Report from Lab Assignment #2
TDT4258 Energy Efficient Computer Systems

Emil Taylor Bye Sigve Sebastian Farstad Odd M. Trondrud

March 14, 2013

Abstract

This report presents a solution to assignment #2 of TDT4258 at NTNU, spring 2013. The assignment was to write a program in C that plays sound effects and music on an Atmel STK1000 development board without an operating system. The solution program presented in this report is capable of generatively playing configurable sound effects and four-channel MOD music in real time.

Contents

Part I
Introduction

This report presents a solution to assignment #2 of TDT4258 at NTNU during the spring of 2013. The objective of this lab assignment is to write a program in C for the STK1000 development board which causes different sounds to play when different buttons on the board are pressed. An interrupt routine should be used to pass audio samples to the board's ABDAC (Audio Bitstream Digital to Analog Converter). The program is to run directly on the board, without an operating system. A minimum of three different sound effects should be made, as well as a "start up melody". [**?**]

The goal of this assigment is to introduce students to programming in C, I/O control for AVR32 in C, use of the microcontroller's ABDAC for sound generation and Interrupt handling in C for AVR32.

1 The STK1000

The STK1000 is a development board from Atmel which offers a complete development environment for Atmel's AT32AP7000 processor. It offers a multitude of different peripheral I/O devices, of which this assignment will be using an array of LEDs and some push buttons, as well as the Audio Bitstream Digital-to-Analog Converter. The processor is an ARM32 processor, and will for this assignment be only running the assembled output of a mix of hand-coded and tool-generated C code, without the support of an operating system. Additional information about the STK1000 can be found in [5].

2 The physics and digitalization of sound

In order to write a program to generate sound, one should first study the physical properties of sound, and research different strategies to generate sound in a digital environment.

Sound is a mechanical wave that is an oscillation of pressure composed of frequencies within the range of hearing[1]. Humans can percieve sounds with frequencies that range from about 20Hz - the lowest of basses - to about 20kHz - the highest of high-pitched whining[2]. Sound is inherently analog, and requires some form of digital representation to be able to be generated by the AP7000, which is a digital device. Regarding a sound wave as a continuous signal representing wave amplitude with respect to time, one straight-forward way of representing a sound wave digitally is to simply have an list of integer signal samples at a fixed, preferrably small, time interval. This format is called PCM, or Pulse Code Modulation, and is indeed the format the AP7000 expects for its digital-to-analog converter.

There are different strategies available for preparing the stream of integers that needs to be sent to the digital-to-analog converter to generate a sound. One strategy is simply to store the prepared list of integers somewhere in memory, and then copy it over to the DAC integer by integer as they are consumed. This strategy is analogous to rasterized bit maps in the image world. This strategy, while easy to implement, and is able to represent all kinds of sounds, requires a great deal of memory (integer size * sample rate of bytes per second, in fact). As an example, a three minute song, when stored at 16 bits per sample at a generously low sample rate of 22050Hz, requires 16 bits/sample * 180 seconds * 22050 samples/second = ca 7.57 megabytes. To put this in perspective, the entire flash memory of the AP7000 is 8 megabytes. On a low memory platform like the STK1000, this is therefore not a great strategy.

Another strategy is the generative approach. This strategy is analogous to vector based images in the image world. The idea is to generate samples at run-time based on configurations read from memory, rather than reading the pre-generated values from memory. This is a more CPU-hungry approach, but requires less memory than the previous strategy. This strategy is used for the sound effect synth in the presented solution program.

A third strategy is a hybrid approach, where small sample lists are pre-bundled with the program, and generative rules are used to play back the samples at different times with different parameters. This is the approach used in the music player in the presented solution program.

[1] http://en.wikipedia.org/wiki/Sound
[2] http://en.wikipedia.org/wiki/Audio_frequency

3 About the MOD file format

The MOD file format is an old music tracker file format originally created for the Commodore Amiga, a series of computers from the late eighties. TODO: IMAGE OF AN AMIGA, YO The file format is tightly optimized for playback on the Amiga's audio hardware, so to understand the inner workings of the MOD file format, one should first know a little about how the Amiga's audio hardware works.

The Amiga's sound chip, called Paula, is capable of powering four simultaneous DMA-driven 8-bit PCM sample sound channels. Each of these channels can be independently set to different sample frequencies many times per second. The MOD format exploits this - it supports 4 simultaneous channels of sample playback, using the frequency modulation to change the pitch of the samples played in the different channels.

Internally, the music in a MOD file is stored as a set of PCM-coded predefined sounds, as well as a large table of note patterns containing information about which sounds should be played at which frequencies and at which time. The MOD format also includes a large set of musical effects such as tremolo, vibrato, arpeggio, portamento and so on, a subset of which are implemented in the presented solution program.

The MOD file format is not a defined standard, and does therefore not have a formal specification. The MOD format grew organically from the early Amiga demoscene in the eighties, so many different variants exist, each with with their own specialities and quirks. The MOD Player presented in the solution is tailored to read so-called 'M.K.' MODs, generated by a MOD creator program ("tracker") called ProTracker. These MOD files are called 'M.K.' MODs because they contain the magic number 'M.K.' in the file header. This is one of the most popular MOD formats, and has become a sort informal standard amongst MOD trackers.

'M.K.' MODs can have a maximum of 31 bundled PCM-coded sounds, 128 patterns, each with 64 note divisions for each of the four channels, and a a 128 item long list of which patterns should be played in what order.

Image XXX shows an image of a MOD file being edited in a tracker program. Each column represents one channel, and each row represents one of the 64 divisions of a pattern. The currently played division is traditionally kept vertically centered in the middle of the screen, as in this image.

[Image of a MOD file being edited in a tracker to show what they look like]

Part II
Description and Methodology

This section describes the sound program, and details how it was developed. This section covers procedure, setup and configuration, tools and program details.

need to include a bit about the Makefile here

4 Functional description

The solution program plays sound effects and music, and is controlled by the eight buttons on the STK1000. The LEDs are used to indicate which sound is playing.

When the program is started, the board is in idle mode, ready to react to button presses. Pressing any of the buttons SW0-SW3 plays a piece of music, which loops until another sound is selected. Pressing any of the buttons SW4-SW6 plays a sound effect, which is not looped. Pressing SW7 stops all playback.

4.1 Sound effects

The sound effects are generatively composed by wrapping a generator signal in a configurable ADSR volume envelope. TODO: talk about ADSR. The available generator signals in the program are NOISE, SAWTOOTH and SQUARE.

TODO: talk about NISE, SAWTOOTH and SQUARE, with photos.

4.1.1 Explosion

'Explosion' is a NOISE-based sound effect with the following ADSR envelope: Attack: 0 ms Decay: 1000 ms Sustain: 0% Release: 0 ms The effect is held for 0 ms.

Total length: 0 ms + 1000 ms + 0 ms + 0 ms = 1000 ms 'Explosion' can be triggered by pressing SW6. TODO: add photo

4.1.2 Air horn

'Air horn' is a SAWTOOTH-based sound effect with the following ADSR envelope: Attack: 100 ms Decay: 100 ms Sustain: 70% Release: 500 ms The effect is held for 0 ms.

Total length: 100 ms + 100 ms + 500 ms + 0 ms = 700 ms 'Air horn' can be triggered by pressing SW5. TODO: add photo

4.1.3 Teleport

'Teleport' is a SQUARE-based sound effect with the following ADSR envelope: Attack: 500 ms Decay: 1250 ms Sustain: 20% Release: 250 ms The effect is held for 0 ms.

Length: 500 ms + 1250 ms + 250 ms + 0 ms = 2000 ms 'Teleport' can be triggered by pressing SW4. TODO: add photo

4.2 Music

The music pieces in the solution program are played by the MOD player.

4.2.1 Tuulenvire by Dizzy/CNCD

Tuulenvire is a 2:09 long 808KB composition in the ambient genre, featuring piano and accordion, amongst other instruments. This composition was chosen to demonstrate how careful composing can render realistic compositions with a relatively small memory footprint. It uses 25 different PCM-coded sounds. Tuulenvire can be triggered by pressing SW3.

4.2.2 Boesendorfer P. S. S. by Romeo Knight

Boesendorfer P. S. S. is a 3:22 long 211KB solo piano composition, chosen to illustrate the possibilities enabled by a hybrid generative/recorded approach. It uses 9 different PCM-coded sounds. Boesendorfer P. S. S. can be triggered by pressing SW2.

4.2.3 Drop The Panic by H0ffmann

Drop The Panic is a 4:05 long 702KB "glitch-hop" composition. It was chosen to show how MOD files can support embedded vocals. It uses 31 different PCM-coded sounds. The composition was tweaked by adding some extra inaudible notes in the beginning of the song to decrease critical cache misses by the MOD player during playback on the STK1000. Drop The Panic can be triggered by pressing SW1.

4.2.4 Bacongrytor by Maktone

Bacongrytor is a 15Kb endless loop chiptune-style composition, chosen to demonstrate the compactness of the MOD format, and therefore its aptfulness for use on microcontrollers. It uses 7 different PCM-coded sounds. Bacongrytor can be triggered by pressing SW0.

5 Solution components

The solution consists of three separate components. This section describes each of the three.

5.1 main program

5.2 synth

5.3 libmodam

Libmodam is the name of the MOD file parser and player written for this assignment. It is a portable, cross-platform c library. It has been tested on avr32 and on x86.

Libmodam is statically linked in the main solution program.

Libmodam includes a python utility script for platforms without filesystems. Normally, to use libmodam, a programmer must read a MOD file to to a byte buffer, and pass the buffer to libmodam, as it is done in the sample usage program test.c. This is sadly impossible in environments without a file concept. The python script solves this by reading arbitrary files and converting them to C-code declaring large static const char* arrays representing those files. For most platforms, this means that the files will be bundled with the executable in the .text section, making them read only.

By the programmer using libmodam, two main functions must be called repeatedly to generate output. One of these functions is `MOD_Player_play(...)`. This function returns a single sample of output, and must be called `sample rate` times per second for real time playback. The other function is `MOD_Player_step(...)`, and advances logical state, such as which notes should be playing and what effect should be applied. This function should be called no less that 50 times per second for real time playback.

These two driving functions are split up like this to minimize and constantivize the time spent generating the next output sample, to make libmodam more real-time-friendly.

Libmodam implements support for `M.K.` type MODs. A subset of the playback effects from `M.K.` have been implemented. Which effects to implement was chosen so that the most popular and often used effects were prioritized. A list of the effects in `M.K.`, and their implementation status in libmodam can be found in table XXX.

6 Experimental Procedure

7 Configuration

We can reference our previous report here, so that we don't have to write so much.

8 Jumpers

As before.

9 GPIO connections

As before, + maybe mention audio.

10 Audio

Jack etc

11 Development of the program

11.1 Sound effect synth

11.2 Libmodam

Libmodam was initially developed in a linux environment on an x86 PC, and later ported to avr32 and the STK1000. This was done for comfort reasons: there was no need to be in the lab to develop, testing and

iterating went a lot faster when code and data didn't need to be uploaded after each modification, and there was no need to focus on performance until features were confirmed to be working properly. As an added bonus, this also ensured a certain degree of portability, which is a nice property for a library to have. The python file conversion tool was developed alongside libmodam.

When the code was run on the STK1000 for the first time, the performance difference between the Intel i7 multi-GHz linux laptop and the STK1000 became immediately apparent. The sound produced by libmodam on the STK1000 at this point did not even remotely sound like music. In fact, it was more akin to a bowl of half-eaten oatmeal porridge left overnight, were porridge as audibly dull as it is bland in taste. One of the main reasons for the abysmal performance on the STK1000 was the lack of a floating point unit in the AP7000. Libmodam used floating point numbers heavily. The library was rewritten to use integer arithmetic, which helped immensely. Further optimizations were iteratively applied after this, until the code reached a point of acceptable performance.

11.3 The main program

How we assembled everything in the main program.

11.3.1 Setting up the LEDs

11.3.2 Setting up the buttons

11.3.3 Setting up the audio

rant about sample rates, div, diven

list of what we did:

* Board was set up w/ jumpers and such * Leds and buttons were hooked up in hardware * Leds and buttons were hooked up in software * Code was split into separate files * Audio was hooked up with apropriate settings * Sound was tested to work using random noise

the following two groups of bullet points happened in parallel:

* a C sound effect synth inspired by sfxr was prototyped on a PC * the synth was ported to avr32 * the synth was developed further on the avr32 * the synth was used in the stk1000 program to play various sound effects

* a C MOD player library + python tools inspired by amiga trackers were developed on a PC * the library was tested on the avr32 and needed a great deal of optimization * a great deal of optimization occurred * the library was used in the stk1000 program to play selected mods

finally:

* the actual main program flow was decided upon and written, hooking button and led behaviour together with sound effects and music

12 Programming environment

12.1 JTAGICE

We can reference our previous report here, so that we don't have to write so much.

12.2 GNU Debugger

since last time: * discovered tui mode: looks nice, breaks the makefile * avr32gdbproxy -f0.8Mb -a 0.0.0.0:1024 and target extended-remote:1024

12.3 Make

we can reference.

12.4 Other tools

* OpenMPT was used to examine MOD files for libmodam * vim * git * github * latex * avr32 toolchain

table XXX: Arpeggio: implemented Slide up: implemented Slide down: implemented Slide to note: implemented Vibrato: not implemented* Continue slide to note and volume slide: implemented Tremolo: not implemented Unused: implemented Set sample offset: not implemented Volume slide: implemented Position jump: implemented Set volume: implemented Pattern break: implemented Set filter on/off: not implemented Fineslide up: not implemented Fineslide down: not implemented Set glissando on/off: not implemented Set vibrato waveform: not implemented Set finetune value: not implemented Loop pattern: implemented Set tremolo waveform: not implemented Extended unused: implemented Retrigger sample: not implemented Fine volume slide up: implemented Fine volume slide down: implemented Cut sample: implemented Delay sample: not implemented Delay pattern: not implemented Invert loop: not implemented Set speed: implemented

*was implemented at one point, but was later removed for performance reasons

For more details about what these effects do, refer to XXX.

Part III
Results and Tests

Remember to bring an oscilloscope to have a look at different waveforms! Also, we could have a look at the random noise from the stk1000.

13 Energy Efficiency

(we threw this out the window :3) because of reasons. don't forget the reasons.

Some talk about how we chose to not turn off the abdac, even though it would be more efficient.

14 Testing

We loaded up the code and pushed a button and omg it worked because it played sounds. Also that one test where we just loaded the code to play a sound and it also worked.

15 Discussion

Part IV
Evaluation of Assignment

Writing the lab-reports using the required design (Abstract, introduction, description and methodology, results and tests, evaluation of assignment, conclusion, references) does not feel entirely appropriate. Mostly because the entire report seems to end up in description and methodology. Also there really aren't any relevant tests to perform (that are relevant to the assignment) beyond "does it work per the assignment's specification?".

Part V
Conclusion

References

[1] Atmel. Avr assembler user guide. http://www.atmel.com/Images/doc1022.pdf.

[2] Atmel. Avr32 architecture document. http://www.atmel.com/images/doc32000.pdf.

[3] Atmel. Avr32 at32ap7000 preliminary. http://www.atmel.com/Images/doc32003.pdf.

[4] Atmel. Power jumpers on ap7000. http://support.atmel.no/knowledgebase/avr32studiohelp/com.atmel.avr32.tool.stk1000/html/jumper_settings.html#Power_jumpers.

[5] Emil Taylor Bye, Sigve Sebastian Farstad, and Odd M. Trondrud. Report from lab assignment #1, tdt4258 energy efficient computer systems.

[6] Caltek/BST. Bst bs1901w manual. http://www.docstoc.com/docs/67828383/BS1901W.

[7] University of Toronto Faculty of Applied Engineering. Lab reports. http://www.engineering.utoronto.ca/Directory/students/ecp/handbook/documents/lab.htm.

[8] Stefano Nichele. Introduction, tdt4258 energy efficient computer systems. http://www.idi.ntnu.no/emner/tdt4258/_media/intro.pdf.

[9] Stefano Nichele. Tutorial lecture for exercise 1. http://www.idi.ntnu.no/emner/tdt4258/_media/ex1.pdf.

[10] Department of Computer and NTNU Information Science. Lab assignments in tdt4258 energy efficient computer systems. http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf.

All internet resources were checked on ¡¡ENTER DATE HERE¿¿.