1 Introduction

In this lab you will practice performance debugging techniques for embedded systems by further exploring the audio processing program from lab02. Specifically, in this lab you will determine system limitations, estimate performance by counting instructions, and measure performance with an oscilloscope / logic analyzer.

1.1 Timing Matters

Recall from lab02, we have created a system that samples analog audio signals and recreates equivalent analog signals in a pulse-width modulated (PWM) waveform. Also, recall that we must satisfy the Nyquist theorem for sampling audio signals. The human audible range is less than 20 KHz, so we must sample at more than twice that frequency. We must also output the PWM square wave at a high frequency to avoid hearing the rich harmonics from the square wave in the audible spectrum.

Meanwhile, we want to maximize the resolution of the analog-to-digital (ADC) and digital-to-analog (DAC) conversions to reduce quantization errors. The ADC on the AT91SAM7L can repeatedly sample with 10-bit resolution at 460 Ksamples/sec or 8-bit resolution at 660 Ksamples/sec. The resolution of the DAC, which is implemented using PWM, is controlled by the number of possible duty cycles between 0% and 100%. For example, two bits of resolution would offer only four possible duty cycles, 0%, 33%, 66%, and 100%. Resolution of the duty cycle directly relates to the resolution of the resulting analog output, but to obtain more resolution in the duty cycle, we must increase the number of PWM clock cycles per PWM output sample.

For example, to obtain a 10-bit input /10-bit output conversion from ADC to PWM, we must provide 1024 PWM clock cycles for each PWM output sample. In order to satisfy Nyquist's Theorem, we must sample at 44.1 KHz. Therefore, the speed of the PWM Clock must be greater than 45.1584 MHz (44.1 KHz * 1024). Unfortunately, the PWM clock cannot exceed the system clock, and the system clock is limited to 36 MHz. Therefore, we must try to maximize the resolution while not exceeding this limit.

Yet another limitation of the PWM is its limited PWM Clock resolution. The PWM Clock is derived from the Master Clock (MCLK) through a process of dividing the MCLK by a selected power of two and dividing the resulting quotient again by a selected value between 1 and 255 (See Section 32.5.1 PWM Clock Generator in AT91SAM7L128/64 Preliminary document). This typical clock generation scheme leaves us with very little resolution in the upper frequencies of operation, and we may have to settle with a different, but satisfactory, sampling frequency. The following table summarizes the timing goals and challenges of this system.

Table 1: Timing Matters Summary

Goals	**Limitations**
Satisfy NyQuist Theorem – Sample > 40KHz	Master Clock (MCLK) <= 36MHz
Hide PWM Harmonics	PWM Clock <= MCLK
Maximize resolutions for ADC and duty cycle	PWM Clock = max_duty_cycle * $f_{Sampling}$
	Poor Resolution of PWM Clock Frequency
	ADC 10-bit resolution at 460 Ksamples/sec

Using known frequencies and clock speeds, we can calculate values for maximum delays that remain consistent with the strict audio timing requirements. Next we will analyze instructions themselves to ensure that we write software that remains beneath our timing limits.

1.2 Making Sense of Instruction Counting on the ARM

One way of estimating the delay of a block of code is to analyze each instruction and to look up the delay of each instruction. Because the speed of execution is dependent on the master clock, the delays are calculated in terms of clock cycles. Idealistically, one could accumulate number of clock cycles consumed by a block of code and calculate the execution time. However, counting instructions does not consider architectural complexities such as the pipeline and the cache.

ARM has provided a method for calculating the incremental cycles for each instruction, which estimates the effect of the 3-stage pipeline. Refer to the *ARM7TDMI Technical Reference Manual*, **Chapter 3.3 Bus cycle types** and **Chapter 6 Instruction Cycle Timings (specifically 6.1 and 6.20)**. In summary, ARM considers four different types of bus cycle:

N-cycle, a non-sequential cycle
S-cycle, a sequential cycle
I-cycle, an internal cycle
C-cycle, a coprocessor cycle

Table 6-23 ARM instruction speed summary shows us the incremental number of N, S, I, and C cycles for many instructions under various conditions. Summing the number of N, S, I, and C cycles can show us a more accurate estimation of the software execution time.

1.3 Debugging with Parallel I/O Outputs

To test our estimations, we need to reliably know when a specific piece of code begins executing and when it ends without drastically disturbing the timing of the program (or being intrusive). One of the most reliable techniques for analyzing program state is to change the voltages of individually accessible I/O pins according to where the program is executing. The overhead for setting and clearing a pin is low and the output of the pins can be viewed with an oscilloscope or logic analyzer.

1.4 Setting and Clearing Pins with the AT91SAM7L

The AT91SAM7L has three parallel I/O controllers: PIOA, PIOB, and PIOC. Additionally, the pins of PIOC have been routed to the J6 header on the AT91SAM7L-STK. Each pin is individually configurable to be set as an output pin; however, care must be taken not to use a pin for debugging if it is used by the program. For example, we do not want to modify the configuration of the PWM pin located at pin 8 of PIOC. Read the section **3.11 PIO Usage** in the **AT91SAM7L-STK Rev. A Starter Kit User Guide** and read **27. Parallel Input Output Controller (PIO)** in the **AT91SAM7L128/64 Preliminary** document.

Furthermore, Atmel has created an Application programming interface (API) to make setting and clearing individual bits simple. Read the definition for the Pin structure at the bottom of pio.h (in the peripherals). Each pin is individually accessible and configurable by using the Pin structure. Each pin object contains five parts: bitmask, PIO controller, peripheral ID of PIO controller, pin type, and pin attributes. We will configure our debugging pins to be on PIOC as output pins with an internal pull-up resistor. See the lab source code for an example of how to use the pin API.

2 Lab Procedure

2.1 Estimate ISR Limits

From lab02 we know that the PWM output and ADC input is handled by an interrupt, and the software inside of the PWM interrupt service routine is executed at a rate defined by our PWM Frequency. Assuming the PWM Frequency is 44.1KHz, how much time does the ISR have to execute all of its code before another interrupt occurs during the ISR of the current interrupt (i.e. the interrupt trips on itself). Approximately how many clock cycles is that?

2.2 Download and Run lab03 Project

1. Unzip the lab03 source code from the lab03.zip file attached with this lab assignment. Note the similarities between lab03 and the finished source code from lab02.
2. Use IAR Embedded Workbench IDE to make, download, and run the program.

2.3 View the Disassembly

1. From within the IAR EWARM debugging environment (after clicking Download and Debug), set a breakpoint for the beginning of the PWM ISR.
2. Execute until you hit the breakpoint
3. Open the Disassembly view by navigating to View → Disassembly.
   The Disassembly window shows the assembly instructions overlaid with the corresponding lines of C code. This assembly code is received by a disassembler which reads in binary machine code and reconstructs its assembly instructions.
4. Find the segment of assembly code that executes during an ISR.

2.4 Look up the instructions

1. Using the method found in the *ARM7TDMI Technical Reference Manual*, create a table that contains each instruction, its corresponding incremental cycle counts, and the totals for each cycle type. Include only those instructions found in the PWM ISR.
2. Interpret the table and convert the ISR cycles into a rough execution time in seconds.

2.5 Implement Pin I/O and test

1. Write code to implement a debugging pin that you use to measure the execution time of the ISR.
2. Have the pin voltage go high upon entering the ISR and go low upon leaving the ISR.
3. Measure the time each ISR takes to execute on an oscilloscope. How near or far were your estimations?
4. Measure the frequency and period of the PWM interrupt.
5. Calculate the slack (amount of time per interrupt period where the ISR is not executing)
6. What happens when you decrease the PWM_Frequency?
7. What happens when you decrease the MAX_Duty_Cycle?
8. Demonstrate your work to the TA and place the answers to these questions in the lab report.

TA Initial Lab03 Box 1

2.6 Implement another Pin I/O

1. Implement another pin as output for use in the main function.
2. Have the pin toggle on and off as fast as it can from within the infinite while loop in the main function. This pin will simulate a process in the foreground.
3. What characteristic do you observe with the toggling pin in the foreground when the PWM interrupt is enabled?

2.7 PWM Frequency Error

1. Measure the actual output frequency of the PWM waveform. Setting the duty cycle to 50% helps to restrict the waveform, allowing you to measure it more accurately.
2. Go to the definition of PWMC_ConfigureClocks to understand how the PWM Clock is derived from the MCLK.
3. Demonstrate to a TA your understanding of why the PWM Clock may not exactly equal the value set to PWM_FREQUENCY.

TA Initial Lab03 Box 2

3 Lab Report

Your lab report for lab03 should include answers to the following questions:

1. From section 2.1: Assuming the PWM Frequency is 44.1KHz, how much time does the ISR have to execute all of its code before tripping on itself (another interrupt occurs during the ISR of a previous interrupt). Approximately how many clock cycles is that?

2. Provide the ISR cycle table from Section 2.4. Include interpretation of table and estimation for execution time in seconds. What effect would increasing the value of SUBSAMPLES have on your estimations?

3. From Section 2.5, provide your measurements and answers to the listed questions.

4. From Section 2.6, what characteristic do you observe with the toggling pin in the foreground when the PWM interrupt is enabled?

5. From Section 2.7, why is the PWM frequency not exactly 44.1KHz when we set PWM_FREQUENCY to 44100?

6. Compare your cycle counting estimate to actual measured delay in terms of both cycles and seconds. Explain your findings in detail.