1 Introduction

In this lab you will learn debugging techniques for embedded systems while implementing a program that performs digital audio processing on the AT91SAM7L microcontroller. Audio signals will be input with a 10-bit Analog-to-Digital Converter (ADC) and output with Pulse Width Modulation (PWM) operating as a Digital-to-Analog Converter (DAC).

1.1 Analog-to-Digital

In order to process an analog audio stream, an embedded system must first convert the received analog signals from continuous voltages to digital values at a specified sampling frequency. In order to perfectly reconstruct an analog signal, the ADC must sample the audio source at a rate that satisfies the Nyquist Sampling Theorem. The Nyquist Theorem dictates that the minimum sampling frequency is at least twice the highest frequency of the analog source. Since we are working with audio signals for human consumption and our maximum audible frequency is approximately 20kHz, we will require an ADC sampling frequency of at least 40kHz to accurately reproduce the audio signal.

The ADC logic employed by the AT91SAM7L is based on the Successive-Approximation-Register (SAR) architecture. The SAR architecture encodes an analog voltage into a binary value through a repetitive, binary-search process. In each step of the process a test analog signal is created internally and compared against the input analog signal. The test signal is adjusted until it most closely represents the input signal, and then the binary representation of the test signal is representative of the original analog signal. See how the analog test signal successively converges to the input analog signal in the figure below.

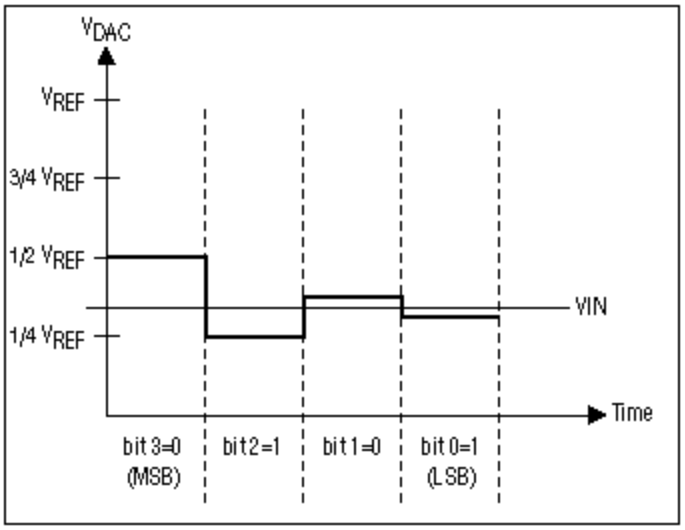

Figure 1: Successive Approximation Register Operation

The operation of the ADC for the AT91SAM7L is well documented in the **AT91SAM7L128/64 Preliminary** Datasheet (Section 33. Analog-to-Digital Converter). In summary, the AT91SAM7L offers 10-bits of precision and four simultaneous analog inputs. With 10 bits of precision analog signals are represented in the decimal range 0 to 1023. All conversions are relative to the ADVREF voltage, which is hard wired to VDDOUT, or the voltage currently supplied by the batteries. Any analog voltage greater than VDDOUT saturates to 1023.

1.2 Pulse Width Modulation

In embedded systems, we often need to generate analog signals for certain I/O devices. Many embedded systems come equipped with a dedicated Digital to Analog Converter (DAC), however the AT91SAM7L does not include this feature, and an alternative method is required for generating analog signals.

Pulse Width Modulation (PWM) is a technique for creating analog signals by toggling a digital output between VDD and GND, such that the average voltage over the sum of one PWM period is the desired analog output. The duty cycle represents the percentage of each PWM period where the digital output voltage high. Therefore, the duty cycle directly controls the resulting analog output voltage. See the figures below for an illustrative example.

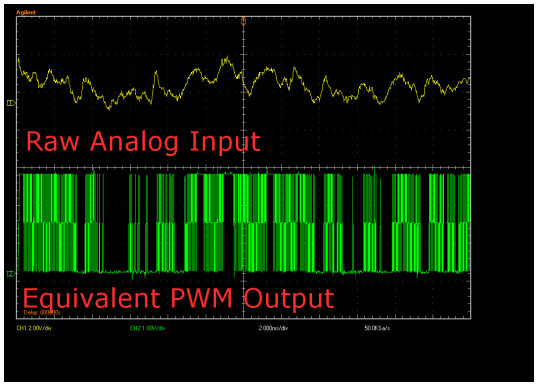

Figure 2: An analog signal is sampled and reproduced with PWM

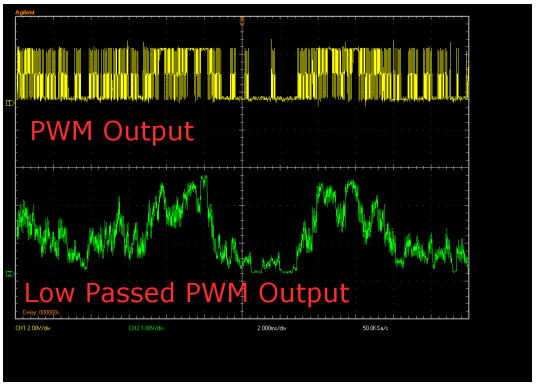

Figure 3: Filtering the PWM signal with a low-pass filter nearly recovers the analog signal.

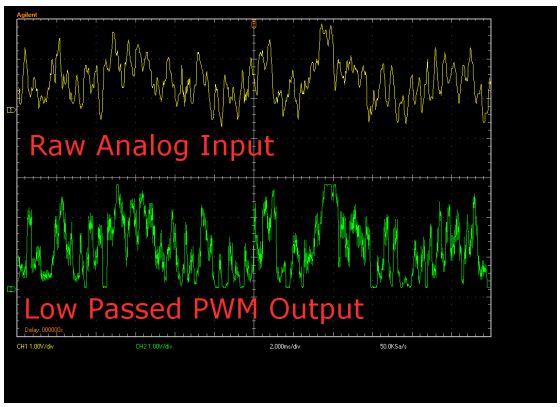

Figure 4: Comparing the raw analog signal to the filtered PWM signal.

To recover the original signal, a low-pass filter can remove most of the high frequency data from the PWM signal. Square waves, like the PWM signal, generate rich harmonics in the frequency domain, which can be difficult to suppress. However if we output PWM samples at a faster frequency, the harmonics become farther away from the baseband signal and are therefore easier to filter. Therefore, many applications employ PWM frequencies in the megahertz range. The PWM clock for the AT91SAM7L can go as fast as the Master Clock (36 MHz).

This lab uses a passive, first-order low-pass filter (see figure below), but more advanced filters, such as a multi-order RC filter or a synchronized integrator filter can recover the signal more accurately.

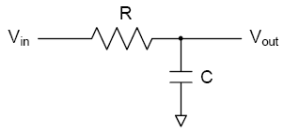

Figure 5: First Order Passive Low-Pass Filter

The operation of the PWM for the AT91SAM7L is well documented in the **AT91SAM7L128/64 Preliminary** Datasheet (Section 32. Pulse Width Modulation Controller). In summary, the PWM peripheral contains four PWM channels of 16-bit of precision with configurable output frequencies, duty cycles, and alignment/polarity.

2 Lab Procedure

2.1 Download and Run lab02 project

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

2.2 View the PWM wave on the oscilloscope

1. Print the Pin Mapping for the 40-pin header on the AT91SAM7L-STK, cut it to size and place over the J6 header on the board with pin 1 on the paper aligned with pin 1 on the board.
2. Observe the PWM signal emitted from port PWM2/SCK0 (pin 9 on J6) on the oscilloscope.
3. Vary the value of PWM_FREQUENCY from 176400 to 88200. What effect do you see? What about when PWM_FREQUENCY = 44100?

2.3 Low-pass Filter the PWM output

1. Using a protoboard, wires, a 750Ω resistor, and a 0.01μF capacitor, construct a passive, first-order low-pass filter for the PWM output (corner frequency 21,231Hz).
2. Observe the filtered PWM signal on the oscilloscope.
3. Here we use controlled inputs (sine wave, square wave) as a debugging technique so that we can isolate any bugs specific to the PWM operation.

2.4 Breakpoints and the Watch Window

1. Breakpoints are an essential tool in embedded debugging.
2. Make sure the ISR_PWM routine is actually being visited by the processor by setting a breakpoint somewhere within the ISR.
3. Execute the program until the breakpoint halts the processor. If the processor never halts, there is likely a configuration bug in the PWM initialization routine.
4. Once a breakpoint has halted the processor, you can add variables to the watch window (Right click on a variable in the scope of the breakpoint and click "Add to Watch")
5. Once a variable is in the Watch Window, you can view and modify its current value by changing the corresponding value cell in the watch window.
6. Execution can resume after a breakpoint, and the modified variable will contain the new value until changed either by code execution or user interference.
7. Without recompiling the source code use breakpoints and the watch window to switch between outputting a sine wave and a square wave to the PWM channel.
8. Show the TA your ability switch back and forth between sine and square waves.

TA Initial Lab02 Box 1

2.5 Implement Analog-to-Digital Converter

1. Another effective debugging technique is to isolate different pieces of the program independently from the rest of the program. We are now going to focus on just the Analog to Digital conversion process, and should therefore disable the PWM features of the program.
2. Comment out the line that enables the PWM interrupt in the Advanced Interrupt Controller (AIC) so that the ISR_PWM routine is never executed.

   ```
   //   AIC_EnableIT(AT91C_ID_PWMC); // ISR_PWM never executed
   ```

3. The ADC peripheral of the AT91SAM7L is controlled by writing to the ADC registers defined in the **AT91SAM7L128/64 Preliminary** Datasheet (Section 33. Analog-to-Digital Converter).
4. Copy and paste the following code into the section of main.c labeled "Insert ADC Initialization Below Here"

   ```
   /* Insert ADC Initialization Below Here*/
   //ADC Initialization:
   //Configure ADC Mode Register
   //see AT91SAM7L128/64 Preliminary Datasheet Section 33.6.2 'ADC Mode Register'
   AT91C_BASE_ADC->ADC_MR =
           0<<0 | // hardware triger enable 0/1
           0<<1 | // triger source selection (0-6)
           0<<4 | // enable low resolution (8 bit) 0/1
           0<<5 | // use sleep mode (auto. wakeup on conversion 1mA -> 1uA) 0/1
           4<<8 | // ADC clk prescaler ADC_clk = MCI/2 / (prescaler+1)
           // ADC clk max: 5 MHz for 10 bit; 8 MHz for 8 bit
   ```

```
        // prescaler = MCI/2/ADC_clk - 1 = 3.7923 (round up to 4)
        // --> ADC_clk = 4.7923 MHz
        11<<16 | // STARTUP: Start up Time (hardware needs max 20us)
        // time = (STARTUP+1)*8 ADC_clk cycles
        2<<24; // SHTIM: track and hold time (min. 600ns)
        // time = (SHTIM+1) ADC_clk cycles
    // configure channel enable register
    AT91C_BASE_ADC->ADC_CHER = 1<<3; // enable channel 3
    AT91C_BASE_ADC->ADC_CR = 1<<1; // start a new conversion
    /* Insert ADC Initialization Above Here*/
```

5. Copy and paste the following code into the section of main.c labeled "Insert ADC Foreground Below Here"

```
/* Insert ADC Foreground Below Here */
// Wait until Analog Conversion completes
while(AT91C_BASE_ADC->ADC_SR & (1<<3) != 0);
adcValue = AT91C_BASE_ADC->ADC_LCDR; // read result of last conversion
AT91C_BASE_ADC->ADC_CR = 1<<1; // start a new conversion
printf("%x \n\r", adcValue);
/* Insert ADC Foreground Above Here*/
```

6. To debug and verify the ADC logic, we can use one of the most useful debugging tools, the print statement. Because we have configured the DBGU to send all print statements to through the serial port, we can view the contents of each print statement in HyperTerminal.
7. Print statements are an intrusive form of debugging because they consume several instruction cycles, but printing out the values of variables to HyperTerminal can quickly reveal bugs.
8. Open and configure HyperTerminal for serial communication with the AT91SAM7L.
9. Compile and run the program with ADC code added.
10. You should see several values for adcValue scrolling across the HyperTerminal.

2.6 Debug/ Verify ADC

1. The ADC initialization code initializes the AD3 input (pin 35 on the J6 header) as the only analog input.
2. Wire a potentiometer to the AD3 input so that an analog voltage between VCC and GND arrives at the AD3 pin. A multimeter will be necessary to determine how to wire the three terminals of the potentiometer so that it creates a voltage divider.
3. Run the program and observe the HyperTerminal output as you vary the analog voltage supplied to the AD3 pin.
4. Use a breakpoint to observe the value of adcValue in the debugging environment.
5. Demonstrate your progress to a TA.

TA Initial Lab02 Box 2

2.7 Move the ADC to the ISR

1. Once the ADC has been proven to work, we can move the ADC conversion into the ISR_PWM interrupt service routine and re-enable the PWM Interrupt.
2. Remove or comment the "ADC Foreground" from within the infinite while loop. Also, remove or comment the section of the ISR_PWM() that sets duty to either SIN64[count] or SQUARE64.
3. Copy and paste the following section of code into the section of ISR_PWM() labeled "Insert ADC ISR Code Below Here"

```
/* Insert ADC ISR Code Below Here*/
// if Analog conversion ready
if(count >= SUBSAMPLES && (AT91C_BASE_ADC->ADC_SR & (1<<3))) {
        adcValue = AT91C_BASE_ADC->ADC_LCDR; // read result of last conversion
        duty = adcValue>>3; // Right Shift by 3 converts 10-bit into 7-bit
        AT91C_BASE_ADC->ADC_CR = 1<<1; // start a new conversion
        count = 0;
}
count ++;
/* Insert ADC ISR Code Above Here*/
```

The new ISR_PWM should look identical to the below code:

```
static void ISR_PWM(void)
{
static unsigned int count = 0;
static unsigned int duty = MAX_DUTY_CYCLE >> 1;
// If this interrupt was triggered by PWM channel 1
if ((AT91C_BASE_PWMC->PWMC_ISR & AT91C_PWMC_CHID1) ==
AT91C_PWMC_CHID1) {
/* Insert ADC ISR Code Below Here*/
// if Analog conversion ready
if(count >= SUBSAMPLES && (AT91C_BASE_ADC->ADC_SR & (1<<3))) {
adcValue = AT91C_BASE_ADC->ADC_LCDR; // read result of last conversion
duty = adcValue>>3; // Right Shift by 3 converts 10-bit into 7-bit
AT91C_BASE_ADC->ADC_CR = 1<<1; // start a new conversion
count = 0;
}
count ++;
/* Insert ADC ISR Code Above Here*/
// Set new duty cycle
PWMC_SetDutyCycle(1, duty);
PWMC_SetDutyCycle(2, duty);
}}
```

4. Uncomment the AIC_EnableIT(AT91C_ID_PWMC); function call to re-enable interrupts.
5. Debug the new code using breakpoints and the watch window.

2.8 Music Testing

1. Wire the analog input pin to a streaming music source (ipod, PC audio out, etc.)
2. Use the provided audio jacks to connect the music source to the microcontroller.
3. Observe the raw analog signal with the PWM signal and the filtered PWM signal on an oscilloscope.
4. Listen to the quality of the music and modify the following variables: SUBSAMPLES, PWM_FREQUENCY, MAX_DUTY_CYCLE.
5. What observations can you make that relate the value of the each variable to the perceived quality of music?
6. Compare the FFT of the raw analog input verses the PWM output verses the filtered PWM output. What do you notice different among all three?
7. Demonstrate your music player to the TA.

TA Initial Lab02 Box 3

3 Lab Report

Your lab report for Lab 1 should include answers to the following questions:

1. List four debugging techniques exercised in this lab.
2. What differences do you observe in the FFT of the raw analog input verses the PWM output verses the filtered PWM output?
3. What is the best sounding combination of SUBSAMPLES, PWM_FREQUENCY, and MAX_DUTY_CYCLE? What kind of measurements could you take to objectively classify quality?

4 References

Maximum Audible Frequency - http://hypertextbook.com/facts/2003/ChrisDAmbrose.shtml
SAR - http://www.maxim-ic.com/appnotes.cfm/appnote_number/1080/
PWM - http://focus.ti.com/lit/an/spraa88/spraa88.pdf