ME 333: Introduction to Mechatronics

Assignment 3: More blinking lights and the C compilation process

Electronic submission due **before** 12:30 p.m. on February 1st

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

1 Understanding the Compiler

1.1 Before you begin this section

Create a project called SimpleIO.mcp from SimpleIO.c found in the Assignment3 zip folder. Many questions in Problem 1 will refer to files created by SimpleIO.mcp.

1.2 The C compilation process

When you compile a C program there are several command-line tools that go into converting your .c file (like SimpleIO.c) into a .hex file. The compiler, assembler, and linker are the three main programs that the MPLAB IDE invokes behind-the-scenes when building your program. The job of the C compiler[1] is to translate your C source code to the assembly language of the target device. The compiler has the freedom to do this in several ways and compiler options can produce significantly different assembly-level code. The assembler performs the task of taking the assembly language files (*.s file) created by the compiler and outputs object files (*.o file). The object files are the machine code equivalent of your C source files, for example, for every SimpleIO.c there is a SimpleIO.o.[2] In the final step, the linker takes all of the object files and outputs the executable file (*.elf file). The role of the linker is to put all of the object files into one file and assign memory addresses to each function and global variable defined in your C program according to a set of rules in a linker script (*.ld file, such as procdefs.ld). The IDE then runs a utility program to convert the .elf file into a .hex file.

Questions

(a) What files were generated after you compiled the SimpleIO project?

(b) Draw the compilation process (on a computer) starting with the SimpleIO.c file. Clearly denote the input files and output files of the compiler, assembler, and linker.

(c) Go to *Help→Topics* and read page 2 of the MPLAB ASM32 .chm file (page numbers are located on the lower right-hand corner of the .chm), what are the names of the compiler, assembler, and linker executables used in MPLAB?

[1] Technically the Microchip compiler is a cross-compiler, because the "host" machine (e.g., your PC or laptop) has a different hardware architecture (Intel, AMD, ARM, and MIPS are examples of companies that sell computer chips with different architectures) than the "target" machine (the PIC32).

[2] In fact, an object file can stand in as a replacement for the corresponding source code. If I wanted to compile my program with your proprietary functions, you can send me your object files and I would be able to compile my program without your source code.

(d) What are the names of at least two utility programs, listed as "Other Utitlities" on page 2 of the MPLAB ASM32 .chm reading? (Hint: you have to go to the language tools directory mentioned in the reading)

1.3 The Linker Map File

An interesting and useful output file from the linker is the map file (*.map file). You can actually see where the linker placed functions and global variables in memory by opening the map file. The map file is a text file that can be opened in any text editor, even MPLAB's editor. The two most useful parts of a map file are the *Microchip PIC32 Memory-Usage Report*, which summarizes memory usage in the various memory partitions, and the *Linker script and memory map*, which details the memory address each .o file and its functions and global variables were assigned to. You should keep in mind while exploring the map file that in addition to your code, the final executable will also include additional object files that contain functions and global variables found in the C and Microchip libraries (like PORTSetPinsDigitalIn(), PORTSetPinsDigitalOut(), printf(), malloc(), errno, etc.) that you called in your program.

Questions

(a) At what memory addresses did the linker place the main() and delay() functions in SimpleIO.elf? (Look in the *Linker script and memory map* section of SimpleIO.map that was generated after compiling the project, the important columns are the second column, which is the memory address, and the fourth column, which is either a .o file or function name)

(b) How large is SimpleIO.o in bytes? (SimpleIO.o should be directly above main from part (a). For this question the column of importance is the third column, which is the size of the object file in bytes) Note that our object file is in the .text section. By default, the .text section is were the linker puts executable lines of code.

(c) Refer to the *Microchip PIC32 Memory-Usage Report* in SimpleIO.map and notice that the size in bytes of the .text section is larger than the memory footprint of SimpleIO.o. How large is the memory difference? What do you think explains the discrepancy in size?

1.4 The Disassembler

It is very easy to go from an object file to the original assembly file because a computer's assembly language is a direct one-to-one mapping to the computer's machine code. Going from a .o file to a .s file requires a disassembler –the MPLAB language toolsuite provides us with pic32-objdump.exe, which produces a disassembly listing exactly like the one shown when you click on *View→Disassembly Listing* in MPLAB's IDE. If we have access to the source code, the disassembly will show C code followed by the assembly language it generated. The format of each line that contains assembly is a memory address in hex, machine code in hex, followed by the code in assembly language. For example, the for loop in the delay function generated 11 assembly-level instructions. The first line of assembly generated by the for loop in the delay function (underneath line 36) reads:

9D0013BC AFC00000 sw zero,0(s8)

which states that at virtual memory address 0x9D0013BC on the PIC32, we will see the machine instruction 0xAFC00000, which in human-readable form is "sw zero,0(s8)" (i.e., store the value zero at memory address (s8 + 0), which corresponds to the C statement i = 0 in the for loop).

Questions

(a) Go to *View→Disassembly Listing* in MPLAB's IDE. How many assembly instructions does the assignment TRISGCLR = 0x3000; generate in main() (line 19)? Write the equivalent one-line statement using the bitwise-and operator and TRISG on line 20. How many assembly instructions does your TRISG statement take?

(b) Earlier we mentioned that different compiler options can produce different assembly code. The code optimization options[3] are examples of this. To change the optimization level go to *Project→Build Options...→Project*, click on the "MPLAB PIC32 C Compiler" tab, and select "Optimization" under Categories. To change the optimization level click inside the circle with a 0, 1, 2, s, or 3 in it. Note where each option lies on the code size vs. speed curve. What is the program memory size for optimization level 0 and 3? (To see program memory usage go to *View→Memory Usage Guage*) How many more assembly language instructions were generated in the delay function for optimization level 0 than for level 3 (look at the disassembly listing for comparison)? Before moving on, return everything back to its **default optimization** value by clicking "Restore Defaults" under Optimization.

2 A more complicated blinking light program

This programming assignment uses pins E0-E7 as output to turn on and off the LED bar and pin E8 as input to read the state of the push button. The following is a walk-through on how to assemble the circuit.

2.1 Before you begin this section

1. Read the NU32v2: Digital Input and Output wiki page to familiarize yourself with digital I/O.
2. Gather the following material from your NU32v2 Kit.
    - LED bar
    - Resistor array (don't forget that the dot is common ground)
    - Extra push button
    - 10k resistor

2.2 Assembling your circuit

1. Insert Pin 1 of the LED Bar in hole E5 on the right-hand side of the breadboard. It should be in contact with pin E7 on the PIC32 board.
    - Pin 1 of the LED bar is located next to the notch on the LED Bar.
2. Insert Pin 1 of the Resistor array in hole G15 on the right-hand side of the breadboard.
    - Pin 1 of the resistor array is directly underneath the black dot. If the letters on the resistor array are facing you, then pin 1 is the leftmost pin.

[3] If you want to learn more about the optimization options, feel free to refer to MPLAB C Compiler for PIC32 MCUs Users Guide, section 1.8.6.

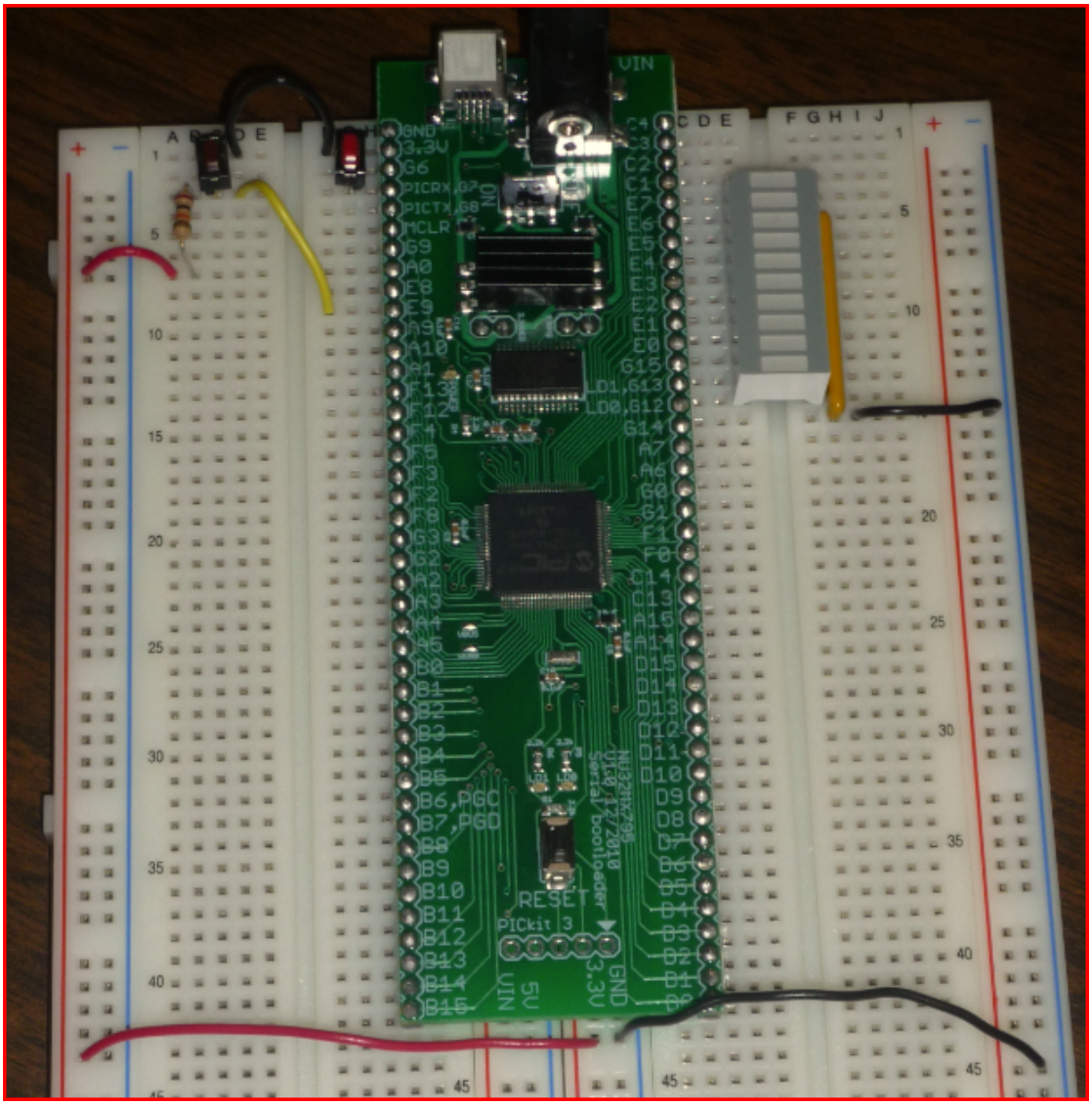

Figure 1: Final Layout of circuit

3. Connect Pin 1 of the resistor array to the ground rail on the right-hand side of the breadboard.
    - Your circuit should resemble Figure 2
4. Insert your extra switch in holes C1 and C3 on the left-hand side of the breadboard.
5. Connect Row 1 to ground by placing a wire in holes D1 and F1 on the left-hand side of the breadboard. The top of the switch is now connected to ground
6. Insert the 10k resistor in hole B3 and B7 of your breadboard.
7. Connect A7 to 3.3V using the power rail on the left-hand side of the breadboard.
8. Insert a wire in holes D3 and F9 on the left-hand side of the breadboard.
    - Your wire should eletrically connect the bottom of your switch to pin E8 on the PIC32 board.
    - The left-hand side of your circuit should resemble Figure 3
9. Connect the 3.3V rail underneath the PIC32 board to the power rail on the left-hand side of the breadboard.
10. Connect the Ground (GND) rail underneath the PIC32 board to the ground rail on the right-hand side of the breadboard.
    - Your final circuit should resemble Figure 1

Questions

(a) Draw (on a computer) the pull-up resistor and switch circuitry that you built on the left-hand side of the breadboard

(b) Draw (on a computer) one row of the LED bar circuitry.

(c) Refer to Table 4-31 in the PIC32 Data Sheet,
    - What is the virtual base address of PORTE?
    - What is the default value of TRISE after reset? What does the value represent in terms of the state of the I/O pins?
    - Do all of the bits in the TRISE register affect the pins on PORTE? If not, which bits are relevant?

(d) Refer to the Reference Sheet, Section 12: I/O Ports, what are LATE, TRISE, and PORTE? What C data type(s) should you use to read or write these registers?

(e) How are LATE and PORTE the same and how are they different?

(f) Somewhere in the Microchip header files there exists a constant, _PORTE_BASE_ADDRESS, which defines the base memory address of PORTE for our PIC32. In what header file is this constant defined? (Hint: start with <plib.h> and work your way down the chain of #includes.)

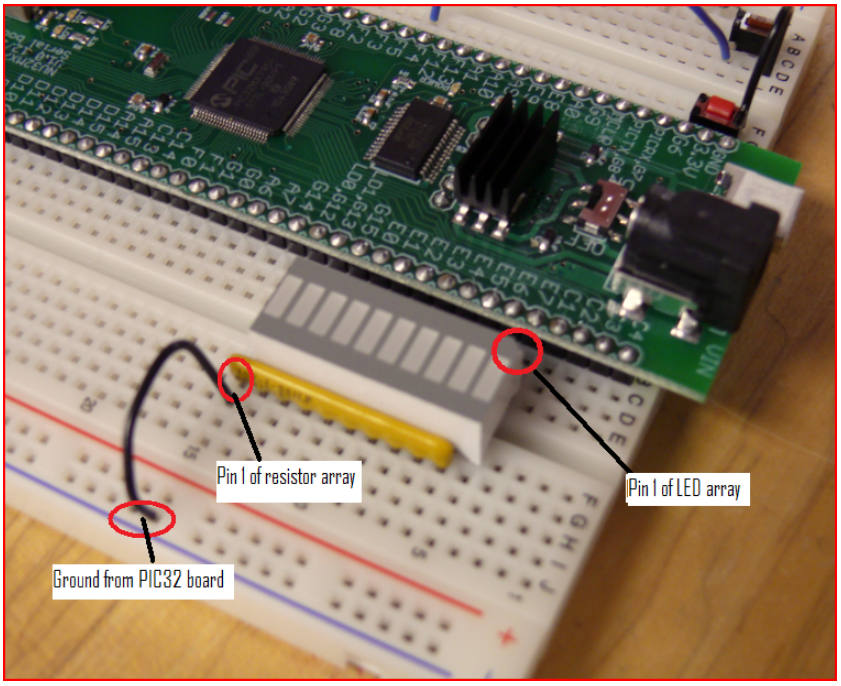

Figure 2: End of Step 3 of circuit assembly

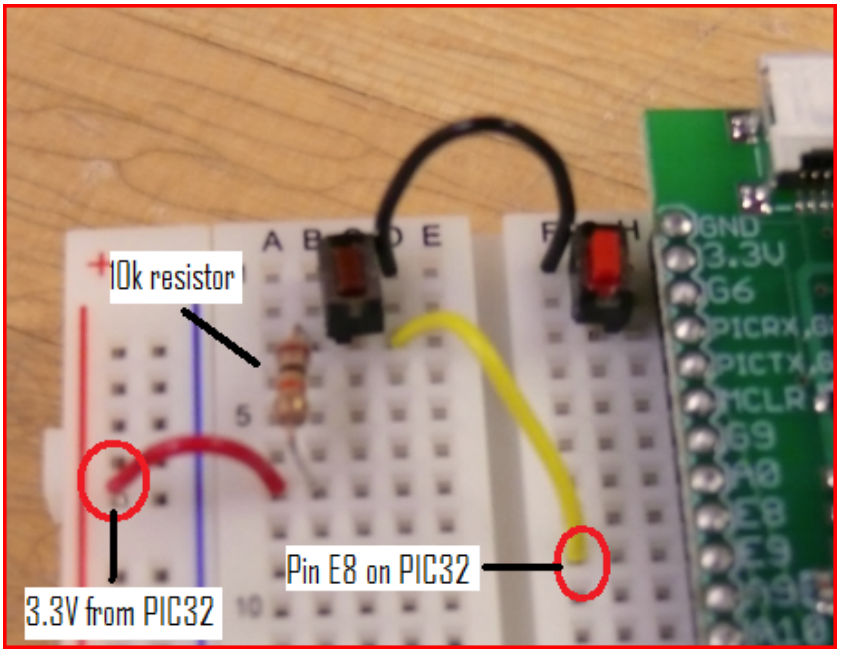

Figure 3: End of Step 8 of circuit assembly

2.3 Programming Assignment

Using the sample code in NU32v2: Digital Input and Output as a reference create a program with the following behavior:

- LED Bar behavior 1: Alternate (at a reasonable frequency) between blinking LEDs connected to even pins E0-E6 and odd pins E1-E7.
- LED Bar behavior 2: Alternate between blinking LEDs connected to pins E0-E3 and pins E4-E7.
- With a push button on pin E8, "LED Bar behavior 1" is active when the button is pushed down, otherwise "LED Bar behavior 2" is active.

3 What to turn in

For the questions, 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_a3.zip. For example, the TA would submit his homework as Rosa_a3.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 LED bar/push button circuit with the three programmed behaviors.

For full credit you must follow these instructions exactly.