character must be type at the beginning of the line. Notice that the value of a variable X is retrieved in the form: $(X).

Now, we are going to present a simple Makefile to illustrate this concepts as a little introduction.

```
# this is a comment
# macro to perform reordering
define reorder
	mv fb bin
endef
# declare some variables
CC = gcc
PROG = fb
OBJ = main.o foo.o var.o
$(PROG): $(OBJ)
	$(CC) $^ -o $@ && $(reorder)
main.o: main.c foo.h bar.h
	$(CC) -c $<
foo.o: foo.c foo.h var.h
	$(CC) -c $<
bar.o: bar.c bar.h
	$(CC) -c $<
clean:
	rm *.o
```

We build the target with the command:

```
make
```

we suppose there is a folder named bin to move the target to it; the last line is for delete all the generated object files and is executed with the command:

```
make clean
```

notice that we can also use wildcards like '*' as we do in the command shell.

A more detailed description of the make utility is beyond the scope of this article, but we recommend some readings below.

To learn about C, you can consult the book:
Kernighan, Ritchie: The C Programming Language 2nd ed, Prentice Hall

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/

4 Design and Implementation

Maybe the best way to solve a problem is to decompose it into a small ones, then, solve and test each subproblem independently, and put all of them to work together to solve the hole problem. That is what we are going to do, constructing incrementally the code of the entire library in this project, so we have to define a set of data types suitable to manage the data flow.

Data Types

Since the comunication in the MIDI protocol is achieved through messages, what we need first is a MIDI message data type. Since the C programming language has not a byte atomic type, we decided to define one just for the sake of clarity in the code:

```
typedef unsigned char byte;
```

On the other hand, we know that a MIDI message has 3 bytes as much, what we can represent as an array of bytes.

The multimedia API offered by the Microsoft Windows system requires sending the message as an unsigned long quantity, so we need a structure allowing this issue; the solution is to have one which could be a byte array and an unsigned long at the same time; a union is the perfect one for this purpose.

```
typedef union
{
unsigned long _dispatch; /* packed message */
byte _data[4]; /* message */
}MIDIMessage;
```

We said before that a MIDI message has 3 bytes as much, but since we have to send the message as a 32 bit quantity, we need a 4 byte array. For example, suppose you want to send a program change message to channel 1 with program number 2, then it would be as follows:

```
MIDIMessage* msg = ...
msg->_data[0] = 0xc0; /* program change - channel 1 */
msg->_data[1] = 0x01; /* program number 2 */
```

```
msg->_data[2] = 0x00; /* this message has 2 bytes */
        msg->_data[3] = 0x00; /* not used - always set to zero */
```

Then, the _dispatch field takes the value:

0x000001c0 hexadecimal (448 decimal).

Now, we need some functions to manage data. Functions to create, clone and destroy a message will be enough, here there are their prototypes:

```
MIDIMessage* create_midi_message(byte status, byte data1, byte data2);
       MIDIMessage* clone_midi_message(const MIDIMessage* msg);
             void destroy_midi_message(MIDIMessage* msg);
```

In addition, we define some macros to access data an to dispatch a message, you can do that directly but we think that in this way the code is clearer; anyway, once the macros are expanded by the C preprocessor, the resulting code will be the same.

```
#define status_msg(msg) ((msg)->_data[0])
 #define data1_msg(msg) ((msg)->_data[1])
 #define data2_msg(msg) ((msg)->_data[2])
#define dispatch_msg(msg) ((msg)->_dispatch)
```

Now that we already have a message data type, we can continue our incremental development. The next one we are looking for is an event data type. We know that a MIDI event has a message and a timestamp, and that an event can be also a meta event or a system exclusive event, and that in case of meta or sysex the event has an extra buffer of bytes. We also know that a track is a list of events. So, we have to define a MIDI event as a node which has inside it a pointer to the next node in the track.

```
                typedef struct _event
                        {
                  MIDIMessage* _msg;
          unsigned long _ticks; /* timestamp */

                     int _is_meta;
                    int _is_sysex;
      int _data_size; /* size for the extra data buffer */
            byte* _data; /* buffer of extra bytes */

        struct _event* _next; /* pointer to next node */
                          }MIDIEvent;
```

The name of the fields in this structure are illustrative of their purpose. As we did for the message data type, we need some functions to manage the MIDIEvent data type. In this case, there are more functions and macros, so we are going to present you the prototypes grouped in some way. The functions to create an event are as follows:

```
MIDIEvent* create_midi_event(MIDIMessage* msg, unsigned long t);
                MIDIEvent* create_midi_meta_event
       (MIDIMessage* msg, unsigned long t, int data_size);
                MIDIEvent* create_midi_sysex_event
       (MIDIMessage* msg, unsigned long t, int data_size);
```

The prototype to create what we can call a ‘short event’ is the first one. It has a MIDIMessage and an unsigned long; we use that function to create an event with a channel voice message and a timestamp. The next two are to create a meta or a sysex event and, as we can see, they have an extra parameter to know the size of the extra data

buffer. Remember that you can look at the source code to know the implementation details. The functions to clone and destroy a MIDIEvent are self-explanatory, here there are their prototypes:

```
MIDIEvent* clone_midi_event(const MIDIEvent* event);
     void destroy_midi_event(MIDIEvent* event);
```

In addition, we declare a function to know if a concrete event is a short event and another one to check for tempo events:

```
int is_short_event(const MIDIEvent* event);
int is_tempo_event(const MIDIEvent* event);
```

Later, in the playback section of this chapter we will explain in more detail the purpose of these two functions.

The macros defined to manage this data type are also self-explanatory, here they are:

```
#define ticks_event(event) ((event)->_ticks)
#define msg_event(event) ((event)->_msg)
#define next_event(event) ((event)->_next)
#define is_meta_event(event) ((event)->_is_meta == 1)
#define is_sysex_event(event) ((event)->_is_sysex == 1)
#define data_size_event(event) ((event)->_data_size)
```

A track is, in our case, an event list. The MIDIEvent data type has been defined as an abstraction of a node, which has inside it a pointer to the next node in a list. So, we are ready to write the code for the MIDITrack data type definition.

```
typedef struct
        {
     int _items;
int _name_length;
   char* _name;
MIDIEvent* _head;
MIDIEvent* _tail;
   }MIDITrack;
```

Since a track, as we said before, is a list of events, we declare the head and the tail each one as a MIDIEvent node. A track can have a name, declared as a pointer to char, we also added an integer fiel to store the length of the name and another integer to count the items in a track. The function prototypes to create, clone and destroy a MIDITrack are as follows:

```
           MIDITrack* create_midi_track();
MIDITrack* clone_midi_track(const MIDITrack* track);
     void destroy_midi_track(MIDITrack* track);
```

Function prototypes to add an event to a track and for chaeck if a concrete track has tempo events ar:

```
void add_midi_event(MIDITrack* track, MIDIEvent* event);
     int has_tempo_events(const MIDITrack* track);
```

The macros defined to access data in a MIDITrack structure are the following ones:

```
#define head_track(track) ((track)->_head)
#define is_empty_track(track) ((track)->_items == 0)
```

The first macro is to get the head node (first node of the track), and the other one tell us if a concrete track is empty.

The last data type that we need is a MIDISong. A song is composed by a list of tracks; in our case, we implemented it as an array of pointers to tracks:

```
typedef struct
        {
int _ticksperq;
 int _ntracks;
MIDITrack** _tracks;
}MIDISong;
```

The field _ticksperq (ticks per quarter note) refers to what we can also call ‘division’, and the _ntracks field, obviously, is the number of tracks in the song; remember that if the sequence has only one track its format will be 0 and if it has multiple tracks its format will be 1.

The function prototypes to create, clone and destroy a MIDISong are as follows:

```
MIDISong* create_midi_song(int ticksperq, int ntracks);
MIDISong* clone_midi_song(const MIDISong* song);
void destroy_midi_song(MIDISong* song);
```

And here you are the macros defined to access data in this structure:

```
#define song_ticksperq(song) ((song)->_ticksperq)
#define song_tracks(song) ((song)->_ntracks)
```

As you may have guessed, to access a concrete track in a song, it is done as follows:

```
int i;
MIDISong* s = create_midi_song( ... );
For(i = 0; ... )
s->_tracks[i] = ...
```

The set of fundamental data types for our work is already completed. Now, we are ready to use them in the required algorithms. We are going to start with the implementation to load and store SMF.

Data Storage

This section is devoted on how to read and write standard midi files. In the chapter 2 we presented the structure of SMF in the related section; here, we describe in a like pseudo-code way how to implement functions using our set of data types in order to load and store MIDI files. The function prototypes are so intuitive:

```
MIDISong* read_midi_file(const char* filename);
int write_midi_file(const char* filename, const MIDISong* song);
```

These functions deserve an explanation about the algorithms employed. Although in the header file there are only declared these two functions, in the implementation one a number of static variables and static helper functions needed to achieve our purpose are implemented; that is, we decomposed the problem into small ones and, once all of them were correct, we put them to work together to solve the hole algorithm implementation. The static helper functions that we have implemented in the midi_file.c are used to check for correct headers and, in the case of read a file they are to read a byte, read a 32-bit value, read a 16-bit value and to read variable length quantities. Following, we describe in pseudo-code how to read a SMF.

Read a Standard MIDI File:
Open the requested file to read bytes
Check if it begins with a SMF header (“MThd”), if not, return NULL
Read a 32-bit quantity, if it is not equal to 6 return NULL
Read a 16-bit value; this value is the format of the sequence, it must b 0 or 1, otherwise return NULL
Read a 16-bit value, this value tell us the number of tracks in the file.
Read a 16-bit value, this value is the division (ticks per quarter note).
Create a midi song: song = create_midi_song(division, ntracks)
Loop to traverse track after track:
for(i = 0; i < ntracks; i++)
{
Check for track header (“MTrk”), if not, return NULL
Read a 32-bit value, this is the length in bytes of the current track
bytes read = 0
Create a track: track = create_midi_track()
while(bytes read != number of bytes in the track)
{
Read timestamp as a variable-length quantity
Read a byte; each time you read a byte the number of bytes read is incremented
Test byte read to know the type of event rlated to timestamp, if it is a meta or sysex event, read the length of the extra buffer of bytes as a variable-length quantity. Check also if it is a short event, that is, an event with a channel voice message and a timestamp and create the needed event and add it to the current track.
Add_midi_event(track, event)
}
Add track to song:
Song->_tracks[i] = track
}
Close the opened file.
return song

The pseudo-code for write a SMF is similar, we also have static variables and helper functions available. A Function to write the file header, write a 32-bit value, write a 16-bit value and to write variable-length quantities. To write the entire file, we use a const MIDIEvent data type to iterate each track and store the bytes according to the data stored in a MIDISong structure.

Write a Standard MIDI file:
Open the file to write bytes.

If cannot open the file return 0
Write the header file.
We defined a 32-bit symbolic MThd constant:
#define MThd 0x4d546864
To write the file header, we first write the MThd constant as a 32-bit value, we also write the value 6 as a 32-bit quantity, then we write the format (0 if the sequence has only 1 track and 1 if it has multiple tracks), number of tracks and division as a 16-bit value.
We defined a 32-bit symbolic constant equivalent to 'M''T''r''k'
#define MTrk 0x4d54726b
This is the track id to be written as a 32-bit quantity
Loop track after track:
For(i = 0; i < ntracks; i++)
{
write track header
offset = save position in the file (we do not know the length in bytes of the track yet)
write 0 as a 32-bit quantity (afterwards, we return to this position to write the track length in bytes)
bytes written = 0 (when you write a byte this counter is incremented)
pick the first event in the current track
while(event != NULL)
{
write the event timestamp in ticks as a variable-length quantity
write the current event (short, meta ...)
pick the next event in the current track
}
write the end of track
end = save file position
go to offset position
write the length in bytes of the track as a 32-bit quantity
turn to end position
}
close the opened file
return 1

The implementation of the data types needed and the support for storage are already done. Now that we can read a MIDI file and store the data in a song structure, we can implement a player in order to reproduce the loaded sequence.

Playback

First of all, what we are going to do is to prepare the data previously loaded and stored in a MIDISong data type. What we need is to merge all the tracks in the song into a single track; for this purpose, we implemented a function whose prototype is as follows:

```
MIDITrack* merge_midi_tracks(const MIDISong* song);
```

This function merges all events in each track in the song ordered by timestamp. To do that, not all of the events are needed just to play a song; actually, we are only interested in short events (channel voice message and timestamp), and tempo events. To know if a concrete event is a short event or a tempo event we remember that we have available the folloeing functions:

```
int is_short_event(const MIDIEvent* event);
int is_tempo_event(const MIDIEvent* event);
```

This functions return 0 if the event is not a short or tempo event respectively and 1 if afirmative. To do the task, we iterate all the tracks and build a single track with all the events (short and tempo), ordered by timestamp; in case of several events having the same timestamp, and among them were a tempo event, the tempo event is placed first in the group.

We also need some functions to manage the tempo and if necessary, comvert ticks to milliseconds. These functions are defined in the midi_tempo.h header:

```
unsigned long get_tempo_mu(const MIDIEvent* meta);
unsigned long ticks2milliseconds
(unsigned long ticks, int division, unsigned long tempo);
```

The signatures for these functions are so intuitive about what they do. The first one computes the time in microseconds for a concrete tempo event; if the event passed as parameter is not a tempo event it returns 0. The other one convers delta time in ticks to milliseconds. These functions are used to syncronize time while playback.

In the rest of this section, we are going to describe an implementation for a player capable of reproduce MIDI sequences. In the midi_player.h header, there are defined the function prototypes:

```
char** get_midi_out_devs(int* numdevs);
```

This function returns by reference the number of MIDI out devices in your system, and also a char array of strings with a description for each one.

```
void set_midi_out_port(int port);
```

This function allow us to set the wanted MIDI out port; take in account that the port ID must be in the range 0 .. (num devs – 1).

```
int load_midi(const char* filename);
```

This function loads a MIDI file. It returns 1 if success or 0 if failed. Internally, it uses the read_midi_file function to store the data in a song structure; afterwards, the tracks in the song are merged into a single track using the merge_midi_tracks function.

If all was right, we should have a multichannel track ready to be played. All the data types and helper functions are declared in the midi_player.c implementation file.

The helper function to reproduce the loaded sequence has the prototype:

```
static void play_midi_track
(const MIDITrack*track, int ticksperq);
```

As you can see, the first parameter is the merged track to be played, and the second one is the ticks per quarter note, that is, the division, which is needed to

compute the conversion to delta time in ticks to milliseconds. So, a pseudo-code for this function could be:

Open the MIDI out device.
Take the first event in the track.
Since that event should has timestamp in ticks = 0, then send all events where ticks = 0 to the output device; in case of tempo event, store the tempo in microseconds in a variable, for instance, tempo = get_tempo_mu(e).
Now, send the rest of messages to the output device taking in account the delta time in ticks after its conversion to milliseconds applying the function ticks2milliseconds ; as above, if there are any tempo event, store it in a variable; remember that the tempo and division are needed to time conversion:

```
unsigned long ticks2milliseconds
(unsigned long ticks, int division, unsigned long tempo);
```

When the playing loop is completed or aborted by stopping it, we close the opened output device and the executing function is finished.
On the other hand, we must launch the player as an independent thread in order to keep the system responsive. To do that, we use the pthreas package previously installed with the mingw-get command line installer for MinGW. If you do not have it already installed yet, go to the folder where the mingw-get installer is, for example, cd c:\MinGW\bin and type and execute the command:

```
mingw-get install pthreads-w32
```

To prepare the system to play, we first define a MIDI player type:

```
typedef struct
{
void (*_play_midi)();
}MIDIPlayer;
```

The defined player type has only a field, that is, a pointer to a function.
To init an instance of a player, we declared the following helper function:

```
static void init_midi_player(MIDIPlayer* p, void (*func)())
{
p->_play_midi = func;
}
```

To create and launch the thread, we first init the player with a thread_code function, which has inside it a call to the already presented one to play our multichannel track.
We also define a static function to pass as parameter to the one which actually launches the thread:

```
static void* splay_midi_thread_safe(void* ptr)
{
((MIDIPlayer*)ptr)->_play_midi();
return NULL;
}
```

The following static variables are declared in the midi_player.c implementation file:

```
static pthread_t _pt;
static MIDIPlayer _player;
```

to init the player, we call the init function with the thread_code one as parameter:

```
init_midi_player(&_player, thread_code);
```

So, the play_midi function declared in the header file (midi_player.h):

```
int play_midi();
```

Has the body:

```
int play_midi()
{
      if(_is_playing) return 1;
      return pthread_create
(&_pt, NULL, splay_midi_thread_safe, &_player);
}
```

The play_midi() function returns 0 if success and 1 otherwise. We put the code for these functions here, because it is not too large, and in order to clarify the concepts behind them.

The last declared function prototype in this header is:

```
void stop_midi();
```

As its signature tells, a call to this function stops playback immediately. With this last function, we finish this chapter devoted to explain the design and implementation for our work. We hope that with the concepts presented and the source code, which you should be available, you will not have difficult to properly understand how all things described were done.

5 Conclusions

We successfully implemented a MIDI subsystem capable of load and store SMF, and a player suitable to reproduce any MIDI file accomplishing with SMF.

The code was divided into several archives, each one with a header for its interface and an implementation file; the purpose of the functions in a concrete header are all related to the same subject, so, we isolate each implementation in order to avoid coupling.

So, you can add extra features without changing the existing code. Although this implementation works in Microsoft Windows OS, the only code depending on that is the one in the player implementation, that is, in the midi_player.c file; you can modify that, for example, with conditional compilation to add support for another operating system; that means that all the rest of the code compiles and works in any system with a C compiler, since uses just standard libraries and pure C code.

Each header file has comments explaining what does every function and macro declared in it. The code for the implementation is not difficult to understand for anyone who knows something about the C programming language.

As we already say, we build all the code in a shared library using the MinGW (Minimalist GNU for Windows) compiler, but you can change the building process to work in another compiler under MS-Windows.

We expect that this article can be useful to an interested reader who wants to know about the MIDI protocol.

6 Appendix

In this section, we furst explain what you will find in the ‘cwmidi’ package. The root folder has the same name as the package (cwmidi), inside, there are the following directories and files:

```
< example > < include >
< lib >     < src >
Makefile    readme.txt
```

Maybe the shared library ‘cwmidi.dll’ is already inside the lib folder in this deployment, if not, just type and execute the command:

```
make
```

This will compile the dll and move it to the lib folder. Now that we have that library built, we can compile and execute the example; to do that, go to the example folder: cd example; you will see another Makefile there, so, type and execute the make command to build the example application command line, the executable file will be moved to the bin folder; then you have to go to there: cd bin. To run the program, you must execute the run.sh script if you are working in the MinGW Shell, or the run.bat one if you are under a cmd Windows command line shell. The interface for this command line application is as follos:

```
1 - show menu
2 - select midi out device
3 - load midi
4 - play midi
5 - stop midi
6 - exit

option:
```

You have just to type an option (1..6) to use this program; the interface is so intuitive that we do not need to tell how it works.

In the bin folder there are two midi songs, the one named ‘mysong.mid’ was composed by the author years ago by using a known sequencer (Cake Walk). It was made writing musical notes directly to the staff; the other one is ‘myway.mid’ what anybody should know, and it was downloaded from the web.

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.