ME 333: Introduction to Mechatronics

Assignment 4: Towards real-time control with the PIC32

Electronic submission due **before** 12:30 p.m. on February 8th

All questions asked in this problem set must be typed and submitted via Blackboard.

1 Code efficiency

For questions in this section (unless otherwise noted), you are to run your code with **optimization level set to zero**. Do not submit any code related to this section.

Successful real-time control applications require intimate knowledge of how long a task takes to complete. The main purpose of this homework assignment is to build your understanding of how many cycles the PIC processor takes to perform mathematical operations. As you answer the questions in this section, keep these points in mind:

1. One clock cycle (or simply cycle) is 1/`SYSFREQ`, so we're assuming that one instruction takes 12.5ns on our 80MHz PIC32.
2. You can assume that one assembly-level instruction takes one cycle to complete. This isn't entirely true, but it's a very good approximation for most instructions you are asked to look at.
3. The PIC has no hardware to compute floating-point operations. This piece of hardware is known as a floating-point unit or FPU. The calculations are instead performed in software.

1.1 Speed

Questions

(a) How long does it take to add, subtract, multiply, and divide each of the C variable types? In order to answer this question, create a table similar to Table 1.[1] Each entry in the table

	`char`	`short`	`int`	`long long`	`float`	`double`
add (+)	5 (1)					
sub (-)						
mul (*)						
div (/)						

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

[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.

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 (see Algorithm 1 in Section 1.4) 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 (i.e., the first number in the cell) normalized to the entry with the least number of cycles in the table. For example, in Table 1, 20 `char` additions took 101 cycles to complete (round(101/20) = 5) and, assuming that it is also the shortest cycle in the table, we normalize it to 1 (5/shortest cycle) in the parenthesis.

(b) How many cycles does adding two `int`s and subtracting two `long long`s take by looking at the disassembly listing and counting the number of assembly instructions generated from the two expressions? For integral addition, subtraction, and multiplication, we should get a pretty good estimate of the number of cycles by looking at the assembly code.

(c) How many cycles does it take to compute `sinf()`, `sin()`, `sqrtf()`, and `sqrt()`? Again, write two numbers here 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 (a)). Note that `sinf()` only takes a `float` as input and `sin()` only takes a `double` as input (same for `sqrtf()` and `sqrt()`), make sure to call each function with the correct data type. Again, the ratio should be relative to the fastest cycle time from your table in part (a).

(d) How many cycles does it take to left bit shift `int`s and `long long`s? Produce a similar table for bit shifting `int`s and `long long`s as in the example table above, repeating each operation 20 times as in part (a). The operations in your columns for this table 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? Verify your results by looking at the generated assembly code. Remember the syntax for bit shifting is `x << n`, where `x` represents the bits to shift and `n` are the number of slots to shift each bit over to the left. They must both be integral types. 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.)

1.2 Compiler optimizations

If certain operations aren't fast enough, we have additional tools at our disposal. We could use the compiler's optimization settings to optimize our code (which can make code run faster and consume less program memory). If an optimization level > 0 is set, then the compiler will rewrite or eliminate inefficient code without changing our program's behavior. We will illustrate this functionality using optimization level 1 on Algorithm 1. You should be aware that Microchip disables optimization levels >= 2 after a 60 day trial period has elapsed.

(a) Using the code in Algorithm 1, how long does each operation and data type take with optimization level set to 1? Does the disassembly support your results, why?

(b) Using optimization level 1, what is the return value stored in `ans` in Algorithm 1? Given what the code does, is this return value what you would expect? By convention the return value is placed in register `v0`, the line in your disassembly window that you are looking for is right above the statement `jr ra` in the `timeCode()` section of the disassembly listing.

1.3 Memory vs. speed

Another common practice in speeding up computation is to use a look-up table. An important consideration in creating a look-up table is how much space in memory the table will occupy. This often leads to a trade-off between the accuracy of your stored values, the speed at which the PIC can access the values from memory, and the number of bytes each value requires in memory. For example, the `sin()` function is a good candidate to create look-up values for, but how should we create the table? We could create an array of doubles for our look-up table and write a function, `double mySinFcn(double x)`, that maps the interval 0 to 2π to the indices of the array. With a little cleverness on our part, we can save a lot of memory using a fixed-point representation for the input, `x`, and use an array of integral types instead. A fixed-point representation treats an integral type as if it were a real number, you decide after what digit the decimal point is located. For example, double d = 6.284 can be represented as int i = 6284. Any arithmetic operations are equivalent —we just have to remember that conceptually there is a decimal point after the most significant digit in the variable i.

Questions

(a) Define global arrays of 100 elements for the following C data types: `char`, `short`, `int`, `long long`, `float`, and `double`. How much space in bytes (and in base-10) does each array take and what are the memory addresses (in base-16) of the first element and last element of each array? Note that the first element is always located at the start address of each array. You can find the size of each array underneath the "Allocating common symbols" section of your map file. The .bss section[2] of the same map file under the category "COMMON" is where you'll find the start addresses of your arrays in memory. Because the memory addresses reserved for an array are sequential, you can calculate the address of the last element in the array using the start address, number of total bytes of the array, and the size of each element in the array.

(b) Don't forget that pointers also take up memory when declared. On the PIC32, a pointer can access up to 4GB of virtual memory. How much space does an array of 100 pointers to `double`s and 100 pointers to `int`s take up (e.g., `double *dbl_ptr_array[100]`)? Why does the `double *` array take up less memory than an array of 100 `double`s from part (a)? Think about what a pointer represents and why this purpose dictates how many bytes in memory it should be.

(c) Create a look-up table, call it `LookUpTable`, with indices 0 to 6,284 (where 6,284 is our fixed-point approximation of 2π) that can hold values from -10,000 to 10,000. For the elements in your array, you must choose the smallest data type in bytes that can store these values. Initialize the array using the built-in `sin()` function from <math.h> and correctly map 0–6.284 radians to indices 0–6,284 and the output of `sin()`, -1–1, to -10,000–10,000. Answer the following questions:

- How much RAM does your array use?
- What is your initialization code? It should be a loop that is about 3–5 lines of C code total.
- If we are trying to map 0–6.284 radians to discrete index values 0–6,284, what is the resolution of your inputs into the array (i.e., the indices) in radians?

[2]If you remember from Assignment 3, the .text section is where program code is placed. The .bss section is where uninitialized (at least by you) global variables are stored and the C programming language guarantees that their value will be zero when your program starts.

- Determine the time it takes to look up a value in the table. Do this by replacing `ans = a + b` in Algorithm 1 with `ans = LookUpTable[a]` 20 times and making any other necessary modifications, like making `ans` the same data type as the elements in your table. Make sure that the **optimization level is zero**. How much faster is a look-up table over calling `sin()`? Write your answer as a ratio between the average number of cycles to compute `sin()` in part 1.1 (c) over the average number of cycles to look up a value in `LookUpTable`.

1.4 How to time your code

The following code snippet demonstrates how to time each cell in Table 1. Note that `buffer` is a `char` array that you define somewhere in your C program. It needs to be large enough to hold the longest string you plan on sending to your PC (e.g., 1,024 bytes). On the other hand, WriteString(...) is defined for you in NU32v2.c. Don't forget to add NU32v2.h and NU32v2.c to your MPLAB project (see NU32v2:_Serial_Communication_with_the_PC).

Algorithm 1 Example of timing your code

```
int timeCode() {
  int ans, a, b;
  unsigned int e;
  a = 17;
  b = 12;
  OpenTimer2(T2_ON |T2_SOURCE_INT |T2_PS_1_1, 0xFFFF);
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  ans = a + b;
  e = ReadTimer2();
  sprintf(buffer, "int (+): %f (total: %d)\r\n", e/20.0, e);
  WriteString(UART3, buffer);
  return ans;
}
```

2 PWM and counter/timers

In this program you will use `OC1` and `TMR4`'s interrupt service routine to generate the first two terms of a Fourier Sine Series.

2.1 Baby steps

This section is optional. Before getting to the actual programming assignment, make sure you can write the programs below.

(a) Write a program with an interrupt service routine that triggers 10,000 times a second using Timer 4. Each time you enter the routine, you add 1 to a global counter. When the counter reaches a constant `SWITCH_TIME`, say 5000, the LEDs are toggled and the counter is set back to zero.

(b) Copy the wiki code for the 20 kHz PWM and verify using your oscilloscope (in analog mode) that you see the digital PWM output. Change the duty cycle (in range 0–999) and see the effect on the pulse train. Now attach the low-pass filter to the `OC1` output pin. Use your oscilloscope to view the voltage across the capacitor. What is the cut-off frequency of your RC low-pass filter?

2.2 Programming assignment

In this assignment, you will be graded on correctness, coding efficiency, program readability, and program maintainability. When we test your code for correctness, we can change your input values to make sure your program behaves as expected. There is an accompanying MATLAB file that you can use as a guide for your program found in the Assignment4 zip file. You can port (convert code from one language to another) various aspects of the MATLAB code into your C program.

The inputs: `BASEFREQ` (Hz), `BASEAMP` (volts), `HARMONIC` (integer value > 1), `HARMAMP` (volts), `PHASE` (degrees), `MAXSAMPS` (maximum number of samples of a sine wave to compute), and `ISRFREQ` (Hz)

The output: A sine wave, centered around 1.65V, approximating:

$$\texttt{BASEAMP}*\sin(2\pi*\texttt{BASEFREQ}*t)+\texttt{HARMAMP}*\sin(2\pi*\texttt{HARMONIC}*\texttt{BASEFREQ}*t+\texttt{PHASE}*\pi/180)$$

The details:

Write a program that uses a timed interrupt and PWM to "play" a periodic output voltage waveform. In particular, this waveform will consist of the sum of two sinusoids, one with frequency `BASEFREQ` and the other at frequency `HARMONIC*BASEFREQ`, where `HARMONIC` is an integer > 1. Based on five inputs defined in your program and the frequency of your interrupt service routine, you will calculate the elements of an array,

```
int dutyvec[MAXSAMPLES],
```

which holds the signal. Your program should:

1. Create a look-up table, `dutyvec`, given `BASEFREQ`, `BASEAMP`, `HARMONIC`, `HARMAMP`, `PHASE`, `MAXSAMPS`, and `ISRFREQ` (see MATLAB code for an example).

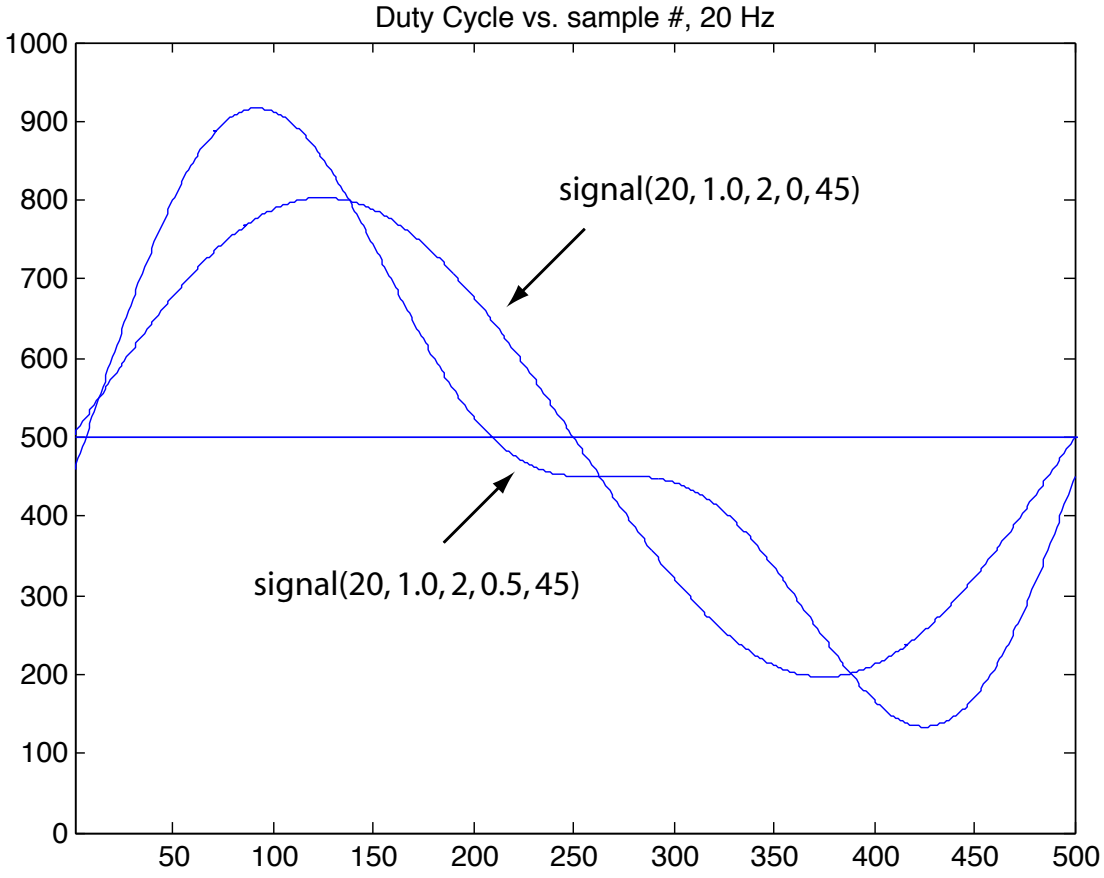

Figure 1: Output signal for Tuesday's demo

2. Set up the PWM output using `OC1` and create a timed ISR at `ISRFREQ` using TMR4.

3. In the ISR, add 1 to a global counter, set the counter back to zero when it reaches `numsamps` (the number of samples used in the `dutyvec` array), and set the output PWM duty cycle to `dutyvec[counter]`; and

4. Assume all inputs will be >= 0. You must still set reasonable upper-bounds. In your file header, the comment block you saw at the top of `invest.c`, effectively convey your reasoning behind the upper-bounds you have set and list and discuss the RC values you used in your LPF (make sure to mention the rise-time and cut-off frequency of your filter).

If you are successful, when you low-pass filter the OC output, you should see voltage waveforms similar to those in Figure 1. Try different frequencies and amplitudes. Turn in your code and demo in class, using your oscilloscope, with `BASEFREQ` = 20 Hz, `BASEAMP` = 1.0 volt, `HARMONIC` = 2, `HARMAMP` = 0.5 volt, `PHASE` = 45 degrees, `ISRFREQ` = 10 kHz, and `MAXSAMPS` = 1,000.

2.3 Using MATLAB to verify your results

The example MATLAB code, signal.m (found in the zip file), generates various waveforms based on its inputs. The output of the MATLAB code should be the same as the output of OC pin on your PIC. You can call the program from the MATLAB prompt as follows,

```
signal(20, 0.5, 2, 0.25, 45)
```

for example, which corresponds to $voltage(t) = 0.5 * \sin(2 * \pi * 20 * t) + 0.25 * \sin(2 * \pi * 2 * 20 * t + 45deg)$. This is converted into a single period of duty cycles, stored in an array. Two sample outputs are shown in Figure 1. You should note that the MATLAB code, as written with `ISRFREQ` = 10,000 and `MAXSAMPLES` = 1,000, does not allow a `BASEFREQ` of less than 10 Hz. Your C program should handle cases like these in a similar fashion.

3 What to turn in

For the questions in Section 1, you must submit typed responses. Place your responses and your .c file from the programming assignment 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 lastname_a4.zip. For example, the TA would submit his homework as Rosa_a4.zip.

In addition to submitting your assignment via Blackboard, you will also demonstrate your program to the teaching staff at the beginning of class on the date the assignment is due. For this assignment you will demo the output waveform with input parameters as specified at the end of the programming assignment section.

As always, for full credit, you must follow these instructions exactly.