ECE 3849 B2022

Real-Time Embedded Systems

Lab 3: Advanced I/O

Professor Stander

William Tyrrell

12/16/22

Introduction:

In this lab we will be adding a frequency counter and audio output functionality to the lab 2 oscilloscope. Much of the same structure using the RTOS and tasks remain. I will also optimize the ADC I/O using DMA and bump the sampling rate to 2 Msps. I will subsequently measure the CPU load using the code written in lab 1 to measure the performance difference of using the old sampling method from lab 2 and the new one using DMA. I was able to complete all these tasks within this lab.

Discussion and Results:

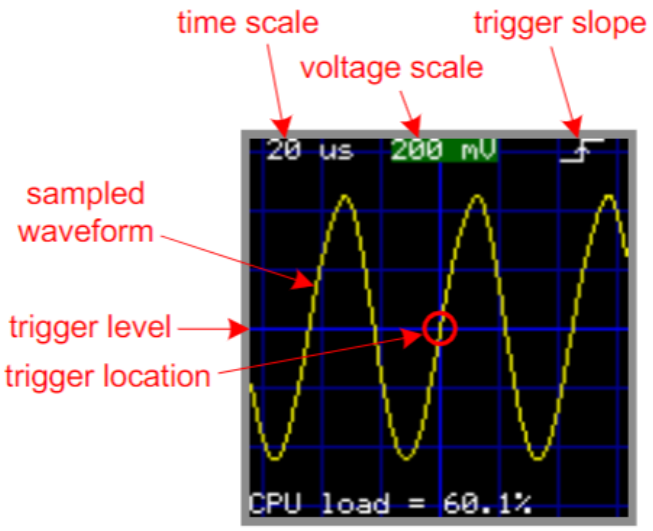

Figure 1: Sample Screen for Oscilloscope

Challenge #1: ADC Optimization:

ADC Configuration	Sampling Rate	CPU Load	ISR Relative Deadline
Single-sample ISR	1 Msps	76.3%	1 usec
DMA	1 Msps	2.1%	1.024 msec
DMA	2 Msps	2.3%	512 usec

Figure 2: CPU Load Chart

Single Sample ISR Deadline = 1sec/Sampling rate of 1 Msps = 1 usec

DMA Deadline = Channel Size / Sampling Rate

DMA 1 Msps = 1024 items / 1 Msps = 1.024 msec

DMA 2 Msps = 1029 items / 2 Msps = 512 usec

The first step in this lab was to make a copy of Lab 2 code and implement the CPU Load function from lab 1. This code is used to measure the CPU Load of 76.3% for the Single Sample ISR implementation of the ADC.

To lower the CPU Load percentage, we will utilize DMA. The Direct Memory Access (DMA) controller can perform most of the duties of the Lab 2 ADC ISR without using any CPU time. An ISR is still required, but the relative deadline is nowhere near as short. A DMA interrupt occurs only when a long DMA transfer completes, not every ADC sample. Figure 3 shows the initialization for the DMA.

```
void DMA_init(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    uDMAEnable();
    uDMAControlBaseSet(gDMAControlTable);
    uDMAChannelAssign(UDMA_CH24_ADC1_0); // assign DMA channel 24 to ADC1 sequence 0
    uDMAChannelAttributeDisable(UDMA_SEC_CHANNEL_ADC10, UDMA_ATTR_ALL);
    // primary DMA channel = first half of the ADC buffer
    uDMAChannelControlSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT,
                          UDMA_SIZE_16 | UDMA_SRC_INC_NONE | UDMA_DST_INC_16 | UDMA_ARB_4);
    uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT,
                           UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                           (void*)&gADCBuffer[0], ADC_BUFFER_SIZE/2);
    // alternate DMA channel = second half of the ADC buffer
    uDMAChannelControlSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT,
                          UDMA_SIZE_16 | UDMA_SRC_INC_NONE | UDMA_DST_INC_16 | UDMA_ARB_4);
    uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT,
                           UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                           (void*)&gADCBuffer[ADC_BUFFER_SIZE/2], ADC_BUFFER_SIZE/2);
    uDMAChannelEnable(UDMA_SEC_CHANNEL_ADC10);
}
```

Figure 3: DMA Initialization

The DMA controller configuration is as follows:

- Allocate the DMA control table in RAM (#pragma specifies alignment)
- Assign a supported DMA channel to ADC1 sequence 0
- Configure the primary and alternate DMA control blocks for this channel:
    - Source = ADC1 Sample Sequence Result FIFO 0 register (direct access)
    - Primary channel destination = first half of the ADC buffer
    - Alternate channel destination = second half of the ADC buffer
    - Data size = 16 bits (one ADC sample)
    - Source address increment = 0 (always read the same register)
    - Destination address increment = 16 bits (same as data size)
    - Arbitration size = 4 transfers
    - DMA mode = ping-pong

In ping pong mode, the DMA operates on one of the two buffers, while the CPU is supposed to access the other. Then they swap, achieving a continues transfer. Note that uDMA transfer size is limited to 1024 items. An ADC buffer size of 2048 samples the maximum possible for this simple static, where each DMA channel is assigned to a fixed buffer.

In ping-pong mode, we are configuring two DMA channels that are assumed linked. When a ping-pong DMA channel reaches the end of transfer, it starts its complementary channel, triggers an interrupt, then stops and waits for further instructions. It does not automatically reset itself for another transfer. If left alone, the DMA transfer will halt when the alternate channel finishes its transfer. Resetting the DMA channels, such that, continuous DMA is maintained, is the job of the new ADC ISR. DMA interrupts are passed on to the peripheral where the DMA transfer is occurring. Instead of the normal ADC1 sequence 0 interrupt, I enabled the ADC1 sequence 0 DMA interrupt. The next ADC ISR is shown in Figure 4.

```
void ADC_ISR(void){
    //lab 2
    //    ADC1_ISC_R = ADC_ISC_IN0; // clear ADC1 sequence0 interrupt flag in the ADCISC register
    //    if (ADC1_OSTAT_R & ADC_OSTAT_OV0) { // check for ADC FIFO overflow
    //        gADCErrors++; // count errors
    //        ADC1_OSTAT_R = ADC_OSTAT_OV0; // clear overflow condition
    //    }
    //    gADCBuffer[
    //               gADCBufferIndex = ADC_BUFFER_WRAP(gADCBufferIndex + 1)
    //               ] = ADC1_SSFIFO0_R; // read sample from the ADC1 sequence 0 FIFO

    ADCIntClearEx(ADC1_BASE, ADC_INT_DMA_SS0); // clear the ADC1 sequence 0 DMA interrupt flag
    // Check the primary DMA channel for end of transfer, and restart if needed.
    if (uDMAChannelModeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT) == UDMA_MODE_STOP) {
        uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT,
                               UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                               (void*)&gADCBuffer[0], ADC_BUFFER_SIZE/2); // restart the primary channel (same as setup)
        gDMAPrimary = false; // DMA is currently occurring in the alternate buffer
    }//end if
    // Check the alternate DMA channel for end of transfer, and restart if needed.
    // Also set the gDMAPrimary global.
    if(uDMAChannelModeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT) == UDMA_MODE_STOP) {
        uDMAChannelTransferSet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT,
                               UDMA_MODE_PINGPONG, (void*)&ADC1_SSFIFO0_R,
                               (void*)&gADCBuffer[ADC_BUFFER_SIZE/2], ADC_BUFFER_SIZE/2); // restart alternate channel
        gDMAPrimary = true;
    }//end if
    // The DMA channel may be disabled if the CPU is paused by the debugger.
    if (!uDMAChannelIsEnabled(UDMA_SEC_CHANNEL_ADC10)) {
        uDMAChannelEnable(UDMA_SEC_CHANNEL_ADC10); // re-enable the DMA channel
    }
} //end ISR
```

Figure 4: ADC ISR

The gADCBufferIndex is no longer updated by the DMA mechanism. Therefore, the trigger can't find the newest samples in the ADC buffer. This is evident by the waveform being jittery and skipping at points. Using a new function, uDMAChannelSizeGet(), we will attempt to fix this skipping. The global variable, gDMAPrimary, from the ADC_ISR tells us which channel the DMA is writing to using a RTOS GateHwi. Without the gate task we would run into non-atomic read issues. Figure 5 shows the getADCBufferIndex() function. It returns the location where the newest sample are.

```
int32_t getADCBufferIndex(void)
{
    int32_t index;
    IArg gatekey = GateHwi_enter(gateHwi0);
    if (gDMAPrimary) { // DMA is currently in the primary channel
        index = ADC_BUFFER_SIZE/2 - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT);
        GateHwi_leave(gateHwi0, gatekey);
    }
    else { // DMA is currently in the alternate channel
        index = ADC_BUFFER_SIZE - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT);
        GateHwi_leave(gateHwi0, gatekey);

    }
    return index;
}
```

Figure 5: getADCBufferIndex

We will also change the priority of the ADC Hwi to not be a zero-latency Hwi. The worst thing that can happen then is that the DMA channel we detected as currently active finishes, and its remaining transfer size reads 0. Since the other channel has just started, our estimate of the newest sample location is then only slightly off. We will not be using gADCBufferIndex variable anymore. Instead, we will put another variable to display this in the waveform task. In my case, I

named mine DMAIndex. After these modifications, we were able to take the CPU load with 1Msps and 2Msps. Figures 6 and 7 show the CPU loads for 1Msps and 2Msps, respectively.

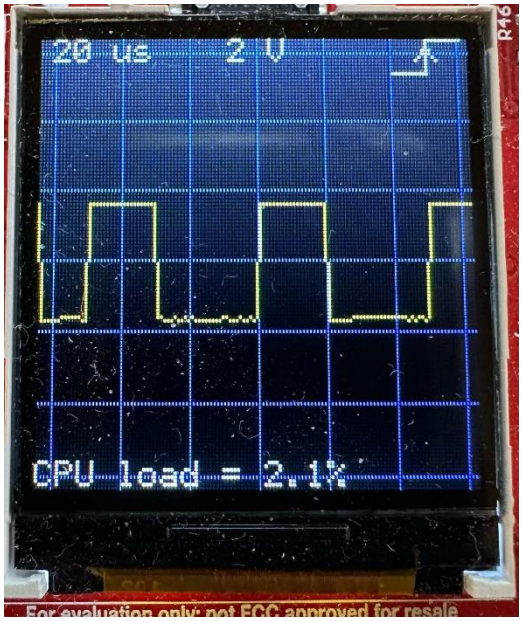

Figure 6: CPU load 1 Msps

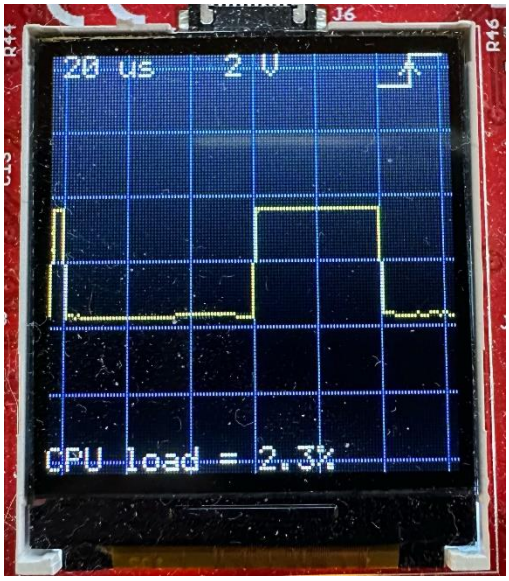

Figure 7: CPU load 2 Msps

Challenge 2: Frequency Counter

For this challenge, we will implement a simple frequency counter. We will configure a timer in Capture mode to measure the period of the input PWM signal. Note that this setup has a flaw (which we will not fix): An external signal is generating interrupts. The period of these interrupts may be uncontrolled. At a high enough input signal frequency, the Timer Capture ISR can starve lower priority tasks of the CPU. Configuring Timer0A for Edge capture mode is shown in Figure 8. In the RTOS config we will change the Timer ID to anything other than 0. Timer ID0 is in use. In my case, I made it 1.

```
// configure GPIO PD0 as timer input T0CCP0 at BoosterPack Connector #1 pin 14
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
GPIOPinTypeTimer(GPIO_PORTD_BASE, GPIO_PIN_0);
GPIOPinConfigure(GPIO_PD0_T0CCP0);

SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerDisable(TIMER0_BASE, TIMER_BOTH);
TimerConfigure(TIMER0_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_CAP_TIME_UP);
TimerControlEvent(TIMER0_BASE, TIMER_A, TIMER_EVENT_POS_EDGE);
TimerLoadSet(TIMER0_BASE, TIMER_A, 0xffff);   // use maximum load value
TimerPrescaleSet(TIMER0_BASE, TIMER_A, 0xff); // use maximum prescale value
TimerIntEnable(TIMER0_BASE, TIMER_CAPA_EVENT);
TimerEnable(TIMER0_BASE, TIMER_A);
```

Figure 8: Timer Setup

We also need to connect the PF3/M0PWM3 signal generator pin to the PD0/T0CCP0 timer input pin using a jumper wire. I also configured a Hwi to handle the Timer0A interrupts. Since this Hwi will not need to be disabled by the RTOS in any critical sections, it may be a “zero-latency Hwi.” In the Hwi function remember to clear the Timer0A Capture interrupt flag, different from the Timeout flag. Figure 9 shows the timerCapture function. Figure 10 shows the displaying of the frequency and period in Hertz and Seconds respectively. Using the TimerValueGet() function

to read the full 24-bit captured timer count (includes prescaler). Calculate the period as the difference between the current and previous captured Timer0A count, taking care of wraparound. Save the period into a global and verify that its value is what you expect using the debugger. When adjusting the period, I also updated the duty cycle of both outputs to keep it at 40%. Two PWM outputs were set up as copies of each other to make it convenient to wire one to the ADC and another to the timer input.

```
void timerCapture(void){
        // Clear the TIMER0A Capture Interrupt flag
        TIMER0_ICR_R = TIMER_ICR_CAECINT;
        current_count = TimerValueGet(TIMER0_BASE , TIMER_A);
        Period = (current_count - previous_count) & 0xffffff;
        previous_count = current_count;
}
```

Figure 9: Timer Capture Function

```
//Frequency f
snprintf(str1, sizeof(str1), "f=%5d Hz", gSystemClock/Period);
GrStringDraw(&sContext, str1, -1, 0, 120, false);
//Period T
uint32_t PWMPeriod1 = roundf((float)gSystemClock/(PWM_FREQUENCY + PWM_STEP_SIZE*PWMperiod));
snprintf(str1, sizeof(str1), "T=%4d CC", PWMPeriod1);
```

Figure 10: Display Frequency and Period

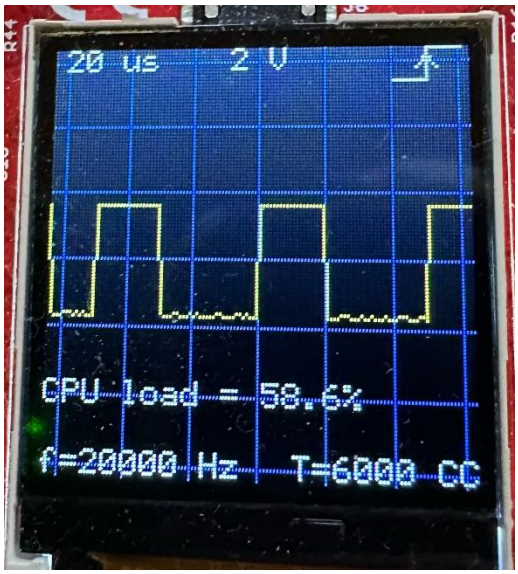

Figure 11: Frequency and Period

Challenge 3: PWM Audio Output:

In this final challenge I will be using pulse-width modulation (PWM) as a form of digital-to analog conversion and drive a speaker to produce audio. By low pass filtering a PWM signal, we obtain a voltage proportional to the PWM duty cycle. By varying the duty cycle, we can produce a low-frequency waveform, below the cutoff frequency of the low-pass filter. The given audio waveform.c and .h are inserted into the project. We need to configure the PWM signal to drive the speaker on the BoosterPack. Using Lab 1 as a template PWMinit is shown in Figure 12:

```
void PWMinit(void){
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
       GPIOPinTypePWM(GPIO_PORTG_BASE, GPIO_PIN_1); // PG1 = M0PWM5
       GPIOPinConfigure(GPIO_PG1_M0PWM5);
       GPIOPadConfigSet(GPIO_PORTG_BASE, GPIO_PIN_1, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);

       // configure the PWM0 peripheral, gen 2, outputs 5
       SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
       PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1); // use system clock without division
       PWMGenConfigure(PWM0_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
       PWMGenPeriodSet(PWM0_BASE, PWM_GEN_2, roundf((float)gSystemClock/AUDIO_PWM_FREQUENCY));   // Initialize to period of 258 clock cycle
       PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, roundf((float)gSystemClock/AUDIO_PWM_FREQUENCY*0.5f)); // Initialize to 50% duty cycle
       PWMOutputState(PWM0_BASE, PWM_OUT_5_BIT, true);
       PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_2, PWM_INT_CNT_ZERO);    // configures PWM interrupts (every time the counter reaches 0)
       PWMGenEnable(PWM0_BASE, PWM_GEN_2);
}
```

Figure 12: PWM Initialization function

I initialized the PWM0 generator 2, output 5, which drives the speaker on the BoosterPack. As well as configure the corresponding GPIO pin PG1 as M0PWM5 in the GPIOPinConfigure() call. I Initialized this PWM generator to a period of 258 clock cycles (defined a constant for this). It is the shortest period that achieves 8-bit duty cycle resolution. I will not be using the 0% and 100% duty cycle settings (to avoid nonlinearity), so the PWM period is (28 + 2) system clock cycles. This translates to a PWM frequency of 465 kHz, far above the audible range. I also set the initial duty cycle to 50%. Use the following call to configure, PWM interrupts (every time the counter reaches 0). I Did not enable the interrupt in this part of the code, adding the enable is done later in the button processing task. We need an ISR to periodically update the duty cycle to create a low-frequency waveform. I did this in the PWMISR function shown in Figure 13. In the PWM ISR the following is accomplished.

• Clear the PWM interrupt flag (you may do this using the given function call or using direct register access)

• Determine when the end of the waveform array has been reached (the length of this array is given in “audio_waveform.h”

• Calculate the sampling rate divider

```
void PWM_ISR(void)
{
    PWMGenIntClear(PWM0_BASE, PWM_GEN_2, PWM_INT_CNT_ZERO); // clear PWM interrupt flag

    int i = (gPWMSample++) / gSamplingRateDivider; // waveform sample index
    PWM0_2_CMPB_R = 1 + gWaveform[i]; // write directly to the PWM compare B register
    if (i == gWaveformSize - 1) { // if at the end of the waveform array
        PWMIntDisable(PWM0_BASE, PWM_INT_GEN_2); // disable these interrupts
        gPWMSample = 0; // reset sample index so the waveform starts from the beginning
    }
}
```

Figure 13: PWM ISR

It was found using this formula:

((gSystemClock) / (PWMPer * AUDIO_SAMPLING_RATE))

Where: gSystemClock is 120MHz

PWMPer is the period of the input waveform in clock cycles

AUDIO_SAMPLING_RATE is the sampling rate of the ADC. Specifically, 16,000 samples per second.

I also needed to configure a TI-RTOS a Hwi for the PWM generator 2 ISR. I made this a zerolatency interrupt because it must meet its deadlines.

Lastly, I added the PWMIntEnable() call to the Joystick press for the audio to be enabled. It reads out “Welcome to ECE 3849”

The CPU load percentage was a lot higher when the audio is playing back. This is shown in Figure 14.

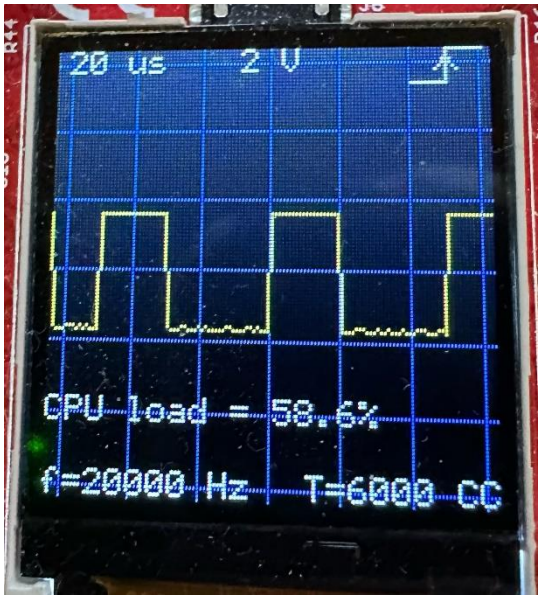

Figure 14: CPU load Percentage during Audio playback

Conclusion:

I found this lab to be straight forward considering that a large part of the code was either given or taken from lab 2. I ran into a bit of a problem trying to configure DMA so that the waveform did not jitter or studder but I able to locate a small error in the DMA configuration which was causing this. The important things that I learned from this lab were how to setup and use DMA, the effects on the CPU load percentage that this has, how to manipulate the frequency of the given square wave and calculate its period using a timer, and how audio can be outputted from the lab board. It was a valuable lab that did a good job wrapping up the course considering DMA and timers were topics we talked about towards the end of the class. I do not believe this lab needs any improvements as it demonstrated topics from class without being too difficult.