ME 333: Introduction to Mechatronics

Assignment 4: Interrupts, Serial Communication, and PIC Programming

Electronic submission due **before** 11:00 AM on Thursday February 9th

All questions asked in this problem set must be typed and submitted via Blackboard. See the ME333 homepage for more details. An in-class demo of the programming problem will be given on Thursday, February 9.

1 Chapter Review Questions

The following true/false questions are based on the reading in chapters 3 and 6. If the answer is false, make sure to provide the correct statement. We explore topics in Chapter 5 in a later section.

1. The input of the assembler is a ".s" file. True
2. The linker script is an output of the linker. False; the linker script is an *input* to the linker
3. If you accidentally return from your main function, the PIC will not power off but remain in an infinite loop. True
4. `crt0.S` is the C run-time clean up code False; it is the run-time *startup* code.
5. `elf32pic32mx.x` is the linker script. True
6. libmchp_peripheral_32MX795F512L.a is the assembly code for the PIC32 peripheral library. False; it is the object code
7. Interrupts are generated by the processor core, peripherals, and external inputs. True
8. If an ISR's priority is not greater than the CPU's priority at the time that the interrupt occurs, then the interrupt is dropped and never gets processed. False; the interrupt remains pending until the CPU's priority drops
9. The names of the interrupt vectors and interrupt requests are in `ppic32mx.h`. True
10. When the PIC32 is in "single vector mode", it can only service one interrupt of your choosing and when "multi-vector mode" is enabled, the PIC32 can service all interrupts. False; when "single vector mode" is enabled, all interrupts are serviced by the same ISR
11. There are 64 interrupt sources and 16 interrupt vectors. False; there are 96 different interrupt request sources, and up to 64 different interrupt vectors can be supported

2 Serial Communication with the PIC32

Serial communication will be a big part of your PIC32 programs throughout the term. In order to make the process easier, we have provided `NU32.c` and `NU32.h` to help you get quickly up and running with sending information between your PIC and computer. In order to get you acquainted with the functions in `NU32.c`, please read about them on the wiki page, *NU32: Serial Communication with the PC*. The following questions refer to the new versions of `NU32.c` and `NU32.h`, which have been posted to the wiki as of February 1st, 2012.

1. The functions `NU32_Startup()` and `NU32_Initialize()` should be called early on in your main function. Looking at the source file, what do these functions do? For now, ignore any references to `SYSTEMConfig(...)`.

`NU32_Startup()` enables multi-vector interrupt mode. `NU32_Initialize()` sets up the LED's to be digital outputs, and sets their initial state. It then configures UART1 for general serial communication, and UART4 for bootloading serial communication. Interrupts on UART4 are enabled, and the ISR for UART4 is set to reset the PIC if the letter "B" is received.

2. What is the default baud rate (bits per second) for serial communication on the PIC32 and how are `NU32LED1`, `NU32LED2`, and `NU32USER` defined?
   Baudrate = 115200 (bits/ second); `#define` preprocessor commands are used to define the LEDs and the USER button. The code looks like the following:

```
#define NU32LED1 LATAbits.LATA4
#define NU32LED2 LATAbits.LATA5
#define NU32USER PORTCbits.RC13
```

3. Modify Invest.c to run on the PIC32. The *NU32: Serial Communication with the PC* wiki page is a useful resource. Here are a few hints to get you started: Remove `<stdio.h>` and replace it with the typical PIC32-related header files, replace instances of scanf and printf with `NU32_ReadUART1(...)` and `NU32_WriteUART1(...)`, and if `\n` gives you undesirable output formatting, try `\r\n`. Submit your modified Invest.c program and a screenshot of sample output for an initial amount of $1000.00 at a rate of 5% over 5 years using a terminal emulator that is not the one provided by the NU32_utility Processing app (i.e., use an external program, like Putty, miniterm, or ZTerm. This assignment will not cover how to use these programs. If you have trouble using these programs, you can get help from your peers, office hours, or the forum.). The formatted output should match what you see in the Invest.c output on the *Installing a C Compiler and IDE* wiki page.
   The solution to this is linked on the wiki.

3 Timing Mathematical Operations

Successful real-time control applications require intimate knowledge of how long a task takes to complete. The main purpose of this section is to build your understanding of how many cycles the PIC processor takes to perform mathematical operations. Run `MathTiming.c` on your PIC and view its output in a terminal. This program times various mathematical operations and returns the average number of cycles and total number of cycles over 20 repetitions and then exits. Use the output to answer the questions below. To rerun the program, press the reset button on your PIC.

Questions

1. How long does it take to add, subtract, multiply, and divide each of the C variable types? In order to answer this question, run `MathTiming.c` and create a table similar to Table 1.[1] Each entry in the table will comprise of two numbers. The first number is the average number of cycles 20 copies of the same arithmetic operation took to complete for a particular data type rounded to the nearest integer. After you've filled in the table, add a second number, in parenthesis, to each cell. The second number is the number of cycles the operation took to complete normalized to the entry with the least number of cycles in the table. For example, in Table 1, 20 `int` additions took 106 cycles to complete (round(106/20) = 5) and, assuming that it is also the shortest cycle in the table, we normalize it to 1 (5/shortest cycle) in the parenthesis.
   The solutions are seen in Table 1
2. How many cycles does it take to compute `sinf()`, `sin()`, `sqrtf()`, and `sqrt()`? Follow the layout in Table 2. Again, write two numbers here for your results. The first is the average number of cycles over

[1] We've omitted unsigned types from the table because they are equivalent in cycle times to their signed counterparts. Also, the long data type is not present, because a long and int are the same number of bytes on the PIC32. This is generally not true on most other platforms, like your PC.

	char	short	int	long long	float	double
add (+)	6 (1)	6 (1)	5 (1)	11 (2)	66 (13)	66 (13)
sub (-)	6 (1)	6 (1)	5 (1)	11 (2)	81 (16)	81 (16)
mul (*)	8 (1)	8 (1)	7 (1)	25 (5)	56 (11)	56 (11)
div (/)	17 (3)	17 (3)	17 (3)	134 (27)	151 (27)	151 (27)

Table 1: Average time to compute arithmetic operations for various data types

sinf	161 (1)
sqrtf	309 (2)
sin	161 (1)
sqrt	309 (2)

Table 2: Average time to compute sin and sqrt function

20 repetitions and the second is a parenthetical normalized number (use the same normalization value as in 3.1). Note that `sinf()` only takes a `float` as input and `sin()` only takes a `double` as input. The solutions are seen in Table 2. Note that the time it takes to calculate the trig functions varies significantly depending on what argument they are passed. For example sin(0) computes much more quickly than sin(0.123). This implies that Microchip is using some look-up tables to quickly obtain the answers for common trig math.

3. How many cycles does it take to left bit shift `int`s and `long long`s? Produce a similar table for bit shifting as in Table 3. For each cell, write two numbers for your results. The first is the average number of cycles over 20 repetitions and the second is a parenthetical normalized number (use the same normalization value as in 3.1). The bit shift operations in `MathTiming.c` are performing 1 bit shift, 10 bit shifts, and 31 bit shifts. Does bit shifting n times take n times as long as bit shifting once? If `n = 32`, what value will an `int` variable have, regardless of its initial value? (Hint: think about what value is shifted in from the left into bit b_0, as the bits are shifted each time.)
Bit shifting takes the same amount of time regardless of how many bits we shift. A shift of 32 bits on an `int` will always result in all zeros in the 32 bits reserved for the `int`.

	int	long long
<< 1	5 (1)	26 (5)
<< 10	5 (1)	26 (5)
<< 31	5 (1)	26 (5)

Table 3: Average time to compute bit shifting operations for `int` and `long long`

4 Programming Challenge

This programming challenge is to code a full-functioning stopwatch that runs on the NU32 using change notification (CN) and core timer interrupts. A demo of this will be given in class on Thursday, February 9. A template has been provided to assist. To help you get started with finishing the problem, we are going to briefly discuss the functionality of the change notification module.

4.1 Change Notification Summary

All of the details on the CN module can be found in the *PIC32MX Family Reference Manual* in *Section 12. I/O Ports*. Especially useful information is found in parts 12.2.6, 12.3.9, and 12.4, as well as the register tables and examples section. The CN module provides a convenient way to generate an interrupt when the state of any number of selected pins changes. As an example, imagine a scenario with a lot of buttons

attached to the PIC where different combinations of button presses would accomplish different things. With the CN module, an ISR could be triggered when any button was pressed, and then logic inside the ISR could determine which buttons were pressed, and what to do about it.

There are a number of I/O pins that can also act as CN pins[2]. To use them in this capacity, they must be configured as digital inputs, and their corresponding bits in the `CNEN` register must be set. The CN module itself must also be enabled by setting bit 15 in the `CNCON` register. Finally, interrupts for the CN module must be turned on using the `CNIE` bit. Note that there are also optional pull-up resistors attached to each CN pin that can be enabled. This is useful when interfacing with external components because the internal pull-ups allow the use of less complex external circuitry.

If enabled, on every SYSCLK cycle, the CN module compares the previous value of a PORT with the current value of the PORT and signals a "mismatch value" if any configured CN pin changed between the two readings. If the CN interrupt is enabled, the mismatch value triggers an interrupt. This illustrates two idiosyncrasies with the CN module. The first is that when the ISR is triggered, the user must perform their own logic to determine which pin changed state, and whether there was a falling or a rising edge. The second is that during the ISR the correct PORTs must be read to clear their mismatch values. For example RD4 is also CN13. If CN13 is the only CN pin enabled, then PORTD must be read during the ISR (as well as clearing the ISR flag) to properly reset the CN module.

4.2 Primer Questions

Answer these questions by reading the reference manual.

1. True or false, you should disable CPU interrupts before modifying the CN control registers?

   True. It says this, for example, in section 12.3 of the *PIC32MX Family Reference Manual.*

2. Let's say only CN1 (RC13)[3] is enabled, write the code snippet that would be in the ISR that determines if CN1 triggered on a falling edge i.e., use an `if` statement that tests true if there was a falling edge and false if there was a rising edge on CN1.

   ```
   if (!PORTCbits.RC13)
   {
     // If RC13 is low, then we have found a falling edge
   }
   ```

3. We are trying to declare a function to be an ISR for the CN module with a priority of six. Replace the question marks in the following code snippet with the correct syntax for a CN ISR:

   ```
   void __ISR(_CHANGE_NOTICE_VECTOR, IPL6) cn_isr(void) {
   ```

4. Looking at the provided template, you will see a function called `display_time`. This function takes a double representing elapsed time in seconds, and converts it into hours, minutes and seconds. You will notice the use of the "%" operator in this function. This is called the "modulo" operator (or "mod"), is mentioned in *Crash Course in C*. What is the value of 10%4 and 97%60?

   $$10\%4 = 2$$
   $$97\%60 = 37$$

[2]Up to 22 input pins may be selected to generate an interrupt.
[3]This correspondence was determined by looking at the *Chapter 2 - Looking Under the Hood: Hardware* handout.

4.3 Assignment Details

As previously mentioned, you are going to write a program that implements a stopwatch on the NU32. You will use the core timer to perform the timing, and a core timer interrupt ISR to prevent the core timer from overflowing. There will be two buttons on the stopwatch, a "start/stop" button (that will be hooked up externally), and the USER button that will act as a "reset" button[4]. The change notification module with an ISR will be used to determine when and which button was pressed.

The basic logic is illustrated with the following sequence of button presses:

1. "start/stop" button is pressed
    - feedback provided to the user that timing has started
    - timing begins
2. after 10 seconds the "start/stop" button is pressed again
    - timing stops
    - feedback that the timing has been suspended is provided
    - feedback of 10 seconds elapsed is provided
3. after 5 seconds the "start/stop" button is pressed again
    - feedback is provided stating that timing has resumed
    - timing resumes
4. after 1 second the "start/stop" button is pressed again
    - timing stops
    - feedback that the timing has been suspended is provided
    - feedback of 11 seconds elapsed is provided
5. after 30 seconds the "reset" button is pressed
    - timing stops
    - feedback that timing has been reset to zero is provided
    - total time counts are reset such that repeating the above cycle would produce the same results

To get you started a template has been provided[5]. In the template there are comments about where individual functions should go, and what they should accomplish. Helper functions for calculating the total time, displaying the time in a nice format, handling time calculations introduced by rollovers, and more is already provided. Several functions are completely missing, and several functions have blanks that must be filled in by the student. For full credit, please follow all directions that are included in comments in the template.

[4] If you believe your CN ISR is correct, and you notice that sometimes pressing the button once triggers several interrupts (especially with the USER button), it is likely you are experiencing button "bouncing". This is high frequency electrical noise caused by mechanical vibrations in the button. The elasticity of the components in the switch causes vibrations that quickly open and close the switch for a few milliseconds after it is toggled. There are many ways to accomplish "debouncing" such as passive filtering with electronics, or intelligent software. In this application, it doesn't matter much, but in a more complex project steps should be taken to avoid this

[5] The template utlizes functions from the newest versions of NU32.h and NU32.c. Thus you will need to create a project that has the correct NU32.c and NU32.h included, and add procdefs.ld to the directory

5 What to turn in

An in-class demo of the stopwatch problem will be given on Thursday, February 9. For the questions, you must submit typed responses. Place your typed responses (the TAs prefer a PDF file) in a **zip** file and submit the zip file through Blackboard before class on the date the assignment is due. The name of the zip file you submit will be of the form lastname_firstname_a4.zip. For this assignment we expect only 5 files, your answers to all short questions, your modified Invest.c, and the three modified template files.