ME 333: Introduction to Mechatronics

Assignment 3: The C compilation process

Electronic submission due **before** 11:00 AM on Thursday February 2nd

All questions asked in this problem set must be typed and submitted via Blackboard. See the ME333 homepage for more details. To make it easier to find the questions in this document, **questions are in bold**.

1 Understanding the Compiler and the Includes

The questions in this section are about "Simple_pic.c", which we discussed in class. This program can be found on the Mechatronics wiki at `http://hades.mech.northwestern.edu/index.php/Image:Simple_pic.c`. Before answering the problems in this section, you should create an MPLAB X project with this file and compile it.

As mentioned in the *Crash Course in C* document, there are several steps in the C compilation process that takes a piece of source code in C all of the way to a .hex file that you can put on your PIC. As with any C compilation process, the main components in the mplabc32 compiler suite utilized by MPLAB X are the compiler the assembler, and the linker. In this section, we are going to look at each of components in more detail.

1.1 The C compilation process

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[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 process up to this point is very similar to what happens when we compile C code for the computer. When compiling code for the PIC there is an additional step that involves converting the *.elf executable into a *.hex file that we then load onto the PIC.

Questions

(a) **What files were generated after you compiled the "Simple_pic.c" project?** This includes any *.o, *.o.d, *.elf, and *.hex files. This does not include any Makefiles, *.mk, *.bash, *.xml or *.properties files.

[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.

(b) In MPLAB X, click *Help→Help Contents* to bring up the documentation for both MPLAB X and the compiler. In the "Contents" tab you will find a topic titled "C32 Toolchain" under the "Language Tools" heading. After reading the first couple of pages[3], **what are the names of tools used for compiling, assembling and linking?** These are the executables that the IDE is ultimately calling when building your program.

(c) In class you explored `<plib.h>`, which turned out to be a convenient way to include all of the Microchip PIC32 definitions. Go to the top-level Microchip directory on your computer (on my PC it's C:\Program Files\Microchip) and search for files named `<plib.h>` in that directory. **How many files named `<plib.h>` do you find?** If you open them, you'll notice that they each have subtle differences. How does the C compiler know which file to use? How can you even tell what the compiler is doing? A handy compiler switch is `--verbose`, which tells the gcc compiler to tell you everything it is doing. To enable this switch, go to *Run→Set Project Configuration→Customize...*. In the left window pane, select pic32-gcc. This changes the right window pane where you should see a textbox labeled "Additional options:". In this box type in `--verbose`. Press OK and recompile your project. After compiling your project look through the output window and you'll find the `#include` search path. Using this information, **which plib.h file is the compiler using?** Write down enough of the file's path name to distinguish it from the other file locations. If in doubt, give the absolute path.

(d) Eventually everything should map down to the hardware. Open up the `p32mx795f512l.h` file and go to line 5,924. You'll see the definition for `PORTA`. Judging from the attribute tag, **where is the header file telling the compiler to place `PORTA`?** When all of the source code has been compiled into object files it becomes the job of the linker to make sure that PORTA resides at the correct address in memory. All of the SFRs, including `PORTA`, are defined in a special object file, `processor.o`, that contains all of the SFR addresses. Our PIC's processor.o file is in pic32mx\lib\proc\32MX795F512L. **What else is in this folder that might be useful to the linker?** If you have some experience with the command line you can inspect this object file with a utility called objdump:

```
./pic32-objdump -t processor.o
```

Some sample output from executing this command is shown below:

```
bf886000 g     O *ABS*    00000000 TRISA
bf886004 g     O *ABS*    00000000 TRISACLR
bf886008 g     O *ABS*    00000000 TRISASET
bf88600c g     O *ABS*    00000000 TRISAINV
bf886010 g     O *ABS*    00000000 PORTA
bf886014 g     O *ABS*    00000000 PORTACLR
bf886018 g     O *ABS*    00000000 PORTASET
bf88601c g     O *ABS*    00000000 PORTAINV
bf886020 g     O *ABS*    00000000 LATA
bf886024 g     O *ABS*    00000000 LATACLR
bf886028 g     O *ABS*    00000000 LATASET
bf88602c g     O *ABS*    00000000 LATAINV
```

At what address (the first column) is processor.o telling the linker to place `PORTA`? Is the address a virtual or physical address? This address should line up with the address given in the PIC32 data sheet. You can verify this by looking at Table 4-24 in the data sheet. A useful piece of information to know is the default value of PORTA at startup. Refer to Table 4-24, **what is the reset value of `PORTA`?**

[3]Documentation is also available in *.pdf form. It is available on Microchip's website or from within the directories where MPLAB X and your C32 compiler are installed.

(e) We will sparingly use the functions and macros defined in the Microchip library, but you should still be aware of what it offers. In your PIC32 C tool suite directory, you'll find a header file for each important peripheral on the PIC32 in the proc/ folder. Open the file port.h in the proc directory and go to line 1,171. Here you'll find macros for manipulating `PORTA`. **How would you use `mPORTAReadBits(...)` to read the values on pins `A0` and `A9`?** Your answer should involve calling the macro function with the proper input.

1.2 The Linker

Recall that it is the job of the linker to stitch the object files into a .elf file. Passing the option "-M" to the linker tells the linker to generate a "map" file. A *.map file is a simple text file that contains a report on the output of the linking process. It details how much memory was used as well as the address assigned to your program's global variables, functions, and object files. To tell MPLAB X to generate this file, click *File→Project Properties* and then select the `pic32-ld` section of the options. From the *Option categories* dropdown menu, select *Diagnostics*; then in the box to the right of "Generate map file" enter a name for the map file to be saved as. For example, "simple_pic_memory.map".

Note that the *.map file is organized following a standard convention. Briefly, the .text section is where the machine instructions get grouped together. The other sections like .sbss and .data, and .bss are where constants and global data is stored.

Questions

(a) By searching the generated *.map file, determine the following: **what memory addresses (in hex) did the linker place the main() and delay() functions in the output .elf file?** (Use a text editor with searching capabilities, you shouldn't read the whole *.map file.)

(b) Add a global variable and a global constant to your file by adding the following line just below the `#include <plib.h>` line:

```
int glob = 21;
```

Now clean and rebuild the project and open the *.map file again. Find the reference to "glob" in the *.map file. It should be in a section called ".sdata". How many bytes does this variable use (in hex)? Change the data type to a `char`. **How many bytes does it use now (in hex)?** Look at the "Microchip PIC32 Memory-Usage Report" towards the top of the *.map file, and find the .sdata section. **Where in the PIC's memory does `char glob` get placed (in hex)?**

(c) **How large is the total .o (in hex bytes) produced by this simple C file?** This will be found in a .text section. The 32 bit hex number is the address in the virtual memory map where the object file will be placed. The shorter hex number in the third column is the total memory usage of that object file.

(d) The procdefs.ld file is used by the linker to determine where in memory the object files go. **Is the location of the "Simple_pic.c" consistent with procdefs.ld, in other words, does the location of the .o get placed within kseg0_program_memory as we would expect? Are the instructions here cacheable?**

(e) Temporarily move procdefs.ld out of the directory that contains the source code. Clean and build the project to re-compile everything. Look at the new map file. **Where does "Simple_pic.o" get placed now (in hex)? Has it changed?** Note that the NU32 bootloader also has code in this region and it won't erase itself. Knowing this, **will the newly compiled version (that is missing procdefs.ld)**[4]

[4]In the absence of a local procdefs.ld file, the linker automatically uses a default one that is included with the installation of the C32 toolsuite.

run on the NU32? Having control of where programs are placed in memory according to a linker file is a powerful tool that allows different programs, like a bootloader and your code to be placed in flash without interfering with each other.

1.3 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 toolsuite uses `pic32-objdump`. 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. To see the disassembly in MPLAB X, click *Window→Output→Disassembly listing window*. Note that a file containing the disassembly can also be generated by passing command-line arguments to the compiler, or calling `pic32-objdump` manually.

Questions

(a) In the delay() function there is the following line:

```
LATAbits.LATA5 = !LATAbits.LATA5; LATAbits.LATA4 = !LATAbits.LATA4;
```

How many assembly instructions does this line generate when we view the disassembly? Just below this line is a different version of the same thing that uses the "fast bit manipulation" INV register for Latch A. Change your code to use this line instead, recompile, and analyze the new disassembly. **How many instructions are produced now?**

(b) Assume that you have the value `0xF0CA` in LATA. You are trying to get the latch to contain the value `0xAAAA`. **What are the two lines of C code that write to the LATASET and LATACLR addresses to accomplish this? What value do you write to the LATAINV address to accomplish the same thing?**

(c) MPLAB X has some very useful tools for exploring the files that your project includes. Clicking *Window→Classes* from the top menu will bring up a listing of all classes, functions, structs, macros, and constants that are available for your use. As described in detail at `http://hades.mech.northwestern.edu/index.php/NU32:_A_Detailed_Look_at_Programming_the_PIC32_on_the_NU32` the inclusion of `plib.h` at the top of this simple file actually includes many .c, .h, and .o files written by Microchip that define all of the SFRs, the addresses of the SFRs, the macros to interface with the PIC, and much more. The *Classes* window provides a convenient interface to all of these files. Using this window, **what file defines the struct we are using when we type `LATAbits.LATA5`?**

(d) We can roughly estimate the time that a given function or chunk of code will take to run by analyzing the assembly code. The architecture of the PIC32 is such that most instructions can be executed in a single clock cycle. Looking at the disassembly for the delay() function, find the line that looks something like:

```
9D0080B0  AFC00000   SW ZERO, 0(S8)
```

located just below the C line `for (j=0;j<10000;j++)`. Now find the line

```
9D0080D0  00000000   NOP
```

just above the "}" that closes the `for` loop. **How many lines of assembly code are there between these two lines?** Each of these assembly lines gets executed once for a single pass through the `for` loop. Knowing that our CPU runs at 80MHz, **how long does a single pass through the `for` loop take?** Using this number, **at approximately what frequency should the lights be toggling?** Looking at the running program, **does your estimate seem reasonable?**

(e) Modify the C code in this simple program to make pin `A0` a digital output. Then add lines above and below the call to `delay()` that allow you to toggle the logic level that pin `A0` has. Now with your NUScope, calculate the time that the delay loop takes. This is a useful trick to time how long your functions are taking. **How long is delay? How far off was your estimate?**

1.4 The Bootloader

When the PIC32 is reset it automatically starts executing whatever it finds at memory address 0xBFC00000. For us this happens to be the start of the NU32 bootloader program. Downlod the bootloader source code from `http://hades.mech.northwestern.edu/index.php/Image:NU32_boot.zip`. In MPLAB X load the project in the PIC32_UART_Bootloader_BootFlash_Explorer16.X project folder. The code in this project is complicated, but BootLoader.c is simple enough to follow along at a high level. For example, you can tell that lines 49-54 are where the configuration bits are set. The main program calls two functions, `ValidAppPresent()` and `JumpToApp()`. Take a look at these functions and answer the questions below.

Questions

(a) In the function JumpToApp(), **what address is the application expected to begin at?** You should use the tools built into the IDE to quickly find the definition of `USER_APP_RESET_ADDRESS`. For example, *Navigate→Go To Declaration.*

(b) **How does the bootloader know in `ValidAppPresent()` that there is a valid application at `USER_APP_RESET_ADDRESS`?** Because the PIC32 uses NAND flash memory, a long string of this value represents unused space on a NAND-based flash device, like the memory card found in cameras or USB sticks. Note: If you are unsure of what a `DWORD` is, use the IDE to help you find its definition.

2 What to turn in

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_a3.zip.