ECE 3849: RTOS Oscilloscope with DMA

B-Term 2020

Adam Yang
Prudence Lam
Sirut Buasai

December 11, 2020

1. Introduction

This lab aims to successfully program a digital oscilloscope with extra features such as frequency counter and audio output using DMA structures in a RTOS environment. With the extension from the previous lab where a digital oscilloscope functionality was implemented in a RTOS environment, DMA are used to optimize ADC I/Os as well as increasing its sampling rates and audio configuration. These features are further fed to the interface for users to interact with the data using the LCD displays and Launchpad control buttons.

2. Discussion and Results

2.1. System Requirements

2.1.1. ADC I/O Optimization

In the first task, a new ADC ISR replaces an existing one from the previous lab. In addition, the sampling rate is increased to 2 Msps. This new implementation utilizes the DMA interrupt instead of an existing sequence 0 interrupt. And thus, the relative deadlines with the DMA interrupt differ from the previous implementation. A CPU load for each sampling rate is determined using a code from lab 1. This outputs a relative deadline for the tasks which influences the following DMA configuration.

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

Figure 1: The figure above depicts the snippet of code used to enable the ADC1 Sequence 0 DMA interrupt.

The ADC_ISR interrupts when every DMA channel successfully loads. The ISR first clears the interrupt flags. Then it checks the primary DMA channel in the gADCBuffer for data transfers from the ADC. If the DMA channel successfully loads data, the ISR restarts the primary channel and looks at other DMA channels.
The process then repeats as the ISR checks whether the DMA channel successfully loads the data from the ADC or not.
In addition, a helper function is implemented to obtain the ADC buffer index in the DMA channels. The function subtracts the current number of items left in the channel from the channel size. However, due to the risk of shared data issues as the method uses non-atomic read operation, GateTask is implemented to protect the measured index value. This function was used to replace the variable “gADCBufferIndex” from previous labs.

The figure below demonstrates the implementation:

```
int32_t index;
IArg gateKey = GateTask_enter(gateTask0);
if (gDMAPrimary) {  // DMA is currently in the primary channel
    index = ADC_BUFFER_SIZE/2 - 1 -
            uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT);
    GateTask_leave(gateTask0, gateKey);
}
```

Figure 2: This figure depicts the snippet of code used in the getADCBufferIndex() function. A GateTask was used to resolve a shared data issue from a non-atomic read operation.

With the DMA ADC Configuration and the ADC sampling rate of 1 Msps, the CPU load was determined to be 0.8%, which is expected because the DMA controller performs most tasks of the previous ADC ISR. Moreover, with the ADc sampling rate increased to 2Msps, the measured CPU load 1.0%. The low CPU loads reinforce the idea that the utilization of DMAs allow ADC tasks to perform more effectively. Moreover, the absence of the previous ISR allows other interrupts to take priority. And thus, the PWM and timer interrupts take priority over the newly implemented DMA interrupt.

For these measurements, please refer to table 1 in the appendices.

2.1.2. Frequency Counter

The second task requires the implementation of a frequency counter to measure the period of the sampled input waveform. The input waveform from previous Labs was duplicated in the separate GPIO pin PF3, and was connected to the timer input GPIO pin PD0. To accomplish this, the Timer0A was configured for Edge Time Capture Mode.

This is depicted in the following figure:

```
// From Lab 3 (Challenge #2)
// 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 3: This figure depicts the code snippet was to configure Timer0A for Edge Time Capture Mode. This timer is used to measure the period of the input waveform.

The timer Hwi takes top priority as it handles the capture interrupts. The interrupt flags are first cleared. Then, the Timer ISR uses the getTimerValue() function to obtain the current timer

count. The period of the input waveform is calculated by subtracting the old timer count from the new timer count; the difference is bitwise logical ANDed with 0xFFFFFF to handle overflow. The result is saved into the global variable “pwmPeriod”.

The following figure depicts the implemented Timer ISR:

```
void timerCaptureISR(void) {     // From Lab 3 (Challenge #2)
    // Clear the TIMER0A Capture Interrupt flag
    TIMER0_ICR_R = TIMER_ICR_CAECINT;

    current_count = TimerValueGet(TIMER0_BASE , TIMER_A);
    pwmPeriod = (current_count - last_count) & 0xffffff;
    last_count = current_count;
}
```

Figure 4: This figure depicts the timer capture ISR was to calculate the period of the input waveform.

The frequency in Hz was determined by dividing the System Clock frequency by the value in “pwmPeriod”; though the variable “pwmPeriod” is in clock cycles, rather than seconds. The Display Task displays this frequency value onto the LCD screen.
The left and right joystick buttons were used as user interaction for the variability in the period of the PWM signal. This was accomplished by reading the left and right presses on the joystick and decrementing or incrementing the variable “pwmp” respectively. This variable represents the step change in the period of the PWM signal.

Within the signal initiation code, the following changes were implemented:

```
PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, roundf((float)gSystemClock/(PWM_FREQUENCY + PWM_STEP_SIZE*pwmp)));
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, roundf((float)gSystemClock/(PWM_FREQUENCY + PWM_STEP_SIZE*pwmp)*0.4f));
PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, roundf((float)gSystemClock/(PWM_FREQUENCY + PWM_STEP_SIZE*pwmp)*0.4f));
```

Figure 5: This figure depicts the code snippet used to vary the input waveform.

The variable PWM_FREQUENCY was defined as 20,000 since the starting frequency is 20,000 Hz. PWM_STEP_SIZE is the step size of the frequency. And as described above, “pwmp” is the number of steps from the starting frequency. In addition, the period in clock cycles of the input waveform was displayed on the LCD screen in the Display Task by printing the value stored in “pwmPeriod”.

2.1.3. PWM Audio

The third task required the implementation of the speaker on the Boosterpack to produce an audio message. This was accomplished by first configuring the GPIO pin PG1 as output on the PWM0 peripheral. This audio function calls onto a similar PWM initialization method used in lab 1.

The initialisation method is depicted in the following figure:

```
* Initializes 465 kHz PWM Audio wave on PG1
*/
void PWMInit(void) {    // From Lab 3 (Challenge #3)
    // configure M0PWM5, at GPIO PG1
    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));    // Init
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_5, roundf((float)gSystemClock/AUDIO_PWM_FREQUENCY*0.5f)); //
    PWMOutputState(PWM0_BASE, PWM_OUT_5_BIT, true);
    PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_2, PWM_INT_CNT_ZERO);    // configures PWM interrupts (eve
    PWMGenEnable(PWM0_BASE, PWM_GEN_2);
```

Figure 6: This figure depicts the code snippet used to initialize the audio PWM signal.

The constant “AUDIO_PWM_FREQUENCY” was defined to be 465,116 Hz, which configures the PWM signal to have the desired period of 258 clock cycles with a duty cycle of 50%.
A new PWM ISR had to be implemented to update the duty cycle in order to create a low frequency sound.

This is depicted in the following figure:

```
void PWM_ISR(void)  // From Lab 3 (Challenge #3)
{
    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 7: This figure depicts the PWM_ISR used to update the duty cycle of the sampled PWM signal.

After first clearing the interrupt flag, the ISR determines the sample index, and uses it to sample the PWM waveform. The ISR also checks for the termination of the audio waveform. If the audio reaches its terminal, the ISR will disable the PWM interrupts. The variable gSamplingRateDivider was calculated using the formula (gSystemClock) / (pwmPeriod * AUDIO_SAMPLING_RATE), where gSystemClock has the value 120 MHz, pwmPeriod is the measured period of the input waveform in clock cycles, and AUDIO_SAMPLING_RATE is defined as 16,000 samples per second. To handle the PWM interrupts, a new high-priority PWM Hwi was configured to call the PWM ISR function depicted above. Lastly, a PWMIntEnable() function was added to the User Input Task in response to a press on the joystick. This enables the

PWM interrupts and starts the audio message playback, which can be made out as “Welcome to ECE 3849!”.
After the implementation of the audio playback feature, the CPU load was significantly higher at 55.5% compared to the 2.5% CPU load when audio playback is disabled.

2.2. Difficulties

The only main difficulty encountered in this lab was when the program was aborting moments after the code was uploaded to the Launchpad. The debugger was used to determine that the stack size of the Display Task was too low for the needs of the new functionalities. The solution to this issue was to double the stack size of the Display Task from 1024 to 2048.

```
ti.sysbios.knl.Task: line 373: E_stackOverflow: Task 0x20013018 stack overflow.
xdc.runtime.Error.raise: terminating execution
```

Figure 8: This figure depicts the Stack overflow error message due to the stack size of Display task not being large enough.

2.3. Results

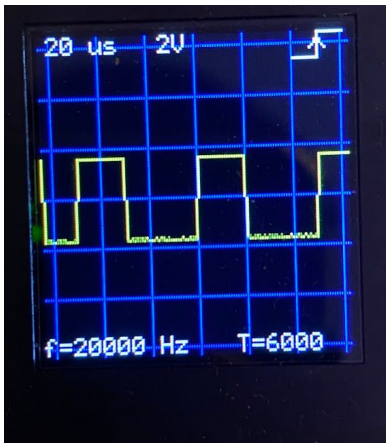

Figure 9: Displayed Starting input waveform with period of 6,000 clock cycles.

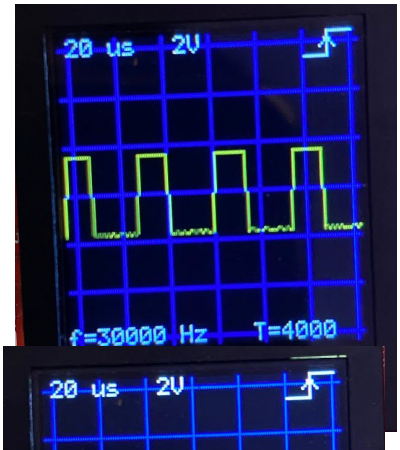

Figure 10: Displayed modified input waveform with period of 4,000 clock cycles.

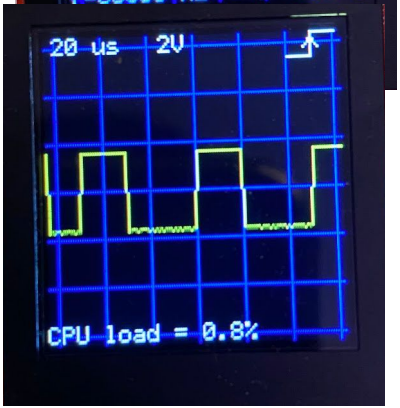

Figure 11: Displayed CPU Load of 0.8% with DMA and 1 Msps ADC sampling rate.

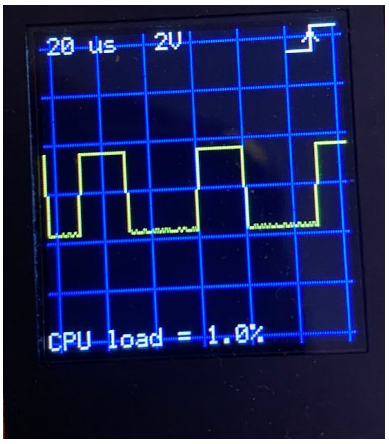

Figure 12: Displayed CPU Load of 1.0% with DMA and 2 Msps ADC sampling rate.

3. Summary and Conclusion

As an extension to the second lab, the 2 Msps digital oscilloscope with an audio playback used RTOS environment along with DMA features to optimize the CPU utilization load. An ADC configuration was transferred from a sequence 0 interrupts into a DMA interrupts, thus freeing up space for other demanding tasks to take priority. Building upon our previous lab, RTOS objects such as gates were used to handle potential shared variable issues. Moreover, timer I/O are configured and used such that Timer ISR were able to count the frequencies outputted from the waveform. Lastly, the use of RTOS objects were further explored and interfaced for the audio processing and playback.

4. Appendix

Table 1.

ADC Configuration	ADC Sampling Rate	CPU Load	ISR Relative Deadline
Single-sample ISR	1 Msps	75.4%	1 µs
DMA	1 Msps	0.8%	1 ms
DMA	2 Msps	1.0%	0.5 ms

During Audio-Playback: CPU → 55.5%

DMA relative deadline = (DMA channel size)/(ADC sampling rate)