MIDI player

Davide Botti,Cristiano Di Marco

15/03/2016

Contents

1 Introduction

The goal of this project is to implement a simple MIDI file player inside the board STM32f4 discovery. The board is equipped with a common DAC by which the music can be reproduced, a 192KB RAM memory,1MB flash memory (non-volatile memory), in which we loaded the code and other useful files (more details in next sections) and with an ARM Cortex M4 processor; so it's quite a powerful board. We use the operating system Miosix as it provides functionalities such as primitives to read/write to the serial line, and to send data to the DAC peripheral.

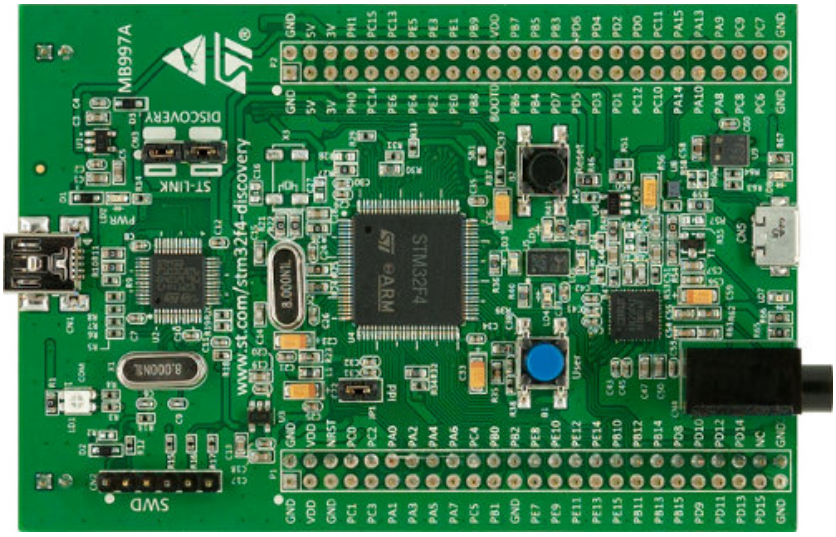

Figure 1: STM32f4

In order to implement what is written further, you need a board, a program to send binaries into the board (like STLink), and a patched C compiler in order to compile for the ARM Cortex M4 (more details in the 5th section).

2 MIDI structure

One important thing (actually the most important) in order to implement a MIDI player, was to understand the basic structure of a MIDI file: the file indeed has a fixed structure that divides it in three subsections:

header section The header chunk consists of a literal string denoting the header, a length indicator, the format of the MIDI file, the number of tracks in the file, and a timing value specifying delta time units. This is an introductory section, so apart from the information about the time, can be skipped by our program.

body section the body section consists of a series of MIDI events, such as the reproduction of a note, a pause, a change in time or a change of the instrument (so called meta-events). Actually this is the most important part, because here we can find all the notes that create the music.

end section this section simply indicates the end of the MIDI file, and is always composed of 4 bytes: *00 FF 2F 00*.

As you can easily understand, the most important section is the body, where we can actually find the notes that we're going to play. In practice, a MIDI file is a sequence of **MIDI messages** that represent the real music;these messages will be sent by the MIDI reproducer to our board that will take care of playing notes. So our program should listen to these messages coming from the serial line and select the correct note to play.
There are two broad message categories:

system message system messages are non-MIDI data of various sorts consisting of a fixed prefix, type indicator, a length field, and actual event data. For example, this messages indicate what type of instrument is playing, or a change in the rhythm.

channel message consist of a delta time since the last message, and three bytes that can vary according to the specific type of message. For example, these bytes can represent the reproduction of a note, a pause, a control change ect...

In our project we focus our attention on channel-messages, that represent the real music, in particular on the **event 9x** (where x stands for any 4 bits), that indicates the reproduction of a note.

2.1 MIDI messages

Here we would like to underline some MIDI messages and explain a bit their structure. These messages are very important because, as already said, they represent the music; so in order to correctly play the notes a deep understanding of these messages is needed.
Because of the great heterogeneity of MIDI message types, we had to make some assumptions and analyze only a subset of these,actually the most important ones. Every message is composed of three bytes, that indicates respectively the event type, the note and the velocity. In particular, the note is simply a number that encodes the information of the pitch (for example, a C2 is the number 48,C3 is number 60 and so on..) and the velocity represents how fast the note is pressed. Every message is preceded by a timestamp that indicates the delta time from the previous event.
Eventually, a complete channel-message is composed of:
delta-time,event-id,note-number,velocity
Here we provide some more details about these fields:

delta-time it is the relative time from the current time and indicates when the message starts.In particular, if the first byte cannot represent the time (because it is too short), the MIDI standard adds another byte to the first, and so on if also the second byte cannot represent the time interval.

message-id type of message; as already mentioned, an event-id of value 9x represents the reproduction of a note, instead message 8x the stop of a note.

note-number the pitch of the note: these numbers starts from 0 (C-2) and linearly increase until 127 (G8). In this way,every note is uniquely identified by a number; this makes easy the recognition of the note.

velocity a measure of how rapidly and forcefully a key on a keyboard is pressed.

Here we report the messages which we should take care of; the useful messages are only two: the messages that indicate the reproduction of the note, and the messages that indicate the pause of a note.

message 9x as already mentioned, this message represents the reproduction of a note, or a pause if the velocity is equal to zero. The latter 4 bits represent the channel of the note.

message 8x this message represents instead the end of a note. The latter 4 bits represent the channel of the note.

3 Algorithm

The algorithm (all written in C++) implements a simple producer-consumer pattern:there is a main thread that reads from the serial port each byte and sets the note to be played. Another thread takes care of playing the current note (previously set by the main thread).In practice, this is a producer-consumer pattern in which the producer reads bytes from the serial port and the consumer continuously play notes, reading from a one-byte buffer.

3.1 Preliminary steps

First we have to understand how to transfer bytes: we send it via the serial port of the board, but this requires some preliminary steps, in particular:

- open a MIDI port (this passage seems to be done only on Windows).
- open the MIDI file with a program that allows you to send it via a specific port; not all the programs allow this, for example by *MidiOx* you can send it via different output ports.
- connect the MIDI port to the serial port:we did this by *hairless-MIDI-serial*,a simple software that allows to send a data stream over the serial port.
- sample some notes and convert it in order to have a C-style header file which contains a kind of waveform of the notes (more details in 3.2).

Here there is a simple block diagram that shows how the communication happens: the MIDI player in our computer starts to reproduce the song sending midi messages to the midi port; then these messages are forwarded to the serial port thanks to the Hairless-midi-serial program. From the serial port are finally transferred to our board,that reads the bytes and plays notes.

Figure 2: MIDI trasmission block diagram

Following the previous steps, we managed to transfer the bytes through the serial line and to obtain different header files,stored into the board, that contain an array of bytes that represents the waveform of a note. Now we are ready to play music thanks to our algorithm.

3.2 Reading from the serial

The main thread reads from the serial port and waits for an event 0x9x (it is a kind of parser). When that specific byte is sent, reads the note, the velocity and stores this information in a buffer. The producer algorithm is reported below:

```
void parse_byte(char c) {
        char note,velocity;
        //note reproduction
        if((c & 0b11110000)==0x90) {
                note=getchar();
                velocity=getchar();
                //atomic access to the shared variable
                pthread_mutex_lock(&mutex);
                if(velocity==0) current_note=0;
                else current_note=note;
                pthread_mutex_unlock(&mutex);
        }
        //note pause
        else if((c & 0b11110000)==0x80) current_note=0;
}
```

The function parse_byte is called from the main function and takes as input a single byte (the first byte, as already said, represents the message type): if this byte is 0x90 (or 91,92,..,9F), the program fetches the pitch and the

velocity of the next event and locks the mutex in order to set the current note to be played. Instead, if it's a 0x80 (or 81,82,..,8F), the program locks the mutex, set the note to be played to zero (a pause) and unlocks the mutex.Please note that the algorithm does not care about the channel of the note: in fact the following lines:

```
if((c & 0b11110000)==0x90)
...
else if((c & 0b11110000)==0x80)
```

mask the byte in order to discard the latter 4 bits.
Finally, the main thread set the shared variable current_note and unlocks the mutex so the consumer can access the buffer.

3.3 Playing notes

This is the core of the project.In order to play notes, knowing the pitch of the note, we stored in the flash memory of the board arrays that correspond to sampled notes produced with Audacity.These vectors are produced by a program, *convert.exe*, that takes as input a .Wav file and returns a .h header C-style file, that contains the bytes of the notes.Then, we pass the array contained into these headers to a costructor that creates a custom object (*ADPCMSound*, can be found in Miosix).This object represents the "sound" to be played.
The player continuously checks which note it has to play (by a simple switch() statement inside an infinite for loop) and sends the corresponding bytes to the output DAC. Here is the code of our player:

```
void* play_sound(void* argv) {
        char note;
        char previous_note=0;
                for(;;) {
                        pthread_mutex_lock(&mutex);
                        //simple state machine
                        if(current_note==previous_note) note=0;
                        else note=current_note;
                        previous_note=current_note;
                        pthread_mutex_unlock(&mutex);
                        switch(note) {
                                case 0:
                                        Player::instance()
                                        .play(pauseSemicroma_sound);
                                        break;
                                case 63:
                                        Player::instance()
                                        .play(mib3Semicroma_sound);
                                        break;
                                case 66:
                                        Player::instance()
                                        .play(solb3Semicroma_sound);
                                        break;
                                //other switch cases
                                ...
                        }
                }
}
```

The player continuously plays notes, reading from the shared buffer in a secure manner (thanks to the mutex). The consumer does not wait for the producer to store another note in the buffer, but plays the note once and then waits that the note is changed; in this way the rhythm of the song can be managed easily: when the current_note is set, the note is played because the consumer enters in the for loop. But the second time the consumer enters in the loop, checks the current_note variable and, if it's not changed, it sets the note to be played to 0. In this way the note is played once, and the board gives a greater and more accurate timing.

In order to pass the bytes to the peripheral, our player uses a custom Miosix class, *Player*, that manages the transmission between our program and the DAC peripheral taking as input a custom *ADPCMSound* object that contains the bytes of the note. Eventually, we have added only a subset of all the possible notes (the notes used to test the board and to show the final result); a more complete player would require all the possible notes, but this is simply a matter of storing in the board the array that represents the missing notes.

3.4 Known issues

Here we report some issues that are present in our program; these issues are mainly due to the academic purpose of the project. That is, our goal was not to implement a real MIDI player but only to deal with problems studied only theoretically (such as producer-consumer pattern).

instrument type The player does not recognize the playing instrument: so all the notes have the same timbre (this can be easily fixed by checking the channel of the note, instead of masking it).

note duration The player plays a note that is very short, and then waits for the next. In order to play longer notes is necessary a modification to the code. Moreover, because of the notes have a fixed duration, very fast songs cannot be reproduced correctly. In practice we have a kind of "bottleneck" due to the finite duration of the notes.

song speed Because of the fixed length of our sampled notes, the player can reproduce only songs that have a minimum note length equal to our notes. In practice there is an upper limit due to the fixed length of our notes.

3.5 About the code

The entire code can be found in our github repository: `https://github.com/dade92/es_project`

4 Conclusions and further extensions

Using Miosix libraries, the algorithm can get bytes from the serial line and reproduce MIDI songs implementing a producer-consumer pattern: one thread

gets the bytes representing notes, and the other one plays them continuously. In practice the algorithm "chooses" which notes (already stored in the flash memory) send to the DAC peripheral of the board. One improvement could be detect the different channels of the messages and play different instruments (In fact our algorithm does not take care of which channel the message is referred to).

5 Used Software

Miosix-kernel Kernel; can be found at `https://miosix.org/`

Gcc patch patch for the gcc compiler; more info at `https://miosix.org/wiki/index.php?title=Miosix_Toolchain`

Dev-c++ to write c++ code.

Loop-midi to open a MIDI port (necessary only on Windows). Can be found at `http://www.tobias-erichsen.de/software/loopmidi.html`

Midi-ox to reproduce the MIDI and send it to the MIDI port. Can be found at `http://www.midiox.com/`

Hairless-midiserial to interface the MIDI port with the serial port. Can be found at `http://projectgus.github.io/hairless-midiserial/`

QSTlink2 to program the board.

Audacity to produce the notes.

Convert.exe c++ written program to convert the .Wav files of the notes into .h header files.

Logic used to generate MIDI files.