TDT4258 Assignment 2 Group 1

Sondre Lefsaker André Philipp Håvard Wormdal Høiby

March 14, 2013

1 Abstract

For this assignment we wrote a program producing distinct sounds when buttons on the microcontroller where pressed. The program was witten in C without a Operation System. We used the internal DAC to make audible audio waves.

A high-level programing language like C gives you a nice abstraction from the hardware while still maintaining direct control. Compared to assembly it is much simpler to create abstract datatypes and dealing with variables instead of registers.

Programming without an Operation System gives us full control over the hardware, but it does not provide any abstractions, like device drivers. This requires us to implement a simple drivers for each of the devices we want to use. In this assignment we use LEDs, buttons and DAC.

Contents

1 Abstract **1**

2 Introduction **3**

3 Overview of the Solution **4**
3.1 States 4
3.1.1 Modes 4
3.2 Interrupt Routines 4
3.2.1 Button Interrupt Routine 4
3.2.2 ABDAC Interrupt Routine 5
3.3 Datastructures 5
3.3.1 Note structure 5
3.3.2 Sound Tracks 6
3.3.3 Sound Samples 6
3.3.4 Sine Table 6
3.4 Waves 6

4 Solution **8**
4.1 Sound generation 8
4.2 Setting the frequency 9

5 Test Report **10**

6 Discussion **11**
6.1 Deadline 11
6.2 ABDAC interrupt routine and Sample format 11

7 Conclusion **11**

2 Introduction

The second assignment introduces us to a new hardware device, the ABDAC. We also utilize the the buttons and LEDs used in assignment one. This gives an introduction to programming with audio devices and how to produce audio digitally.

To the development environment from assignment one we add the GCC C-compiler.

The gist of this assignment is to produce different sounds when the buttons on the board are pressed. The program should be implemented in the C programming language without the support of a Operation System. We implemented two modes. The first mode is a 7-note piano. The second is a playback function which plays a different predefined sample for each button. The button SW0 is used to toggle between the modes.

In order to make the program energy efficient the CPU is set to sleep and the DAC is shut down when a tone is not playing.

3 Overview of the Solution

3.1 States

3.1.1 Modes

The program has two main modes. These are Piano and Playback. To switch between state the **SW0** is pressed.

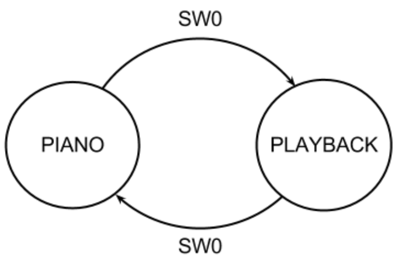


Figure 1: State diagram for modes

3.2 Interrupt Routines

The program is heavily centered around the two interrupt routines, button_isr and abdac_isr, found in **interrupt.c**.

3.2.1 Button Interrupt Routine

The button routine is responsible for switching betweens the modes shown in Figure[1]. If the mode is Piano it saves the button state for the abdac routine. If the mode is Playback it initializes the next sample to be played.

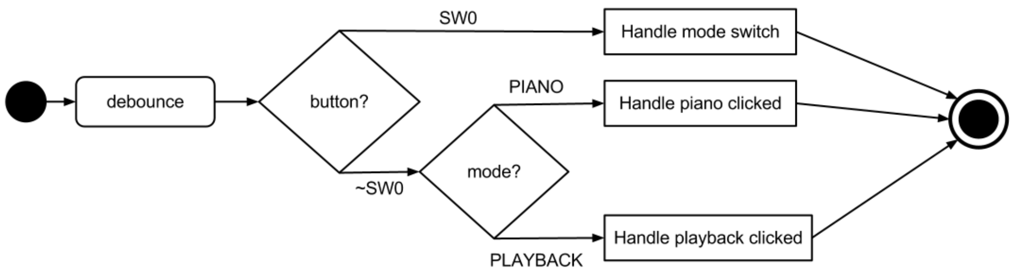


Figure 2: State diagram for buttons

3.2.2 ABDAC Interrupt Routine

The ABDAC routine behavior depends on the the mode. It delegates the work of setting up the next sample to either the Piano or Playback module and ask the gpio to write to the abdac.

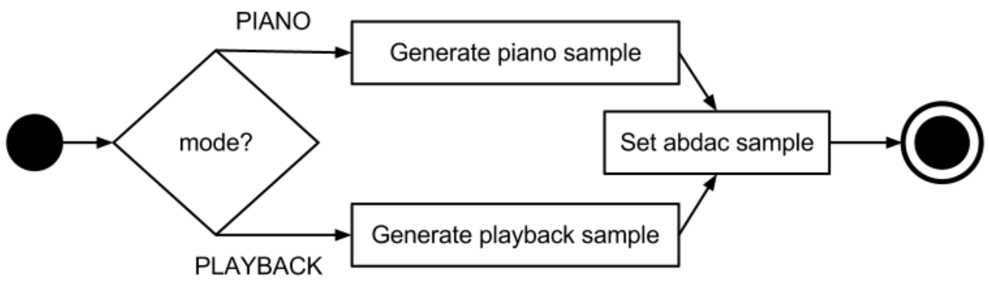


Figure 3: State diagram for abdac

3.3 Datastructures

In order to play sounds the sound samples has to be saved. A sample contains four tracks where each track is a series of notes in a linked list.

3.3.1 Note structure

A musical note is represented in the program by the datatype note_t defined in **note.h**.

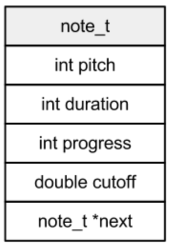


Figure 4: Datatype for notes

This type is used as a linked list to produce a track. The pitch is a number which relates to the frequency of the note. Pitches are defined in **tone.h** (C, D, .., C2, C3, ...). The duration is how long the note lasts, durations are also defined in **tone.h** (WHOLE, HALF, FORTH, etc.). The progress field is a state variable used when the tone is being played in order to know when its done, its always set to 0. The cutoff is to let different tones have different quality.

The range of the cutoff is 0.0 - 1.0 where 0.5 is a very *staccato*[1] tone and 1.0 is *glissando*[2]. The value 0.875 is used for ordinary notes.

3.3.2 Sound Tracks

The **playback.c** file contains a array, *tracks*, of constant size 4. This array has pointers to the current point of each track, it is updated by the **get_track_pich** function. A track is a linked list of notes. The list is NULL terminated. As there are 4 tracks, a sound sample can play four tones at a time.

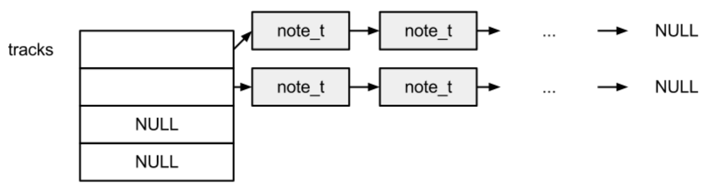


Figure 5: Illustration of tracks setup with 2 tracks

3.3.3 Sound Samples

The 7 different soundsamples are contained within a array of function pointers inside the interrupt.c file. Each function initilizes the tracks array by using the **set_track** function.

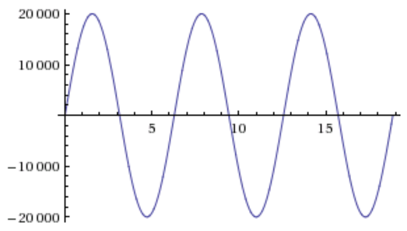
3.3.4 Sine Table

As the abdac interrupt routine is on a deadline, it has to conserve it's computing. Computing a **sin** function on every interrupt is wastefull and to timeconsuming. To make this a constant operation at runtime a sine table is computed on startup and stored in the **sine_table** in the **samples.c** file.

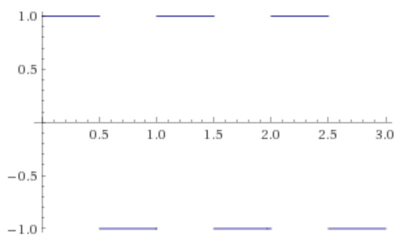
3.4 Waves

To make sound waves we need some functions producing wave signals. The following waves are implemented in **samples.c**.

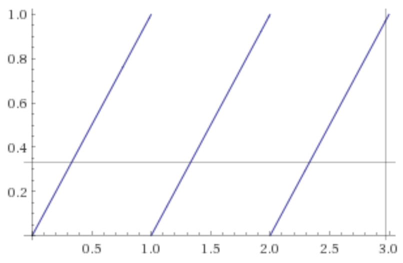
[1]A shortened duration of the tone
[2]When tones glides into each other



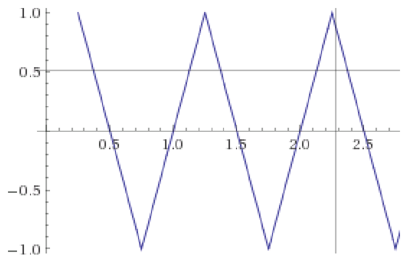
(a) Sine wave



(b) Square wave



(c) Sawtooth wave



(d) Triangle wave

4 Solution

4.1 Sound generation

To generate sound we used the internal ABDAC(Audio Bitstream Digital-to-Audio-Converter) on the AVR32 board. It takes a sequence of samples and converts it to an analog signal amplifies and outputs it. The ABDAC uses 2 of the pins on the PIO port B to send signals to the output, and the 6th clock of the Power Manager to generate interrupts that processes the samples. The clock is set up with oscillator 0 as the source and no division of the frequency. This gives us a clock of 20 MHz and a sample rate of 20MHz / 256 = 81.920 kHz on the ABDAC

```
// Register interrupt handler
register_interrupt((__int_handler)(abdac_isr),
            AVR32_ABDAC_IRQ / 32, AVR32_ABDAC_IRQ % 32,
                ABDAC_INT_LEVEL);

// Disable PIO
piob->PDR.p20 = 1;
piob->PDR.p21 = 1;

// Enable ABDAC
piob->ASR.p20 = 1;
piob->ASR.p21 = 1;

// Set the clock to use Oscillator (OSC0 and OSC1 is 20MHz and
    12MHz)
volatile avr32_pm_t *sm = &AVR32_PM;
volatile avr32_pm_gcctrl_t *clock = &sm->gcctrl[6];

clock->oscsel = 0;
clock->pllsel = 0;
clock->cen = ON;
```

The ABDAC is turned of when it is not used, to save power and keep the output silent.

```
dac->CR.en = OFF;
dac->IER.tx_ready = OFF;
```

To send samples to the ABDAC, in the interrupt routine, each of the stereo-channels are written with the corresponding sample data. We have only used mono sounds in our implementation, so the channels are written with equal samples.

```
dac->SDR.channel0 = sound;
dac->SDR.channel1 = sound;
```

The waveforms that are used as base for the samples are sine, triangle, sawtooth, square and white noise. They are implemented in samples.c as mathematical functions of a counter that ticks for a constant length. To make it possible to play multiple tones at once, the soundwaves are accumulated before written to the channels.

```
        for (i=0; i<7; i++) {
            sound += get_tone_pitch(i);
        }
```

This pretty much works the same way for playing multiple sounds at once in the playback mode, but the playback mode also allows accumulating sounds with different base waveforms.

4.2 Setting the frequency

The abdac_isr function is, as discussed, called with a frequency of 81.910 kHz. This gives us the sample frequency f_s. To get a tone frequency f_t of 440Hz, which is the tone A[3], we have to produce a wave form with this frequency from f_s. We generated a sine table with $SAMPLES = 4096$, meaning we generate 4096 values with even distance from $sin(0)$ to $sin(2\pi)$.

If we play the 4096 samples one by one in a loop the $f_t = f_s/SAMPLES = 20Hz$. To set a f_t of choice the following formula is used:

$$f_t = f_s/(SAMPLES/pitch\_modificator)$$

Calculating this for $f_t = 440Hz$ gives $pitch\_modificatior \approx 22$. We did try but was not able to verify theese calculations on the device, due to a poor cell phone microphone.

[3]440 Hz is called Concert pitch, ("A" one common tongue) the tone used to tune an ensamble of instruments.

5 Test Report

Number	TestCase	Prerequisites	Input	Expected	Result
1	Startup completed	make upload the program, press reset		leds 7 and 3 should be on	Passed 13.03.2012
2	Piano default mode	Startup completed	Button SW7 pressed	LED7 on and tone playing while button is down	Passed 13.03.2012
3	Release piano note	SW7 down, piano mode	Button SW7 released	LED7 off and tone stops playing	Passed 13.03.2012
4	Functional piano	Startup completed	Press button SW7-SW1	Dur-scale played	Passed 13.03.2012
5	Polytone piano	Startup completed	Press button SW7 and SW5	Base tone and major 3th playing	Passed 13.03.2012
6	Switch modes	Startup completed	Press button SW0	All LEDS on	Passed 13.03.2012
7	Switch modes back	Switch modes	Press button SW0	All LEDS off, piano active	Passed 13.03.2012
8	Play sample	Switch modes	Press button SW7	Sample 7 starts playing	Passed 13.03.2012
9	Play another sample	While Playing a sample (sample 7)	Press button SW6	Sample 7 stops playing, sample 6 starts	Passed 13.03.2012
10	Switching modes stops sample	Playbackmode on, sample is playing	Press button SW0	Sample stops, mode is piano	Passed 13.03.2012

Table 1: Test Report

6 Discussion

This assignment was easier getting started with, because we knew the tools, and the programming language. Off course handling I/O devices without an Operating System was a new experience. We found having a little knowledge of music- and wave-theory quite favorable, but the recitation slides and the compendium worked as good summary and reinstatements of the theory.

6.1 Deadline

Deadlines are the time restriction on the interrupt routine. The ABDAC interrupts with a frequency of 81.920 kHz. If it does not get a new sample in SDR before 256 cycles then we have missed a deadline. When we compile with different optimizations flags on gcc (-O0, -O1, -O2, and -O3) we get slightly different sounds and speeds of the audio samples. This indicates that our interrupt routine is to long and that we are missing some deadlines. We found that -O1 gave us the best result.

6.2 ABDAC interrupt routine and Sample format

We choose to save the sound samples as linked lists. This gave us compact storage, flexible notation (tracks are not synchronous), and little overhead at startup. But it means that the interrupt routine has a lot of work to do when it plays a sample. If we had stored the sound samples as an array of the actual samples to be played (which could be built on startup or even precompiled off target), the interrupt routine could be simplified to:

```c
// Predefined
#define N 100000
static int16_t samples[N] = { ... };

__int_handler *abdac_isr( void ) {
    static i = 0;
    if ( i < N ) {
      return samples[i];
      i += 1;
    } else {
      return 0; // silence
    }
}
```

This would have been an simpler design and would have enabled us to easier support sound samples from know formats as wav and midi. This would also have helped on meating the deadlines.

7 Conclusion

During this assignment we have experienced the advantage of higher level programming languages. We followed the recommended approach, and found solving the previous assignment i C to be a fraction of work compared to using assembly. As with the last assignment, we were reminded about the challenges when working with analog signals, and some new challenges were faced when working with digital audio, and real-time requirements.

We learned that the time limitations, limited computing powers and memory size means we have to adapt new ways to solve problems. Problems that might seem trivial when spoiled with GHz and GB in our development environments.