EECS 461 Fall 2009
Lab 5: Interrupts, Timing, and Frequency Analysis of PWM Signals

1 Overview

In the first four labs, you have not dealt with time in the design of your code. For many applications (in fact, for almost all embedded control applications), time is an essential element. For example, sampling an input signal or generating a periodic output requires that your code run periodically. To solve this problem, the MPC5553 provides several timers, all of which can be configured to produce interrupt requests (IRQs) at timed intervals.

In this lab you will learn how to set up the MPC5553's decrementer interrupt to periodically call your own C function. A function that is called in response to an IRQ is referred to as an interrupt service routine (ISR). You will write three ISRs; one of these will periodically sample a sinusoidal input and the other two will generate samples of a sine wave numerically. The samples of a sine wave, whether from an external analog input or internally generated in software, will be used to modulate the duty cycle of a PWM signal. You will then reconstruct the sine wave from the PWM signal using a low-pass filter to attenuate the high frequency content of the PWM, leaving only the (sinusoidal) modulation frequency.

2 Design Specifications

2.1 Hardware

To generate an IRQ at a specific frequency, the decrementer (DEC) uses a 32-bit down-counter, generating an IRQ each time the counter reaches zero. A 32-bit register, called DECAR, contains the value that is automatically reloaded into the DEC counter after reaching zero. In this way, DECAR determines the period of the DEC counter. Because the DEC resets itself automatically, the IRQs are guaranteed to occur with a constant period and the ISR that is called in response to the IRQ is called at a fixed frequency. By adjusting DECAR, the DEC can generate IRQs over a wide range of frequencies. For more details about the configuration and operation of these timing registers, see Section 2.9 of the Freescale e200z6 PowerPC Core Reference Manual.

In this lab, you will write three ISRs: IsrA, IsrB and IsrC, all of which will generate PWM signals where the duty cycle of the PWM encodes the values of a sine wave. IsrA will use the first channel of the eQADC module (AN0) to sample a sine wave produced by a function generator. IsrB and IsrC will generate samples of a sine wave numerically, where IsrB will call the C library function `sin()` and IsrC will use a look-up table. The PWM output generated by each ISR will be sent through a low-pass filter, with frequency response shown in Fig. 1. On an oscilloscope, you will view the input sinusoid, output PWM signal, and filtered PWM signals. The filter in the lab is a Sallen-Key implementation of a Butterworth filter.

2.2 Software

eecs461.h and eecs461.c

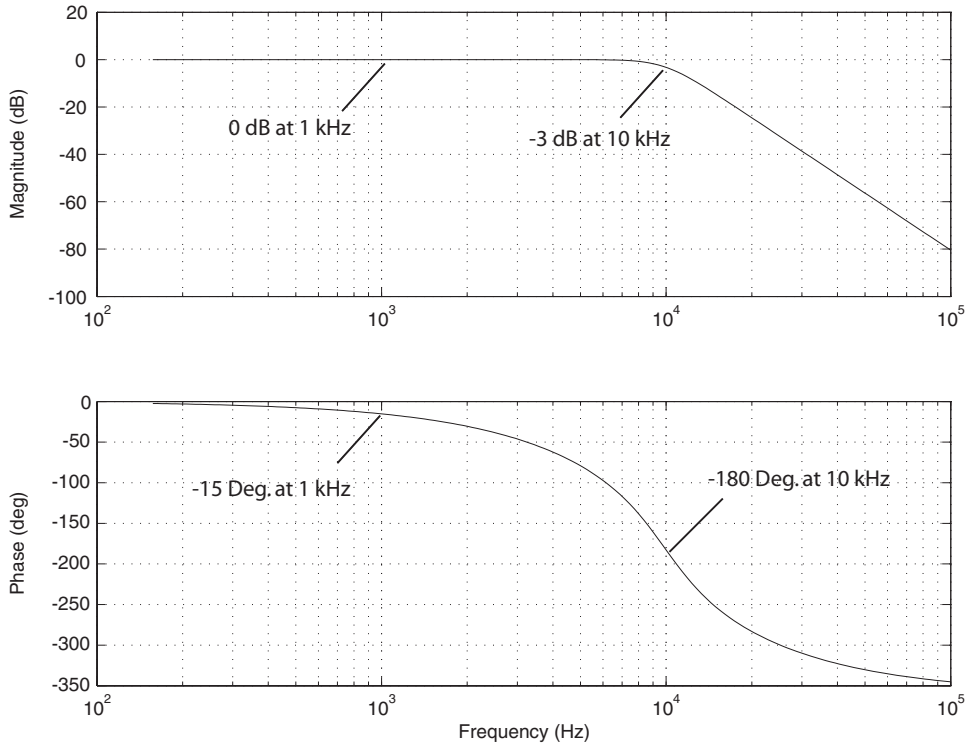

Figure 1: Frequency response of a 4-pole Butterworth low-pass filter with a bandwidth of 10 kHz.

As in previous labs, these files contain system initialization. Call `init_eecs461(5)` from your `main()` function to configure the processor correctly. For this lab the system clock will run at a frequency of 40 MHz.

qadc.h and qadc.c

The *qadc.h* file was given to you in Lab 3 and you completed the *qadc.c* file. The functions defined in *qadc.c* initialize and read analog-to-digital conversions.

mios.h and mios.c

The *mios.h* file was given to you in lab 4 and you completed the *mios.c* file. The functions defined in *mios.c* initialize the eMIOS PWM function and set the frequency and duty cycle of the output. Initialize PWM output on eMIOS channel 1, **not** channel 0, and otherwise use the same initialization process you used in Lab 4.

isr.h and isr.c

The *isr.c* code configures the decrementer ISR and enables the interrupt so the processor will respond properly. For this lab, the function `init_interrupts()` in *isr.c* will initialize the DEC frequency and allow you to specify your own ISR to handle DEC IRQs. The memory location of the DEC_ISR is placed in Interrupt Vector 10, the slot assigned to DEC interrupts. The DEC_ISR within *isr.c* in turn calls your ISR.

To use the interrupts, place *isr.h* in the `include/` directory, and place *isr.c* in the `lib/` directory. Then create a C function that will serve as the ISR. For this ISR to be called in response to the DEC IRQ, call:

```
void init_interrupts(void (*fctn_ptr)(), int freq);
```

Use the name of your ISR (without the parentheses) as the `fctn_ptr` argument and provide the frequency for the DEC as an integer in Hz.

The following sample code shows how to set up a function `my_isr()` as a DEC ISR. Note that once the call `init_interrupts()` is made, the main function enters an infinite loop that is periodically interrupted by calls to `my_isr()`.

```
#include <mpc5553.h>
#include <isr.h>

int my_dec_cntr = 0;
void my_isr (void)
{
   my_dec_cntr++;
}

void main()
{
   unsigned int loopcnt;           /* loop count for debug */
   init_interrupts(my_isr, 1000);  /* connect my_isr() to the DEC IRQ
                                    * and set the DEC freq. to 1000 Hz */

   while (1) {
      loopcnt++;                   /* do nothing between IRQs*/
   }
}
```

lab5.c

The file *lab5_template.c* has been provided as a starting point for your *lab5.c*. It is similar to the sample code provided above.

Makefile

A makefile has been provided to you. Place this file in your `lab5/` directory for proper operation.

3 Pre-Lab Assignment

*Pre-lab questions must be done **individually** and handed in at the start of your lab section. You must also, **with your partner**, design an initial version of your software before you come to your lab section. Your software does not need to be completely debugged, but obviously more debugging before coming to lab means less time you will spend in the lab.*

1. [**7**] Carefully study the PWM generation section of Lecture 6, particularly the section on PWM frequency response. **Write implementations of the ISRs A, B and C described below and insert them into the existing *lab5_template.c*.**

IsrA
ISR frequency: 20 kHz
Sine wave: 1 kHz, 1 to 4 volts, external input
PWM: Varies (see below)
Procedure:

(a) Turn on LED 0.
(b) Read AN0 analog input
(c) Calculate duty cycle
(d) Set the PWM duty cycle
(e) Turn off LED 0.

This ISR samples a 1 kHz sine wave produced by a function generator and regulates the duty cycle of a PWM signal to correspond to the measured voltage of the sine wave. Further, it will illustrate the significance of the PWM duty cycle and frequency range. You should set your ISR up such that when dipswitch 122 is high, the PWM frequency is 60kHZ and when it is low, the PWM frequency is 20kHZ. Similarly, dipswitch 123 should control the duty cycle range. When the dipswitch is high, the PWM duty cycle should be between 40% and 60% and when the dipswitch is low, the duty cycle should be between 10% and 90%. The duty cycle should be made proportional to voltage such that for 1 volt, the duty cycle is 10% and for 4 volts the duty cycle is 90%. You may need to change the software limits imposed on the duty cycle by `set_PWMDutyCycle()` in *mios.h* to achieve duty-cycles beyond 35%-65%. Also, include a line of code to turn on a GPIO LED at the start of the routine, and turn that bit off at the end of the routine. By monitoring that LED on an oscilloscope, you will be able to measure the frequency and run-time of your function by measuring the frequency and length of the pulse. Make sure that your function does not take longer than 50 microseconds.

IsrB
ISR frequency: 1 kHz
Sine wave: 100 Hz, numerically calculated by `sin()` function
PWM: 60 kHz, 40% to 60% duty cycle
Procedure:

(a) Turn on LED 0.
(b) Calculate `sin`(2*pi * i / 10), where i is incremented by 1 each invocation.
(c) Set the PWM duty cycle
(d) Turn off LED 0.

Instead of sampling an externally generated sine wave, this ISR creates the sample points for a 100 Hz sine wave by calling the `sin()` function. Because the ISR is running ten times faster than the sine wave that it is constructing, there will be 10 calls to your ISR for one period of the sine wave. In other words, each time the `sin()` function is calculated, the argument to the function should be incremented by 2*pi / 10. As before, the duty cycle should be set proportional to the sine wave, which is now between 1 and -1, so for a value of -1, the duty cycle should be 40% and for a value of 1 the duty cycle should be 60%. Initially, the DEC frequency should be set to 1 kHz, but you will increase the ISR frequency so your function should be written such that the frequency of the sine wave also increases so as to maintain the 10:1 frequency ratio.

The following function demonstrates how to calculate the duty cycle from the `sin()` function:

```
compute_sin_isr() {
    static double theta;
    double Pi = 3.14159;
    double Ts = 0.001       /* DEC frequency is set to 1 kHz sample rate */
    double delta, dutyCycle;

    f = 100;                /* Frequency is set 100 Hz */
    delta = 2*Pi*f*Ts;      /* delta is the "added" bit for this timestep */
```

```
    theta += delta;

    theta = (theta < 2*Pi) ? theta : theta-(2*Pi); /* Get the
              * remainder after dividing by 2*Pi (otherwise, over time,
              * a very large theta would cause errors) */

    dutyCycle = 0.5 + 0.1*sin (theta);
    /* output this duty cycle to the PWM - ranges from 0.4 to 0.6 */
}
```

IsrC
ISR frequency: 1 kHz
Sine wave: 100 Hz, *pre-computed* by `sin()` function
PWM: 60 kHz, 40% to 60% duty cycle
Procedure:

(a) Turn on LED 0.
(b) Read global look-up table to determine the PWM duty cycle based on a sine wave.
(c) Set the PWM duty cycle
(d) Turn off LED 0.

This ISR is nearly the same as IsrB except you will stream-line the code to produce a sine wave with the maximum possible frequency. To generate a sine wave at the highest possible frequency requires the shortest possible run-time for the ISR, but calculating `sin()` is computationally intensive. Instead of calling `sin()` every time your ISR runs, your program will pre-compute the values it needs once, store them in a global array and the ISR will look-up the appropriate value from the array. Not much memory is required because the sequence of duty cycle values for the PWM should repeat after 10 calls to your ISR. Be creative and try to find a solution that minimizes the run-time of the ISR. You will begin by setting the DEC frequency to 1 kHz and then you will increase the DEC frequency. As you increase the frequency of your ISR, the frequency of the sine wave should increase to maintain the 10:1 frequency ratio such that there are always 10 sample points per sine wave.

2. **[2] Use the frequency response (both magnitude and phase) in Fig. 1 to plot or sketch by-hand the steady-state time response of the filter to an input sine wave at 1 kHz. Indicate both the input sine wave and the output sine wave and specify how the amplitude and phase change.**

3. **[3]** Make another time response of the filter but this time for 10 kHz. **Indicate both the input and output in your plot and specify how the amplitude and phase change.**

4. **[2]** Suppose you have a low-pass filter with pass-band gain of 1, and to its input, you apply a 0-5 volt PWM with a 40% duty cycle and a frequency much greater than the cut-off frequency of the filter.**What is the output of the filter?**

5. **[1]** Complete *lab5.c*.

4 In-Lab Assignment

1. Download the lab files from the course web page, and place .c files and .h files in their associated directories as noted previously.
2. Look at the main function within *lab5.c*. Edit the call to `init_interrupts()` so it uses IsrA. Make sure that the DEC frequency and the PWM frequencies are set correctly for IsrA.
3. Compile your lab5 code using the provided makefile.
4. Connect the signal generator to the input AN0 and connect the output MIOS 1 to the oscilloscope, using the test loop located on the right edge of the interface board. Using a t-splitter at the function generator, make an additional connection from the function generator to the oscilloscope so you can display the input sine wave and output PWM simultaneously.
5. Produce a 1 kHz sine wave with 1.5 V amplitude and 2.5 V offset using a function generator. Make sure the function generator is in "High-Z mode" so that the voltage, as measured on the scope, is between 1 and 4 V.
6. Using the oscilloscope, measure the frequency of IsrA by measuring the frequency of the LED 0. To access the GPIO output, a probe can be connected to the test loop to the right of LED bank.
7. Change dipswitches 122 and 123 and make sure that the PWM output matches what is specified in the IsrA specification.
8. Generate inputs and outputs similar to those shown in Fig. 2.
9. Disconnect the output MIOS 1 from the oscilloscope and instead connect it to the input of a low-pass filter. Connect the output of the filter to the scope and the signal you see should resemble the sine wave input. Make a sketch of what you see on the scope when the PWM frequency is set to 60kHz and 20kHz for duty cycle ranges 40% to 60% and 10% to 90%. The last page of this document can be used for these sketches. Please ask your GSI if you need help connecting the low-pass filter.
10. Modify *lab5.c* so that the DEC interrupts are handled by IsrB. Make sure that the DEC frequency and the PWM frequency are set correctly for IsrB.
11. Again, use a low-pass filter to generate an approximate sine wave out of the PWM signal.
12. Make another sketch of the filtered output.
13. Using the oscilloscope, measure the run-time of IsrB by measuring the time that LED 0 is set. What is the highest possible frequency of IsrB based on this measurement?
14. Find the highest frequency sine wave that can be generated by IsrB. Remember that the sine wave should always be 1/10 the frequency of the ISR, so modify the DEC frequency and be sure your ISR outputs 10 sample points per sine wave.
15. Display 5-6 periods of the LED on the oscilloscope. You should notice that not all executions of the ISR take the same amount of time.
16. Try using `floats` instead of `doubles` in IsrB and repeat Steps 11, 12, 13 and 14. Remember to convert `sin()` to `sinf()`. Is the output different?
17. Modify *lab5.c* so that the DEC interrupts are handled by IsrC.
18. Repeat Steps 11, 12, 13 and 14 for IsrC.

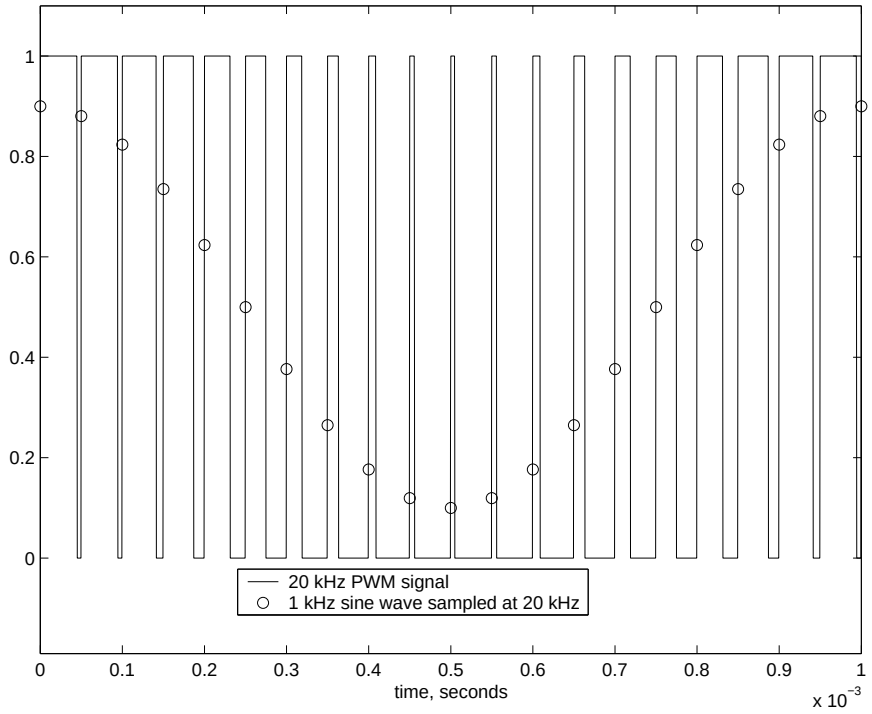

Figure 2: The value of sinusoidal input determines the duty cycle of the PWM signal at the beginning of each period. (A period begins at the rising edge of the PMW signal.) The peak value of the input is 0.9 and the resulting duty cycle of the PWM signal is 90%. Likewise, the lowest value of the input is 0.1 and the resulting duty cycle is 10%.

5 Post-Lab Assignment

1. **[3] Based on what you saw in step 9 why do we use a PWM freqeuncy of 60kHz instead of 20kHz and why you may see a 10kHz interrupt frequency instead of the desired 20kHz frequency?**

2. **[2] Explain any changes you observed when you changed IsrB to use `floats` instead of `doubles`. Under what circumstances would you want to use a `double` instead of a `float`, or vice versa, in an embedded systems application?**

3. **[2] Explain your observations in In-Lab Step 15. Why would different executions of the `sin()` function take different amounts of time to execute? What problems would this cause, and how would you overcome these problems?**

4. **[2] Explain the different range in maximum frequencies you observed between the three measurements you took. (IsrB with `doubles`, IsrB with `floats` and IsrC)**

5. **[1] Suggest ways to make IsrC execute faster.**

6. **[25 in] Include a well-commented copy of your *lab5.c* file.**

If you have comments that you feel would help improve the clarity or direction of this assignment, please include them following your postlab solutions.

[15 pre, 25 in, 10 post = 50 total]

9. 122=0 123=0

9. 122=0 123=1

9. 122=1 123=0

9. 122=1 123=1

12. ISRB

12. ISRC

13,14,16	Runtime	f_{max}
IsrB double		
Isrb float		
IsrC		