ECE 3849 D2018
Real-Time Embedded Systems
Lab 3: Advanced I/O

by

Chaiwat Ekkaewnumchai

Tianyi Xu

Box#: 366

Date:

1 May 2018

Table of Contents

Introduction

The Lab 2 has three main parts. The first part is optimizing ADC I/O using DMA and bump the sampling rate to 2 Msps. We also measured and compared CPU load at different sampling rate and different modes (DMA and normal ADC ISR).

In the second part, we measured the frequency of an analog signal, and displayed the result on the LCD screen. We first convert the analog signal to PWM signal using the analog comparator peripheral, then we measure the period of PWM using a timer in capture mode. Finally, we calculated frequency and displayed the result.

In the last part, we generated a 20 KHz sine wave using PWM output passed through a low-pass filter. We also monitored the CPU load with PWM interrupts enabled.

Discussion and Results

- System requirements and real-time goals. Assign thread types and priorities based on the top-level design.

This lab is very similar to lab2, however, we added one new task (Frequency Task), two semaphores, one clock and two HWi in this lab. The real-time goals are to be able to do six different tasks without error and share data bug. Without extra credit part, we used two clock, three Hwi, six tasks, seven semaphores, and one mailbox. Clock object is used to schedule periodic button scanning, and the Clock function can signal the Button Task and Frequency Task using a Semaphore. TI-RTOS Hwi objects are ADC_ISR, Capture_ISR and PWM_ISR. The six tasks from highest priority to lowest priority are Button Task, Waveform Task, Frequency Task, User Input Task, Display Task and Processing Task. Five of seven semaphores are used to singal and communicate between tasks, and rest two semaphores are used to protect the shared data. Mailbox is used to post and pend button ID between Button Task and User Input Task.

- Implementation of each major lab component.

The goal of the first part is to use a DMA controller to replace ADC ISR because it has a longer relative deadline, which means a lower CPU load. In order to do that, we first need to set up a DMA controller, which includes two DMA channels in ping-pong mode, and each one stores half of ADC buffer. We have to use 2 channels because uDMA transfer size is limited to 1024 items, but our ADC buffer size is 2048. The code to initialize DMA is shown below.

```
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 1. DMA Setup Code

We still need ISR when using DMA configure, but we have to enable DMA interrupts, and disable normal ADC interrupts. However, we only need comment ADC1 interrupt enable function. The ADC1 sequence enable is still necessary here.

```
ADCSequenceEnable(ADC1_BASE, 0);
//ADCIntEnable(ADC1_BASE, 0);
ADCSequenceDMAEnable(ADC1_BASE, 0); // enable DMA for ADC1 sequence 0
ADCIntEnableEx(ADC1_BASE, ADC_INT_DMA_SS0);                // enable ADC1 sequence 0 DMA interrupt
```

Figure 2. Enable DMA Interrupts Code

Then, we edited ADC_ISR function. We used uDMAChannelModeGet() function to know which DMA channel is being used, and the result is saved into a global variable gDMAPrimary. True stands for channel 1, and false stands for channel 2.

```
void ADC_ISR(void)  // DMA (lab3)
{
    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
    }

    // 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 the primary channel (same as setup)
        gDMAPrimary = true;    // DMA is currently occurring in the alternate buffer
    }

    // 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
    }
}
```

Figure 3. ADC_ISR Function Code

We used gADCBufferIndex() in the last lab to find the newest sample in the ADC buffer. However, when we used DMA, this function didn't work anymore, so trigger search part also didn't work also. The way we used to find where DMA is currently writing samples into the ADC buffer is using the uDMAChannelSizeGet() function. In the meantime, the true index was also related with the channel, since we're using ping-pong mode. So, we used if statement to determine which DMA channel was running first, then calculated the true buffer index. Moreover, because of non-atomic read issue, we used GateHwi to protect the shared data.

```
int32_t getADCBufferIndex(void)
{
    int32_t index;

    IArg key = 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);
    }
    else {              // DMA is currently in the alternate channel
        index = ADC_BUFFER_SIZE - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_ALT_SELECT);
    }
    GateHwi_leave(gateHwi0, key);
    return index;
}
```

Figure 4. Get ADC Buffer Index Code

After that, we successfully convert normal ADC to DMA configure. The last step for this part was measuring CPU load in different conditions. In order to do that, we put unloaded CPU measurement code inside the Display task function, and the loaded CPU measurement code inside the lowest priority task function. We first compared the CPU load when using single-sample ISR and DMA configuration. As shown in the table below, the DMA CPU load is much lower than that of normal ISR because DMA has a much longer relative deadline. Then we changed the DMA sampling rate to 2Msps, and we found higher sampling rate caused higher CPU load.

In addition, we also did extra credit in the end of this lab. The code to interrupt every 4 samples is shown below.

```
// 8-sample FIFO with interrupt every 4 times
ADCSequenceStepConfigure(ADC1_BASE, 0, 0, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 1, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 2, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 3, ADC_CTL_CH3 | ADC_CTL_IE);
ADCSequenceStepConfigure(ADC1_BASE, 0, 4, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 5, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 6, ADC_CTL_CH3);
ADCSequenceStepConfigure(ADC1_BASE, 0, 7, ADC_CTL_CH3 | ADC_CTL_IE | ADC_CTL_END);*/
```

Figure 5. Extra Credit Code

The table below compares the CPU load and relative deadline using different configurations.

ADC Configuration	Sampling Rate	CPU Load	ISR Relative Deadline
Single-sample ISR	1 Msps	**71.07%**	**1us**
DMA	1 Msps	**0.903%**	**1024us**
DMA	2 Msps	**1.530%**	**512us**
(extra credit) 8-sample FIFO + ISR	1 Msps	**21.28%**	**5us**

Table 1. Results of measuring CPU in Different Conditions

The second part is measuring the frequency of the a sine wave generated from the breadboard. In order to measure the frequency, we first used ADC comparison(comparator #1) to convert sine wave to square wave. We fed sine wave input into C1, and pin C10 is square wave output. The code for this part is shown below.

```
// Challenge #2
SysCtlPeripheralEnable(SYSCTL_PERIPH_COMP0);
ComparatorRefSet(COMP_BASE, COMP_REF_1_65V);
ComparatorConfigure(COMP_BASE, 1, COMP_TRIG_NONE  | COMP_INT_RISE | COMP_ASRCP_REF | COMP_OUTPUT_NORMAL);

// configure GPIO for comparator input C1- at BoosterPack Connector #1 pin 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
GPIOPinTypeComparator(GPIO_PORTC_BASE, GPIO_PIN_4);

// configure GPIO for comparator output C1o at BoosterPack Connector #1 pin 15
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
GPIOPinTypeComparatorOutput(GPIO_PORTD_BASE, GPIO_PIN_1);
GPIOPinConfigure(GPIO_PD1_C1O);
```

Figure 6. Analog Comparator Setup Code

After generating the square wave that has the frequency as the sine wave, we used timer count to calculated period. We set Timer0A to Edge Time Capture Mode. In addition, we fed the square wave output(C10) back to MCU T0CCP0 pin again.

```
// Timer in Capture Mode

// 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 7. Timer Capture Mode Setup Code

Then, we created an Hwi and wrote a capture_ISR() function. The figure below explains how we measure the period. Timer0A count is a timer, each count increment means 1/120MHz period. T0CCP0 is the square wave output converted from sine wave. Capture_Hwi occurs on the rising edge of square wave. So, we want to measure the time between two capture_Hwi. In order to do that, we used the TimerValueGet() function to read the full 24-bit captured timer count, then we calculate the period as the difference between the current and previous captured Timer0A count.

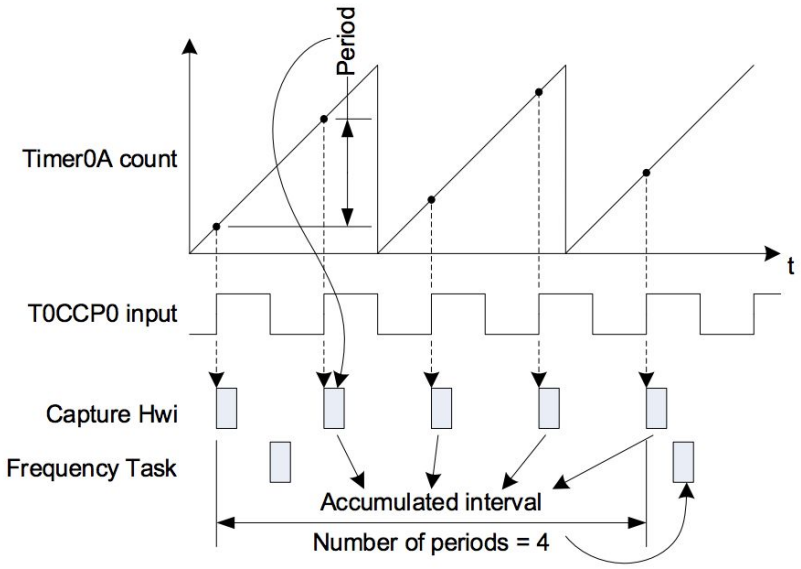

Figure 7. Calculate Period

```
void capture_ISR(UArg arg0)
{
    TIMER0_ICR_R = TIMER_ICR_CAECINT; // clear interrupt flag
    curr_count = TimerValueGet(TIMER0_BASE, TIMER_A);
    uint32_t period = (curr_count - last_count) & 0xFFFFFF;
    last_count = curr_count;

    accu_period += period;
    period_count++;

}
```

Figure 9. Capture_ISR Code

In addition, we created a new clock instance, and a new task, Frequency Task, which is signaled by a periodic Clock instance. In the Frequency task, we calculated the average frequency as the ratio of the number of accumulated periods to the accumulated interval. Also, we used GateHwi to protect access to the globals shared between the Frequency Task and the Timer Capture Hwi.

```
// new low priority -> _ -> 4
void FrequencyTask_func(UArg arg1, UArg arg2){

    uint32_t period, count;
    while(true){

        Semaphore_pend(freq_sem, BIOS_WAIT_FOREVER);

        // retrieve global data
        IArg key = GateHwi_enter(gateHwi0);

        period = accu_period;
        count = period_count;
        accu_period = 0;
        period_count = 0;

        GateHwi_leave(gateHwi0, key);

        // record the data to shared data
        freq_data = (float)gSystemClock / (float)period * (float)count;
    }
}
```

Figure 10. FrequencyTask Code

Finally, we displayed the frequency result inside the DispalyTask. And we successfully saw the correct result, 23KHz on the LCD screen. The picture of the result is in the result section.

The goal of the last part is to generate a PWM which has different duty cycle (pulse width) before the low pass filter, and a 20KHz sine wave after the low pass filter. PWM means pulse width modulation. It's a modulation technique used to encode a message into a rising pulse. The area of each pulse stands for a voltage amplitude. So in order to generate a sine wave which has different amplitude, we have to generate a PWM signal with changing pulse width first.

We first initialize the PWM peripheral(M0PWM1). Then, we set to use system clock, period and duty cycle. Finally, we enabled PWM interrupt in the PWM peripheral. Using code below, we successfully saw a 50% duty cycle PWM from M0PWM1 pin.

```
// Challenge #3
// use M0PWM1 at GPIO PF1 which is BossterPack Connector #1 Pin 40
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);
GPIOPinConfigure(GPIO_PF1_M0PWM1);
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD);

// configure the PWM0 peripheral
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1);              // use system clock
PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, PWM_PERIOD);
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1, PWM_PERIOD/2); // initial 50% duty cycle
PWMOutputInvert(PWM0_BASE, PWM_OUT_1_BIT, true);       // invert PWM output
PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);        // enable PWM output
PWMGenEnable(PWM0_BASE, PWM_GEN_0);                    // enable PWM generator

// enable PWM interrupt in the PWM peripheral
PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_0, PWM_INT_CNT_ZERO);
PWMIntEnable(PWM0_BASE, PWM_INT_GEN_0);
```

Figure 11. PWM peripheral initialization Code

Next, we programed one period of the sine wave into a lookup table.

```
int i = 0;
for(i = 0; i < PWM_WAVEFORM_TABLE_SIZE; i++)
    gPWMWaveformTable[i] = (uint8_t)(sinf(2 * PI * i / PWM_WAVEFORM_TABLE_SIZE) * 255.0 / 2 + 255.0 / 2);
```

Figure 12. Sine Wave Lookup Table Code

Then, we created PWM ISR, and determined the index into the table using a 32-bit phase accumulator. Every interrupt occurs with a period of 2.15 μs (= 258 / (120 MHz)), and we want to generate a 20KHz sine wave. The way we calculated the increment is shown below.

Period = 1/20Khz = 50us
Num of samples = 50us/2.15us = 23.3
Increment = 2^32/23.3 = 184650357

The code for PWM_ISR function is shown below. The last line of code can directly write into compareB that determines duty cycle.

```
void PWM_ISR(void){

    PWM0_0_ISC_R = PWM_ISC_INTPWM0;

    gPhase += gPhaseIncrement;

    PWM0_0_CMPB_R = 1 + gPWMWaveformTable[gPhase >> (32 - PWM_WAVEFORM_INDEX_BITS)];
}
```

Figure 13. PWM_ISR Code

After running the code, we successfully generated the PWM with different duty cycle. When we fed the signal into the RLC filter, we also saw a 20KHz sine wave. The lab results pictures are in the next section.

- Picture of lab results.
  The picture below shows the lab results.

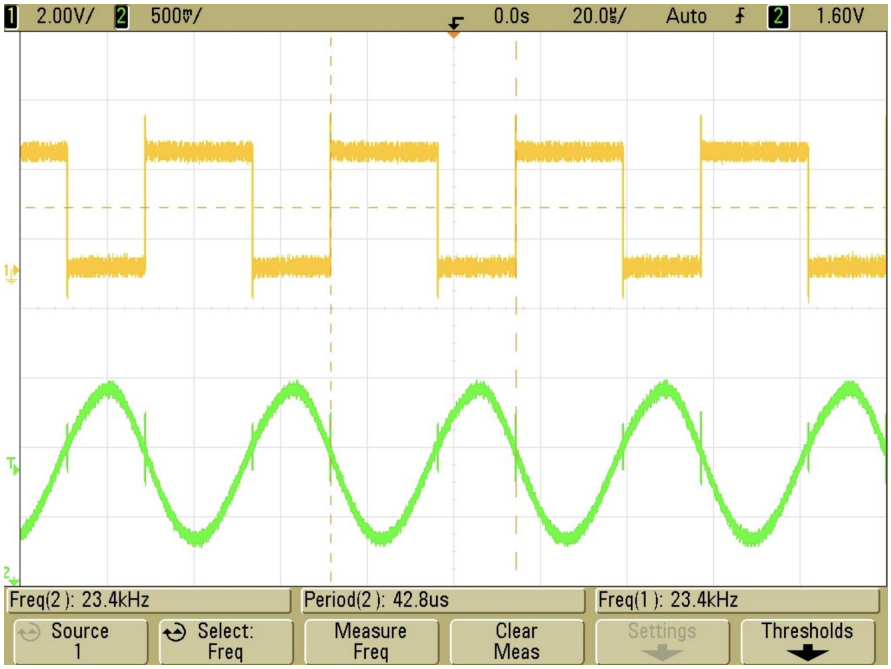

Figure 14. Picture of Lab Result (Part2 ADC Comparion)

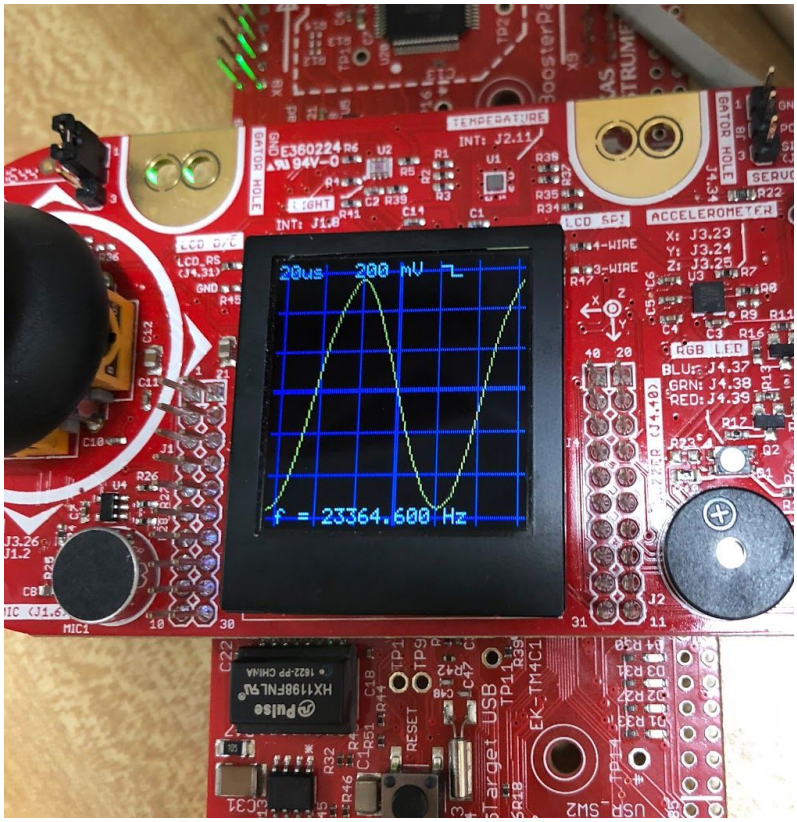

Figure 15. Picture of Lab Result
(Part2 Measure and Display Frequency)

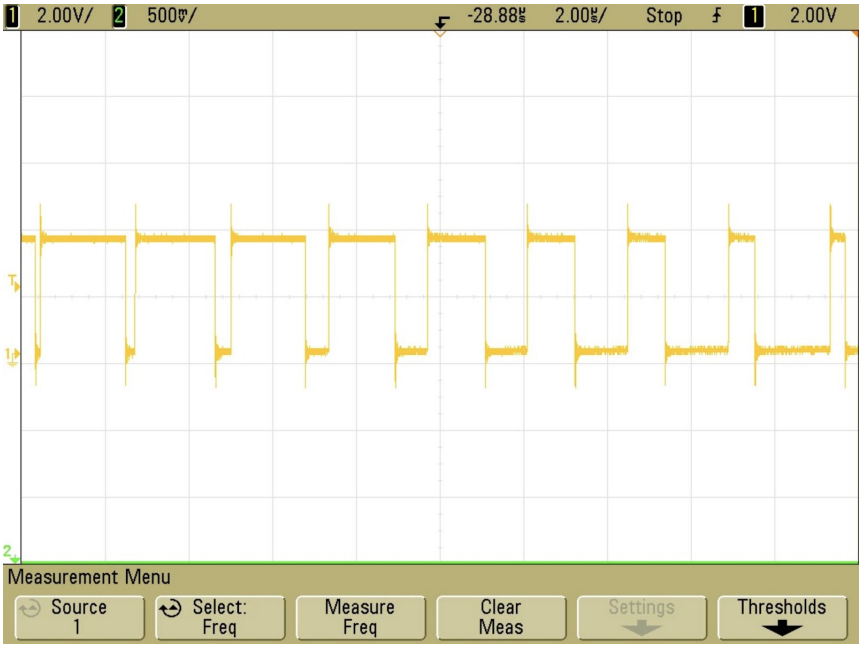

Figure 16. Picture of Lab Result (Part3 Generated PWM)

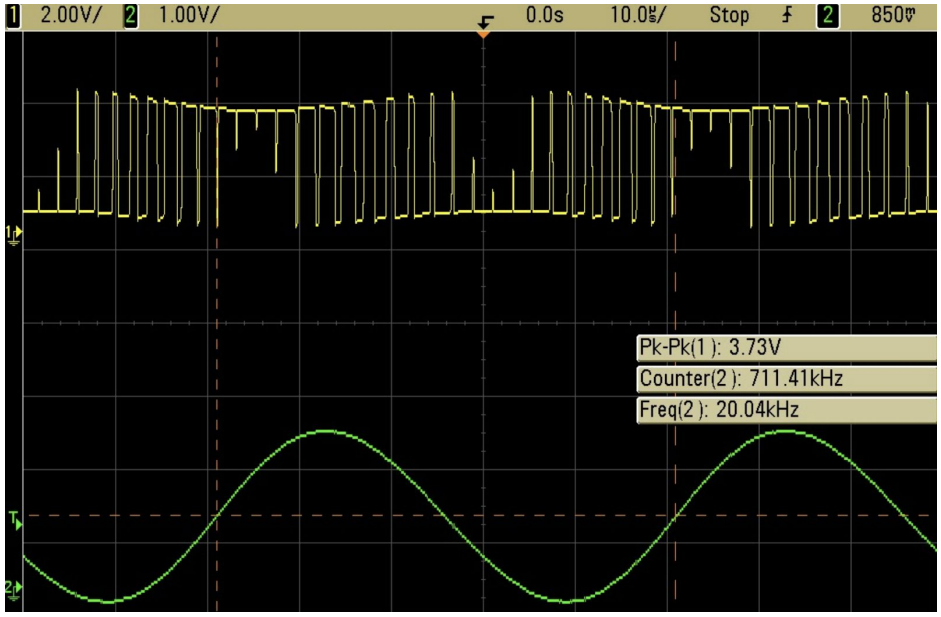

Figure 17. Picture of Lab Result
(Part3 Generated PWM-yellow & Sinewave after RLC filter-green)

- Serious difficulties we encountered.

The first difficulty we encountered was that we didn't get correct square wave output in part2. After asking professor, we found that we connect the wrong pin. Pin3 and Pin15 are on the bottom board instead of the top board.

Another difficulty was calculating frequency in the second part. We successfully got the period based on timer, around 5100. However, we were not sure what's the unit of that period. Finally, we found it's 5100 cycles of the 120MHz clock.

Conclusions

In summary, in this lab, our team successfully replaced normal ADC with DMA. After comparing the CPU load for both configuration, we found DMA CPU load is much lower than that of normal ISR because DMA has a much longer relative deadline. In addition, we found higher sampling rate would increase CPU load. In the second part, we learned how to use analog comparator to convert a sine wave to a PWM signal. Last but not least, we learned PWM. We learned how to generate PWM and how to convert PWM into a sine wave using low-pass filter.