ECE 3849 B2023
Real-Time Embedded Systems
Lab 5: Advanced I/O

In this lab, you will use the freedom provided by DMA to implement some CPU intensive features - a frequency counter and audio output

To complete all labs in this course, you will need to submit your source for that lab, and demonstrate the lab in operation to one of the course staff (a TA or tutor). You will be graded on the completion of the steps in this lab document. After you have received a signoff, follow the instructions at the end of the document to submit your source code.

This lab is due on **Wednesday, December 13**. Late assignments will receive a 20% penalty.

Step	**Max Points**	**Score**
Timer configured in edge capture mode and measuring period	25	
Measured frequency displayed on LCD with 1Hz resolution	5	
PWM source period is adjustable in steps of 1 clock cycle and is displayed on the LCD	15	
Pressing a button starts PWM audio playback. CPU load is reasonable, if high, and message is audible.	30	
Source Code & Report	25	
Total	100	

Objectives

- Measure the frequency of an input signal
- Use a timer in capture mode
- Use PWM with duty cycle adjusted periodically to generate an analog waveform

Assignment

Task 1: Add a frequency counter to your system. Features of the frequency counter:

- Accept the same PWM signal as the AIN3 input
- Measure the period of the square wave input using a timer in capture mode
    - Measurement resolution should be 1 CPU clock cycle = 8.3 ns
    - Longest measured period can be limited to 0xffffff clock cycles or 0.14 s
- Display the measured frequency on the LCD with 1 Hz resolution, e.g. "f = 20127 Hz"

Also make the PWM signal source period adjustable in steps of 1 clock cycle using buttons. Display the PWM period in clock cycles on the LCD, e.g. "T = 5962." Preserve the 40% duty cycle.

Task 2: Implement basic audio output using PWM with periodically adjusted duty cycle.

You are given a sampled waveform (8-bit samples, 16000 samples/sec). Your goal is to play it back when the user presses a button. See part 3 of the lab for more details.

Part 0: Copy CCS Project

Copy your Lab 4 CCS project by selecting it in Project Explorer and pressing Ctrl+c, then Ctrl+v. CCS should prompt you to rename the new project: use the convention for online code submission "ece3849_lab4_username1_username2," substituting your username. When the time comes to submit your source code, follow the instructions at the end of the Lab.

Part 1: Add Second PWM Output

The signal for the oscilloscope to measure with is generated with PWM, but is set to provide only one output. To measure the frequency, a second output will be needed.

GPIO PF3 will be configured as a second output in your signal initialization code. This will require modifying `GPIOPinTypePWM`, `GPIOPadConfigSet`, and `PWMOutputState` to include pin 3. Additionally, a new call to `GPIOPinConfigure` and `PWMPulseWidthSet` will be needed. Consult the pin 2 output code and the peripheral library documentation to find what arguments must be passed to these.

Part 2: Frequency Counter

Here, you will add a frequency counter to measure the period of the input wave.

Configure Timer0A in Capture mode to measure the period of an input square wave. 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.

The setup needed is nearly complete in the following code block. Browse "driverlib/pin_map.h" to locate the correct argument for `GPIOPinConfigure`. Note that TI-RTOS typically uses Timer0 for the Clock module. Change the "**Timer Id**" under the Clock module settings to some unused timer other than 0. Add the LM4-specific module TimestampProvider and configure it to "use Clock's timer."

```
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
#include "driverlib/pin_map.h"
```

```
// config GPIO PD0 as timer input T0CCP0 at BoosterPack Connector #1 pin 14
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
GPIOPinTypeTimer(GPIO_PORTD_BASE, GPIO_PIN_0);
GPIOPinConfigure(...);
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, ...);
// use maximum load value
TimerLoadSet(TIMER0_BASE, TIMER_A, 0xffff);
 // use maximum prescale value
TimerPrescaleSet(TIMER0_BASE, TIMER_A, 0xff);
TimerIntEnable(TIMER0_BASE, ...);
TimerEnable(TIMER0_BASE, TIMER_A);
```

Connect the PF3/M0PWM3 signal generator pin to the PD0/T0CCP0 timer input pin using a jumper wire. The easiest way to find these pins will be to use the silkscreen on the back of the board.

Configure an Hwi to handle Timer0A Capture interrupts. Assign it priority assuming a typical interrupt rate of 20 kHz. 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. Use 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 (see lecture). Save the period into a global and verify that its value is what you expect using the debugger.

Calculate the frequency from the measured period and print it at the bottom of the LCD in the Display Task. The output should look like "f = 20127 Hz."

To give the input signal a bit of variability, configure two user buttons to increment and decrement the period of the PWM signal source. Also print this period on the LCD in the Display Task, e.g. "T = 5962." While this period and the measured period should match, you should still use the measurement to compute the frequency. When adjusting the period, also update the duty cycle of both outputs to keep it at 40%.

Part 3: PWM Audio

Now, you will use 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. While it is best to use an actual low-pass filter on a PWM output (and in past labs we have built them on breadboards), in many applications we can get away without one. For example, a PWM light source does not need low-pass filtering, as our eyes do this for us. Similarly, a speaker, its driver circuit and our ears all have natural low-pass filtering properties. PWM audio without careful filtering will be of lower quality but may still be acceptable for low-end applications.

Download "audio_waveform.h" and "audio_waveform.c" from Canvas and add them to your Lab 3 project. These files define a pre-recorded waveform, its length (number of samples) and sampling rate.

Copy the PWM initialization code used to generate the signal measured by the oscilloscope, and modify it to initialize the **PWM0 generator 2, output 5**, which drives the speaker on the BoosterPack. Configure the corresponding GPIO pin **PG1** as **M0PWM5** in the `GPIOPinConfigure` call. Initialize this PWM generator to a **period** of **258** clock cycles (define a constant for this). It is the shortest period that achieves 8-bit duty cycle resolution. We 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. Set the initial duty cycle to 50%. Use the following call to configure, but not yet enable, PWM interrupts (every time the counter reaches 0).

```
PWMGenIntTrigEnable(PWM0_BASE, PWM_GEN_2, PWM_INT_CNT_ZERO);
```

For your information: The PWM0 peripheral has four PWM generators, numbered 0…3, each with two PWM outputs. Generator 0 controls outputs 0 and 1. Generator 1 controls outputs 2 and 3, and so on. We are using generator 2, output 5, also labeled output B on the generator block.
To verify that your PWM initialization code functions correctly, you may temporarily set the PWM frequency to be in range of your oscilloscope application (i.e. around 20 kHz). You may then temporarily wire PG1 to PE0 to display the generated waveform using your own oscilloscope.

You now need to periodically update the duty cycle in an ISR to create a low-frequency waveform. The PWM ISR is partially given to you in the following code block. Your tasks are to:

- 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

Note that the sampling rate divider is set incorrectly in this code. It will result in too high a sampling rate, producing a higher-pitched sound. Calculate its proper value during PWM initialization from parameters available to you: gSystemClock, PWM period in clock cycles and AUDIO_SAMPLING_RATE.

```
#include "driverlib/pwm.h"
#include "inc/tm4c1294ncpdt.h"
#include "audio_waveform.h"
```

```
uint32_t gPWMSample = 0; // PWM sample counter
uint32_t gSamplingRateDivider = 20; // sampling rate divider
void PWM_ISR(void)
{
    PWMGenIntClear(...); // clear PWM interrupt flag
    // waveform sample index
    int i = (gPWMSample++) / gSamplingRateDivider;
    // write directly to the PWM compare B register
    PWM0_2_CMPB_R = 1 + gWaveform[i];
    if (i ...) { // if at the end of the waveform array
        // disable these interrupts
        PWMIntDisable(PWM0_BASE, PWM_INT_GEN_2);
        // reset sample index so the waveform starts from the
beginning
        gPWMSample = 0;
    }
}
```

Configure a TI-RTOS Hwi for your PWM generator 2 ISR. It should be a **“zero-latency Hwi”** in order to meet its deadlines. Double-check the priority assignments of all three Hwi in your system.

For your information: The PWM module delays updates of the duty cycle to the next time the PWM counter reaches 0. This avoids PWM periods with a bad duty cycle if the update was too early. The deadline for the PWM ISR is the next PWM interrupt.

Finally, add an appropriate PWMIntEnable() call in response to a **button press** in your User Input Task. This should start audio playback. The audio comes out quiet and noisy, but you should be able to tell what it says if you bring the speaker close to your ear. Note the high CPU load during audio playback. This CPU load can only be supported in this system if the ADC uses DMA. Here are some ideas for reducing this CPU load. This is only a hypothetical discussion. **You do not need to implement** any of these techniques:

- We could use a lower PWM frequency. It would be more difficult to filter out, but in our case would not likely make much difference, as long as it is not audible. The 8-bit samples would need to be scaled to properly control the duty cycle.
- We do not really need to interrupt as often as we do, as we only update the PWM duty cycle once per several interrupts. Some PWM hardware can interrupt every 2nd, 3rd or 4th PWM period, but not this one. However, you have the option of configuring another timer to trigger the PWM duty cycle updates. The PWM module contains 4 timers (generators) that can be started synchronously. You could configure an unused PWM generator to a period that is gSamplingRateDivider times longer than the PWM period we used for audio. Use that other generator to trigger PWM interrupts. This would dramatically reduce the CPU load of the PWM ISR.
- The above setup is not even the best that can be done. You could trigger DMA transfers instead of interrupts to update the duty cycle directly from the waveform array. Either use a general-purpose timer that can directly trigger DMA or route a signal externally from PWM to GPIO to trigger DMA. This has not yet been attempted in these labs, so it is unclear if this method has issues. Feel free to experiment with DMA for PWM.

Part 4: Submitting Source Code

Make sure your Lab 5 project is named “**ece3849_lab5_username**” with your username substituted. **Clean** your project (remove compiled code). Right click on your project and select “Export...” Select General → “Archive File.” Click Next. Click Browse. Find a location for your archive and specify the exact same file name as your project name. Click Finish. Submit the resulting .zip file to Canvas. This concludes Lab 5.