

Lab 3: Advanced I/O

Yil Verdeja, ECE Box 349

April 30 2019

Table of Contents

1 Introduction	2
2 Methodology, Results and Discussion	2
2.1 Setting up DMA	2
2.2 CPU Load and Increasing Sampling Rate	4
2.3 Analog Comparator	4
2.4 Edge Timer Capture Mode	6
2.5 Frequency Measurement	7
2.6 Generating an 18 kHz Sinusoidal	9
3 Conclusion	12

List of Figures

Figure 1	GateHwi Protection in getADCBufferIndex	3
Figure 2	Oscilloscope Functionality Working	3
Figure 3	Recommended Implementation for the frequency counter	5
Figure 4	Comparator Input (blue) and Output (yellow) Comparison	5
Figure 5	Timer Capture ISR	6
Figure 6	Period Verification of the Timer Capture ISR	6
Figure 7	Frequency Measurement Setup	7
Figure 8	Frequency Task Implementation	8
Figure 9	Frequency Measurement on Digital Oscilloscope	9
Figure 10	Chebyshev RLC Low-Pass Filter	10
Figure 11	50% Duty Cycle Verification of PWM	10
Figure 12	One Period of Waveform Lookup Table	11
Figure 13	Phase Accumulator Value to a period of sinusoidal waveform	11
Figure 14	18 kHz Sinusoidal Waveform from varying PWM Duty Cycle	12

List of Tables

Table 1	CPU Load at different sampling rates and different ADC configurations	4
----------------	---	---

1 Introduction

The purpose of this project was to (1) optimize the analog to digital converter (ADC) input and output (I/O) by using direct memory access (DMA), (2) measure the frequency of an analog signal, (3) use an analog comparator peripheral, (4) use a timer in capture mode and (5) use a pulse width modulator (PWM) with duty cycle adjusted every period to generate analog waveforms.

After using a DMA to optimize the ADC I/O, the sampling rate was bumped to 2 Msps and the CPU load was measured. Table 1 displays how the CPU load was affected after using the DMA and after switching from 1 Msps to 2 Msps. By using a timer in capture mode, a accurate frequency counter was created to measure the frequency of an oscillating signal. Finally, from a filtered PWM output, an analog function generator was implemented.

2 Methodology, Results and Discussion

This section describes the implementation of each major step in the lab by going through them and describing how each was completed. There were three challenges to this project:

1. Optimize the ADC I/O using DMA
2. Add an accurate frequency counter to the oscilloscope system on the embedded board
3. Implement an analog function generator using a PWM output passed through a low-pass filter

The first challenge was completed by using the initialization code provided and by comparing the CPU load before and after optimization. The second challenge displayed measured frequency on the liquid crystal display (LCD) with mHz resolution. The third challenge was completed after generating an 18 kHz sinusoidal waveform. With all hardware interrupts enabled, the CPU load was measured again and compared to the oscilloscope implemented in the previous lab. By using a DMA, the CPU load on the display task was much less.

2.1 Setting up DMA

The DMA was set up using the initialization code provided. This controller can perform most of the duties of the ADC ISR without using any CPU time. Unlike interrupting after every ADC sample, a DMA interrupt occurs only when a long DMA transfer completes.

Once the DMA controller was set up and the ADC functions were updated, a GateHwi object was added to protect the global shared variables from being accessed by the ADC ISR. The ADC Hwi was then changed from a zero-latency Hwi to having an interrupt of 64.

```
int32_t getADCBufferIndex(void)
{
    int32_t index;
    IArg key;
    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 1. GateHwi Protection in getADCBufferIndex

To verify functionality, a picture of the oscilloscope waveform was taken as shown in Figure 2. After implementing the DMA, the trigger functionality was removed therefore the waveform on the oscilloscope switches between falling and rising trigger. This switch can't be seen in Figure 2.

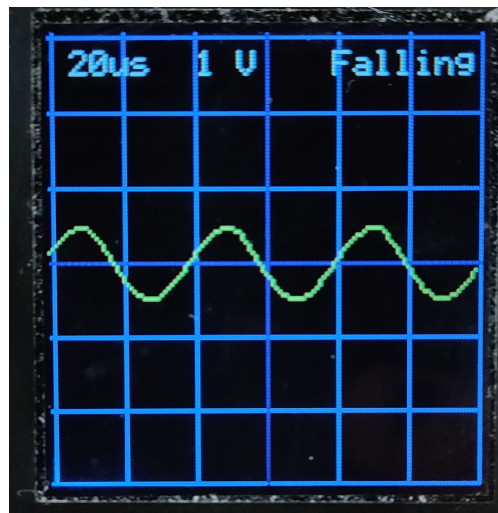


Figure 2. Oscilloscope Functionality Working

In doing this initial part of the lab, there was a single problem that took nearly two days to debug. The problem was in a misspelling of a variable or function in the DMA initialization code. Due to the minor error, it was often looked over.

2.2 CPU Load and Increasing Sampling Rate

This section continues after the DMA initialization. Prior to initializing the DMA controller, the CPU load of the oscilloscope was measured at a value of 75%. After implementing the DMA, the CPU load was measured again with the new DMA implementation of the ADC sampling. As expected, the load was minimal. By adjusting the ADC clock, the sampling rate was increased to 2 mega samples per second and the load was measured once more as shown in the table below.

Table 1. CPU Load at different sampling rates and different ADC configurations

ADC Configuration	Sampling Rate	CPU Load	ISR Relative Deadline
Single-sample ISR	1 Msps	0.75	1us
DMA	1 Msps	0.019	1024us
DMA	2 Msps	0.025	2048us

As shown, it is very clear that using the DMA reduced the CPU load tremendously. Now, having implemented this, there is plenty of CPU available for other demanding real-time tasks.

The ISR relative deadline is equal to the period of the ISR. To determine the period, the inverse of the sampling rate was found. However, since the DMA only interrupts after long DMA transfers complete, the sampling rate is divided by 1024 (the uDMA transfer size is limited to 1024 items). The equation for the relative deadline for the DMA is shown below.

$$\left(\frac{\text{Sampling Rate}}{\text{Buffer Size}}\right)^{-1} \quad \text{Eq. 1.0}$$

2.3 Analog Comparator

This section is the beginning of challenge 2, in implementing the frequency counter of the oscilloscope system. The figure below represents a recommended implementation for the frequency counter. As shown on the left, an analog comparator must be implemented to generate interrupts. This external signal has a flaw - the periods of these interrupts are not controlled, therefore at high enough input signal frequency, the timer capture Hwi can starve lower priority tasks of the CPU.

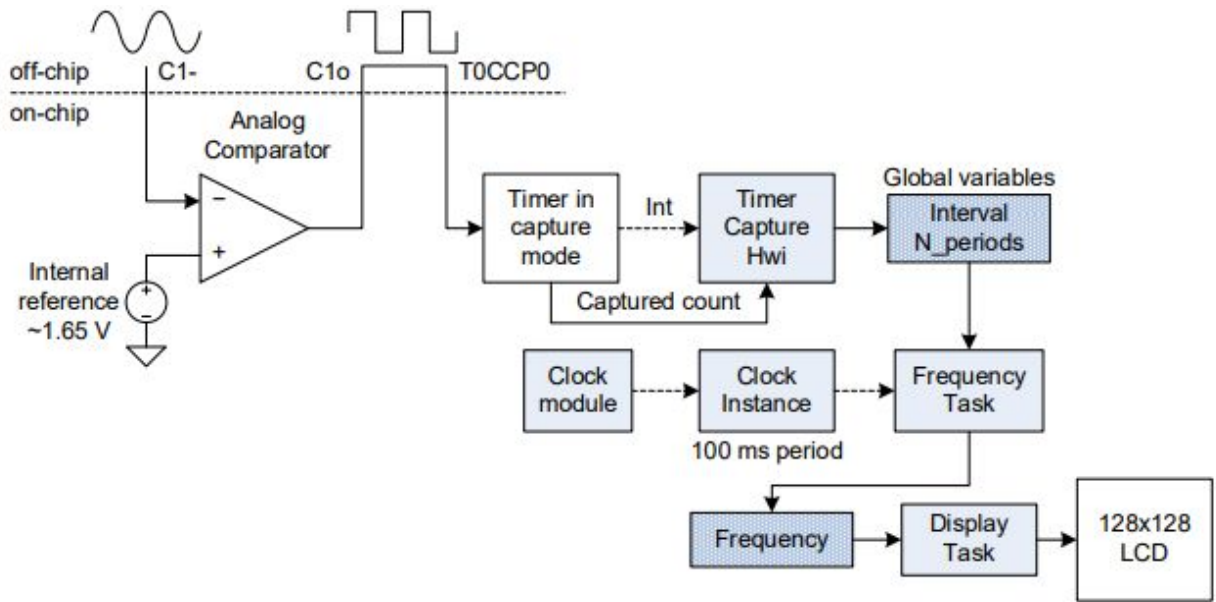


Figure 3. Recommended Implementation for the frequency counter

The analog comparator was initialized using provided code. By connecting the same signal to AIN3 to the C1 comparator input, the C1o comparator output was monitored using the bench oscilloscope. As shown in the figure below, the sinusoidal waveform represents the input to the comparator, while the square waveform is the output.

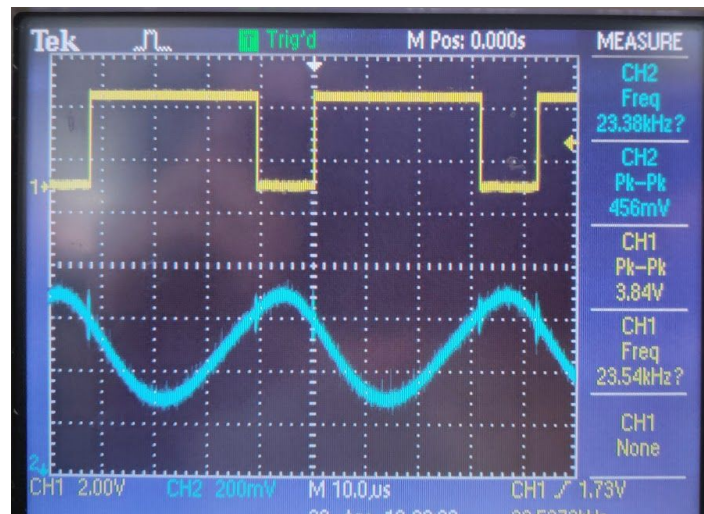


Figure 4. Comparator Input (blue) and Output (yellow) Comparison

2.4 Edge Timer Capture Mode

In this step, the Timer0A was configured for Edge Time Capture Mode. Prior to implementing the Timer0A initialization code, the TI-RTOS clock module ID was changed to use a different timer than Timer0.

After implementation, a timer capture Hwi was configured to handle Timer0A capture interrupts. It was assigned to a priority of 32 which is a higher priority than the ADC Hwi. Inside the timer capture ISR, the flag was cleared and from the number of counts measured between each interrupt, the period was determined. This implementation is shown in Figure 5, and it was verified at a period value of 5000 μ s (see Figure 6).

```
// Timer Capture ISR is an interrupt that calculates the period of the ISR
void timercapture_ISR(UArg arg0){
    // Clear the timer0A Capture interrupt flag
    TIMER0_ICR_R = TIMER_ICR_CAECINT;

    // Use timervalueget() to read full 24 bit captured time count
    uint32_t currCount = TimerValueGet(TIMER0_BASE, TIMER_A);

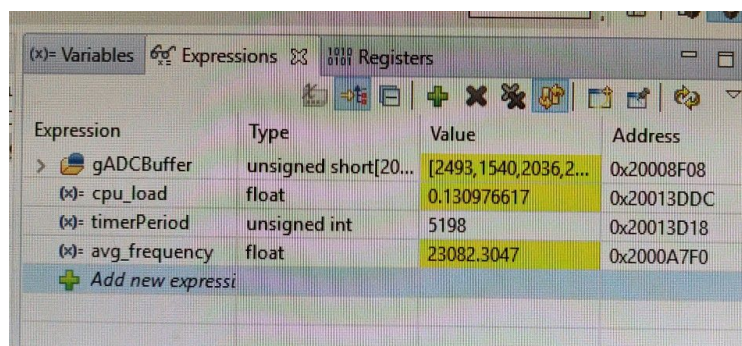
    timerPeriod = (currCount - prevCount) & 0xFFFFF;

    prevCount = currCount;

    multiPeriodInterval += timerPeriod;
    accumulatedPeriods++;
}
```

Figure 5. Timer Capture ISR

The TimerValueGet() function was used to read the full 24-bit captured timer count, while the hexadecimal value (0xFFFFF) at the end of the period measurement took care of wraparound.



Expression	Type	Value	Address
> gADCBuffer	unsigned short[20...	[2493, 1540, 2036, 2...	0x20008F08
(x)= cpu_load	float	0.130976617	0x20013DDC
(x)= timerPeriod	unsigned int	5198	0x20013D18
(x)= avg_frequency	float	23082.3047	0x2000A7F0
+ Add new expressi			

Figure 6. Period Verification of the Timer Capture ISR

On the board, the output of the comparator C1o was connected to the timer input T0CCP0 pin. Every positive edge, the timer capture Hwi interrupted. This is further shown in Figure 7.

2.5 Frequency Measurement

The following figure shows the frequency measurement setup. The frequency task measures the frequency by obtaining the ratio of the accumulated period intervals to the number of periods that have occurred.

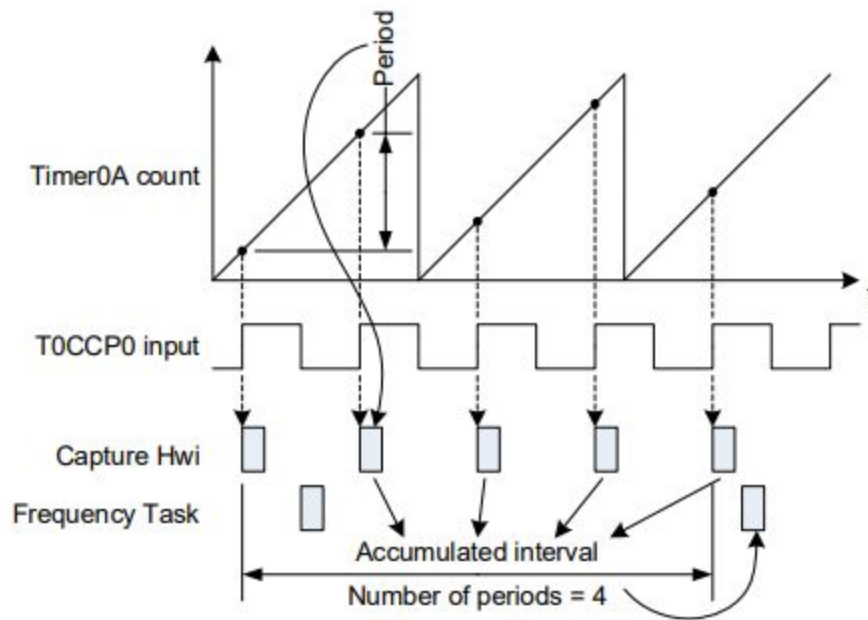


Figure 7. Frequency Measurement Setup

To obtain this ratio value, the Timer Capture ISR was changed to include a multi-period interval and a count of how many intervals have accumulated (see Figure 5). Signaled by a periodic clock instance every 100ms, the frequency task was implemented as shown in Figure 8. For every new measurement, the globals were reset back to zero.


```

// Determines the average frequency as the ratio of the number of accumulated periods
// to the accumulated interval, converted from
// clock cycles to seconds
void frequencyTask_func(UArg arg1, UArg arg2) {
    IArg key;
    uint32_t accu_int, accu_count;

    while (true) {
        Semaphore_pend(semFrequency, BIOS_WAIT_FOREVER); // from clock

        // Protect global shared data
        key = GateHwi_enter(gateHwi0);
        // Retrieve global data and reset
        accu_int = multiPeriodInterval;
        accu_count = accumulatedPeriods;
        // Reset globals back to 0
        multiPeriodInterval = 0;
        accumulatedPeriods = 0;
        GateHwi_leave(gateHwi0, key);

        // determine average frequency
        float avg_period = (float)accu_int/accu_count;
        avg_frequency = 1/(avg_period) * (float)gSystemClock;
    }
}

```

Figure 8. Frequency Task Implementation

The frequency task was created with a priority just below the button task, and the global variables shared between the frequency task and the timer capture Hwi were protected using a GateHwi object as shown in the figure above. To verify whether the frequency measurement, the measurement was saved as a global and displayed at the bottom of the LCD. As shown in Figure 9, the frequency matched the 23 kHz measured in Figure 4.

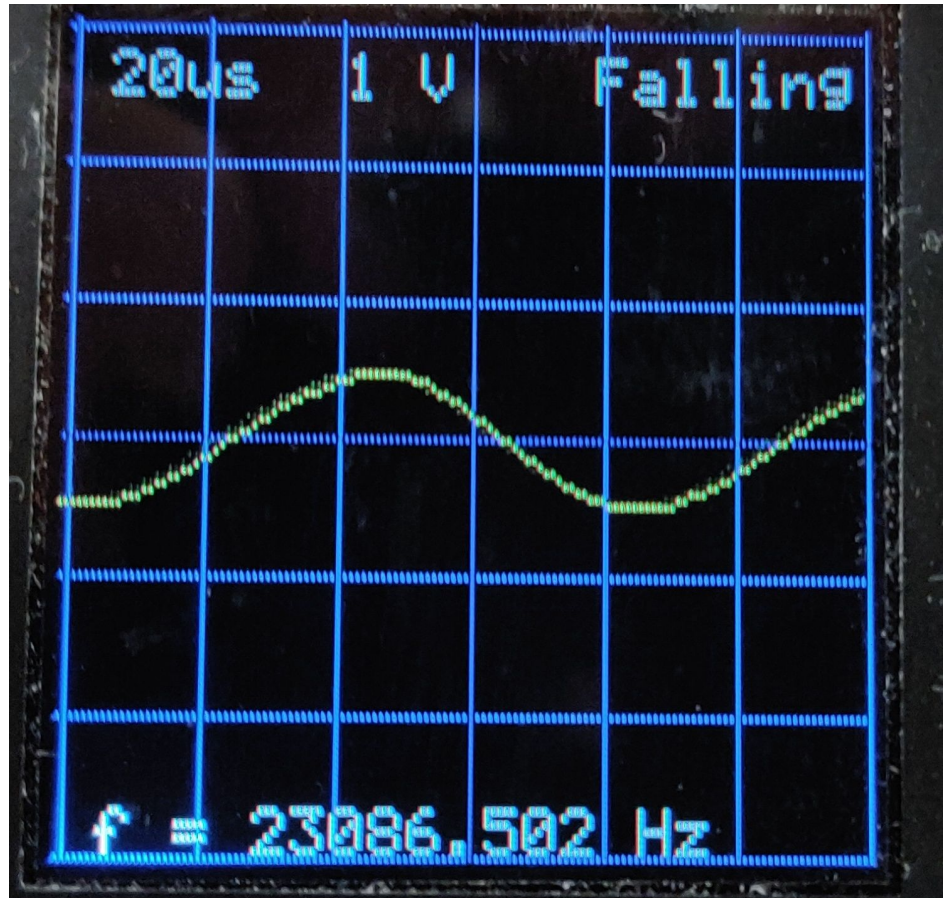


Figure 9. Frequency Measurement on Digital Oscilloscope

This was deemed an accurate frequency measurement tool since it could pick up very sensitive changes. To test this, a metal object was placed near the inductor showing detectable frequency changes.

2.6 Generating an 18 kHz Sinusoidal

Using a PWM output on the board, a form of digital-to-analog conversion was created. This was done by filtering the PWM output using an RLC low-pass filter as shown in Figure 10. The output of the filter produces a voltage proportional to the PWM duty cycle. By changing the PWM duty cycle, it is possible to create a sinusoidal waveform.

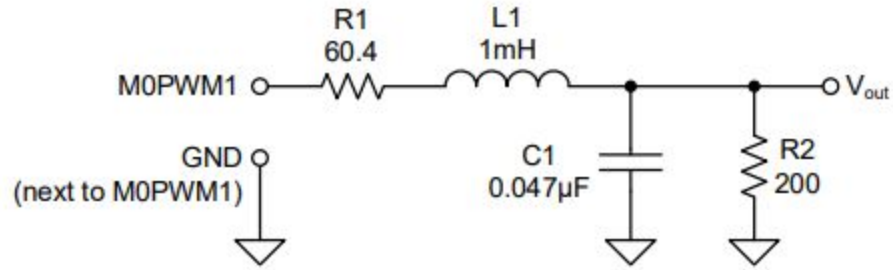


Figure 10. Chebychev RLC Low-Pass Filter

Using the initialization code provided, the PWM peripheral was implemented easily. Since the PWM period achieves an 8-bit duty cycle resolution, the PWM frequency is at 465 kHz. To verify this, the PWM waveform was tested on the bench scope as shown in Figure 11 to show a 50% duty cycle.



Figure 11. 50% Duty Cycle Verification of PWM

To create the 18 kHz low-frequency waveform, the duty cycle needs to vary. In order to obtain this easily, one period of the waveform was programmed using a *lookup table*. As shown in Figure 12, a loop was created to fill the lookup table with the specified values.

```

// Fills the waveform table
void fillWaveformTable(void) {
    int i;
    float a= 255.0/2.0;
    float b = 2*PI/PWM_WAVEFORM_TABLE_SIZE;
    for (i = 0; i < PWM_WAVEFORM_TABLE_SIZE; i++) {
        gPWMWaveformTable[i] = (uint8_t)(a*sinf(b*i) + a);
    }
}

```

Figure 12. One Period of Waveform Lookup Table

As shown in Figure 13, one period of the waveform is mapped to the 32-bit phase accumulator range. Every 2.15 μ s, the phase is incremented by a certain amount. This amount is determined below as the phase increment variable. This variable controls the generated waveform period with high precision.

$$\begin{aligned}
 period &= \frac{1}{18 \text{ kHz}} = 55.56 \mu\text{s} \\
 samples &= \frac{period}{2.15 \mu\text{s}} = \frac{55.56 \mu\text{s}}{2.15 \mu\text{s}} = 25.84 \\
 phase \text{ increment} &= \frac{phase \text{ steps}}{samples} = \frac{2^{32}}{25.84} = 166215234.355
 \end{aligned}$$

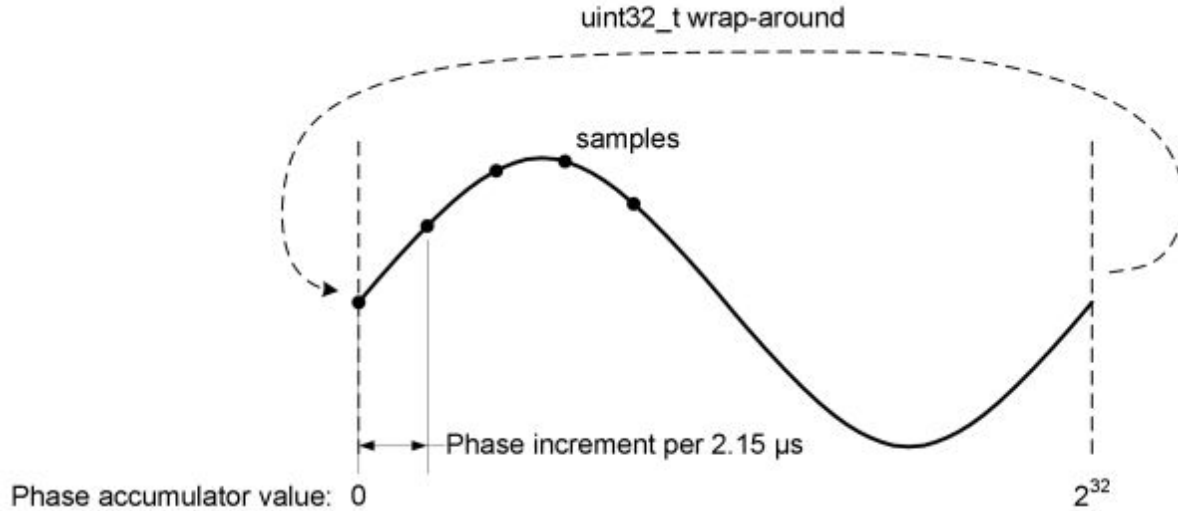


Figure 13. Phase Accumulator Value to a period of sinusoidal waveform

Unlike the previous ISRs, the PWM Hwi was set as a zero-latency Hwi in order to meet all its deadlines. After uploading the code to the board, the output of the RLC low-pass filter was verified on the oscilloscope. Figure 14 shows an 18 kHz sinusoidal waveform besides a PWM signal with varying duty cycle.

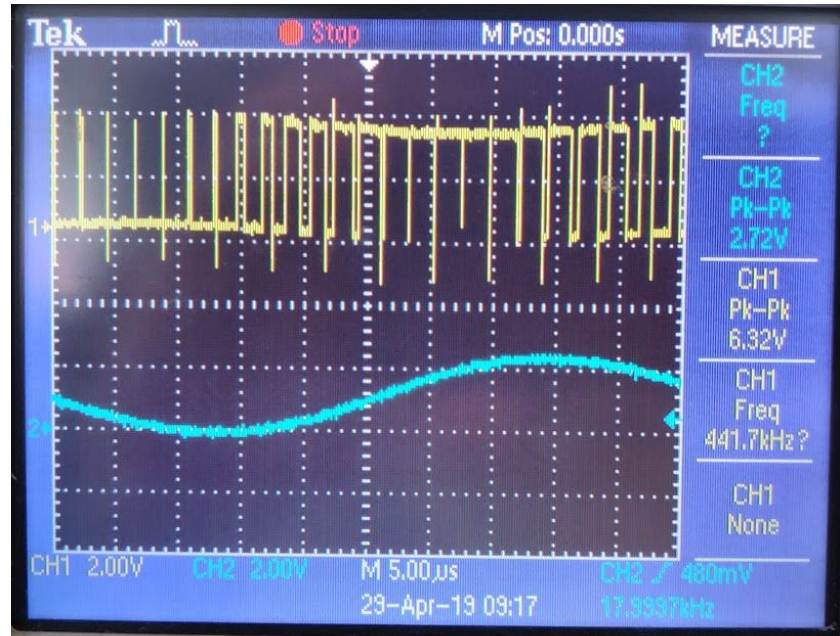


Figure 14. 18 kHz Sinusoidal Waveform from varying PWM Duty Cycle

The final CPU load after the PWM implementation was: 47.45%. This value is still much less than the ADC sampling without the DMA controller.

3 Conclusion

The most important outcome of this project was to implement other demanding real-time tasks such as the PWM with adjustable duty cycle and the timer in capture mode by reducing the CPU load via a DMA controller implementation of the ADC sampling.

Debugging in real-time proved to be a great tool because it allowed to better understand what was happening in the code, but it also helped in fixing errors when they occurred. Two major problems that occurred were from using the wrong variable names or function calls. As mentioned, the first problem in implementing the DMA was deemed very hard to debug. In the end the error was caught by replacing the code written with the *correct implementation*. Although it may *seem* or *look* right, it does not imply that it is right. The second problem encountered was inside the PWM ISR. Since the interrupt was cleared incorrectly, the PWM ISR was always being called. This shows that although these mistakes are minor, they can negatively impact the outcome of the project if they are overlooked.