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

In this "I/O lab," you will add frequency counter and function generator functionality to your Lab 2 oscilloscope. You will also optimize ADC I/O using DMA and bump the sampling rate to 2 Msps. Have the course staff fill in your scores in the following signoff sheet before the signoff deadline. This lab requires a report. See Canvas for due dates. When submitting the source code, first "Clean" the project, and remove all files from the Debug subfolder to reduce the .zip file size.

Step	Max pts	Score
Set up DMA to transfer ADC1 samples directly to memory.	11	
Increase ADC sampling rate to 2 Msps. Fill the ISR vs. DMA CPU load and deadline table.	9	
Configure the analog comparator. Confirm sine to square wave conversion using a bench oscilloscope.	8	
Configure a timer in Edge Time Capture mode. Confirm period measurement using the debugger.	8	
Measure frequency every 100 ms in an RTOS Task and display on the LCD with mHz resolution.	8	
Generate a 20 kHz sinusoid using filtered PWM and interrupts to vary the duty cycle. Record the CPU load.	16	
(*extra credit*) For the ADC, also implement the 8-sample FIFO with an ISR that empties the FIFO (no DMA).	5	
Question about this lab answered.	5	Student1: Student2:
Report and source code (separate due date from signoff)	35	
Total points	100 + 5	Student1: Student2:

Student 1: ______________________________ Student 2: ______________________________

Grader: ________________________________ Date: ______________ Mailbox: __________

Learning objectives

- Use DMA to optimize ADC I/O
- Measure the frequency of an analog signal
- Use an analog comparator peripheral
- Use a timer in capture mode
- Use PWM with duty cycle adjusted every period to generate analog waveforms

Assignment

Challenge #1: Configure the DMA controller to replace your Lab 2 ADC ISR. Measure the CPU load in this configuration. Now increase the ADC sampling rate to 2 Msps and repeat the CPU load measurement. Fill in the following table. Note that DMA also requires an ISR, though different from the Lab 2 ADC ISR. See the Recommended Implementation for details.

ADC Configuration	Sampling Rate	CPU Load	ISR Relative Deadline
Single-sample ISR	1 Msps		
DMA	1 Msps		
DMA	2 Msps		
(extra credit) 8-sample FIFO + ISR	1 Msps		

Challenge #2: Add an accurate frequency counter to your Lab 2 system. Features of the frequency counter:

- Accept the same analog signal as the AIN3 input
- Convert the analog signal to a square wave of the same frequency using the analog comparator peripheral
- Measure the period of the square wave 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
- Average the period measurement over the time interval of 0.1 sec (sampling rate = 10 samples/sec). Convert to frequency in actual time units.
- Display the measured frequency on the LCD with mHz resolution, e.g. “f = 23128.397 Hz”

Challenge #3: Implement an analog function generator using a PWM output passed through a low-pass filter. Your goal is to generate a 20 kHz sinusoid. The PWM duty cycle resolution should be 8-bit. See the Recommended Implementation for details. Continue monitoring the CPU load with PWM interrupts enabled.

Extra credit: Implement the 8-sample hardware FIFO for the ADC sampling at 1 Msps. Include a matching ISR that empties the FIFO into the ADC buffer (not using DMA). Measure the CPU load and complete the last entry in the table above.

Write a detailed lab report documenting your lab implementation. A lab report guide is available in the Assignments/Labs section on Canvas.

Recommended Implementation

Start by copying your Lab 2 project and naming the copy according to our convention.

Challenge #1: ADC I/O Optimization

The Direct Memory Access (DMA) controller can perform most of the duties of the Lab 2 ADC ISR without using any CPU time. While an ISR is still required, its relative deadline is nowhere near as short. A DMA interrupt occurs only when a long DMA transfer completes, not every ADC sample. The DMA setup code is given to you in the following code block.

```
#include "driverlib/udma.h"
```

```
#pragma DATA_ALIGN(gDMAControlTable, 1024) // address alignment required
tDMAControlTable gDMAControlTable[64];     // uDMA control table (global)
```

```
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);
```

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

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

The only mysterious parameter is “arbitration size.” The ADC sequence 0 can be configured as an 8-sample FIFO. Normally, DMA will transfer from the ADC FIFO to RAM even if only one sample is available. If this FIFO becomes half-full (4 samples), this indicates that the DMA subsystem is very busy. At this point, the ADC DMA is given higher priority so it could go ahead despite other DMA traffic competing for bandwidth. For the ADC, the arbitration size defines this priority increase threshold, as well as the longest DMA burst length. Longer bursts are more bandwidth efficient, but introduce longer latency to other DMA transfers. With only the ADC using DMA, this setting has no real effect.

In ping-pong mode, we are configuring two DMA channels that are assumed linked (complementary). 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**, enable the ADC1 sequence 0 **DMA interrupt**. The DMA feature for ADC1 sequence 0 must also be enabled in the ADC peripheral. The additional ADC setup code is partially given in the following code block.

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

The new ADC ISR is partially implemented in the following code block. Complete the missing functionality.

```
volatile bool gDMAPrimary = true; // is DMA occurring in the primary channel?

void ADC_ISR(void)  // DMA (lab3)
{
    ADCIntClearEx(...); // 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(...); // 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.
    <...>

    // 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
    }
}
```

Once you complete the DMA controller setup, ADC1 sequence 0 DMA interrupt setup and the ISR, run your code. You should see a waveform being continuously sampled into the ADC buffer. Unfortunately, `gADCBufferIndex` is no longer updated by the DMA mechanism. So your trigger search does not know where to find the newest sample in the ADC buffer.

Finding where DMA is currently writing samples into the ADC buffer is not too difficult. We can query the number of items **remaining** to transfer in a DMA channel using the uDMAChannelSizeGet() function. The only problem is that there are two DMA channels in ping-pong mode. We need to know which channel is currently sampling, and which is waiting for its turn. The ADC ISR supplies this information through the `gDMAPrimary` global. If we check which channel is currently sampling and then read the remaining transfer size for that channel, we run into the **non-atomic read** issue. By the time we get to read the remaining transfer size, the channel may finish and be restarted by the ISR. The following function returns the location of the newest DMA sample in the ADC buffer, but suffers from the non-atomic read issue. **Fix it using the appropriate TI-RTOS Gate object.** Change the priority of the ADC Hwi to make is **no longer 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.

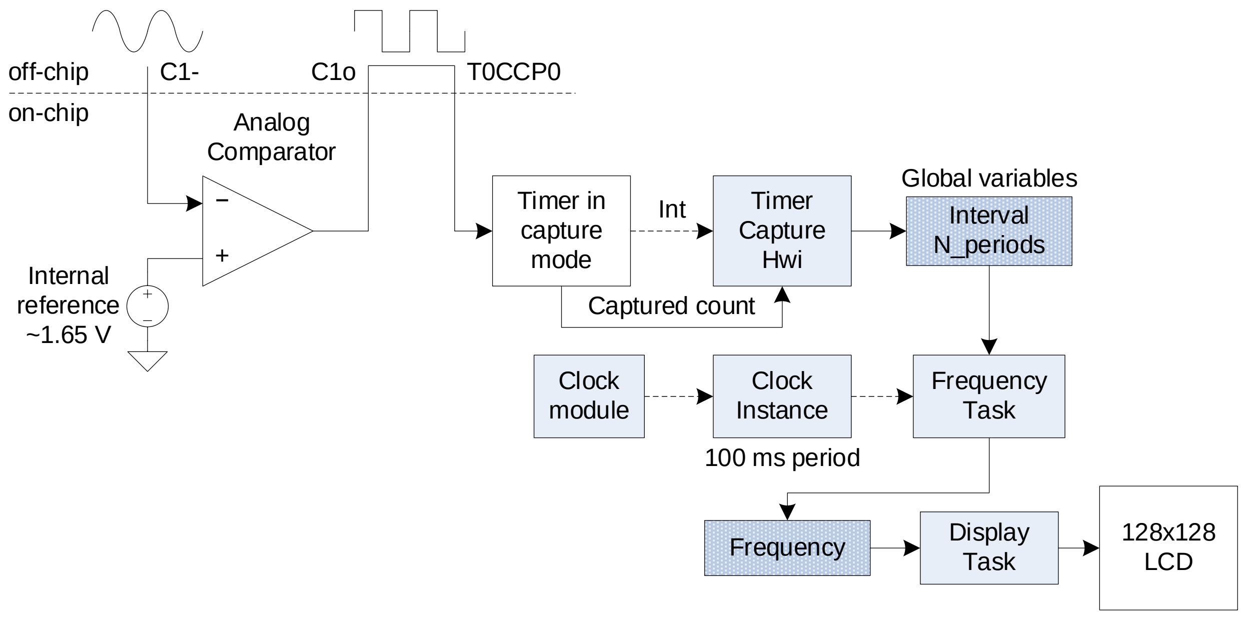
```
int32_t getADCBufferIndex(void)
{
    int32_t index;
    if (gDMAPrimary) {  // DMA is currently in the primary channel
        index = ADC_BUFFER_SIZE/2 - 1 -
                uDMAChannelSizeGet(UDMA_SEC_CHANNEL_ADC10 | UDMA_PRI_SELECT);
    }
    else {              // DMA is currently in the alternate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

Use the above function instead of reading the `gADCBufferIndex` variable in the Waveform Task. You should comment out the definition of `gADCBufferIndex` to make sure you are not using it anywhere anymore.

Measure the CPU load with this new DMA implementation of ADC sampling. Now increase the sampling rate to 2 Msps by adjusting the ADC clock. Measure the CPU load again and fill in the table given in the assignment portion of this document. You should now have plenty of CPU available for other demanding real-time tasks.

Challenge #2: Frequency Counter

The following figure shows the recommended implementation of the frequency counter. Note that this setup has a flaw (which we will not fix): An external signal is generating interrupts. The period of these interrupts is thus uncontrolled. At a high enough input signal frequency, the Timer Capture Hwi can starve lower priority tasks of the CPU.



Analog Comparator

Configure the Analog Comparator peripheral, comparator #1, as shown in the above block diagram. Most of the code is in the following code blocks. Fill in the blanks in the code by browsing the TivaWare Peripheral Driver Library User's Guide. Referring to the TI BoosterPack Checker website, use BoosterPack Connector #1, pin 3 for C1- and pin 15 for C1o. Pay close attention to the GPIOPinConfigure() call. It controls a multiplexer that selects the function of an I/O pin. Find the appropriate constant for the function argument in the header "driverlib/pin_map.h". If you misconfigure this multiplexer, your comparator output will not work. Once you have the comparator set up, run your code, connect the same signal going to AIN3 also to the C1- comparator input and monitor the C1o output using the bench oscilloscope. You should see a square wave.

```
#include "driverlib/comp.h"
#include "driverlib/pin_map.h"
```

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_COMP0);
ComparatorRefSet(COMP_BASE, ...);
ComparatorConfigure(COMP_BASE, 1, ...);

// configure GPIO for comparator input C1- at BoosterPack Connector #1 pin 3
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO?);
GPIOPinTypeComparator(...);

// configure GPIO for comparator output C1o at BoosterPack Connector #1 pin 15
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIO?);
GPIOPinTypeComparatorOutput(...);
GPIOPinConfigure(...);
```

Timer in Capture Mode

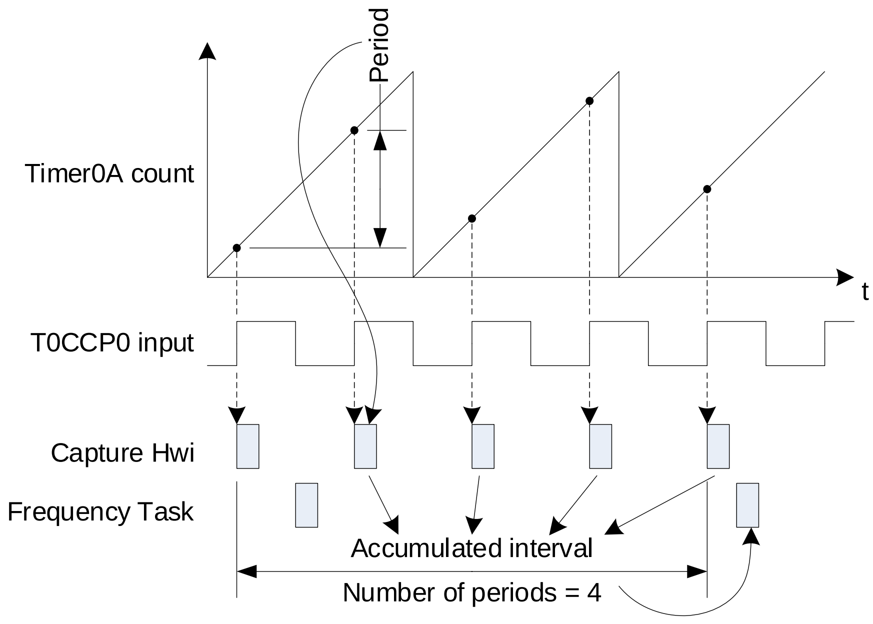
Configure Timer0A for Edge Time Capture Mode. This setup is nearly complete in the following code block. Again, browse "driverlib/pin_map.h" to locate the correct argument for GPIOPinConfigure(). Note that TI-RTOS typically uses Timer0 for the Clock module. Change the "**Timer Id**" under the Clock module settings to some unused timer other than 0. Connect the C1o pin to the T0CCP0 pin (they are conveniently right next to each other).

```
// 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(...);

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, ...);
TimerLoadSet(TIMER0_BASE, TIMER_A, 0xffff);   // use maximum load value
TimerPrescaleSet(TIMER0_BASE, TIMER_A, 0xff); // use maximum prescale value
TimerIntEnable(TIMER0_BASE, ...);
TimerEnable(TIMER0_BASE, TIMER_A);
```

Configure an Hwi to handle Timer0A Capture interrupts. Assign it priority such that it is **not** 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 it is close to what you expect using the debugger.

Frequency Measurement

The following figure illustrates frequency measurement setup.



In the Timer Capture Hwi add the current period into a multi-period interval (global). Also, count how many periods have been accumulated (another global).

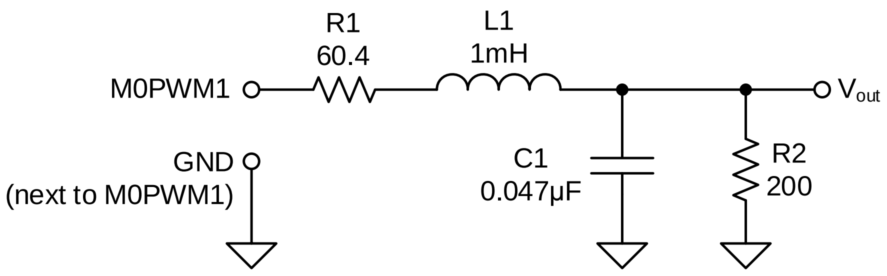
In the Frequency Task (signaled by a periodic Clock instance) determine the average frequency as the ratio of the number of accumulated periods to the accumulated interval. Also, **reset these globals back to zero** to start a new measurement.

Create a Clock instance with a 100 ms period. In the Clock function post to a Semaphore to signal the Frequency Task. Create the Frequency Task with a priority just below the Waveform Task. Create the semaphore for the Frequency Task. Create and use the appropriate TI-RTOS **Gate** object to protect access to the globals shared between the Frequency Task and the Timer Capture Hwi.

Verify that the computed frequency matches what your bench oscilloscope measures as the frequency of the oscillator feeding the C1- pin. Save the frequency into a global variable and then print it at the bottom of the LCD in the Display Task. The output should look like "f = 23128.397 Hz." You may test the sensitivity of your frequency measurement by placing a metal object near the inductor, or heating the inductor with your fingers. You should see detectable frequency changes.

Challenge #3: PWM Function Generator

We will use a PWM output as a form of digital-to-analog conversion. By filtering the PWM output using an RLC low-pass filter, we produce 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). Build the following low-pass filter on your breadboard and connect to the M0PWM1 (GPIO PF1) and GND pins next to each other on EK-TM4C1294XL. Twist these jumper wires together to minimize noise. (If you are interested in the filter design, it is a Chebychev 2nd order low-pass filter with 0.5 dB ripple and 0.5 dB cutoff frequency of 23.1 kHz. Its attenuation increases to 3 dB at 32 kHz and 49 dB at 465 kHz. The resistor R1 is smaller than this design requires because it adds in series to the PWM pin output resistance of ~40 Ω at 8 mA drive.)



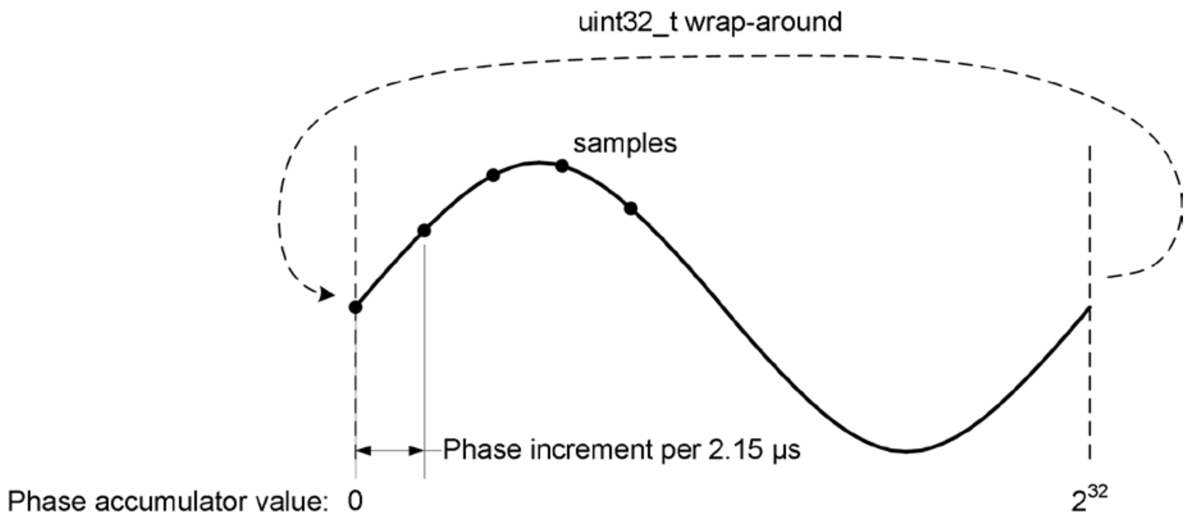
The PWM peripheral initialization is given to you in the following code block.

```
#include "driverlib/pwm.h"
#define PWM_PERIOD 258  // PWM period = 2^8 + 2 system clock cycles
// use M0PWM1, at GPIO PF1, which is BoosterPack Connector #1 pin 40
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);
```

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 0, output 1, also labeled output B on the generator block.

The selected PWM period is the shortest 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 ($2^8 + 2$) system clock cycles. This translates to a PWM frequency of 465 kHz. You may run the above initialization code and verify the 50% duty cycle PWM waveform on the bench scope.

We now need to vary the duty cycle to create a low-frequency waveform. The goal is to generate a 20 kHz sinusoid. The recommended method is to program **one period** of the waveform into a lookup table. Then, in the PWM ISR, determine the index into this table using an unsigned 32-bit **phase accumulator**. The phase accumulator range is mapped to a single cycle of the periodic waveform (one period = 2^{32} phase steps), as shown in the following figure. Every interrupt, occurring with a period of 2.15 μs (= 258 / (120 MHz)), we add a phase increment such that we obtain a sample of a 20 kHz sine wave. When the phase accumulator is incremented beyond the maximum `uint32_t` value, it gracefully wraps around, starting a new period. The phase increment value (it will be quite a large integer) controls the generated waveform period with high precision. We use the most significant bits of the phase accumulator as the index into the waveform table.



The PWM ISR is mostly given to you in the following code block. Your tasks are to determine the value of the phase increment, clear the PWM interrupt flag, fill the waveform table, and configure the PWM Hwi instance. Fill the 1024-sample waveform table with a single period of a sine wave with the maximum amplitude that fits into the unsigned 8-bit integer range. Verify your waveform table contents using the debugger. The PWM Hwi should be a **"zero-latency Hwi"** in order to meet its deadlines. Double-check the priority assignments of all three Hwi in your system.

```
#include "inc/tm4c1294ncpdt.h"
uint32_t gPhase = 0;              // phase accumulator
uint32_t gPhaseIncrement = ?;     // phase increment for 20 kHz

#define PWM_WAVEFORM_INDEX_BITS 10
#define PWM_WAVEFORM_TABLE_SIZE (1 << PWM_WAVEFORM_INDEX_BITS)
uint8_t gPWMWaveformTable[PWM_WAVEFORM_TABLE_SIZE] = {0};

void PWM_ISR(void)
{
    <...>; // clear PWM interrupt flag

    gPhase += gPhaseIncrement;

    // write directly to the Compare B register that determines the duty cycle
    PWM0_0_CMPB_R = 1 + gPWMWaveformTable[gPhase >> (32 - PWM_WAVEFORM_INDEX_BITS)];
}
```

This scheme works correctly because the PWM module delays updates of the duty cycle to the next time the PWM counter reaches 0. Since the interrupts also trigger then the PWM counter reaches 0, the PWM ISR's relative deadline is the same is its period.

Run your code and verify the generated 20 kHz sine wave either on the bench oscilloscope or your own oscilloscope still running on your lab kit. Capture a screenshot of this waveform together with its frequency measurement on the oscilloscope for your report. Also record the CPU load. If the waveform is very noisy, verify your ground connections. The PWM output jumper wire should be tightly wrapped around a GND jumper wire. These wires should plug into the EK-TM4C1294XL board right next to each other. The AIN3 jumper wire should likewise be wrapped around a GND wire that plugs in close by. You may also enable BW limit and noise reduction in the bench scope.

Extra credit: ADC FIFO

So far we have implemented the single-sample ISR and DMA I/O modes for the ADC. For extra credit, configure the ADC1 sequence 0 as an 8-sample hardware FIFO. It should interrupt every 4 samples. You have to program all 8 steps of sequence 0 to sample AIN3, and include the interrupt and end-of-sequence flags in the appropriate steps. Write an ISR that empties the hardware FIFO into the ADC buffer. Sample at 1 Msps. Measure the CPU load in this configuration, with the Timer Capture and PWM interrupts disabled. Complete the table in the assignment part of this document. Leave all three versions of your ADC I/O code available (unused versions can be commented out or conditionally compiled). Submit your Lab 3 source code configured in the DMA mode.