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.

A video demonstration of the solution program running on an STK1000 can be seen at
`http://www.youtube.com/watch?v=cdKfN6vJjk8`

Contents

I Introduction **1**

1 The STK1000 **1**
1.1 ABDAC (Audio Bitstream Digital-to-Analog Converter) . . . 1

2 The Physics and Digitalization of Sound **1**

3 About the MOD file format **2**

II Description and Methodology **4**

4 Functional Description **4**
4.1 Sound effects . . . 4
4.1.1 Explosion . . . 5
4.1.2 Air horn . . . 6
4.1.3 Teleport . . . 7
4.2 Music . . . 7
4.2.1 Tuulenvire by Dizzy/CNCD . . . 8
4.2.2 Boesendorfer P. S. S. by Romeo Knight . . . 8
4.2.3 Drop The Panic by H0ffmann . . . 8
4.2.4 Bacongrytor by Maktone . . . 8

5 Solution Components **8**
5.1 Main Program . . . 8
5.2 Sound Effect Synthesizer . . . 8
5.3 Libmodam - MOD Parser and Player Library . . . 8

6 Configuration **10**
6.1 Jumpers . . . 10
6.2 GPIO connections . . . 10
6.3 Audio . . . 10

7 Development of the program **10**
7.1 Sound effect synth . . . 10
7.2 Libmodam . . . 11
7.3 Main Program . . . 11
7.3.1 Setting up the LEDs . . . 11
7.3.2 Setting up the buttons . . . 11
7.3.3 Setting up the audio . . . 12

8 Programming Environment **12**
8.1 JTAGICE . . . 12
8.2 GNU Debugger . . . 12
8.3 Make . . . 12
8.4 Other tools . . . 12

III Results and Tests **13**

9 Energy Efficiency **13**

10 Testing **13**
10.1 Basic functionality test 13
10.2 Testing of generated waveforms 13
10.3 Testing the ABDAC interrupt frequency 14

11 Discussion **14**
11.1 Ideas for improvement 15

IV Evaluation of Assignment **16**

V Conclusion **17**

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" ([10], page 48).

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.

1.1 ABDAC (Audio Bitstream Digital-to-Analog Converter)

An Audio Bitstream Digital-to-Analog Converter is a component which converts a digital bitstream signal to an analog signal. It is useful for, amongst other things, playing a PCM-coded sample stream as analog audio. The STK1000 has two Audio Bitstream Digital-to-Analog Converters, the internal DAC and the external DAC. The presented solution program uses the internal DAC to play sound.

The internal DAC is capable of converting a 16 bit digital stereo stream with its oversampling architecture. The oversampling ratio is fixed at 128x, and the output is processed with a FIR equalization filter, a Comb4 digital interpolation filter and 3rd order sigma-delta digital/analog converters [3]. The internal DAC is also connected to the DMA, but this feature is not used in the solution.

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. If a sound wave is regarded 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 a list of integer signal samples at a fixed, preferrably small, time interval. The integers represent the amplitude of the sound wave at the given time. This format is called is known as PCM (Pulse Code Modulation) and is the format the AP7000 expects for its digital-to-analog converter.

There are different strategies available for preparing the stream of integers that need 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 processing world. While being easy to implement

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

and capable of representing all kinds of sounds, this strategy requires a great deal of memory (integer size $\times$ sample rate of bytes per second). As an example, a three minute long song stored at 16 bits per sample at a sample rate of 22050Hz, requires 16 bits/sample $\times$ 180 seconds $\times$ 180 seconds $\times$ 22050 samples/second $\approx$ 7.57 MiB. This is not a viable strategy when working with low memory platforms like the STK1000 (the AP7000's flash memory can store 8 MiB).

Another strategy is the *generative approach*. This strategy is analogous to vector based images in the image processing 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.

3 About the MOD file format

The MOD file format is an old music tracker file format originally created for the Commodore Amiga (figure 1), a series of computers from the late eighties.

Figure 1: The Amiga 1000 (1985), the first Amiga model released. Image courtesy of Kaiiv.

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.

Figure 2 shows an image of a MOD file being edited in a tracker program. Each column represents one channel (called *track* in ProTracker), 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.

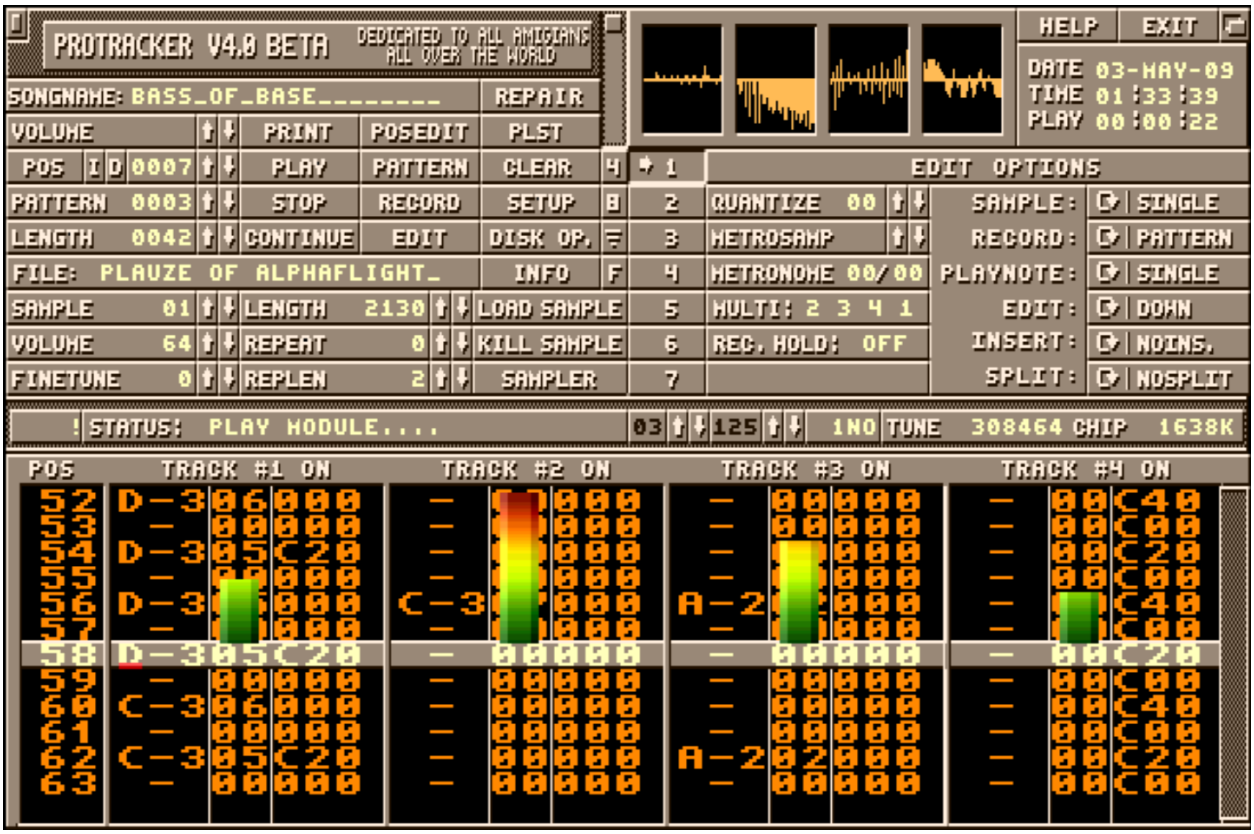

Figure 2: A four-channel MOD being played in ProTracker. Image courtesy of Alec Graggamoor.

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.

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. Because the internal DAC of the STK1000 is rather noisy because of signal bleeding and poorly implemented hardware filters, the sound from the generators can sound rather dirty.

The available generator signals in the program are NOISE, SAWTOOTH and SQUARE.

A sawtooth wave is a wave that increases linearly, until it cuts off, as seen in figure 3. The sawtooth wave includes all the integer harmonics for the given frequencies.

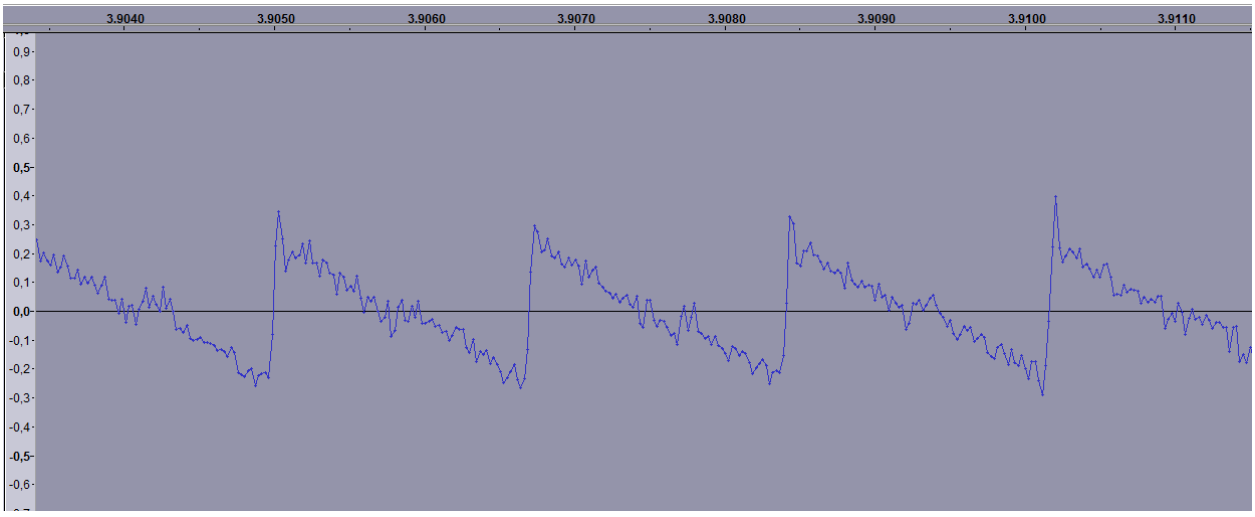

Figure 3: A noisy sawtooth wave recorded from the STK1000.

An ideal square wave is either at a maximum or minimum amplitude as seen in figure 4, and shifts between them instantly. Unlike a sawtooth wave, the square wave only contains odd-numbered integer harmonics.

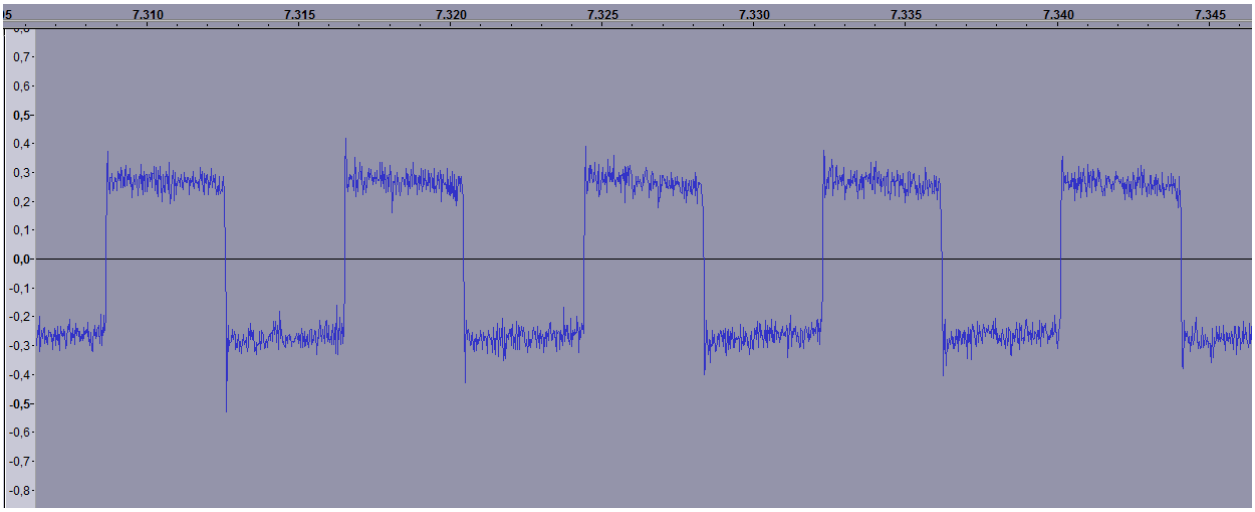

Figure 4: A noisy square wave recorded from the STK1000.

A noise is rather the lack of a waveform, with randomly chosen samples, as depicted in figure 5. Noise can be described as the sound your tv makes when you tune into frequencies that there is no broadcasting on.

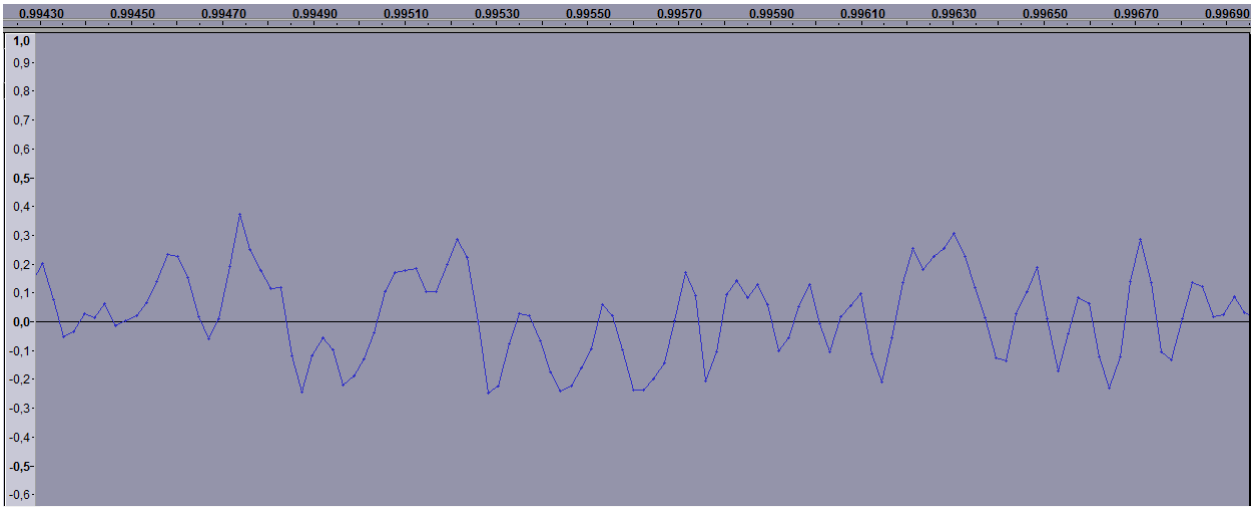

Figure 5: A noise signal recorded from the STK1000

In order to make sound effects we need more than just pure waves, as sound almost never consists of just a wave with constant amplitude. It is the job of the ADSR envelope to solve this issue. An envelope sets bounds for the amplitude of a wave, and an ADSR envelope divides the wave into four parts: attack, decay, sustain and release.

During the attack, the amplitude is gradually increased from 0 to a maximum value. After the attack, the amplitude gradually decays to a sustain level, which is typically a fraction of the maximum value. The sound stays at the sustain level for a given time, until it is released, and the amplitude gradually decreases until it reaches 0 again.

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. The total length of the effect is $0ms+1000ms+0ms+0ms = 1000ms$. 'Explosion' can be triggered by pressing `SW6`. The sound effect is depicted in figure 6.

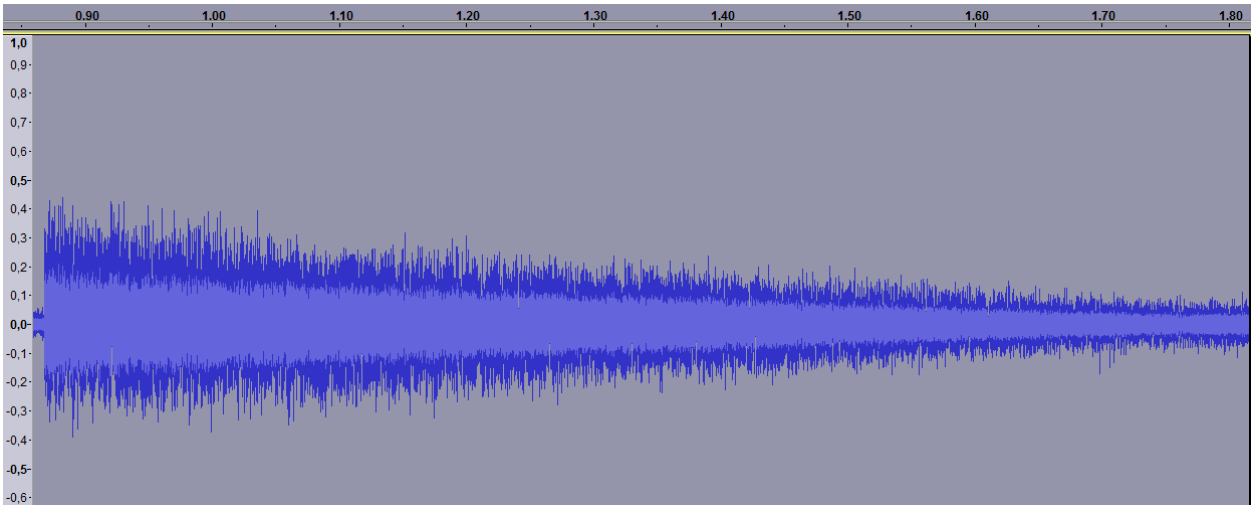

Figure 6: An overview of the Explosion sound effect, recorded from the STK1000.

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. The total length of the effect is $100ms + 100ms + 500ms + 0ms = 700ms$. 'Air horn' can be triggered by pressing `SW5`. See figure 7 for a visualization of 'Air horn'.

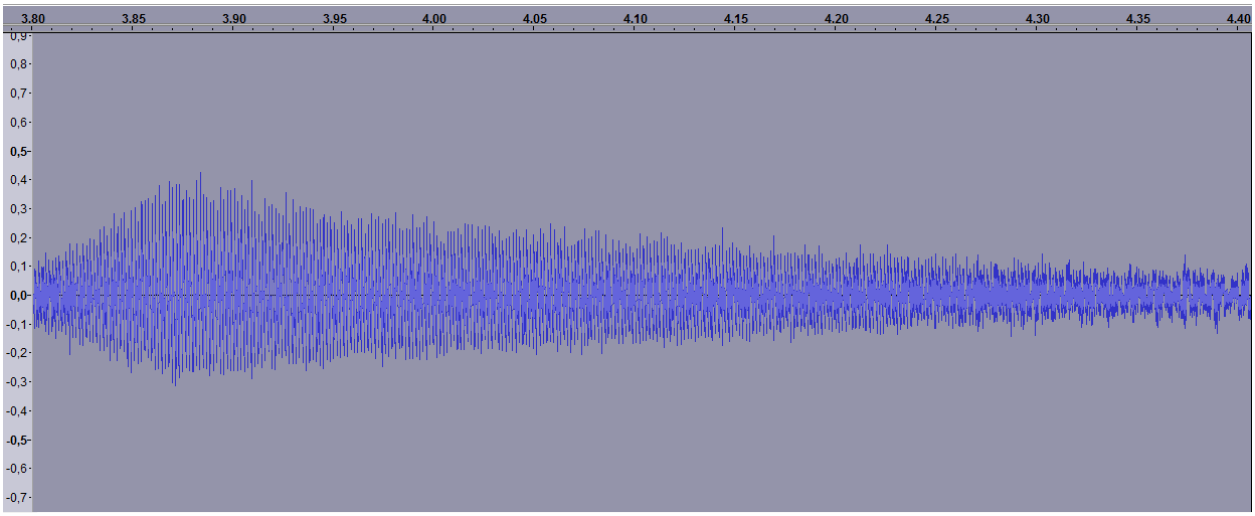

Figure 7: An overview of the Air horn sound effect, recorded from the STK1000.

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. The total length of the effect is $500ms + 1250ms + 250ms + 0ms = 2000ms$. 'Teleport' can be triggered by pressing `SW4`. A picture of the recorded sound can be seen in figure 8.

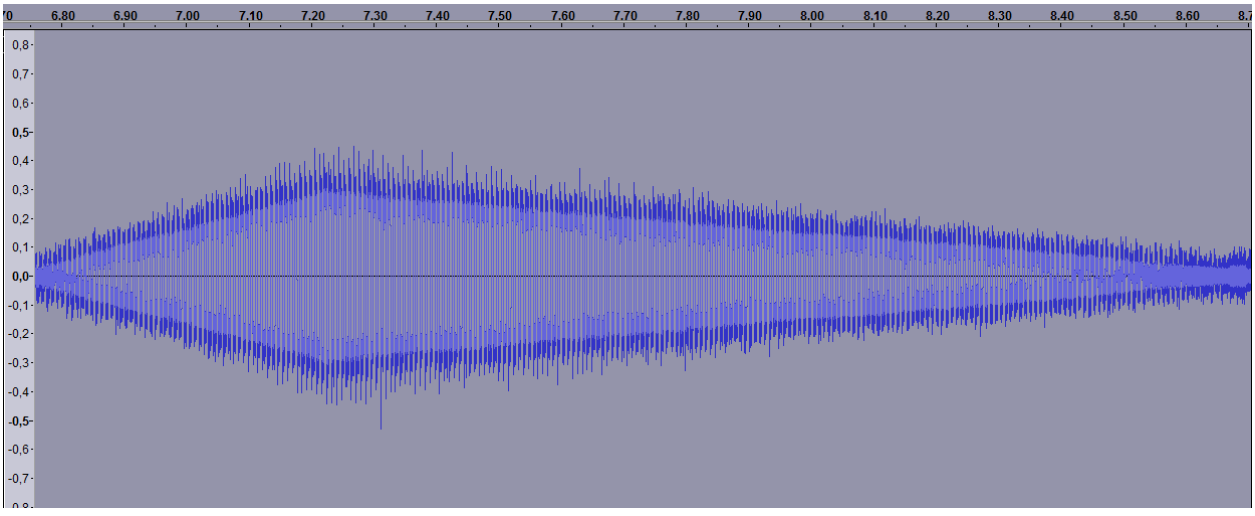

Figure 8: An overview of the Teleport sound effect, recorded from the STK1000.

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

The main program is the C program which runs on the STK1000. It interfaces with the hardware, and administers the user interface and main program logic.

5.2 Sound Effect Synthesizer

The synthesizer is responsible for generating the sound effects real-time as they are played. It generates sounds by producing waveforms as described in the sound effects section, and modifies the amplitude of the signals according to a ADSR envelope.

5.3 Libmodam - MOD Parser and Player Library

Libmodam is the name of the MOD file parser and player written for this assignment. It is written as 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 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 system. 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 keep constant 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 2.

Table 1: A list of the effects in `M.K.` and whether or not they have been implemented in libmodam.

Effect	Implemented?
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

Table 2: A list of MOD effects and their implementation status in libmodam.

*was implemented at one point, but was later removed for performance reasons
For more details about what these effects do, refer to the list below.

- `http://www.mediatel.lu/workshop/audio/fileformat/h_mod.html`
- `http://archive.cs.uu.nl/pub/MIDI/DOC/MOD-info`
- `https://bel.fi/alankila/modguide/`

- `http://16-bits.org/mod/`

6 Configuration

6.1 Jumpers

The jumpers were set up as in the previous assignment. [5]

6.2 GPIO connections

The GPIO connections were set up as in the previous assignment. [5]

6.3 Audio

Headphones were connected to the board's audio jack connector as in figure 9, so people could listen to the sweet, sweet sounds of the solution program.

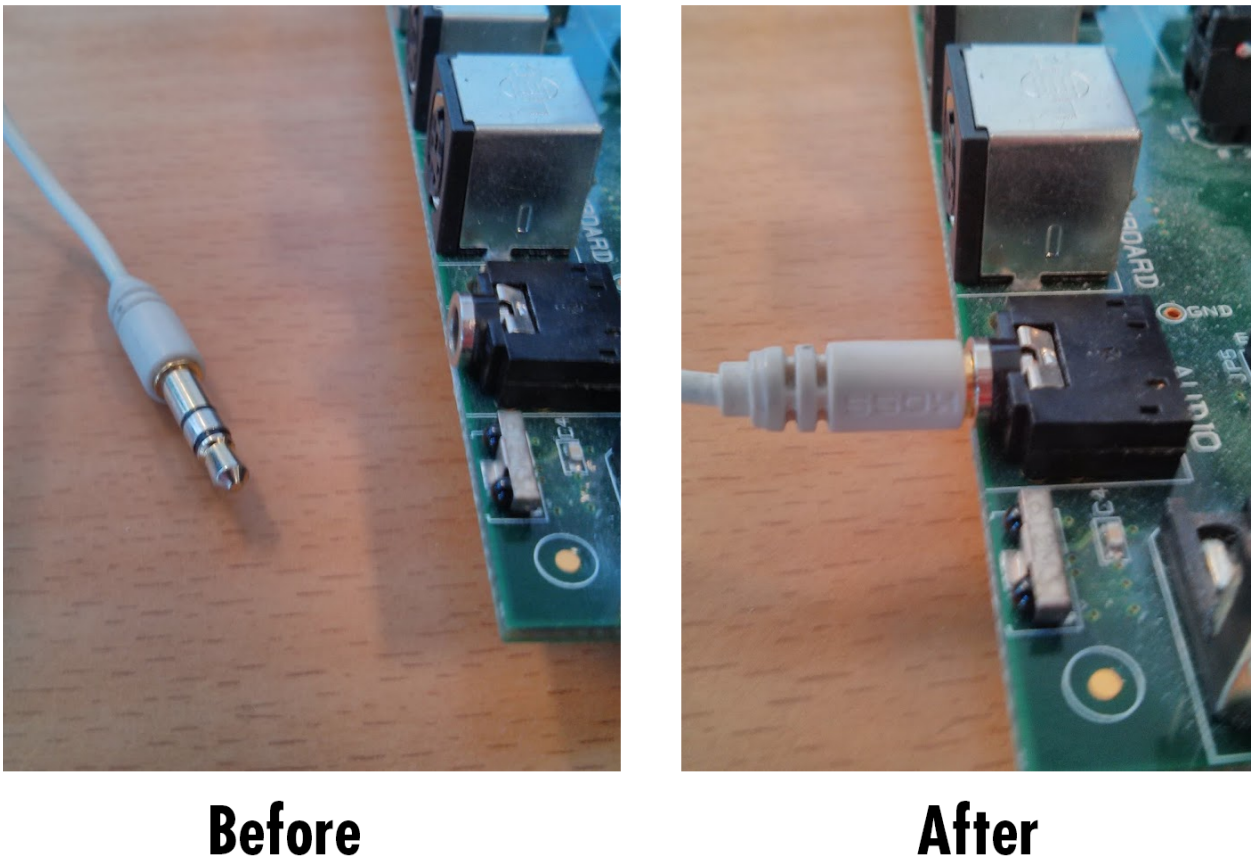

Figure 9: The audio jack port on the STK1000 before and after a headphone is connected.

7 Development of the program

7.1 Sound effect synth

The core of every synthesizer is the oscillator generating signals, so it was essential to develop first in order to produce sound signals to play. Sawtooth and square waves were chosen as they are relatively easy to

implement. A noise signal was added as it required no extra logic, and it is needed to make the explosion sound.

After the oscillator was capable of producing waves with arbitrary frequencies, development began on the ADSR envelope. Code for the sustain part of the sound was the easiest to write, as it just required the amplitude to be scaled down so that the highest possible amplitude was a given fraction of the highest possible amplitude. Scaling for the attack, decay and release part was a bit worse however, as floating point divisions should be avoided at all costs. In the end an acceptable result was reached with a smart ordering of integer multiplications and divisions.

7.2 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.

7.3 Main Program

The main program was developed in the lab, room 458, 4th floor ITV-building, NTNU. First, the board was configured as in the configuration section of this report. Then, LEDs and and buttons were hooked up in the program, as detailed in their respective sections of this report. After this, code was refactored and split into several files, and the ABDAC was hooked up in the program, as detailed in the setting up the audio section.

Once the ABDAC was hooked up, random noise was sent to the ABDAC output register, to see if everything was set up correctly.

At this point, development of libmodam and the synth started. This is detailed in other sections of this report. Once they were finished, they were integrated into the main program. Some wrapper logic, as well as user interface code was hooked up, and hey presto, the program was done.

7.3.1 Setting up the LEDs

On initialization, the address of the `PIO` which the LEDs are connected to is stored as a pointer. A bitfield, represented as an integer, is used to indicate which LEDs to enable. This value is written to the `PER` and `OER` registers of the `PIO` to enable the indicated LEDs. In C, the syntax for this is `pio->reg = bits`, where `pio` is a pointer to an `avr32_pio_t` struct containing the address of the appropriate `PIO`.

For a more detailed explanation of how the LEDs on the STK1000 are set up, see page 4 of [5].

LEDs can then be enabled or disabled by writing appropriate values to the PIO's `SODR` and `CODR` registers, respectively.

7.3.2 Setting up the buttons

The button setup for this assignment is almost exactly the same as the setup from assignment 1 [5]. The only difference is that the setup code is written in C, rather than assembly, which allows for higher lever abstractions. The button code leverages the inversion of control principle, with dependency injection through the argument list of the setup function.

7.3.3 Setting up the audio

The audio setup code is organized in the same way as the LED and button setup code – an injectable setup function in a separate file. The internal DAC, which is used for audio output, uses pin 20 and 21 of PIOB as output pins. These pins must be disabled, because for some reason. What was it again? The pins are instead given to peripheral A, which is the ABDAC.

In order to drive the ABDAC, a clock must be set up. Generic clock 6 is the clock that powers the ABDAC. The generic clock 6 is in the assignment set up to be driven by oscillator 1, which runs at 12Mhz. Because the clock fires an ABDAC interrupt for every 256th clock tick, this means that the sample rate will be 46875. This is a little high. Luckily, the clock speed can be divided using the divide feature. By enabling the divide feature, and setting the `div` to 0, the clock signal is divided by two. This is because the clock speed is divided by $2 \times (\texttt{div}+1)$.

8 Programming Environment

8.1 JTAGICE

See [5], page 2-3.

8.2 GNU Debugger

Picking up where [5] left off, our heroes have since then discovered `tui` mode (text user interface mode), which while looking awfully nice breaks the Makefile. Likely due to `-tui` causing gdb to capture input differently, and the Makefile relies on a hack using cat and pipes.

`avr32gdbproxy -f0,8Mb -a 0.0.0.0:1024` and `target extended-remote:1024`. While not 100 % sure of what this does, it allows the program being debugged to be run again without starting gdb anew.

8.3 Make

See [5], page 3.

8.4 Other tools

- OpenMPT, a MOD tracker, was used to examine MOD files during the development of libmodam.
- `vim` was employed as the authors' text editor of choice.
- `git` was used for version control.
- The project was hosted in a private GitHub repository.
- The report was written with LaTeX.
- AVR32-specific flavors of GNU's `as` and `ld` were used to assemble and link executables.
- `avr32program` was used to program the STK1000 with the JTAGICE.
- The sound effects were recorded using Audacity on an ASUS UL30JT laptop.

Part III
Results and Tests

9 Energy Efficiency

Energy efficiency in computing is ever-important, and is something programmers should be conscious about when developing for microcontrollers.

A number of measures to increase energy efficiency have been considered for the solution program, but not all of them have been implemented.

To save energy, the CPU could be set to sleep when no sounds are playing. If this is done, the clock powering the ABDAC must be turned off before sleeping, so that it does not wake the CPU at once. This is not done in our solution, as the the board makes a loud and ugly popping noise when it is switched on or off. This popping noise is detrimental to the user experience, and ruins the functionality of the program.

Further, the clock rate of the CPU can be lowered to save energy, but this probably requires further optimization of the solution program.

10 Testing

The basic test requirements are the same as on page 15 of [5].

10.1 Basic functionality test

This test aims to uncover whether we have managed to set up the hardware properly and managed to get the sounds and music to play.

Prerequisites:

- One (1) finger
- Functional eyesight
- Auditory perception

Procedure:

1. Upload the code to the STK1000 (e.g using `make upload`).
2. Push the board's `RESET` button.
3. Push the various buttons and verify that the expected piece of music or sound effects play, and that the LED above the button lights up.

This is about as basic as it gets for simple functionality tests. Everything worked as expected, so it was probably mostly fine.

10.2 Testing of generated waveforms

In this test we want to see whether we actually generate the waveforms we expect to with the sound effects.

Prerequisites:

- One (1) finger
- Functional eyesight
- Auditory perception
- A way of recording the sound signal

Procedure:

1. Upload the code to the STK1000 (e.g using `make upload`).
2. Push the board's `RESET` button.
3. Record the sound played when `SW4` is pressed and compare the generated sound waves with a square wave.
4. Record the sound played when `SW5` is pressed and compare the generated sound waves with a sawtooth wave.
5. Record the sound played when `SW6` is pressed and see whether it looks like noise.

The generated sounds were generated more or less as we expected, `SW4`'s sound wave can be seen in figure 4, `SW5`'s in 3 and `SW6`'s in 5.

10.3 Testing the ABDAC interrupt frequency

When first playing sound, we could not quite figure out how often our interrupt routine got called, and therefore had difficulty setting the sample frequency.

Prerequisites:

- One (1) finger
- Functional eyesight
- A clock to keep track of how long the procedure takes
- A simple program to count up every time the interrupt routine is called and changes the LEDs when the counter has increased by a number.

Procedure:

1. Upload the code to the STK1000 (e.g using `make upload`).
2. Push the board's `RESET` button.
3. Take the time it takes for the LEDs to change 10 times.

The point is that when it takes exactly 10 seconds for the LEDs to change 10 times, you know how many times the interrupt routine gets called every second. This test made us realise that we did not manage to control the frequency of the interrupt routine, and we had to implement a little hack to get it to work as we wanted to. We did end up finding how to change the frequency though, and this test allowed us to verify that.

11 Discussion

We managed to play mod-files successfully and generate different sound waves as expected. There was more noise on the internal ABDAC than we expected, and sometimes while developing it was hard to hear whether we had made an error because of all the noise. We did not find any ways to counteract the noise other than turning up the volume, and hoping that the noise was relatively too weak to make much of an impact on the sound waves.

11.1 Ideas for improvement

There are a number of measures that can be undertaken to improve the presented solution program:

- The energy efficiency can be improved, as noted in the Energy Efficiency section of this report.
- To reduce the amount of noise from the board, the external DAC can be used.
- To offload the CPU, the ABDAC can be set up to get samples using DMA.
- To improve MOD compatability, more MOD effects can be implemented.
- To increase the diversity and range of the sound effect synth, more controllable features can be implemented, such as volume slides and filters.
- More generator primitives (sine wave, triangle wave, pulse wave et cetera) can be implemented to increase the range of possible sound for the sound effect synth.
- Polyphony implemented in the sound effect synth can greatly increase the number of possibilities.
- Performance-wise, the samples from the sound effect synth can be pregenerated and stored in memory, to reduce CPU load.

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) felt a bit weird at first, probably because we did not read the supplied material regarding lab reports.

The assignment/compendium could include some questions for the students to reflect over, to push them in the right direction regarding what's important to include in the report. As it is now, there aren't that many relevant tests (in hindsight, at least) beyond "does it work per the assignment's specification?".

All in all though we had fun and the experience was pretty enjoyable.

Part V
Conclusion

The finished program is capable of playing different sounds when different buttons are pushed; including three different sound effects and one start-up melody. It is also capable of playing modfiles, represented as an array of integers. Measures taken to increase the energy efficiency of the program resulted in a reduction of the quality of the sounds, illustrating that energy efficiency cannot always be achieved without sacrifice. The authors feel the assignment has further increased their proficiency and familiarity with C and hardware-level programming, as well as sound theory. In conclusion, the project was completed according to the specifications within the given timeframe. In the world of business, this is commonly known as "great success".

A video demonstration of the solution program running on an STK1000 can be seen at
`http://www.youtube.com/watch?v=cdKfN6vJjk8`

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 2013-03-14.