TDT4258 Microcontrollers System Design Exercise 2

Generating Sound with a DAC on a Microcontroller Using C

Vegard Edvardsen
Jean Niklas L'orange
Caroline Sæhle

March 19, 2012

Abstract

Handling input and output data is essential in all programs. When developing programs for operative systems, the operative system will provide a nice abstraction such that receiving and sending data is made easy. When there is no operating system, one has to handle reading and writing such data manually. This report shows how to create a C program which runs without an operating system on top of an STK1000 board. The program is able to generate sound and send it to audio devices connected through the jack socket on the board. This is done by writing to memory-mapped I/O owned by the internal audio bitstream digital to analog converter, which in turn handles output to the jack socket. The I/O is performed in an interrupt routine which is called by a clock at consistent intervals. We also explain and test out different ways of generating sounds to the interrupt routine fast enough.

Contents

1 Introduction

The objective of this task is that the group members will get a better understanding on how to program in C, how I/O control and interrupt handling is done for AVR32 in C, and how to use the AVR32's audio bitstream digital to analog converter (ABDAC) for sound generation. The task given to us in the course Microcontroller System Design (TDT4258) to learn these things was to "Write a C–program which runs directly on the STK1000 board [...] and which plays different sound effects when different buttons are pressed. [...] The requirement is that you use an interrupt routine to pass the samples to the ABDAC."[3]

The task was solved incrementally, as the group members have had good experience with incremental development of programs in previous projects. Through an incremental approach of trying and testing, we could easily pinpoint where and when a bug in our program was introduced, and could thus eliminate them relatively fast. We also used the AVR32's internal ABDAC, as that was a recommendation from the compendium.

We decided to use equal temperament, due to the ease of implementing songs and tunes as we are already familiar with it. We chose to use sine waves in order to generate smooth tones, and because it posed a greater challenge. This forced us to generate a sound buffer in order to make the interrupt handler fast enough.

2 Description and Methodology

As recommended in the compendium [3], we followed the approach outlined below in doing this exercise.

1. Compiling, uploading and debugging the handout code.
2. Recreating the previous assignment in C code.
3. Implementing the current assigment.

The steps required to implement the current assignment were as follows:

1. Configuring I/O for LED control.
2. Configuring I/O for button handling and implementing a button interrupt routine.
3. Setting up the ABDAC (*Audio Bitstream Digital to Analog Converter*).
   (a) Configuring I/O to allow using the ABDAC.

(b) Setting up a clock source for the ABDAC.

(c) Configuring the ABDAC.

(d) Implementing an interrupt routine supplying samples to the ABDAC.

4. Implementing sounds/tones.
5. Implementing melodies.
6. Letting melodies to be controlled by the buttons.

2.1 Implementation Overview

The source code files in our final implementation are listed in table 1.

File	Function
`ex2.c`	`main` function and initial setup.
`ex2.h`	Header file for `ex2.c`.
`button.c`	Button I/O setup and interrupt routine.
`button.h`	Header file for `button.c`.
`led.c`	LED I/O setup and functions to set and get LED values.
`led.h`	Header file for `led.c`.
`sound.c`	Sound buffer management, ABDAC setup and interrupt routine.
`sound.h`	Header file for `sound.c`.
`melody.c`	Melody playing.
`melody.h`	Header file for `melody.c`.

Table 1: Overview of source code files.

The program is spread across these files to separate the code into logically separate units/modules. The intention was to have separate modules for button handling, LED control, sound playing and melody playing.

2.2 LED and Button Control

The LED control code implemented in `led.c` works similarly to the assembly code in the previous exercise. The same control registers are accessed and used the same way. In this C implementation, however, we utilize the AVR header files to access the various flags and fields as C data structures rather than as bit offsets.

`led_init` initializes the I/O pins for LED control. The global variable `led_setup` contains the current LED state. `led_update` updates the data pins to reflect the value in `led_setup`. The utility functions `led_set`, `led_get` and `led_clear` simplify common LED operations by manipulating `led_setup` accordingly and then optionally calling `led_update`.

Button handling is also similar to the assembly code in the previous exercise. `btn_init` initializes the I/O pins for input and registers `btn_interrupt` as the interrupt routine for button changes. `btn_interrupt` is an extended version of the interrupt routine in the previous exercise, checking the status of *all* of the eight buttons in a `for` loop. When a button is pushed, `mel_set_melody` is called to change the melody.

2.3 Playing Sound Samples

2.3.1 ABDAC Setup

To generate sound, the ABDAC first needs to be set up. As outlined in the beginning of the chapter, there are several steps in doing this.

First, the I/O pins shared with the ABDAC module, pins 20 and 21 on port B, need to be relinquished. This is done by setting the corresponding bits in PIOB's `PDR` (PIO Disable Register) and `ASR` (Peripheral A Select Register) registers.

Next, the ABDAC needs a clock source to time the samples. This is supplied by a *Power Manager* on the chip. The Power Manager has several configurable generic clocks, and the ABDAC is connected to clock number 6. To configure this clock output, we need to specify an internal clock source and an optional frequency divisor. In our program, we use oscillator 0 and no frequency divison. This gives a clock rate of 20 MHz. [2]

Lastly, we need to specify an interrupt routine supplying the samples (`snd_interrupt`), and then the ABDAC needs to be activated. This is done by setting two flags in the ABDAC registers.

These initialization steps are all contained in the function `snd_init`.

The ABDAC will, after activation, trigger the interrupt routine `snd_interrupt` every time it needs a new sample. The sample rate of the ABDAC is the rate of the clock source divided by 256. [2] In our case, the sample rate is $\frac{20000000Hz}{256} = 78125Hz$.

2.3.2 Supplying Sound Samples

To begin with, we only supplied random data for the sample values. This allowed us to verify a working ABDAC before implementing sounds. Afterwards, we moved on to implementing sinusoidal sound waves – sound waves based on sine values.

Experimentation showed us that evaluating the `sin` function in the interrupt routine was too slow. An initial attempt at alleviating this, was to use a lookup table for the sine values. However, this optimization was also insufficient. The

interrupt routine would still need to do multiplications and divisions to get the appropriate sample, and this was too slow.

Our solution is to *precalculate* all of the samples needed for a cycle/wavelength of the current sound wave. These samples are put in a *sound buffer*. The interrupt routine only needs to maintain a step counter, read into the sound buffer correspondingly and return these values. The interrupt routine is left as a quick buffer lookup, and the code for generating specific sounds can be put elsewhere.

The function `snd_note_buffer(frequency)` is used to generate a sample buffer for a sinusoidal sound wave of the given frequency. The function `snd_replace_buffer` is used to change the currently playing sample buffer.

2.4 Playing Melodies

Playing melodies consists of keeping track of the playtime of the currently playing note and then changing to the next note when it is time to do that. This is implemented in `mel_play_loop`. The steps performed by `mel_play_loop` are as follows:

1. If the current melody is done, wait until a new melody is selected (busy waiting).
2. Create a sample buffer for the next note.
3. Change the playback to the new sample buffer (call `snd_replace_buffer`).
4. Reset the step counter (timer) for the current note.
5. Wait for current note to finish (busy waiting).
6. Turn off the LEDs if the melody finished playing.
7. Loop back to step 1 unconditionally.

`mel_play_loop` is an infinite loop entered by a call from `main`. Whereas time efficiency was an important issue in the sample interrupt routine, `mel_play_loop` does not have to be as fast. The sample interrupt routine will keep getting samples from the old sample buffer, until generation of the new buffer is done.

A challenge was to generate a sample buffer to represent silence in a melody. As detailed in the Results and Tests chapter, several attempts at generating silence resulted in noisy sounds from the ABDAC. This is an issue that we did not manage to solve in the final program.

2.5 Ideas for Improvement

We have identified several areas where improvements could be made.

Eliminating busy waiting. Currently the CPU spins in busy wait loops when waiting for the melody to progress. This consumes energy, and could be improved by putting the processor to sleep between calls to the interrupt routines.

Using DMA for sample transfer. In our program, the ABDAC interrupts the CPU over 78000 times a second to get new samples. The DMA controller could be set up instead to transfer samples to the ABDAC when needed, leaving the CPU time to do other tasks.

Using the external DAC. The internal DAC generates noise when outputting silent sound waves. To alleviate this, we could switch to using the board's external DAC instead, which is of higher quality. [3]

3 Results and tests

Testing of the sound production was performed in successive stages, as we refined the code. Tests consisted of uploading the code and listening to the result, using the headphones. One of the group members is a hobby musician with relative pitch and performed the more advanced listening tests.

Prerequisites for testing

- Code compiles and is uploaded to microcontroller.
- Headphones in jack socket.
- For pitch-specific tests, a person easily able to distinguish between frequencies (relative pitch or absolute pitch).
- Trusted sound generator for pitch comparison to absolute frequencies (we used Audacity 1.3 Beta).
- Standard pitch is Concert A, 440Hz.
- 12-tone equal temperament is used, giving a frequency ratio of $\sqrt[12]{2}$

Terms and expressions used in tables

- "Reasonably clear note" denotes a clear tone with only light noise on top. Some light noise is to be expected due to our hardware, and is ignored for testing purposes, as there is no way to change this.
- "Noise" refers to significant noise, such as low buzzes, high beeps, and other noise that goes beyond the expected light noise.
- "Consistent" means that the pitch of the output and the pitch of the trusted sample sound are indistinguishable.

Our first implementation used a random number generator as input.

Test 1

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have noise?	Yes, given our random input.	Yes.
Do we have a reasonably clear note?	No.	No. We did not have a clear note at all, as expected.

After this, we tried implementing a sine wave directly with a frequency of 440 Hz.

Test 2

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	No. *Test failed.*
Do we have noise beyond this?	No.	No.
Do we have a pitch consistent with A440?	Yes.	No. *Test failed.*

As a result of this test, we moved the sine calculation out of the interrupt handler, and into a sine lookup table.

Test 3

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	No.
Do we have a pitch consistent with A440?	Yes.	No. *Test failed.*

However, due to ending up with the wrong tone we suspected that our interrupt handler still took up too much time, and that samples were not delivered sufficiently fast.

We tested this by successively dividing the rate by 2, 4, and 8. Seeing as this was the only difference in our code, we only compared the resulting pitch with 440 Hz.

Rate divisor	Consistent with A440
2	No
4	No
8	Yes

We concluded that our interrupt handler took too much time by a factor of approximately eight. As a result of this, we realised that our method was still too slow, and decided to change the sample generation method to a sound buffer. The buffer contains pre-calculated samples of the sound wave, as seen in section 2.3.2.

Test 4

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	No.
Do we have a pitch consistent with A440?	Yes.	Yes.

We were now able to produce a note given a certain frequency. We continued by implementing octaves. Octaves are half or double the frequency, and are perceived by the human ear to be alike. This makes them easy to test for. We used the buttons to control this. Pressing the left button (button 3) would make the function double the frequency (up one octave), and pressing the right button (button 1) would halve the frequency (going down one octave).

Test 5

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	No.
Do the buttons react to pressing?	Yes.	Yes.
Does the left button shift the pitch up?	Yes.	Yes.
Does the right button shift the pitch down?	Yes.	Yes.
At button press, is the pitch change up or down consistent to octaves?	Yes.	Yes.

We then implemented the chromatic scale using the standard semitone frequency ratio $\sqrt[12]{2}$ and using 440 Hz as a reference point, binding the first eight semitones starting from C4 to the eight buttons, in theory giving us a range of C4 to G4. If correct, the pitch would increase by a semitone for each higher button pressed, and the buttons for C (1), E (5), and G (8) would give us a sequence called a major triad (in this case a C major triad). This sequence is very easily identifiable even to non-musicians, as it is found in most popular music.

Test 6

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	No.
Do the buttons react to pressing?	Yes.	Yes.
Does a successive pressing of buttons 1-8 play the correct notes?	Yes.	No. *Test failed.*
Does the C major triad play correctly?	Yes.	No. *Test failed.*

These results puzzled us. However, double-checking the code showed that we had indeed used the correct formula for the pitch, and we decided to test it again by implementing note sequence arrays (melodies), instead of directly binding the tones to the buttons. For melodies, we chose simple, well-known children's songs so that tone errors would be easily identifiable.

Test 7

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	No.
Do the buttons react to pressing?	Yes.	Yes.
Do the melodies perform in tune as coded with pitch references?	Yes.	Yes.

Our test established that the fault must have been somewhere else, and we continued implementing melodies and note sequences as planned. We decided that we also needed to find a way to make silences, as our melodies needed short pauses between certain notes.

Attempted solution	Result
Very low amplitude.	Faint crackling noise, buzzing and beeping.
Very high frequency.	Loud buzz.
Gradually lowering the amplitude.	Faint crackling noise, buzzing and beeping. Gradual fade-out of tone.

We have not been able to eliminate this noise, but eventually ended up choosing to use a very low amplitude, as this noise was the least annoying. This was implemented by using a buffer with one element of value 0. Having decided this, we implemented "Itsy Bitsy Spider" with pauses as our introduction melody, along with a few other sequences for a win, firing, and a hit.

Test 8

Test	Expected outcome	Observed outcome
Do we have sound?	Yes.	Yes.
Do we have a reasonably clear note?	Yes.	Yes.
Do we have noise beyond this?	No.	Yes, we did, during the recently implemented silences.
Reasonably clear note?	Yes.	Yes.
Do the buttons react to pressing?	Yes.	Yes.
Do the melodies perform in tune as coded with pitch references?	Yes.	Yes.

4 Evaluation of the assignment

This was the first time any of the group members have programmed in C directly on a processor without any operating system in between, and it was educational to experience problems we have not encountered in other environments. Issues such as declaring variables volatile to ensure they are written to memory was one of them, and how interrupts are enabled and set was another. It was also interesting to experience and solve actual issues with real time constraints, such as not being able to calculate sine values at runtime.

However, not every part of this task was as rewarding. It was frustrating to spend a lot of time on issues we later found out were hardware limitations. We found this to be more annoying than educational. However, this is an issue we now will be more familiar with in the future.

It was also hard to debug sound errors, mostly because we did not know where the error occured – was it a mathematical error, issues with real time constraints or issues with hardware limitations? None of the group members have had to deal with the two latter problems before, and thus we had to think outside our normal debug scope. This was however very educational and gave us insight in issues and errors one will have to consider in embedded systems.

5 Conclusion

We completed the assignment with the recommended approach given in the compendium. By doing this, we learnt how programming a microcontroller in C is done, as well as doing interrupt handling and memory-based I/O on an AVR32. In addition, we decided to use a sine wave to generate sound, which required more work than any of the other waves we could have used instead. By doing this, we learnt how to bypass computation heavy sound generation by either pregenerating a single period of a tone just before it is needed, or by pregenerating all tones used at the start of the program.

As we were not completely satisfied with the noise generated during "silence", we tested out many different ways to try to solve the issue. However, though we were unable to solve the problem, we found a plausible reason for the noise, as well as a possible solution. Unfortunately, we did not have time enough to try out this solution, as it required a major refactoring.

References

[1] Atmel, *AVR32 Architecture Document*, Atmel Corporation, 2011.

[2] Atmel, *AT32AP7000 Reference*, Atmel Corporation, 2009.

[3] NTNU, *Lab Assignments in TDT4258 Microcontroller System Design*, Computer Architecture and Design Group, Department of Computer and Information Science, Norwegian University of Science and Technology, 2011.