

ME 461 Laboratory #2

Timers and Pulse-Width Modulation

Goals:

1. Understand how to use timers to control the frequency at which events occur.
2. Generate PWM signals using Timer A.
3. Explore the frequency characteristics of PWM signals.
4. Create a “home-made” DAC.

Exercise 1:

For this exercise, you will change the blink rate of the LED connected to P1.0 using two methods. First, implement a strictly software change, that is, do not modify any control registers. *Hint:* the code you need to modify is in the Timer A interrupt service routine. First halve and then double the blink rate. Print the approximate period (in milliseconds) to the terminal window. Note that in changing the blink rate you also changed how often the program prints to the serial port. **Demonstrate to your TA.** Remember this method, as it is a simple way to implement independent timing intervals (i.e. a schedule) in your program.

Most of the configuration of Timer A has already been done for you in the default project. This would be a good time to review, the MSP 430 User’s Guide Timer_A section, and the “msp430x22x2.h” header file. Recall again that many of the control register names, along with things such as predefined constants with (relatively) easy to remember names, are declared in the “msp430x22x2.h” header file. For example, in the default project code, you will see many examples of things such as the following:

```
TACTL = TASSEL_2 + MC_1;           // source SMCLK, up mode
```

On line 459 of the “msp430x22x2.h” header file you will find:

```
#define MC_1           (1*0x10u)  /* Timer A mode control: 1 - Up to CCRO */
```

On line 468 of the “msp430x22x2.h” header file you will find:

```
#define TASSEL_2      (2*0x100u)  /* Timer A clock source select: 2 - SMCLK */
```

Comparing these definitions of what are essentially constants to the details of the TACTL Timer_A control register (Timer_A section of the MSP 430 User’s Guide), you will see that the #defines and the TACTL line of C code sets bits 9-8 of the TACTL (Timer_A) control register equal to $(10)_2$ and bits 5-4 equal to $(01)_2$, which is exactly what you need to do to configure Timer_A to use the SMCLK as its source and count in UP mode. There will be many examples of this throughout this and the remaining labs. You are strongly urged to become familiar with these concepts and make use of them.

Make sure you understand this default configuration: Timer A running in “up” mode, clock source is SMCLK, TACCR0 interrupt is enabled. Change the period register TACCR0 so that the interrupt service routine is called at a different rate. Again, halve and then double the blink rate of the LED from its default of 1Hz. Print the approximate period (in milliseconds) to the terminal window. **Demonstrate to your TA.**

Exercise 2:

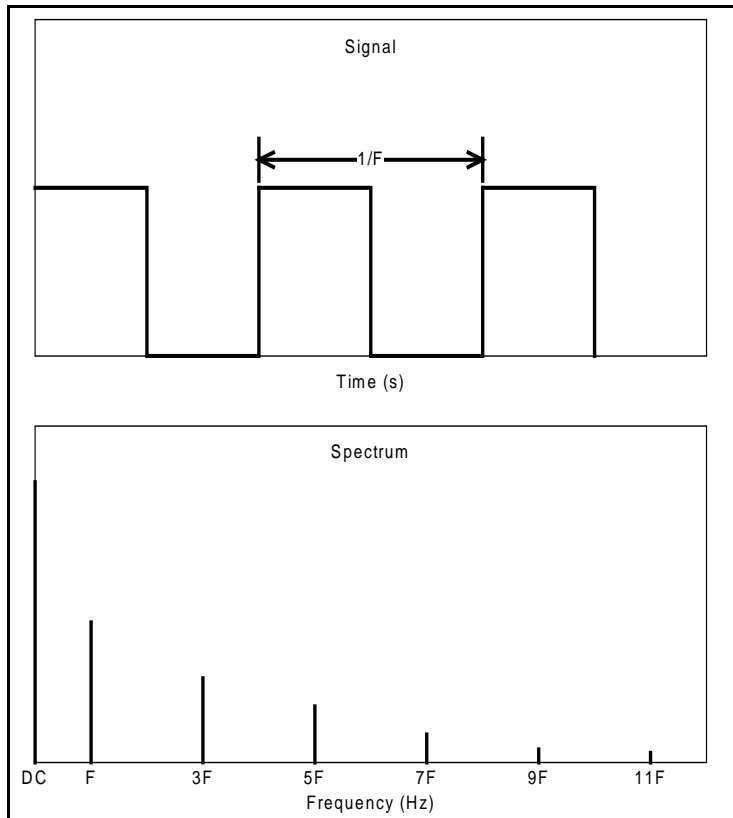
Now you will explore the effects of increasing the blink rate to a much higher frequency. Configure Timer A for PWM generation on channel 1. That is, set the CCR1 OUTMOD bits appropriately; set/reset or reset/set should work for our application (but one is better). Output the TA1 signal to pin 1.2 by setting that pin to its alternative function with the P1SEL register and by setting its direction to output.

First, set the PWM frequency to 40Hz. You will have to use the timer’s input divider to reach this frequency. The PWM frequency can be calculated by the following formula: $CR/CCR0$, where **CR** is the timer clock rate (after any division) in appropriate units. As you should already know from lecture, CCR1 sets the duty cycle of the PWM signal (the duty cycle will be $CCR1/CCR0 \times 100\%$). In your program, set CCR1 so that the LED connected to pin 1.2 flashes with a duty cycle of 50%, 90%, and then 10%. You may find it a useful to scope the voltage at pin 1.2 to help you visualize the PWM signal. **Demonstrate to your TA.**

The code you have developed thus far has produced visible binary intensity levels (ON and OFF) on the LEDs (Figure 1 below). However, it is possible to produce apparent analog intensity levels between ON and OFF without using analog outputs by increasing the flashing frequency and varying the duty cycle. Increase the PWM frequency by an order of magnitude or more (any frequency above 100Hz will do). Now, observe the results by driving the LED with the same duty cycles as before. **Demonstrate to your TA.**

Exercise 3:

In this exercise, you are going to explore the characteristics of PWM signals and exploit those characteristics to create a Digital-to-Analog Converter (DAC). You may recall from your signal processing class that the spectrum of a non-harmonic periodic signal is discrete and infinite. A square wave, in particular, comprises a DC component and an infinite series of sine functions at odd multiples of the fundamental frequency.



A square wave and its spectrum

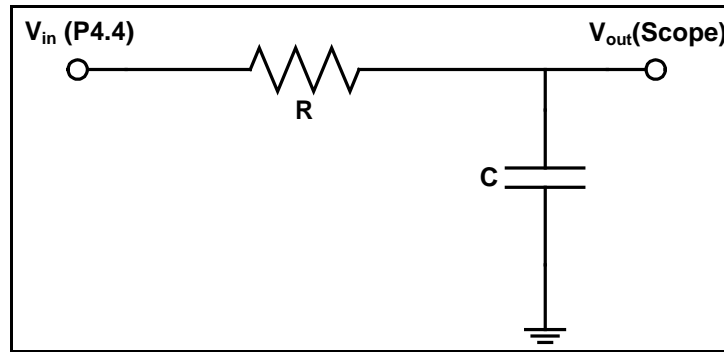
In the case of the PWM signal in question, the fundamental frequency is exactly the PWM frequency. The DC component of the Fourier spectrum is given by the following formula.

$$X[0] = \frac{1}{T} \int_0^T f(x) dx \quad \text{See Alciatore & Hinstand Chapter 4.3 for more explanation}$$

Here, T is the fundamental period of the signal and $f(x)$ is the varying duty cycle square wave called the PWM signal. Thus, we can see that **the DC component of the PWM signal is simply the fraction of the period over which the signal is high (the duty cycle)**. So, if we could simply extract the DC component of the PWM signal, we'd have a DC waveform with amplitude proportional to the duty cycle. This is exactly what you will do. In fact, you have unknowingly done it already in exercise 2. You filtered the PWM signal manifested on the LED with your eyes, so that your brain interpreted the light coming from the LED as an analog, non-blinking signal.

Solder the RC lowpass circuit shown below onto your breakout board. Your TA will provide values for R and C . Record these values below.

R: _____ C: _____

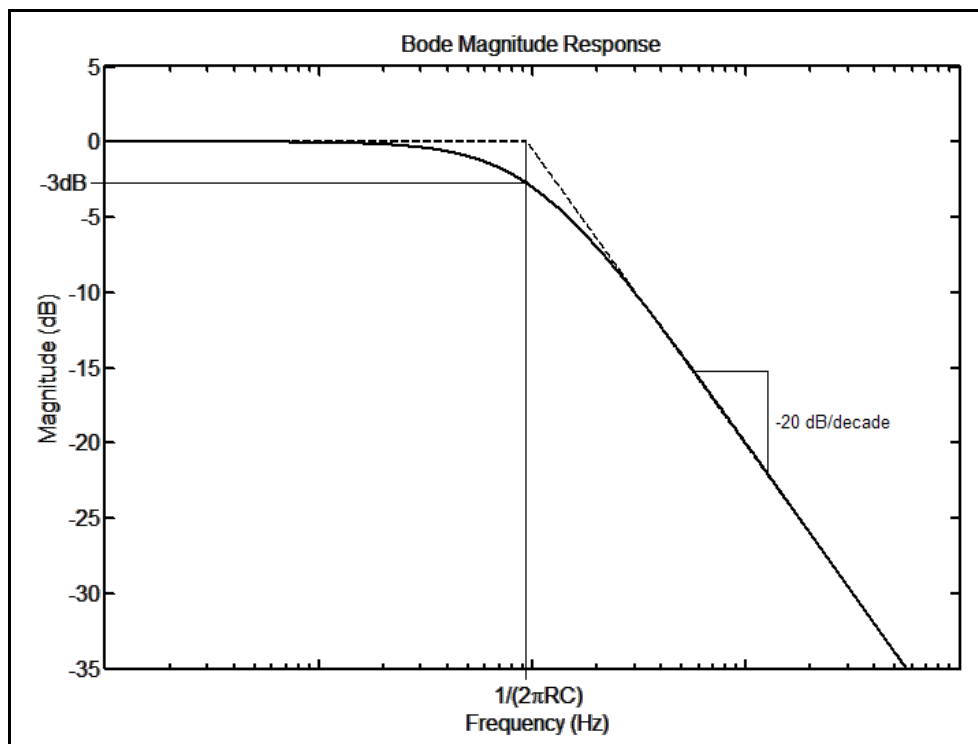


RC Lowpass Circuit

This circuit has a single-pole transfer function with cutoff frequency at $1/RC$ rad/s. Record the actual cutoff frequency below.

f_c : _____ Hz

The Bode magnitude plot of this transfer function is shown below.

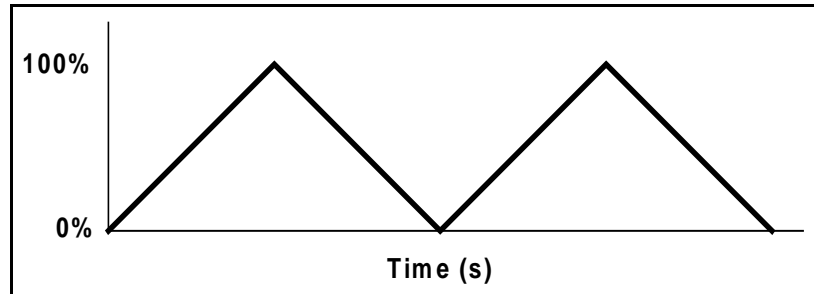


Magnitude Response of 1st-order transfer function $1/(1 + RCs)$

As you can see, the filter attenuates frequencies above the cutoff. However, the roll-off of this simple filter is rather low (-20dB/decade), so you will use a PWM frequency high enough that all of the harmonics are blocked. Configure Timer B to produce a 10kHz PWM signal and output that signal (TB1) on pin 4.4. Scope the output of your filter as you change the duty cycle. You should see that the result is an analog waveform whose amplitude ranges between 0V and V_{CC} . Also, print out the expected value of

the DAC voltage in millivolts to the serial port, calculated through the relation $V_{OUT} = V_{CC} \times \text{Duty Cycle}$. Don't forget that you can't use floating-point numbers, so a **Demonstrate to your TA**.

In your program, vary the duty cycle so that the result is a triangle waveform as shown below.



Ramped DAC Voltage Pattern

The frequency of the waveform is irrelevant so long as it is not so fast as to encroach on the cutoff frequency (~ 100 Hz) of the RC filter. **Demonstrate to your TA**. You should see how, with some straightforward extensions, our home-made DAC could become a home-made function generator! As part of the checkoff, answer the following question.

What is the resolution (in bits) of your home-made DAC? *Hint*: think about how many steps/increments there are between 0 and 100% duty cycle.