Report from Lab Assignment #2
TDT4258 Energy Efficient Computer Systems

Emil Taylor Bye Sigve Sebastian Farstad Odd M. Trondrud

March 14, 2013

Abstract

*purpose of the assignment (learning outcome from the compendium: 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)

*key findings (the board can play sounds if programmed appropriately)

major conclusions (????)

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".

This report details the development of the sound-generating program as a solution to the assignment.

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 the report for the previous assigment of this course. [5]

2 A note on 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 20kHz - the highest of high-pitched whining. 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 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 by the ABDAC. 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

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. 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 could 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 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 'M.K.' MODs, generated by a MOD creator program ("tracker") called ProTracker. This is the most popular MOD format, and has become a sort informal standard amongst MOD trackers.

M.K. MODs can have a maximum of 31 bundled noises, 128 patterns, each with 64 note divisions per channel, 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 how the sound program was developed. Ut covers procedure, setup and configuration, tools and program details.

need to include a bit about the Makefile here

Description

General information about sound

Sound is a mechanical wave that is an oscillation of pressure composed of frequencies within the range of hearing.[http://en.wikipedia.org/wiki/Sound]. Humans can percieve sounds with frequencies that range from about 20Hz - the lowest of basses - to 20khz - the highest of high-pitched whining. Sound is inherently analog, and requires some form of digital representation to be able to be generated by the ap7000. 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 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 simply copy it over to the DAC integer by integer as it consumes them over time. 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). 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. This is the approach used in the music player in the presented solution program.

Development of the program

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

Configuration

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

Programming environment

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

GNU Debugger since last time: * discovered tui mode: looks nice, breaks the makefile

Make we can reference.

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

Setting up the LEDs Setting up the buttons Setting up the audio

Program flow

* main program * synth * libmodam

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. 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 could 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 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 'M.K.' MODs, generated by a MOD creator program ("tracker") called ProTracker. This is the most popular MOD format, and has become a sort informal standard amongst MOD trackers.

M.K. MODs can have a maximum of 31 bundled noises, 128 patterns, each with 64 note divisions per channel, 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]

Functionality of the program

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.

The sound effects

The sound effects are generatively composed by wrapping a generator signal in a configurable ADSR volume envelope. The available generator signals in the program are NOISE, SAWTOOTH and SQUARE. Pressing SW6 triggers the sound effect called 'explosion'. It is a NOISE-based sound effect with the following ADSR envelope: XXX. Length: XXX.

Pressing SW5 triggers the sound effect called 'air horn'. It is a SAWTOOTH-based sound effect with the following ADSR envelope: XXX. Length: XXX.

Pressing SW4 triggers the sound effect called 'teleport'. It is a SQUARE-based sound effect with the following ADSR envelope: XXX. Length: XXX.

The pieces of music

The pieces of music are played by the MOD player.

Pressing SW3 triggers Dizzy/CNCD's Tuulenvire. It 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 samples.

Pressing SW2 triggers Romeo Knight's Boesendorfer P. S. S. It is a 3:22 long 211KB solo piano composition, chosen to illustrate the possibilities enabled by a hybrid generative/recorded approach. It uses 9 samples.

Pressing SW1 triggers a tweaked version of H0ffmann's Drop The Panic, chosen because it sounds cool. It is a 4:05 long 702KB "glitch-hop" composition. It uses 31 samples. The composition was tweaked to decrease critical cache misses by the MOD player during playback on the STK1000.

Pressing SW0 triggers Maktone's Bacongrytor. It is a 15Kb endless loop chiptune-style composition, chosen to demonstrate the compactness of the MOD format, and therefore its aptfulness for use on micro-controllers. It uses 7 samples.

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.

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.

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.

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¿¿.