Lab 3: Advanced I/O

ECE 3849

Samantha Comeau, sjcomeau@wpi.edu, Mailbox 733

William Schwartz, wrschwartz@wpi.edu, Mailbox 300

30 April, 2018

Table of Contents

Introduction

The goal of this lab was use the lab 2 RTOS oscilloscope and add to it. We added a frequency counter and the function generator functionality to the lab 2 oscilloscope. On top of that we also optimized the ADC I/O using DMA with a 2 Msps sampling rate. We also used an analog comparator peripheral, a timer in capture mode and a PWM with duty cycle adjusted every period to generate analog waveforms to make this lab successful. In this lab the previous labs all came together as a cohesive final project. Adding functionality for a function generator and a frequency counter also furthered our knowledge of real time operating systems and how oscilloscopes work.

Discussion

Overview

For this lab we took our lab 2 RTOS oscilloscope and changed it in stages. This was done in three challenges. The first challenge was optimizing the ADC I/O using a Direct Memory Access controller (DMA). The second challenge was implementing a frequency counter. This involved using an external signal to generate interrupts using the onboard analog comparator, the timer in capture mode, and frequency measurements. The final challenge was creating a PWM function generator. The PWM output was passed through a low pass filter to create a sine wave. These three challenges together encompassed the goals of this lab.

Challenge #1. ADC I/O Optimization

For the first challenge we needed to optimize the ADC I/O. We did this using a Direct Memory Access controller which performs almost all the duties of the Lab 2 ADC ISR that we were originally using, but uses a fraction of the CPU. This is ideal because our CPU load is already intense, so optimization will lessen the load so we can add functionality required by other parts of our lab. Although we still need an ISR the relative deadline is much longer because a DMA interrupt only occurs when the DMA transfer complete, not every ADC sample. We were given starter code for this section to enable the DMA.

DMA Configuration

We configured the DMA using functions from the `driverlib/udma.h` library. We needed to allocate the DMA control table in RAM. We also needed to allow the DMA channel to interact with ADC1 sequence 0. Finally, we configure the primary and alternate DMA control blocks for the channel, with a data size of 16 bits which is one ADC sample size. We also configured the DMA to be in ping pong mode which allows the DMA to operate on one of two buffers while the CPU is accessing the other. This allows continuous transfer to be happening. These two DMA channels for ping pong mode are assumed to be complementary linked, which means when a channel reaches the end of transfer it starts the complementary channel and triggers an interrupt. We reset the channels with the ADC ISR.

Since we are no longer using the ADC buffer indexes we have to keep track of the new buffer somehow. This was done using the following code.

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

Code Snippet 1. Indexing the new DMA buffer

Once we've updated this from lab 2 we can then start measuring the CPU load of the new DMA implementation. We found the following values for CPU load and relative deadlines.

Module	**Sampling Rate**	**CPU Load**	**ISR Relative Deadline**
ADC ISR	1 Msps	58%	1uS
DMA	1 Msps	2.24%	1mS
DMA	2 Msps	3.36%	512uS

Figure 1. CPU Load and Relative Deadlines

Challenge #2. Frequency Counter

For the frequency counter we used the following implementation.

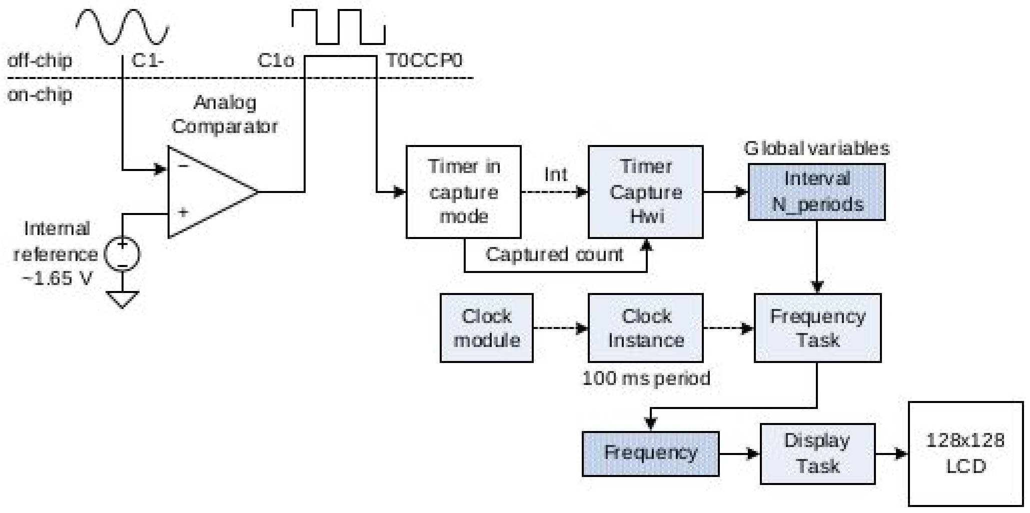

Figure 2. Frequency counter implementation.

This setup has a recognized flaw. Since the external signal is generating interrupts of the highest priority and the period is indeterminate, we can potentially starve the lower priority tasks.

Analog Comparator

For this challenge we needed to configure the analog comparator peripheral. This analog comparator was configured using the `driverlib/comp.h & driverlib/pin_map.h` libraries. We configured the comparator to have input C1- at BoosterPack Connecter #1 to pin 3. The C1+ was internally set to 1.65v This was done as follows.

```
    ///// ANALOG COMPARATOR
    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);
```

Code Snippet 2. Setting up the comparator and reference

```
// Pin 3 Connector 1 = C.4 = C1-
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
GPIOPinTypeComparator(GPIO_PORTC_BASE, GPIO_PIN_4);
```

Code Snippet 3. Setting up the comparator input

We also had to configure the output C1o at BoosterPack Connector #1 to pin 15. This was done as follows.

```
// Pin 15 Connector 1 = D.1 = C1o
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
GPIOPinTypeComparatorOutput(GPIO_PORTD_BASE, GPIO_PIN_1);
GPIOPinConfigure(GPIO_PD1_C1O);
```

Code Snippet 4. Setting up the comparator output

Timer in Capture Mode

For the timer in capture mode we used Timer0A in Edge Time Capture Mode. This involved editing our lab 2 configuration for the clock module Timer ID. We needed to do this because the TI-RTOS typically uses Timer0 for the clock module. Our Hwi was configured to handle the Timer0A Capture interrupts. We assigned it a priority 35 so that it was not a zero-latency Hwi. We clear the Capture interrupt flag and use a TimerValueGet() function to read the full 24-bit captured timer count (which includes the prescaler). Then we calculate the period as a difference between the current and previous captured Timer0A value which involved taking care of wraparound.

Frequency Measurement

Our frequency measurement was setup as follows.

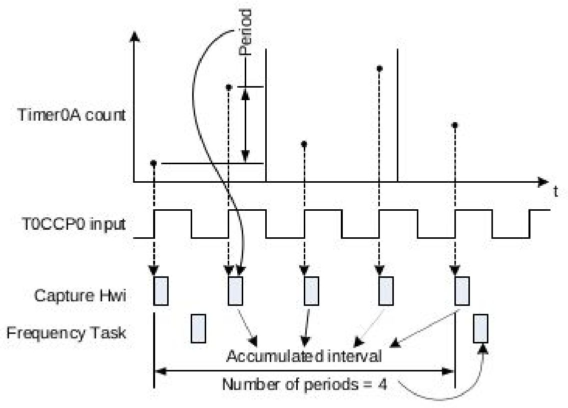

Figure 3. Frequency measurement diagram

For this we used our timer in capture mode to start counting how many periods have accumulated for debugging purposes. For the frequency task, which is signaled by the periodic Clock instance, we used this to determine the average frequency as a ratio of the number of accumulated periods to the accumulated interval. After we've done this we reset our globals back to 0 so we can start our next measurement.

We created a new clock instance for this with a 100ms period and had it post to the Semaphore Waveform Task. We also created a new task for the Frequency with a priority of 4. This tasks was used to compute the frequency and update global variables so we could display the frequency on the screen. On average our frequency was around 22.1 kHz.

Challenge #3. PWM Function Generator

We used the PWM output as a form of digital-to-analog conversion. To make the sine wave output we passed PWM through a low pass filter.

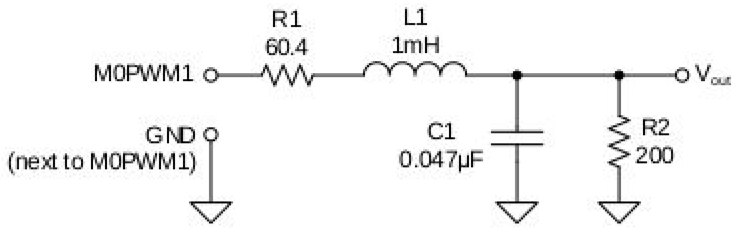

Figure 4. PWM Low Pass Filter

As you can see we use a RLS low-pass filter and produce a voltage proportional to the PWM duty cycle. When we vary the duty cycle we can produce a low-frequency waveform. We did this by connecting pin 1 on our breadboard to the input the diagram above, and recording the waveform on the ouput.

PWM Configuration

Our PWM0 peripheral has four PWM generators, which are 0-3, each with 2 PWM outputs. Generator 0 controls 0 and 1, and so on. Generator 0 output 1 is labeled as output B on the generator block. The selected PWM period is the shortest that achieves 8-bit duty cycle resolution (which means we won't use 0% or 100% to avoid nonlinearity). That means the PWM period is $2^8 + 2$ system clock cyles, which translates to a PWM frequency of 465 kHz.

After we've verified that we vary the duty cycle so that when passed through a low pass filter it generates a low-frequency waveform (20 kHz sinusoid). The figure below shows the goal of this step.

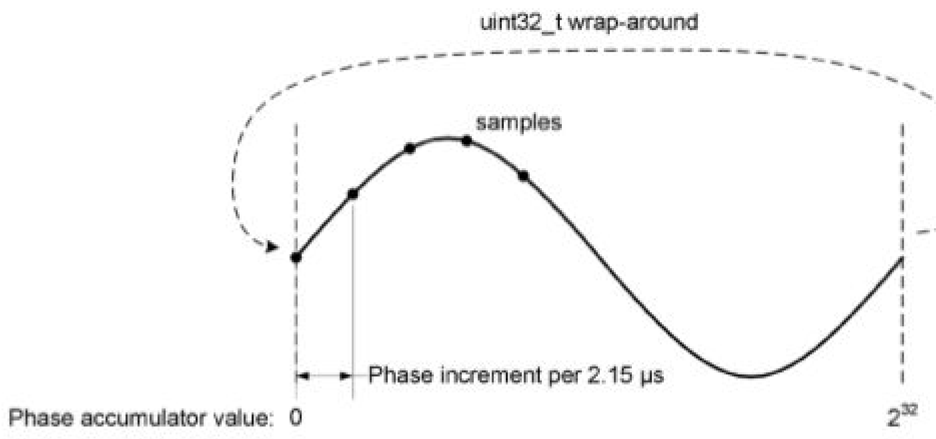

Figure 5. PWM low-frequency sinusoid

Our PWM Hwi is a zero latency Hwi so that it meets its deadlines. We gave it a priority 0 to ensure that. We initialize the PWM as follows, creating a lookup table to generate the sine wave as follows.

```
void pwm_init(void) {
    int i;
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1); // PF1 = M0PWM1
    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);

    for (i = 0; i < PWM_WAVEFORM_TABLE_SIZE; i++) {
        gPWMWaveformTable[i] = (uint8_t)roundf(sinf(2.f * M_PI * i / PWM_WAVEFORM_TABLE_SIZE) *
                                80.f) + 128;
    }
}
```

Code Snippet 5. PWM ISR init

```
void PWM_ISR(void) {
    PWM0_0_ISC_R = 1; // clear PWM interrupt flag
    phase += dphase;
    // write directly to the Compare B register that determines
the duty cycle
    PWM0_0_CMPB_R = 1 + gPWMWaveformTable[phase >> (32 -
PWM_WAVEFORM_INDEX_BITS)];
}
```

Code Snippet 5. PWM ISR code

As you can see in our code, it works properly because the PWM module delays updates of the duty cycle every time the counter reaches 0. The relative deadline is the same as its period and we verified our code by running it and making sure the output as around 20 kHz sine wave.

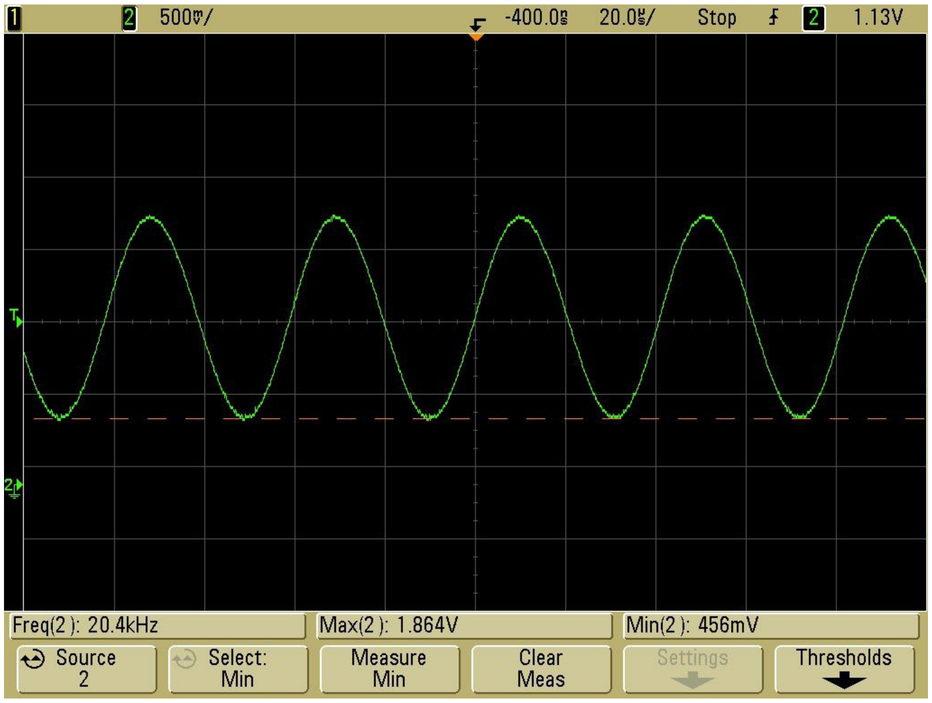

Figure 6. Oscilloscope reading of our PWM Function Generator

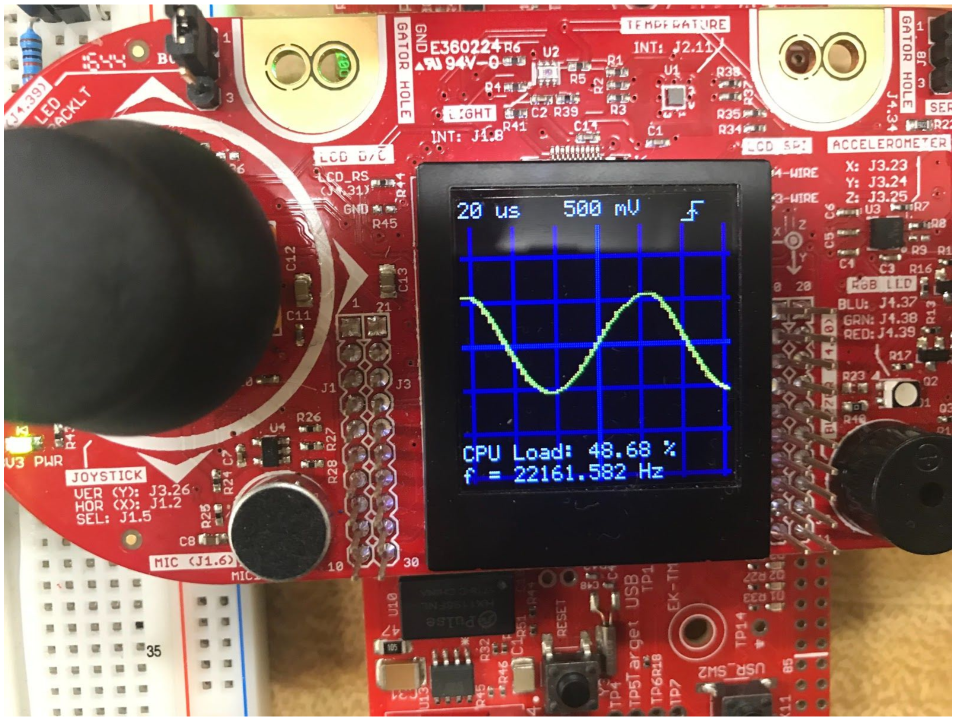

Figure 7. Image Showing CPU load of running complete Project

Conclusion

This lab functioned as it was expected to. It was very rewarding that every challenge built upon the previous, and without offloading ADC sampling task onto the DMA, the rest of this lab would not have been possible. So after we completed that task our CPU load dropped to nearly 2%. We then implemented the frequency counter, adding a significant amount of CPU load. Finally we created a Varying PWM signal, that when passed through a low pass filter gives us the sinusoidal waveform that we see in Figure 5. This waveform had a significant amount of noise, so we followed the lab procedures to limit the amount of noise in our signal, we then set the function generator to acquire, and from there set it to averaging, which gives us this good looking waveform that we see in figure 5. The following Image shows our completed code running, this image was taken at the same time that the Oscilloscope capture was taken. When the project was completed we found our CPU load to be 48%.