EE 2361 - Introduction to Microcontrollers

Laboratory # 2b

Writing a mixed ASM/C Program

Background

In programming, it can be easy to get stuck in doing what is comfortable. For example, the previous lab concentrated on writing code to support iLED's in assembly language. This was justified because we needed a precise 6 cycle pulse to implement the iLED serial communication protocol. It would be tempting to continue to develop further code in assembly to utilize these devices. However, assembly language requires significantly more work to develop complex code. Even moderately complex code can be hard to develop and very hard to read / test. For example, converting a "long int" representing a 24-bit RGB value or, even, computing a color "rainbow" of 1000's of 24-bit RGB values and streaming each in succession out the serial protocol will take up many many lines of assembly code.

For our purposes, remember one easy rule:

- If something can be done in C, then do it in C, else do it in ASM.

Purpose

This lab is an extension on Lab #2a, it will explore how to integrate ASM and C in a single project. You will package up assembly instructions into a C-compatible ASM-based library. Then use that library to reimplement a function from the Adafruit NeoPixel Library on PIC24. In the end you will have all the pieces needed to create sparkly light shows quickly and easily!

Supplemental Resources

PIC24FJ64GA004 - Family Datasheet
16-bit MCU and DSC Programmer's Reference Manual
MPLAB® XC16 User Guide - Section 8.3 on Data Types
C-Programing Tutorial - Header Files
C-Programing Tutorial - Data Types
GNU C Lib "stdint.h"

Required Components:

Standard PIC24 requirements (caps, pullup, debug header, etc.) iLED 100 Ohm Resistor

Pre-Lab

In the pre-lab you will build a mixed C/ASM project. The main() function will be in C and we will provide some of the functions from Lab4a for use in the C function global namespace. Using these ASM functions, you will replicate the functionality of your original ASM-only code.

Setting up the C-based project

1.) Create a new Project for this lab
Remember: Name it something useful like, "orser_lab4b_v001"

2.) Create a new XC16 main file and add it to your project.
Remember: This is under Microchip Embedded → XC16 Compiler → mainXC16.c
Name it something like "orser_lab4b_main_v001.c"
Remember: Creating a new file does *not* add it to your project in some versions of MPLABX.
3.) Modify the main(void) function so that it has a setup() and an endless loop architecture: either an endless loop in main with the main actions of your program inside the body of the loop, or a loop() function and a call to it from an endless loop inside main.
4.) Clean and Build Main Project to check your syntax, etc.

5.) You will need to add full C boilerplate to the top of this file. Go back to your lab 1 project to find these. Here's a reminder about what you need:
- Include xc.h
- Configuration register setup (all those #pragma config lines).

6.) Clean and Build Main Project to check your syntax, etc.

Adding a ASM-based Library to a C-based project

1.) Create a new "assemblyFile.s" and add it to your project
Remember: Assembler → AssemblyFile.s
Name it something like "orser_lab4b_asmLib_v001.s"

WARNING: If you name your c-file and assembly-file with the same beginning file name you will cause cryptic errors in compilation! Make sure filename differ by more than just the extension (ie., orser_lab4b_asmLib_v001.s orser_lab4b_main_v001.c)

2.) You will need a different boilerplate for an ASM library. Use the code included below. Specifically you won't need the following:
- Configuration registers
- Global declaration of the `_main` function/label
- The .bss section: space for stack is allocated by the C compiler before calling main(). Also, we are not going to define any variables in assembly: we can do all our global variable definitions in C. It's safer and easier to let the C compiler handle data organization in data memory.

```
.include "xc.inc"

.text                          ;BP (put the following data in ROM(program memory))

; This is a library, thus it can *not* contain a _main function: the C file will
; deine main().  However, we
; we will need a .global statement to make available ASM functions to C code.
; All functions utilized outside of this file will need to have a leading
; underscore (_) and be included in a comment delimited list below.
.global _example_public_function, _second_public_function
```

Boilerplate code for ASM-based library source files

3.) Add to this file a few of your pre-made functions from Lab 2a. Rename the labels to start with an underscore (_). This is needed for Global Function. You will need at least the following functions:
   - _write_0()
   - _write_1()
   - _wait_50us()

4.) Add each of the above functions to the .global statement.

It should look something like this when you are done (note: “djo” are my initials: David J. Orser. You should use your own initials):

```
.global _write_0, _write_1, _djo_wait_50us

_write_0:
                              ; 2 cycles for a function call
      bset   LATA, #0
```

Example of properly setup .global statement and first function label

5.) Clean and Build Main Project to check your syntax, etc.

6.) Be polite: if you use any registers in your functions (e.g., if you use w5 in _wait_50us as a loop counter), make sure that you save the value of the register at the beginning of the function, and retrieve the value right before you exit. You can to consider 1 cycle for the push and 1 cycle for the pop instruction[1]. The code would look something like this:

```
_wait_50us:
      push    w5
      mov     #0x107, w5
      ...
      pop     w5
      return
```

[1] The reason you save and retrieve the values of registers is that the caller (in our case main()) might be using the register as its own loop counter. We don’t want a seemingly harmless call to a function wreck the value of that counter.

Call etiquette

Creating a Header File

In multiple file c-programming it is common to need to declare functions before we define them. This is because the c-compiler is dumb as a rock and is unable to scan all the files in your directory before it starts doing the actual compile work. Therefore, it requires the input and output arguments of every function to be declared before it is called in *any* file that calls those functions. This is accomplished by the use of a “header” file. This is a file that declares all the functions available in a corresponding C or ASM, but doesn’t actually define them.

Note: Declare and Define are two different things. You declare a function like so:

```
int foo(int);
```

Declare foo(), note this statement only specifies input and output arguments

You define a function like so:

```
int foo(int arg) {
    int bar;
    bar = arg + 1;
    return bar;
}
```

Define foo()

We will now create a header file for our ASM library, the include it in our C program.

1.) Right Click on your Lab4b project in the Projects pane
2.) Click New → Other
3.) Select C → Header File
4.) Name it exactly the same thing as the .s source file, for example “orser_lab4b_asmLib_v001.h”
5.) Click “Finish”
6.) Add the header file to your project under “Header Files” if it is not added already.
7.) Open up for edit your header file and add declarations for each of your assembly function. Remember these assembly functions do not take any arguments. In the future, if you should need arguments in assembly functions, please see the Quick Lesson - Assembly Function Argument Passing.

Your header files should look something like this:

```
#ifndef ORSER_LAB4B_ASMLIB_V001_H
#define      ORSER_LAB4B_ASMLIB_V001_H

#ifdef __cplusplus
extern "C" {
#endif
void djo_wait_50us(void);
void write_0(void);
void write_1(void);
#ifdef __cplusplus
}
#endif
```

Example header file declaring multiple global assembly functions

8.) Now include your header file in your main C file. It should look something like this:

```
#include "xc.h"
#include "orser_lab4b_asmLib_v001.h"
```

Example of a ASM header inclusion in standard C file

9.) Clean and Build Main Project, now to verify syntax, etc.

Replicate main() from Lab4a in C

Now that we have added our assembly functions for writing 0's and 1's and creating delays, we need to fill in the rest of the code in C. We have to set up RA0 as a digital output port, and setup the clock to get a 16MHz instruction clock frequency.

Mentally partition Lab4a into setup() and the infinite loop sections.

1.) Translate the code from Lab 4a before "foreverLoop:" into setup()
Hint: You can also go back to your Lab 1 project to help remember what to do here.
For example:

ASM code	C Code
`mov #0xffff,w0` `mov w0,AD1PCFG`	`AD1PCFG = 0x9fff;`

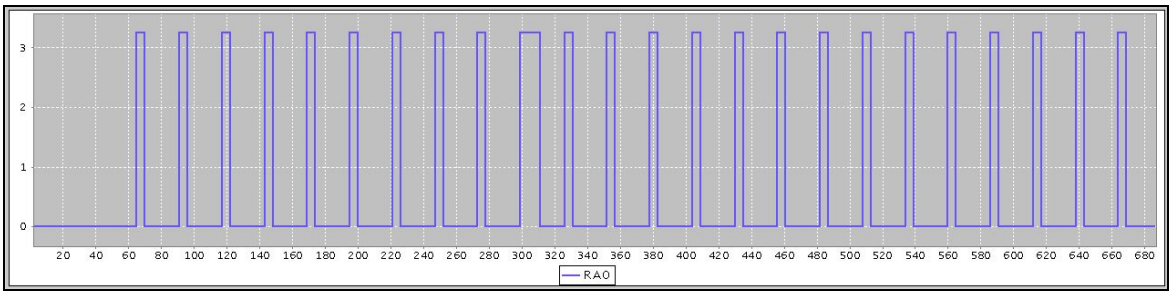
2.) Translate the "foreverLoop:" section of your Lab4a assembly file into the endless loop inside main while(1) {...}. Don't forget to call setup() in main.
For Example:

`call _write_0`	`write_0();`

ASM code	C Code

HINT: There should be combined 24 calls to **`write_0()`** and **`write_1()`** and one call to **`djo_wait_50us()`** preceded by resetting of RA0.

3.) Clean and Build Main Project
4.) Make sure you have “Simulator selected”
5.) Add a breakpoint on your wait() function call.
6.) Simulate your program. It should look something like this:



Example of 24-bit pulse train showing the color #004000, utilizing C with ASM functions

7.) Switch to your programmer and program your PIC24. Hopefully your iLED changes color!

Pre-Lab Deliverables

- C Program that contains boilerplate, setup(), endless loop in main(), and when run correctly sets the color of your iLED
- The following ASM function are accessible in the C namespace:
  - `void wait_50us(void);`
  - `void write_0(void);`
  - `void write_1(void);`

Pre-Lab Checklist

- ❏ Take pre-lab quiz on header files
- ❏ Complete the walk-through to create a mixed C and ASM project in MPLAB X
- ❏ Bring to lab a single assembly project that fulfills the Pre-Lab Deliverables

Lab Procedure

During this lab you will create support C code to make your iLED shine.

Create a decoder function

The first component needed is a decoder function that will translate three 8-bit color values into 24 consecutive calls to **`write_0()`** or **`write_1()`**. The typical method used for this type of serialization of data is called "mask and shift", but there are many ways to implement it. You will have to choose a method and implement it.

Finally, don't forget to hold the output low for >50us to latch the data into the iLED (**`wait_50us();`**)

1.) Write a function that does the above with the following declaration:

```
void writeColor(int r, int g, int b);
```

2.) Run a simulation of your "hardcoded" color loop() function, look at the resulting output on the Logic Analyzer.
QUESTION: How many cycles does your hard coded program take to write 24-bits?

3.) Replace the hardcoded calls to **`write_0()`**, **`write_1()`**, and **`wait_50us()`** with your new **`writeColor()`** function.

4.) Simulate your new function.
QUESTION: How many cycles does your new function take to write 24-bits?

5.) Verify your new function operates correctly in hardware.

Implement color cycler function

It is time to create our first animation! We need to decide on the sequence for our animation. One option is to create a continuous change from one color to another (often called a gradient).

Example of a gradient

Mathematically this is described by the following psuedo-code:

```
assume byteFrameNumber ∈ {0, …, 255}
byteRed = byteFrameNumber;
```

```
byteBlue = 255 - byteFrameNumber;
```

1.) Write a function, that takes a frame number and calls **`writeColor()`** as described above:
`void drawFrame(`unsigned char **`frame);`**

2.) Add **`drawFrame()`** to **`loop()`** wrapping it with code that increments the frame from 0 to 255, then decrements the frame from 255 to 0. Follow each call of **`drawFrame()`** by a delay of 1ms.
3.) When your code compiles successfully, load it on your PIC24.

Due to the difference between human and machine perception rates, it is useful to control the rate of an animation with a compiler constant.

4.) Write a new C function that calls your ASM function **`wait_1ms()`** a variable number of times (this doesn't have to exact N milliseconds, a minimal amount of overhead is ok):

```
void delay(int delay_in_ms)
```

5.) Create a compiler constant:

```
#define FRAME_PER 5 //Period (1/FrameRate) in milliseconds
```

6.) Utilize the above constant and the new delay function to make your program easily change frame rate.
7.) Modify your frame rate to that specified by your TA and demonstrate your program.

Adafruit Wheel() function

NOTE: This section is less guided than your previous lab procedures. Expect to get stuck occasionally. When you get stuck, consult with your neighbors, talk to your TA, ask questions. If you're working outside the lab, ask questions on the forums.

In this section you will translate an existing Adafruit Library function into code compatible with your newly built functions.

The Adafruit NeoPixel Library has a neat function buried in their example code. The `Wheel()` function cycles through the 3-dimensional RGB color space, with a 1-dimensional frame number input. The is useful for testing all three LEDs inside the iLED. It also looks kinda cool.

The Adafruit NeoPixel Library → https://github.com/adafruit/Adafruit_NeoPixel

```
// Input a value 0 to 255 to get a color value.
// The colours are a transition r - g - b - back to r.
uint32_t Wheel(unsigned char WheelPos) {
  WheelPos = 255 - WheelPos;
  if(WheelPos < 85) {
    return strip.Color(255 - WheelPos * 3, 0, WheelPos * 3);
  }
  if(WheelPos < 170) {
    WheelPos -= 85;
    return strip.Color(0, WheelPos * 3, 255 - WheelPos * 3);
  }
  WheelPos -= 170;
  return strip.Color(WheelPos * 3, 255 - WheelPos * 3, 0);
}
```

Excerpt from “strandtest.ino” from the Adafruit NeoPixel Library

The first thing to notice is that Adafruit NeoPixel library operates on 32-bit “packed color” integers. These are simply a convenient way of moving 3-variable data around. They are formatted as 0x00RRGGBB.

The Microchip compiler does not inherently support the datatype syntax “uint32_t”. This syntax is a compiler agnostic method of specifying data types. It is synonymous with the Microchip data type “unsigned long int”. If you want to use compiler agnostic methods just include “#include "stdint.h"” at the top of your main C file.

The second thing to notice is that Adafruit uses a datatype called “byte” (again not supported by default by Microchip’s compiler XC16). A byte is an “8-bit unsigned two’s complement number”. For our purposes just replace “byte” with “unsigned char”.

Packing and unpacking variables is actually very easy and can accomplished in several ways, here are some hints:

```
    unsigned char Red    = 0x40;
    unsigned char Green  = 0x20;
    unsigned char Blue   = 0x60;
    unsigned long int RGBval = 0;
    RGBval = ((long) Red << 16) | ((long) Grn << 8) | ((long) Blu);
```

Packing of an RGB value into a single 32-bit unsigned long int

```
unsigned char Red    = 0;
unsigned char Green  = 0;
unsigned char Blue   = 0;
unsigned long int RGBval = 0x00402060;
Red   = (unsigned char) (RGBval >> 16);
Green = (unsigned char) (RGBval >> 8 );
Blue  = (unsigned char) (RGBval >> 0 );
```

Unpacking of an RGB value (NOTE: types do matter here!)

Both of these examples use a technique called “type casting”, if you’re unfamiliar with this notation, please read up on it at TutorialsPoint.

To implement `Wheel()` you will need the following:

- A color packer:
  - `uint32_t packColor(unsigned char Red, unsigned char Blu, unsigned char Grn)`
- Three color un-packers:
  - `unsigned char getR(uint32_t RGBval)`
  - `unsigned char getG(uint32_t RGBval)`
  - `unsigned char getB(uint32_t RGBval)`
- A packed-color compatible version of writeColor():
  - `void writePacCol(uint32_t PackedColor)`

Your final objectives for this lab are:

1.) Create a Microchip PIC24 compatible version of the Wheel program
2.) Animate Wheel() at a specified frame rate
3.) Demonstrate it in hardware to your TA

Lab Procedure Checklist

- ❏ Demonstrate your color cycler program to your TA
- ❏ Demonstrate your `Wheel()` program to your TA
- ❏ A single file that contains all the requested ASM functions
- ❏ A “main” C file that animates `Wheel()` (must includes all called functions)
- ❏ The answers to two questions from the lab procedure

Lab Report

1. Answer the two questions on Page 8. (How many cycles does your hard coded program take to write 24-bits? How many cycles does your new function take to write 24-bits?)
2. In the color cycling function, the color is jumping from blue 0x0000ff to red 0xff0000 at the end or beginning of the cycle. Please write a psuedo-code for a smooth transition.

Appendix: Implications of ASM vs. C code

The following are the number of cycles between bits and the maximum bit updates rate for iLED's written using three different methods during the development of this lab:

- Hard Coded ASM: 26 cyc = 1625ns = 615 kHz max update rate
- Dynamic ASM: 31 cyc = 2000ns = 500 kHz max update rate
- Dynamic C: 40 cyc = 2500ns = 250 kHz max update rate

As one can see, with 24-bits of data, that results in a visual update rate of 10kHz (most TV's update at ~60 Hz), thus there is no need for further hand tailored ASM code.

It is possible to foresee a use case where there are more than 1,000 iLEDs in a single chain. In that case a 500kHz update rate would allow you to string together twice as many LEDs on a single output while maintaining a reasonable refresh rate. If this is your use case it may be worthwhile to look at hand writing a Dynamic ASM iLED driver.