ME 461 Homework #1

Building Green Board and C Programming Exercises

Check off of all Items: Due September 13th, 1:00 PM

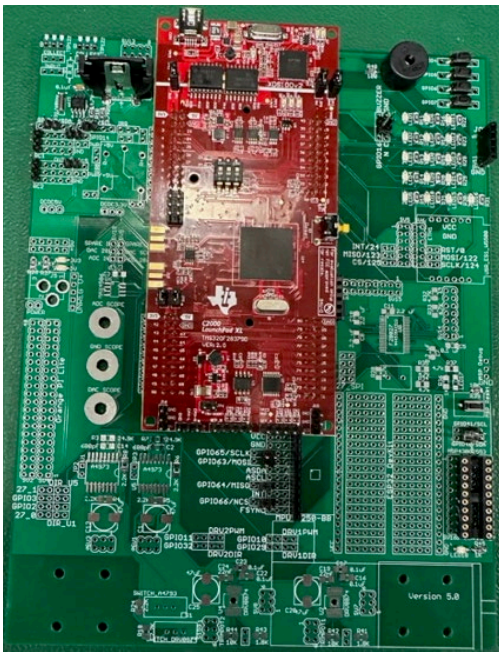
Submission at Box: Due September 13th, 11:59 PM

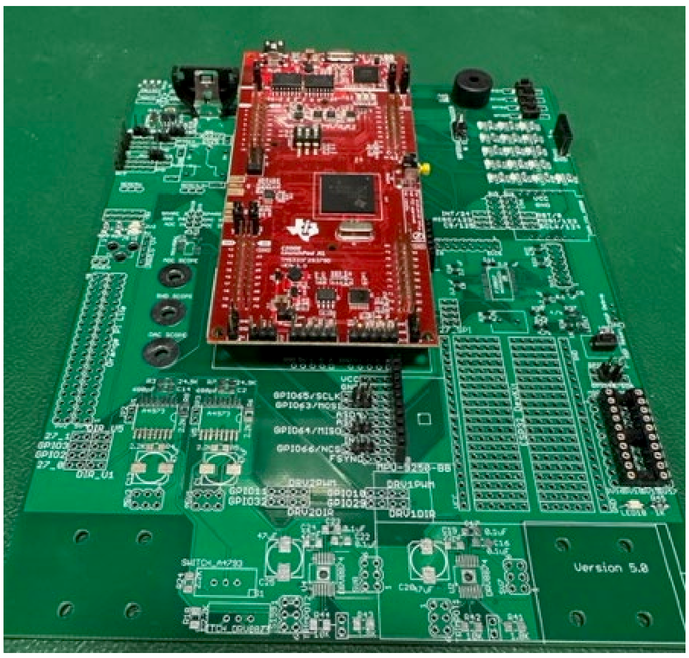
Exercise 1

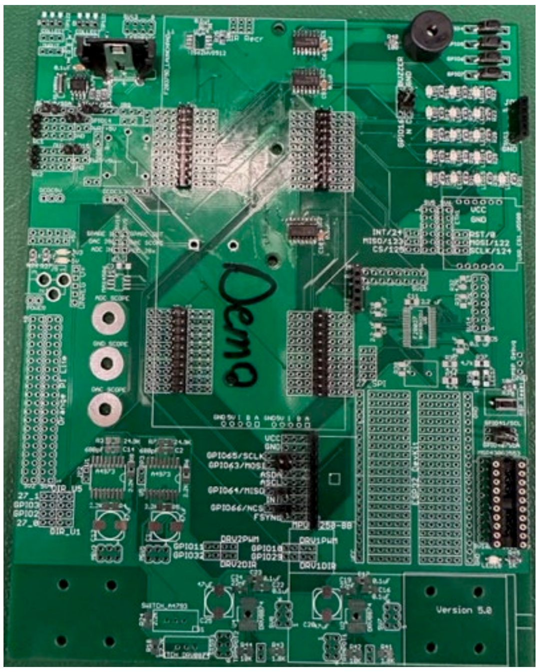
The first part of this homework coincides with Lab #1 which has you solder your breakout board. In addition to soldering your breakout board, you need to assemble much of your kit and get it tested by me. Items to Complete by Due Date for this first exercise:

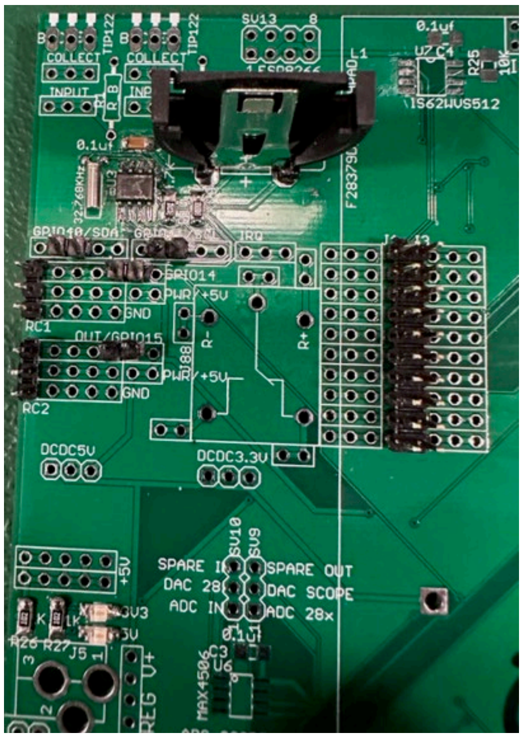
1. Make sure you have soldered all items in the pictures below or look at the "Demo" board in lab.
2. Any other items that I have not thought of as of this writing and announce later in lecture.
3. Finish soldering your breakout board and have it tested by your TA.
4. Make an appointment with me and get your entire kit tested to verify that it is working correctly.

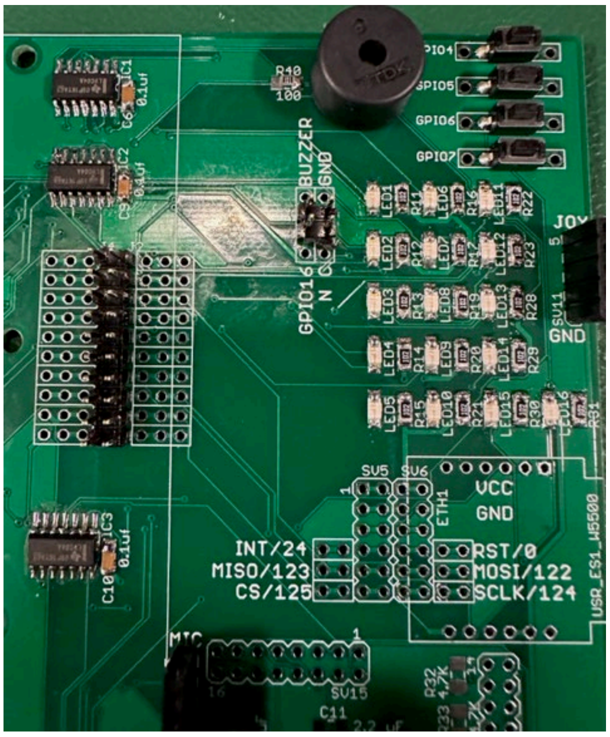
Here are some pictures of what your final soldered board should look like.











Exercise 2

For this exercise, I would like you to go through the "C Essentials Training" course at https://www.linkedin.com/learning/c-essential-training (You first need to login through the UIUC LinkedIn Learning portal with your university NetID to get past the paywall : https://web.uillinois.edu/linkedinlearning). I am not going to check if you went through every exercise, but I highly recommend it. Go through each chapter. Then for your homework submission, submit your code and any screen shots showing the code working for

1. Chapter 3's final challenge, Binary Math
2. Chapter 4's final challenge, The Grid Challenge
3. Chapter 5's final challenge, Complete the Code

Yes, I know the solution code is given to you in the course's download file. Try your best to not look at the solutions, as that will help you teach yourself C. DO NOT just submit the solution code! The areas I want you to really think about as you are going through this training are: The different types of variables in C and how to declare global and local variables. The "printf" statement and how to use it to print integer variables and floating point variables. Writing functions and how to call those functions in your C code.

There are quite a number of free C compilers that you can use to run the C code you write for these training exercises. I am thinking using one of the online C compilers may be the easiest way to go. You can google search for "online C compilers" and see which one you like or you can try this one that seems to work pretty well https://www.onlinegdb.com/online_c_compiler#. I think the biggest issue you will have with the online compilers is saving your files. So, I recommend cutting and pasting your code into any text editor and save your work with the text editor instead of saving from the online webpage. If you would like you could also download something like CodeBlocks and compile your C code on your PC or laptop.

Explanation of Given Starter Code to help you with the remaining tasks.

Below is an introduction to the starter code given to you for homeworks and labs. I used green for the code and black for my commentary. *This listing leaves out sections of the starter code in order to not make this lab write up too long.*

```
//#############################################################################
// FILE: labstarter_main.c
//
// TITLE: Lab Starter
//#############################################################################

// Included Files
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <limits.h>
#include "F28x_Project.h"
#include "driverlib.h"
#include "device.h"
#include "f28379dSerial.h"
#include "LEDPatterns.h"
#include "song.h"
#include "dsp.h"
#include "fpu32/fpu_rfft.h"
```

```
#define PI 3.1415926535897932384626433832795
#define TWOPI 6.283185307179586476925286766559
#define HALFPI 1.5707963267948966192313216916398
// The Launchpad's CPU Frequency set to 200 you should not change this value
#define LAUNCHPAD_CPU_FREQUENCY 200

// Interrupt Service Routines predefinition
__interrupt void cpu_timer0_isr(void);
__interrupt void cpu_timer1_isr(void);
__interrupt void cpu_timer2_isr(void);
__interrupt void SWI_isr(void);

void serialRXA(serial_t *s, char data);

// Count variables
uint32_t numTimer0calls = 0;
uint32_t numSWIcalls = 0;
extern uint32_t numRXA;
uint16_t UARTPrint = 0;
uint16_t LEDdisplaynum = 0;
```

For these C exercises, this is where I would like you to create any global variables or global functions. Actually, the only kind of functions we will create this semester will be global functions. We will never need to create a function inside another function and that includes not creating functions inside your main() function. You can of course put your global functions anywhere outside of other functions. It is just nice to have all your functions defined in one spot of your code so they are easier for you to find. The same goes for global variables.

```
void main(void)
{
 // PLL, WatchDog, enable Peripheral Clocks
 // This example function is found in the F2837xD_SysCtrl.c file.
 InitSysCtrl();

 InitGpio();

        // Blue LED on LaunchPad
 GPIO_SetupPinMux(31, GPIO_MUX_CPU1, 0);
 GPIO_SetupPinOptions(31, GPIO_OUTPUT, GPIO_PUSHPULL);
 GpioDataRegs.GPASET.bit.GPIO31 = 1;

        // Red LED on LaunchPad
 GPIO_SetupPinMux(34, GPIO_MUX_CPU1, 0);
 GPIO_SetupPinOptions(34, GPIO_OUTPUT, GPIO_PUSHPULL);
 GpioDataRegs.GPBSET.bit.GPIO34 = 1;

… Purposely left code out here in this listing because it is all initializations that we will discuss
in future labs and not important here …

 EALLOW; // This is needed to write to EALLOW protected registers
 PieVectTable.TIMER0_INT = &cpu_timer0_isr;
 PieVectTable.TIMER1_INT = &cpu_timer1_isr;
 PieVectTable.TIMER2_INT = &cpu_timer2_isr;

… Purposely left code out of listing …

 // Configure CPU-Timer 0, 1, and 2 to interrupt at given period in microSeconds:
 // 200MHz CPU Freq, Period (in uSeconds)
 ConfigCpuTimer(&CpuTimer0, 200, 10000); //Timer0’s interrupt will happen every 10ms or 0.01seconds
 ConfigCpuTimer(&CpuTimer1, 200, 20000); //Timer1’s interrupt will happen every 20ms or 0.02seconds
 ConfigCpuTimer(&CpuTimer2, 200, 40000); //Timer2’s interrupt will happen every 40ms or 0.04seconds

 // Enable CpuTimer Interrupt bit TIE
 CpuTimer0Regs.TCR.all = 0x4000;
 CpuTimer1Regs.TCR.all = 0x4000;
 CpuTimer2Regs.TCR.all = 0x4000;

 init_serialSCIA(&SerialA,115200);
```

… Purposely left code out of listing …

```
 // Enable global Interrupts and higher priority real-time debug events
 EINT; // Enable Global interrupt INTM
 ERTM; // Enable Global realtime interrupt DBGM

 // IDLE loop. Just sit and loop forever (optional):
 while(1)
 {
```

Look below in the timer 2's interrupt function, cpu_timer2_isr(), and you will see that UARTPrint is set to 1 every 50th time the timer 2's function is entered. So both the rate at which timer 2's function is called and this modulus 50 determines the rate at which the serial_printf function is called. serial_printf prints text to Tera Term or some other serial terminal program. Also notice that after the serial_printf function call, UARTPrint is set back to 0. Think about why that is important. Add a comment to the UARTPrint = 0; line of code explaining why it must be set back to zero here to make this code work correctly. (Correctly means that the serial_printf function is called at a periodic rate.)

```
 if (UARTPrint == 1 ) {
```

For this exercise, I would like you to put most of your written code here. In future labs you will find that code run here, in this continuous while loop "while(1)", is less important code. It will be your lower priority code that does not have as strict of timing. Also this is the only place in your code that you should call serial_printf. serial_printf is somewhat of a large function and depending on how many variables you print can take some time and is not deterministic. Technically, you can call multiple serial_printf functions in a row here but it is better if you just call serial_printf once with all the variables you want to print.

```
        serial_printf(&SerialA,"Num Timer2:%ld Num SerialRX: %ld\r\n",CpuTimer2.InterruptCount,numRXA);
 UARTPrint = 0;
 }
 }
}
```

Right now consider the calling of this function, cpu_timer0_isr() "magic". (We will explain this in detail soon in the course.) "cpu_timer0_isr() is called every 10ms, without fail, in this starter code. (It actually "interrupts" the code running in the main() "while(1)" while loop.) In main() you can change the 10000 (microseconds) in the line of code "ConfigCpuTimer(&CpuTimer0, 200, 10000);" to have it be called at a different rate.

```
// cpu_timer0_isr - CPU Timer0 ISR
__interrupt void cpu_timer0_isr(void)
{
 CpuTimer0.InterruptCount++;

 numTimer0calls++;

 if ((numTimer0calls%250) == 0) {
 displayLEDletter(LEDdisplaynum);
 LEDdisplaynum++;
 if (LEDdisplaynum == 0xFFFF) { // prevent roll over exception
 LEDdisplaynum = 0;
        }
 }
 // Blink LaunchPad Red LED
 GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;
 // Acknowledge this interrupt to receive more interrupts from group 1
 PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

Right now consider the calling of this function, cpu_timer2_isr() “magic”. (We will explain this in detail soon in the course.) “cpu_timer2_isr() is called every 40ms, without fail, in this starter code. (It actually “interrupts” the code running in the main() “while(1)” while loop.) In main() you can change the 40000 (microseconds) in the line of code “ConfigCpuTimer(&CpuTimer2, 200, 40000);” to have it be called at a different rate.

```
// cpu_timer2_isr CPU Timer2 ISR
__interrupt void cpu_timer2_isr(void)
{
 // Blink LaunchPad Blue LED
 GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;

 CpuTimer2.InterruptCount++;
```

Since “CpuTimer2.InterruptCount” increments by 1 each time in this function, this below if statement sets UARTPrint to 1 every 50th time into this function “cpu_timer2_isr()”. The % in C is modulus. Modulus returns the remainder of the divide operation. So 23 % 50 equals 23, 67 % 50 equals 17, etc.

```
 if ((CpuTimer2.InterruptCount % 50) == 0) {
 UARTPrint = 1;
 }
}
```

Exercise 3

1. Study the above partial code listing and read through my commentary in black.

2. Starting with a new Labstarter project, first change either the periodic rate that Timer 2’s interrupt function is called or the 50 modulus in Timer 2’s interrupt function to make the default `serial_printf` function be called every 250ms. Debug your code and check that the default text is being printed, to the TeraTerm Windows application, 4 times a second.

3. Answer this question after finishing Task 2 above. “CpuTimer2.InterruptCount” is incremented by 1 each time `cpu_timer2_isr()` function is run. (Remember the processor, “by magic”, runs this function at the periodic rate you set in “ConfigCpuTimer”.) If CpuTimer2.InterruptCount has incremented to 3721, how much time has gone by? This answer could be different from another student depending on what you changed to make serial_printf print every 250ms. Show your math for this calculation.

4. Write a global function and name it `saturate`. Do not use any global variables in this function. This function should have two float parameters, `input` and `saturation_limit` and return a float. This return value is saturated by `saturation_limit`. `input` is the input signal (or input value) that will be saturated if its value is greater than +`saturation_limit` or less than –

`saturation_limit`. For example, if `input` is passed 3.4 and `saturation_limit` is passed 5.5 then `saturate` will return 3.4. If `input` is passed 9.5 and `saturation_limit` is passed 5.5 then `saturate` will return 5.5. If `input` is passed -7.4 and `saturation_limit` is passed 5.5 then `saturate` will return -5.5. You will test this function in Task 5.

5. We are going to use the `sin()` function to generate a sine wave at the sample rate of 250ms. Create five global float variables and one global 32-bit integer:

    a. `float sinvalue = 0;`

    b. `float time = 0;`

    c. `float ampl = 3.0;`

    d. `float frequency = 0.05;`

    e. `float offset = 0.25;`

    f. `int32_t timeint = 0;`

    Then inside the same if statement where "serial_printf" is called and before "serial_printf" write three lines of code:

    g. `timeint = timeint+1;`

    h. `time = timeint*0.25;`

    i. `sinvalue = ampl*sin(2*PI*frequency*time) + offset;`

    j. Create one more global float variable, float `satvalue = 0;`

    k. In another line of C code, set `satvalue` equal to your `saturate` function that is passed `sinvalue` to the input parameter and constant 2.65 passed to the `saturation_limit` parameter.

    l. As a final step, modify the `serial_printf` statement so that it prints on one line: `timeint` (note it is a 32-bit integer), float variable `time` with 2 digits of precision, `sinvalue` with 3 digits of precision and `satvalue` with 2 digits of precision. %.3f is the formatter for printing a float with 3 digits of precision. %ld is the formatter to print a 32-bit integer.

    m. Build and run this code on your Launchpad (red board). Does everything work correctly? Check TeraTerm for the prints. Demo this code working to your TA. The print out should look something like:

```
Timeint = 30, Time = 7.50sec, Input = 2.371, SatOut = 2.37

Timeint = 31, Time = 7.75sec, Input = 2.198, SatOut = 2.20

…etc
```

n. Come up with a way for you to remember the floating point format in these printf statements. You will need to use this throughout the semester. Just using %f is normally not the best because that prints 6 digits of precision. Most of the time when I print floats I use either %.2f or %.3f.

o. A common mistake with `serial_printf` is to use the %d formatter for 32-bit integers. %d is for 16-bit integers on the F28379D processor and %ld is for 32-bit integers. Change your code to add one more global 32-bit integer and two global 16-bit integers. Modify these three new variables somehow each time your CPU Timer 2 interrupt function is called. Then in the `serial_printf` statement (inside the continuous while loop), change it so that it now prints `timeint` along with the three new variables. Print a 32-bit integer, then a 16-bit integer, then a 32-bit integer, and finally a 16-bit integer. Demo this working to your TA. Then change the printf formatter to all %d instead of the two that are %ld. What do you notice that happens to the printing? In the same way change all the %d to %ld. What happens? Figure out a way that you will remember to always use %d for 16-bit integers and %ld for 32-bit integers.

Submissions

- In your Box folder for this class, create a HW1 folder.
- In this folder put the following:
  - Exercise #2: Submit your code and any screen shots showing the code working for Chapters 3, 4 and 5. This is the only code of the semester that you do not need to comment.
  - Exercise #3:
    - Your commented code file. Comment the code you wrote, remembering to put your initials (use at least three letters) by your comments so I can find them easily.
    - A screen shot of Tera Term printing the correct text for part m.
    - A screen shot of TeraTerm printing out incorrect values seen in part o due to using %d instead of %ld, etc., part o.