TDT4258
Energy Efficient Computer Design
Exercise 2

Lundal, Per Thomas
perthol@stud.ntnu.no

Normann, Kristian
krinorm@stud.ntnu.no

Selvik, Andreas Løve
andrels@stud.ntnu.no

March 14, 2013

Abstract

This report presents a MIDI player implemented on an AVR32 microcontroller. This is the result of assignment 2 given in TDT4258, for which the requirements are to make a program that allows one to play different sound effects by pushing buttons on the STK1000. It is done by first parsing the MIDI files into a simpler intermediate event-based format which can be played efficiently by the STK1000.

Contents

I Introduction

Our focus for this assignment was to generate as smooth a sound as possible. As opposed to the previous assignment, this time around we did not give much thought to energy efficiency and tried instead to make sure the resulting music would be able to make use of as many tones as possible without loss of notable speed, while also minimizing background noise.

Physically, sound is longitudinal waves that propagate through a medium, often the air. Which means that we have to produce waves, and that gives us the properties of waves to work with; frequency, amplitude, waveform and interference. Frequency is the property that decides the perceived pitch of the sound and amplitude, see figure 1a, decides the loudness. The waveform, see figure **??**, has impact on multiple aspects of the sound, the most prominent being that the average amplitude is higher and thereby sound produced will be louder, it also changes sounds characteristics. When there is multiple sounds, that is, basic harmonic waves, at the same time they interfere, which gives the result of the amplitude in a given point being the sum of the amplitude in all the waves in that point.

To produce sound digitally, one has to be able to generate samples. A sample is the numerical value of the current wave amplitude in a point. In order to capture the frequency of the intended wave, the frequency of the samples have to be at least twice the maximum frequency of the sound produced. Humans are capable of perceiving sound with a frequency between 20Hz and 20kHz, so in order to produce the whole range we would need a sample rate above 40kHz. As we'll see, we have a sample rate of 23.4kHz, which means that we can produce sound with a maximum frequency of 11.7kHz.

The STK1000 board has an ABDAC: write about it here

By hooking an interrupt routine up with a clock, we were able to deliver samples to the ABDAC at a consistent rate.

A little about notes and octaves.

Talk about midi and vaguely about structure of application. During the development we got to face the limited computing power of the STK1000. Oh yes! Per-Thomas?
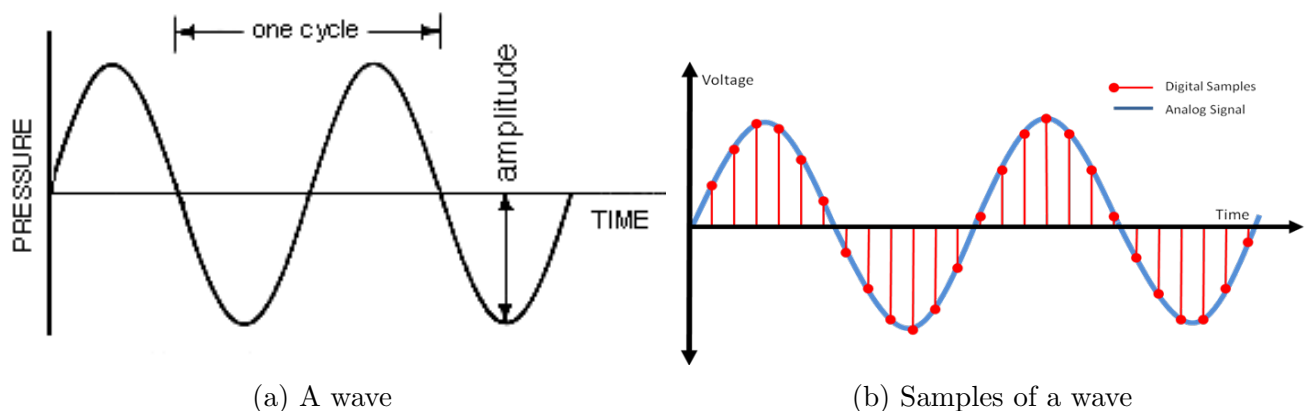

(a) A wave
(b) Samples of a wave
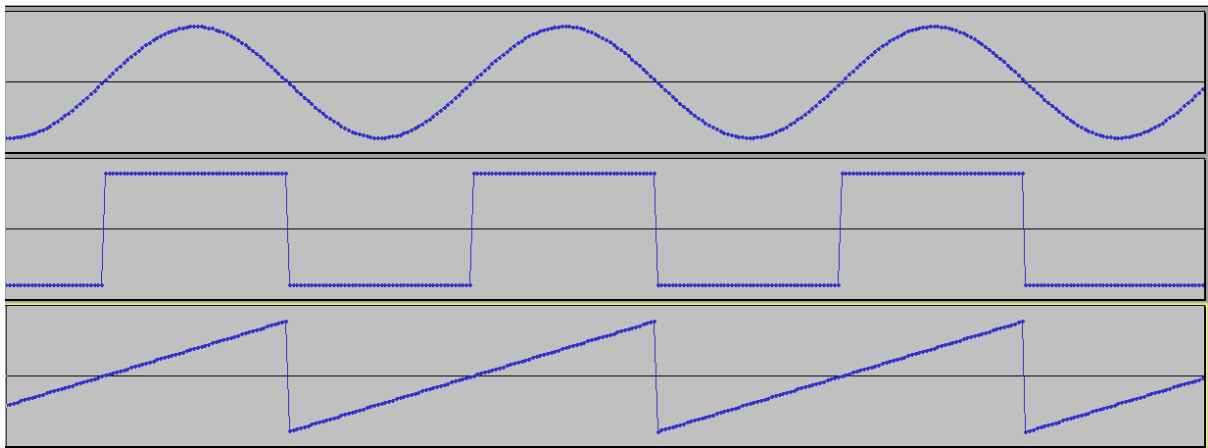
Figure 1: Sound waves

Figure 2: Different waveform in audacity

II   Description And Methodology

The first step was to read the exercise 2 chapter[ref] in the compendium[ref], where it was discovered that the purpose of the assignment was to learn how to use the features of the STK1000 while using the C programming language.

The assignment was solved in the following steps:

1. Implement the first assignment in C

2. Generate noise with the ABDAC

3. Generate tones with different wave types and frequencies

4. Play MIDI tunes

2.1   Jumper and cable configuration

- The LED's were connected to PIOC by a flat-cable from GPIO pins 16-23(J3) to the LED-pins(J15) on the STK1000 board [1, section 2.4.1].

- The buttons were connected to PIOB by flat-cable from GPIO pins 0-7(J1) to the SWITCH-pins(J25) on the STK1000 board [1, section 2.4.1].

- On the AT32AP7000 we set the jumpers SW6 and SW4 to GPIO(so that PIOC and PIOB would be connected to GPIO)[1, table 2.3].

2.2   LEDs and Buttons

The LEDs and buttons were to be operated in a similar manner as in Assignment 1. However, as it now was to be written in C, the method of accessing the registers was a bit different. The AVR32 C libraries makes these available through a number of simple structs though, so the main difference was the syntax. The code for initialization can be seen below:

```
// Initialize the buttons
register_interrupt(button_isr, AVR32_PIOB_IRQ / 32, AVR32_PIOB_IRQ % 32, BUTTONS_INT_
piob->per = 0xFF;
piob->puer = 0xFF;
piob->ier = 0xFF;

// Initialize the LEDs
pioc->per = 0xFF;
pioc->oer = 0xFF;
pioc->codr = 0xFF;
```

The interrupt routine was implemented by first debouncing, then determining what button was pressed by looking at the ISR (Interrupt Status Register) and PDSR (Pin Data Status Register), and finally updating the lights by writing to SODR (Set Output Data Register) and CODR (Clear Output Data Register).

2.3 ABDAC

Once the LEDs were shining and reacting to the buttons, the next objective was enabling the ABDAC. By following the procedure in the compendium [ref here], we started by releasing the ABDAC pins (20 and 21) from PIOB by writing 1 to its PDR (Pin Disable Registers) and followed by connecting them to Peripheral A (ABDAC) by writing to it's A Select Register (ASR). The procedure for enabling the clock was not described in the compendium, so for this we searched for an example on the internet. There we found the code file for an AVR32 MP3 player [5] which led us to the following code for using oscillator 1:

```
pm->GCCTRL[6].pllsel = 0;
pm->GCCTRL[6].oscsel = 1;
pm->GCCTRL[6].cen = 1;
```

Finally we activated the ABDAC by writing 1 to its enable bit in CR (Control Register) and set the interrupt routine to write random data to SDR (Set Data Register). This made the board output white noise, which meant we could proceed to generating some tones.

2.4 Tones

At the the beginning of the main method we initialize a set of twelve notes (an octave) corresponding to the wave type parameter given to the tones initialization function. The notes are built from this wave type and and a given, hardcoded frequency. Each note is an oscillation variable computed by multiplying the sample_amplitude with some value corresponding the the wave type. $A * f$, (where A is the amplitude, f is the wave type function).

2.5 MIDI

With the ability to play different tones and the desire to play some sweet tunes in mind, we decided to implement a system for playing MIDI files. Since the raw MIDI format is not especially well designed for playback on the board, and we would like to play as many simultaneous tones as possible, it was decided to create a simplified format. In addition, some means to pass the data to the board and access it was required. This was accomplished by designing a set of structs to hold the data:

```
typedef struct {
    int32_t delta_time;
    int8_t channel;
    int8_t tone;
    int8_t volume;
} midi_event_t;

typedef struct {
    int32_t num_events;
    midi_event_t events[];
```

```
} midi_soundtrack_t;
```

This solution uses 7 bytes of data for each event, which is slightly more than the average for MIDI. In addition, it only supports up to 128 simultaneous tones (opposed to the 4096 of MIDI), and does not take instrument types into account. But there is only one stream of events compared to the possible $2^{(16)}$ of MIDI, which makes it much easier to process for the board.

Next up was the playback system. It was needed to keep track of all playing tones and their volume. Additionally, we didn't want to create different samples for all 128 tones as this would require a lot of memory. Therefore we also needed to keep track of the pitch shift which essentially changes the playback speed. This was accomplished by creating yet another struct:

```
typedef struct {
    sample_t *sample;
    int32_t sample_point;
    int8_t pitch_up;
    int8_t pitch_down;
    int8_t volume;
} midi_channel_t;
```

The reason for both pitch_up and pitch_down instead of a single variable for pitch_shift is so that we avoid an if-test in the playback procedure to determine the shift direction, which in turn allows us to play more tones simultaneously. A single global variable is used to keep track of the current soundtrack.

Now we could focus on the main playback procedure $midi_tick(), which is to be called once for each ABDACi$

```
// Retrieve tone and pitch
sample = event.tone  % 12;
pitch = event.tone / 12;
```

After processing events, the time is incremented. Then it advances and pulls output data from all channels by calling $midi_channel_advance$ for each of them. The output is summed and then normaliz

Issues during development:

At one point during development, we noticed that with eight tracks enabled, we could play two tracks simultaneously without loss of speed and clarity of sound(i.e no particularly greater amount of crackling, or noise in the background). Any more tracks played at the same time however, would result in slower and more faded sound. After some inquiries we came upon this stackoverflow post (http://stackoverflow.com/questions/9742934/multiple-source-file-executable-slower-than-single-source-file-executable), which submitted that having multiple source files(which we do), decreased run-time performance. We then made an "all" c file that included all the c files and compiled this "all" file. We are inclined to believe that the post's claim was correct as doing this let us play a whole eight tracks without loss of speed and seemingly, clarity of sound as well, though it should be noted at this point that we observe in Audacity that the sound isn't quite as we expect[kanskje legge inn screenshot av Audacity bildet] and we wonder if this might be on account of hardware deterioration.

Tools

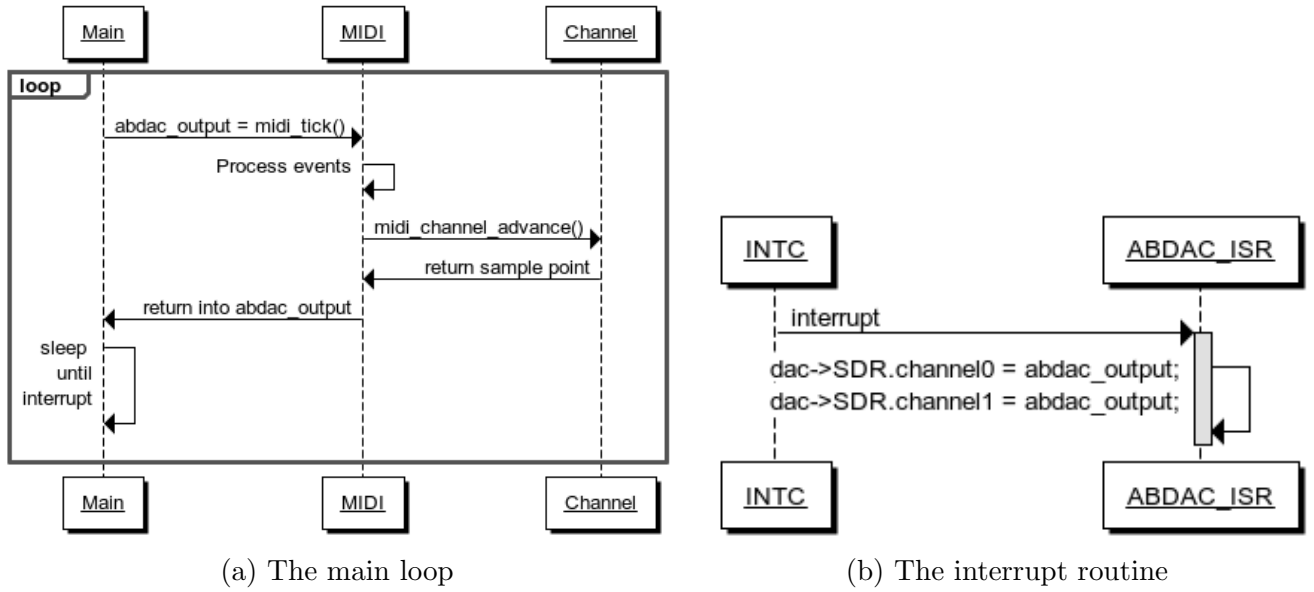(a) The main loop   (b) The interrupt routine

Figure 3: Overview of the system

- Audacity for observing and measuring sound output and waveforms

- GitHub for handling version control

- Vim as main code-editor

- Google Docs for report collaboration

- LaTeX for report markup

- JTAGICE mkII for connecting the computer to the board

- avr32gdbproxy for connecting to JTAGICE mkII

- avr32-gdb for connecting to the board, allowing us to debug the code

III   Results And Tests

We designed some tests for the final implementation, see Appendix A. The final version passed all of these tests, save one instance during development. The "Two interleaving button-clicks" test failed when including button SW0 in the test. As it turned out, this was because the 'debounce-loop' was removed in optimization by the compiler. This was fixed by making the debounce-variable volatile. After this all tests worked without fault.

IV   Evaluation Of Assignment

V Discussion

During the start-up-fase, after turning on the STK1000, just before LED0 is lit and its corresponding music starts to play, some random light-arrangement occurs based on what is present in memory from previous. This does not interfere with any expected functionality.

Problems:

A problem that has continued through the whole assignment is noise from the ABDAC. During pauses in the music, where the ABDAC is fed the value 0 repeatedly, a tone with seemingly random frequency can be heard. All attempts at correcting this in software have only masked the problem by another sound and thus failed. However, by intense searching of the internet we found that the audio output of the STK1000 might not be designed for load-heavy devices such as headphones, and that it might sound better when connected to an amplifier[6]. By connecting the output to some speakers we noticed a slight reduction in the noise level, but it was still present. This leads us to think it is a hardware issue that we can not prevent.

Another problem is that oscillator division does not seem to be working as no matter what values were set as division factor, some test samples were played at the same rate.

MIDI files do not always follow the standard

Challenge: MIDI files can theoretically play 16*128 = 2048 notes simultaneously, but our board can only play 12 at 24kHz

Oscillator division seems not to be working, therefore we just pretend it is running at targeted speed and make the ABDAC underrun (So that the abdac will only receive data every other time).

5.1 Modularity

VI   Conclusion

In the end we did not only have a system that satisfied the requirements, but a full-fledged MIDI player. This gives us a great deal of flexibility when we

References

[1] Computer Architecture and Design Group, *Lab Assignments in TDT4258 Energy Efficient Computer Systems*. Department of Computer and Information Science, NTNU, 2013, `http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf`.

[2] Atmel. *AVR32 Architecture Document*, 2011, `http://www.idi.ntnu.no/emner/tdt4258/_media/doc32000.pdf`.

[3] Atmel. *AT32AP7000 Preliminary*, 2009, `http://www.idi.ntnu.no/emner/tdt4258/_media/doc32003.pdf`.

[4] `http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=750034`

[5] Atmel. "abdac.c" *asf.atmel.com*, Web.09.03.2013. `<http://asf.atmel.com/docs/latest/avr32.applications.audio-player.mp3.evk1104/html/abdac_8c_source.html >`

[6] AVR32 *avrfreaks.net*, Web 09.03.2013, `<http://www.avrfreaks.net/index.php?name=PNphpBB2&file=printview&t=53711&start=0 >`

Bilder: `http://projeneering.com/technical_articles/images_pid_primer/sampling.png` `http://www.sfu.ca/sonic-studio/handbook/Graphics/Sine_Wave.gif`

A   Tests

Test 1:	Starting the machine
Action:	Turn on the STK1000
Preconditions:	Uploaded program on microcontroller and power cable is connected
Wanted outcome:	LED0 should light up and 'Under pressure' should play.

Test 2:	Switch song
Action:	Press button SW X
Preconditions:	LED Y lit and the music corresponding to SW Y should be playing.
Wanted outcome:	LED Y turns off and LED X turns on. Music corresponding to SW Y stops and the music corresponding to SW X starts from the beginning.

Test 3:	One click, one action
Action:	Wait two seconds and release the button.
Preconditions:	Press button SW X and hear the music start playing.
Wanted outcome:	Upon releasing the button, the music should continue and not start over.

Test 3:	Two interleaving button-clicks
Action:	First press button SW X and hold it, press press button SW Y (where $X \neq Y$), release button SW X and finally, release button SW Y.
Preconditions:	The board is turned on and some LED Z is lit with its corresponding music playing.
Wanted outcome:	Upon pressing button SW X, its corresponding music begins to play. Then, SW Y is pressed, the music changes to button SW Y's corresponding music. When button SW X is released, it changes nothing and the music continues to play, and lastly, when button SW Y is released, nothing changes and the music continues to play uninterrupted.

Test 4:	Quick click
Action:	Press button SW Y 5 times with 1-second intervals.
Preconditions:	The board is turned on and some LED X is lit with its corresponding music playing.
Wanted outcome:	LED Y lights up and stays lit upon the first pressing of the button, and the music corresponding to button SW Y starts over each time the button is pressed.