

EE 2361 - Introduction to Microcontrollers

Laboratory # 2a

Bit Banging an RGB LED in Assembly



Background

Serial communications protocols are a fundamental component of operating microcontrollers in today's connected devices. Serial communication allows for complex information to pass between devices over 4, 3, 2 or even just 1 wire. Allowing everything from displays to sensors to function. These protocols are so common and important, that they often have custom digital logic, referred to as peripherals embedded on the MCU to improve efficiency. In EE1301, we commonly used the "Serial" connection to output data to a computer over USB. In later labs, we will learn how to use these optimized digital circuits to communicate over UART, I2C, or SPI.

However on the cutting edge of development in custom systems, electrical engineers often need to develop software code to implement serial communication protocols from scratch.

PIO = GPIO

In the PIC24 Reference Manual, digital input or output ports are called Parallel I/O Ports (PIO). This is synonymous with the term General Purpose I/O (GPIO) that you may remember from EE1301 and several other platforms (ie., RaspPi, Arduino, Photon, etc.)

In our case the Individually Addressable RGB LED (iLED), introduced in EE1301, has no such dedicated digital support circuits. In this lab, we will create our own implementation of a custom serial communication standard.

Purpose

In this lab you'll familiarize yourself with the basic assembly instructions for changing the state of the GPIO pins on your PIC24 microcontroller. You'll explore the timing implications of various instructions and their impact on the resulting output waveform. You'll combine these skills to change the color of an individually addressable RGB LED (iLED). In the next lab, you'll package up your assembly instructions into a C-library so that you can create sparkly light shows quickly and easily!

Supplemental Resources

[PIC24FJ64GA004 - Family Datasheet](#)

- Section 10.1 Parallel I/O (PIO) Ports

[16-bit MCU and DSC Programmer's Reference Manual](#)

- REPEAT, CALL, and RETURN Functions

iLED from Adafruit References:

- [WS2812 Datasheet](#)
- [Device Description - Individually Addressable LEDs](#) (from EE1301)
- [EE1301 - IoT Lab #2 - Section Individually Addressable LEDs](#)

Required Components:

Standard PIC24 requirements (caps, pullup, debug header, etc.)

iLED
100 Ohm Resistor
Standard LED or LED strip
220 Ohm Resistor

Pre-Lab

Creating an assembly project in MPLAB X

As discussed in Lab 1, you will need to create a new project for this lab. Here is a brief outline of the steps:

- 1.) In MPLAB X, click on “File” → “New Project...”
- 2.) Select Microchip Embedded → Standalone Project
- 3.) Select 16-bit MCU (PIC24) and PIC24FJ64GA002
- 4.) Leave the Debug Header set to “None”
- 5.) Select your Tool as “Simulator”
- 6.) Select your Compiler as “XC16”
- 7.) Finally, give your project a descriptive name!
(for example “x500_labX_vXXX” → “orser_lab2_v001”)
- 8.) Click Finish

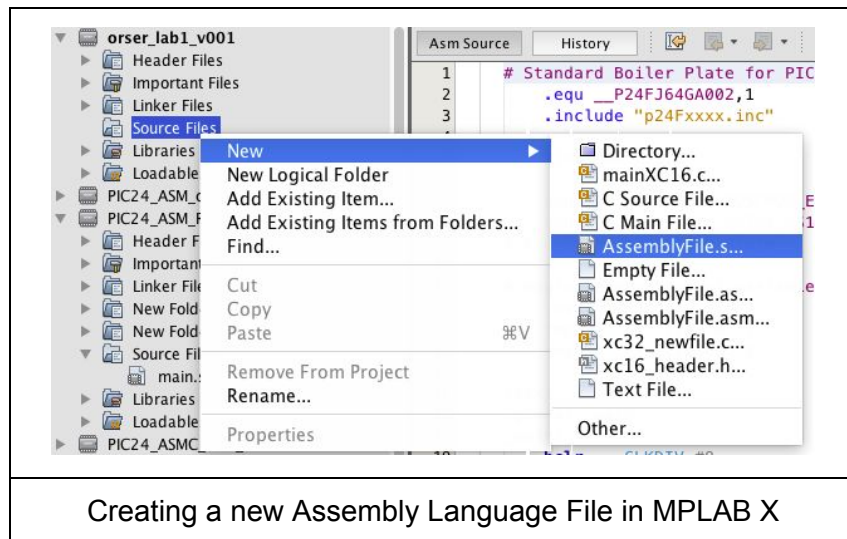
Add boilerplate code for assembly language project

The boilerplate code for an assembly-based project is slightly different than the C-based project we did in Lab 2. First, create a new assembly language file.

- 1.) In the “Projects” tab of MPLAB X, right-click on the “Source Files” directory and select “New” → “AssemblyFile.s” Careful! There are three options that look similar.

File Types

There are several suffixes for assembly files in MPLAB X, “.s”, “.as”, “.asm”, and “.S”. Each suffix is processed slightly differently by the compiler. For example “.s” uses “;” as the comment, “.S” uses “/” Be wary of source code on the internet that may be in a slightly different format. In this lab, All labs will be using the “.s” suffix and syntax.



- 2.) Give the file a name like, "orser_lab1_core_v001", leave Project, folder, and Create File alone. Make the name unique, as it gets confusing if there are three files open called "main.s"
- 3.) In some versions of MPLABX, this doesn't add the new file to your project! It just creates an empty file and opens the file for editing. To add the file to your project: Right-click "Source Files", select "Add Existing Item...", click on your new file and click "Select". It should appear in your Project under the "Source Files" heading (or complain that the file is already part of the project, in which case no harm is done).

4.) In the editor enter the following “boilerplate” text into your new source file:

```
.include "xc.inc"          ; required "boiler-plate" (BP)

;the next two lines set up the actual chip for operation - required
config __CONFIG2, POSCMOD_EC & I2C1SEL_SEC & IOL1WAY_OFF & OSCIOFNC_ON &
FCKSM_CSECME & FNOSC_FRCPLL & SOSCSEL_LPSOSC & WUTSEL_FST & IESO_OFF
config __CONFIG1, WDTPS_PS1 & FWPSA_PR32 & WINDIS_OFF & FWDTEN_OFF && BKBUG_ON &
GWRP_ON & GCP_ON & JTAGEN_OFF

        .bss          ; put the following labels in RAM
counter:
        .space 2      ; a variable that takes two bytes (we won't use
                        ; it for now, but put here to make this a generic
                        ; template to be used later).
stack:
        .space 32     ; this will be our stack area, needed for func calls

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

;because we are using the C compiler to assemble our code, we need a "_main" label
;somewhere. (There's a link step that looks for it.)
.global _main          ;BP
_main:

        bclr    CLKDIV,#8          ;BP
        nop
        ;; --- Begin your program below here ---
```

Code Example X: Boilerplate for ASM-only projects

This boilerplate contains all the same functionality as our c-code boilerplate. There are calls to supporting library definitions with `.include()` and `#include` statements. You can see the flash configuration word declarations (`__CONFIG1` and `__CONFIG2`). And the Clock Divider setting of 1:1 for full 16 MIPS operation (`bclr CLKDIV,#8`).


Turn on RA0

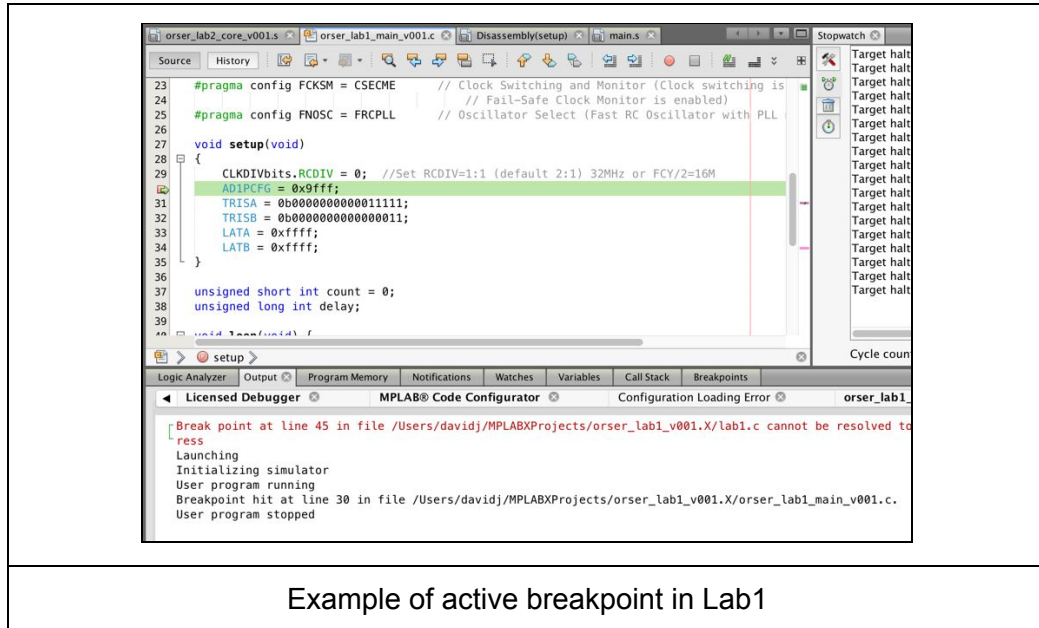
In this part of the lab we will prepare a simple assembly program that turns on the RA0 pin. This will demonstrate some basic structures of assembly code.

- 1.) Just like the C-program we need to add code that configures LATA, TRISA, and AD1PCFG registers. Remember, you can't load an immediate directly into a general file register (generic memory location): you must first use a special function register as a middle-man.
- 2.) Just for the sake of clarity, let us go look at how the C-program compiles the first few lines of Lab 1.
 - a.) If you see the Lab1 Project on the left pane, under “Projects”, Right-Click on the Lab1 Project, and click “Set as Main”. Otherwise, just use File > Open Project to open it. You will later go back to your assembly project.
 - b.) Right-Click on Lab 1 project, click Properties, select Simulator as the target

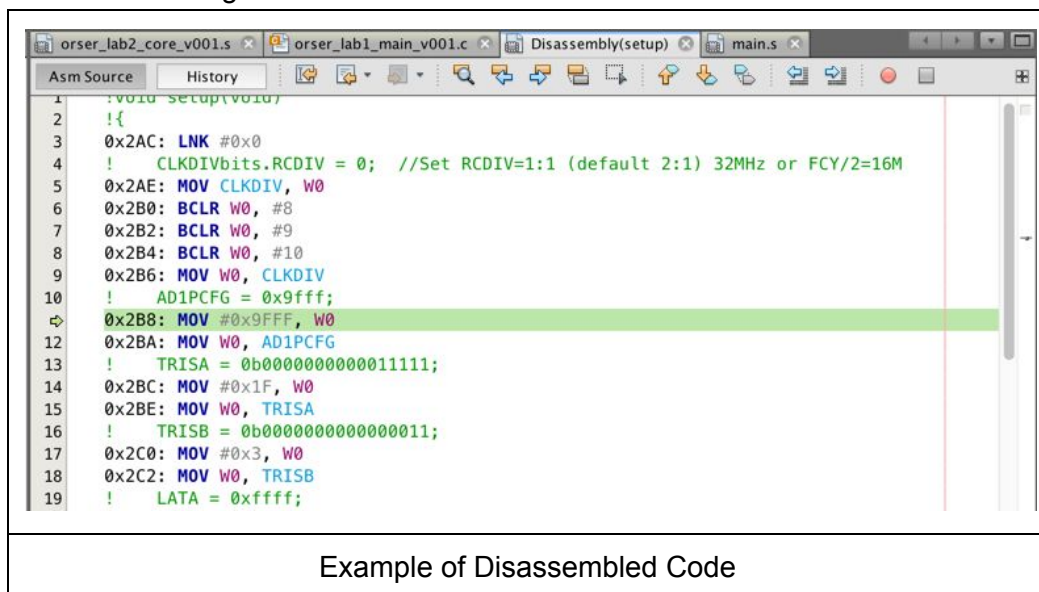


c.) Add a breakpoint on the line “AD1PCFG = 0x9fff;”

d.) Click Debug Main Project (), if all goes well it should look something like this:



e.) Click on the menu item Window → Debugging → Disassembly, it should look something like this:



You can see the C-code gets directly converted into two assembly instructions. The first loads a register (w0) with an immediate value (0x9FFF). The second copies the contents of the register (w0) into the memory location of the register (AD1PCFG).

3.) Now mimic this to set AD1PCFG, TRISA, and LATA to the following values:

- a.) Set AD1PCFG to 0x9FFF // This sets all pins to digital mode
- b.) Set TRISA to 0b1111111111111110 // This sets the RA0 pin to output mode and the rest of the PORTA pins to input (as a matter of fact, the chip that you work with has only 5 bits on PORT A, but the microcontroller doesn't mind us setting all those 16 bits in TRISA. It will ignore bits 5..15).
- c.) Set LATA to 0x0001 // This sets RA0 to output a logic high

```

mov    #0x9fff,w0
mov    w0,AD1PCFG          ; Set all pins to digital mode
mov    #0b1111111111111110,w0
mov    w0,TRISA            ; set pin RA0 to output
mov    #0x0001,w0
mov    w0,LATA             ; set pin RA0 high

```

- 4.) It is “bad form” to let your program counter continue to execute past the end of your program. Add a forever-do-nothing loop:

```

foreverLoop:
nop
bra    foreverLoop
nop
.end    ; this doesn't actually end anything. Does not translate to assembly
        ; code. Just a way to tell the compiler we are done with this file.

```

Example of a forever-do-nothing loop


NOTE: Don't forget to set Lab 2 as your main project again! This is a very common mistake while working in MPLAB X.


- 5.) Right click on Lab Project, click “Set as Main Project”, or, if you don't see the project on the left, open it.

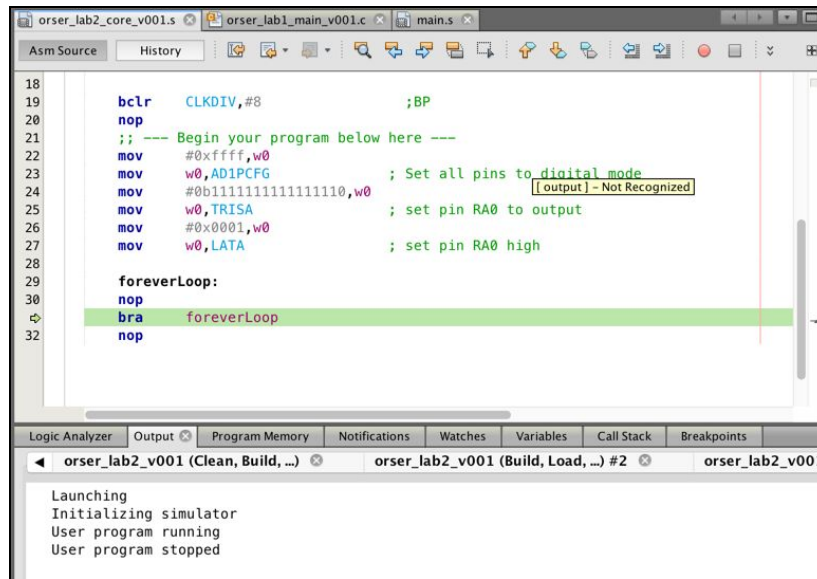


- 6.) Now compile your code, click , clean up any errors that result.

- 7.) Bring up your logic analyzer and add the signal RA0

- 8.) If you click () to simulate your code, your code runs forever because there are no breakpoints that stop the program. Try it!

- 9.) Click the Pause button (). It should look something like these screenshots:



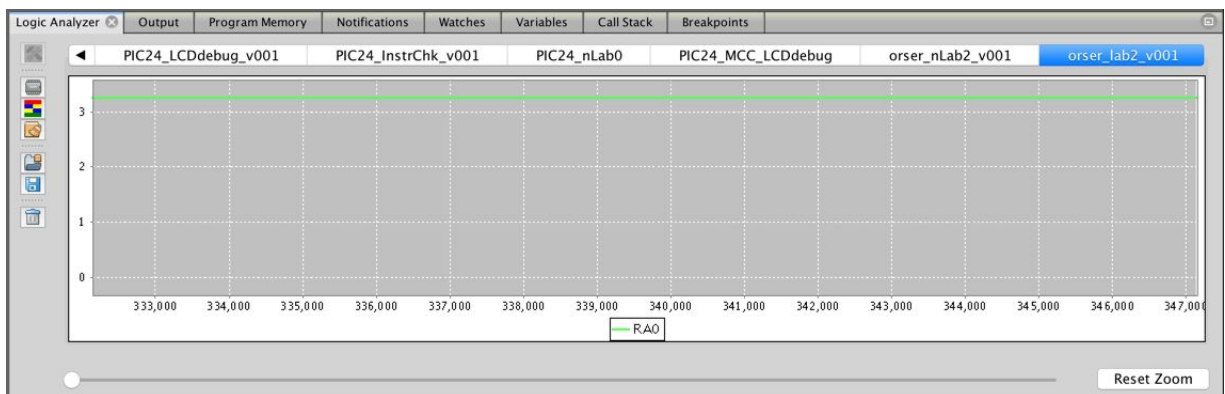
```
18      bclr    CLKDIV,#8           ;BP
19      nop
20      ;; --- Begin your program below here ---
21
22      mov     #0xffff,w0
23      mov     w0,AD1PCFG          ; Set all pins to digital mode
24      mov     #0b11111111111110,w0 ; [output] - Not Recognized
25      mov     w0,TRISA            ; set pin RA0 to output
26      mov     #0x0001,w0
27      mov     w0,LATA             ; set pin RA0 high
28
29      foreverLoop:
30      nop
31      bra     foreverLoop
32      nop
```

Logic Analyzer | Output | Program Memory | Notifications | Watches | Variables | Call Stack | Breakpoints



orser_lab2_v001 (Clean, Build, ...) | orser_lab2_v001 (Build, Load, ...) #2 | orser_lab2_v001

Launching
Initializing simulator
User program running
User program stopped

Example of paused Simulation



Example of RA0 output on Logic Analyzer

10) Now press the “stop” button to stop simulating the program. Add a breakpoint on the line that says “mov w0,LATA”. Press the Debug Project button () again, and when the program stops at the breakpoint (make sure the “Logic Analyzer” window is up), press F8 (Step Over ) a couple of times. You will notice that RA0 makes a transition to 1 in the logic analyzer window.

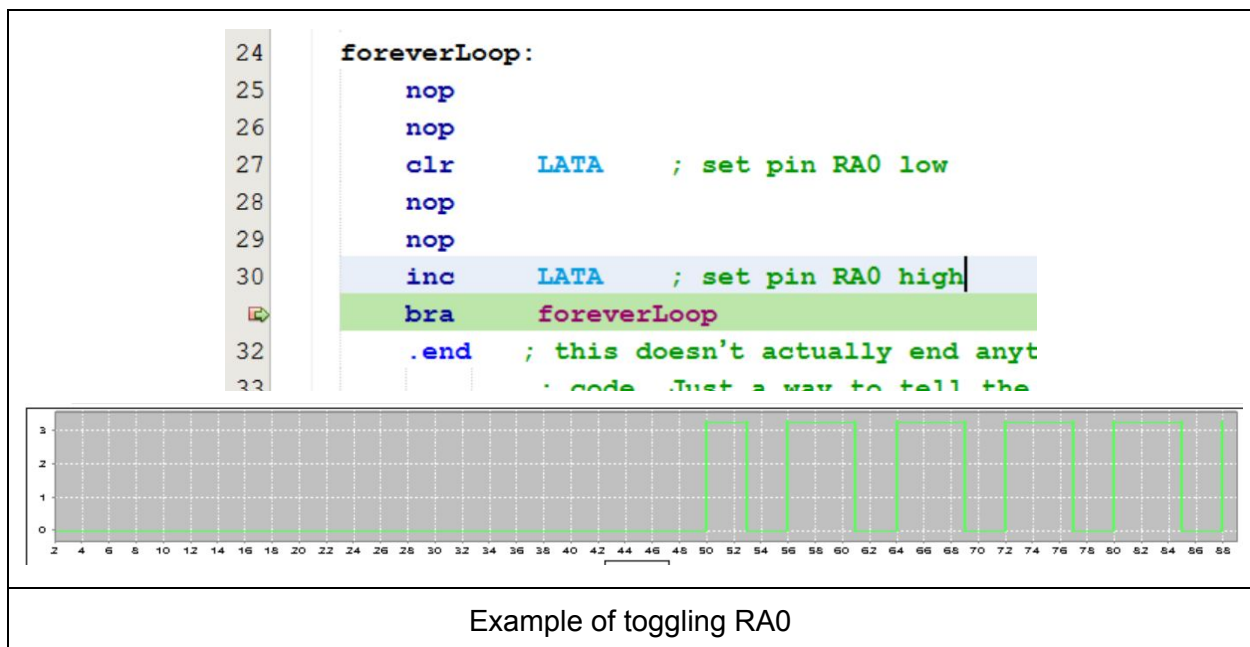
Toggle RA0

In this section of the lab you will toggle RA0 and learn to carefully control the duration of the high/low logic pulses.

When properly configured our PIC24 operates on a 32 MHz internal Clock, which results in a 16 MIPS (million instructions per second). This is called $F_{cy}=16\text{MHz}$ in the PIC manual. This means most instructions take $1/16\text{MHz} = 62.5\text{ns}$ to execute. We can use a combination of two programming methods to create blocking delays of precisely N times 62.5ns.

NOP Delay Method

The most basic instruction for this purpose are “nop”. Which does exactly what it says “No Operation” for 62.5ns. For example the following code produces the following output.



Notice how the first pulse is narrower than the subsequent pulses? The first rising edge on RA0 happened because of the “mov w0, LATA”, before the “foreverLoop” label. There were only two nop instructions between that instruction and the clr instruction, which results in a falling edge on RA0. However, when we are inside the loop, there are three instructions between the instruction causing a rising edge (inc LATA), and the falling edge (clr LATA). The branch instruction (bra foreverLoop) takes two instruction cycles, resulting in a total of 5 instruction cycles for the high part of the signal as opposed to only 3 during the low-time of the signal.

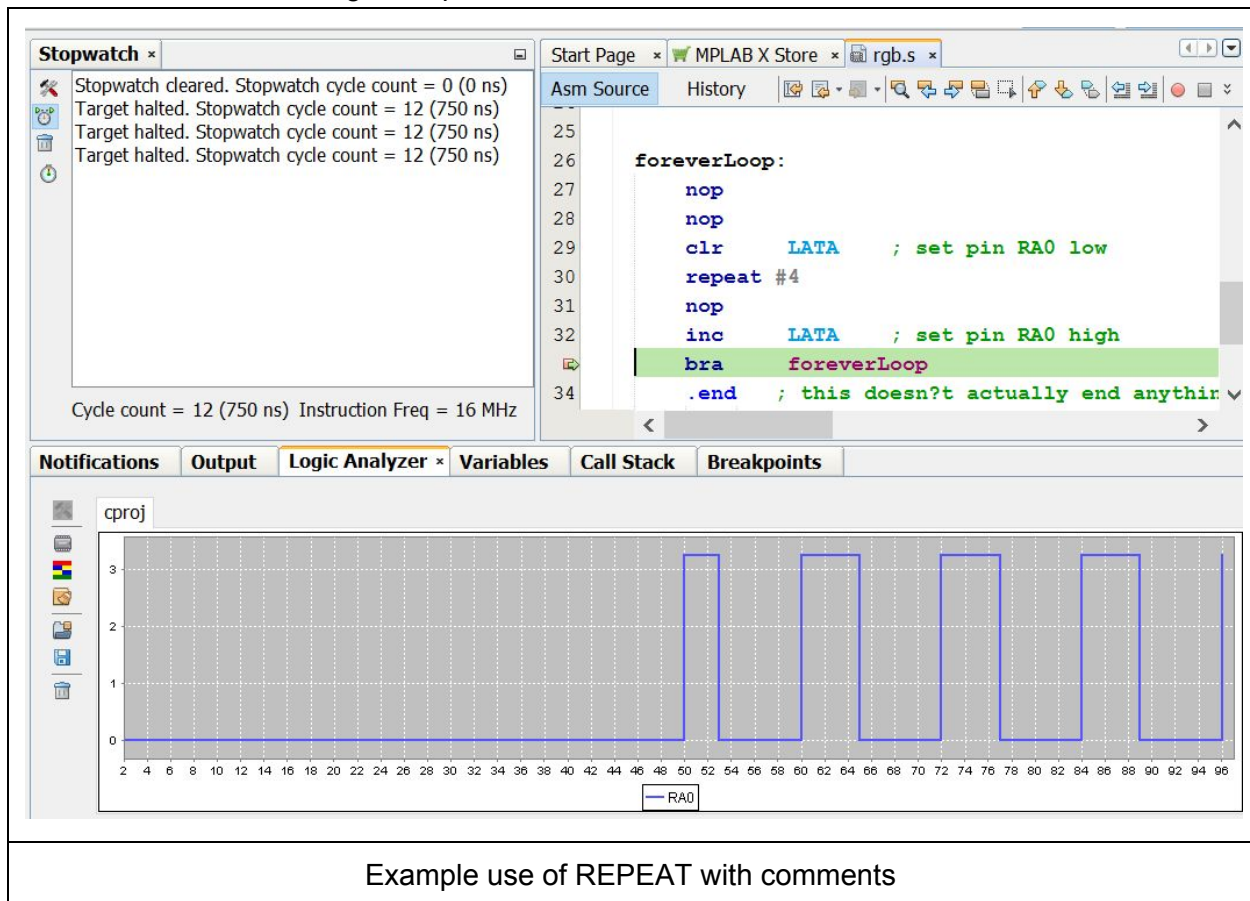


REPEAT Delay Method

The second method is excellent for creating precise delays of longer duration. The REPEAT instruction can be found in the 16-bit PIC Programmers Reference Manual (page 355 as of the current printing.)

Basically the REPEAT #N command repeats the next instruction N+1 times, where N is the argument of REPEAT, a `#14lit` (a 14-bit integer number between 0 and 16383).

Take a look at the following example code and simulation:



REPEAT can create precise delays of between $(3) \cdot 62.5\text{ns} = 187.5\text{ns}$ and $(16385) \cdot 62.5\text{ns} = 1,024,062.5\text{ ns}$ (slightly more than 1 ms). This makes it a fantastic tool for precise delays less than or equal to 1ms. The code is readable and takes a minimum of program memory space.

There are several instructions that cannot follow a REPEAT (including the CALL instruction discussed in the next section) see the 16-bit PIC Programmers Reference Manual for a complete list.

CALL and RETURN instructions

Function calls (often referred to as subroutines in the PIC documentation) are executed in assembly via the CALL and RETURN instructions. These instructions take several cycles each to store/retrieve important registers, update the program counter, and flush the instruction fetch+decode stage. They are both defined in detail in the 16-bit PIC Programmers Reference Manual. We'll briefly review them here, but if you need further information please read the manual before approaching your TA or instructor.

The CALL instruction requires one argument, the address of the subroutine to be called (in our case actually a label.) CALL takes 2 cycles to execute. The RETURN instruction doesn't require any arguments. RETURN takes 3 cycles. These extra cycles need to be accounted for when building precise timing code.

In assembly, a function is simply a label with a RETURN instruction at the end.

```
wait_10cycles:
    ; 2 cycles for function call
    repeat #3    ; 1 cycle to load and prep
    nop         ; 3+1 cycles to execute NOP 4 times
    return      ; 3 cycles for the return
```

Example of a function in assembly

You call this function with the CALL instruction.

```
foreverLoop:
    call    wait_10cycles    ; 10 cycles
    clr     LATA             ; set pin RA0 low = 1 cycle

    nop     ; 2 cycles to match BRA delay
    nop     ;
    repeat #8    ; 1 cycle to load and prep
    nop     ; 8+1 cycles to execute NOP 9 times
    inc     LATA    ; set pin RA0 high = 1 cycle
    ; Total = 12 cycles high, 12 cycles low

    bra     foreverLoop
```

Example of a function call in assembly

In the pre-lab deliverables section below you will have to modify this above example to make the pulse generation its own function.

HINT: It is possible to create wait functions for each pulse width (high and low). However, if you never reuse the function, it's a waste of time and energy to write it. Instead, you should write the REPEAT/NOP instruction in-line.

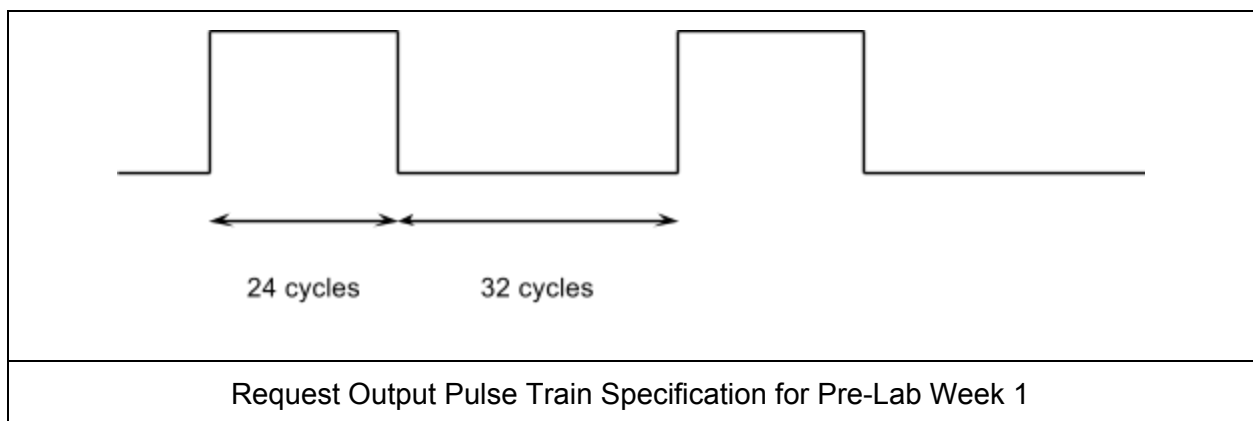
Pre-Lab Deliverables

For the Week 1 Pre-Lab, on your own, you need to create several pieces of complete assembly code. These will be used in lab!

- 1.) Create an all assembly function and calling assembly program
 - a.) The assembly function should be of the form (pseudocode):


```
void write_bit_stream(void) {}
```

(NOTE: Remember your function should be in assembly, where inputs and outputs are passed by either the “stack” or registers. In this case there are neither.)
 - b.) The function should generate a 24-cycle high, 32-cycle low pulse
- 2.) Place your function within your program within a forever loop (HINT: you'll have to add a couple NOPs to account for the forever loop BRA instruction and trim a couple cycles from the high/low delay in the function.) The waveform is shown in the diagram below. Be prepared to demonstrate this program to your TA at the start of lab (ie., hit simulate and view the logic analyzer.)



- 3.) Create four additional subroutines using REPEAT to provide exact delays:
 - Delay 1us
 - Delay 10us
 - Delay 100us
 - Delay 1ms

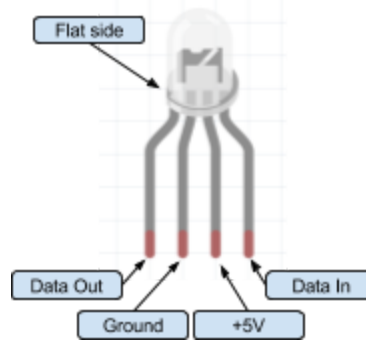
Pre-Lab Checklist

- ☐ Take pre-lab quiz on LATx, PORTx, ADxPCFG
- ☐ Complete the walk-through to create an assembly project in MPLAB X
- ☐ Bring to lab a single assembly project that fulfills the [Pre-Lab Deliverables](#)
- ☐ Read through the [iLED specification](#) (and Quick Reference Document). Note that the package and pins on our iLED are different from the figure in the specs document.

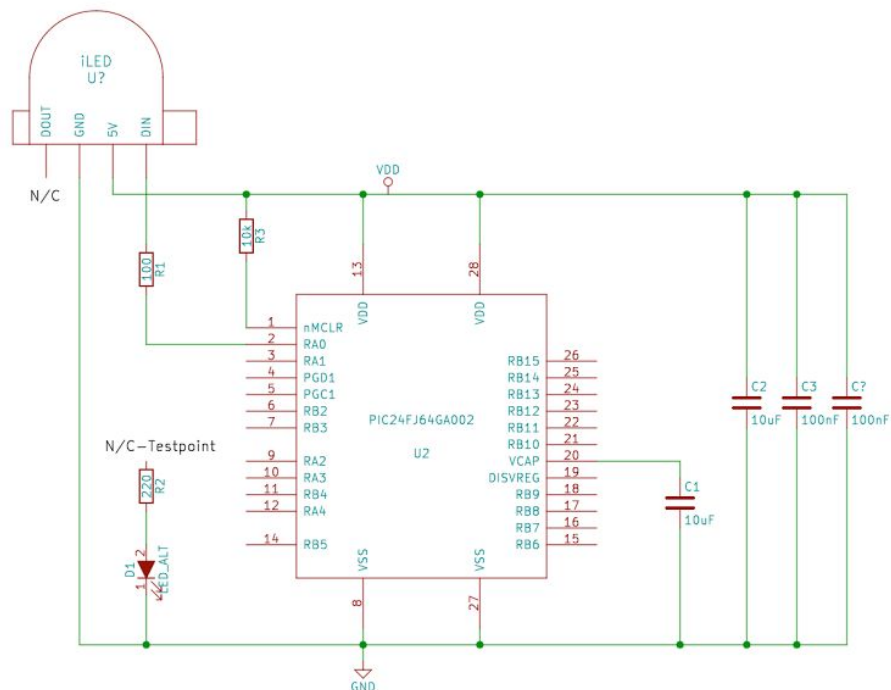
Lab Procedure

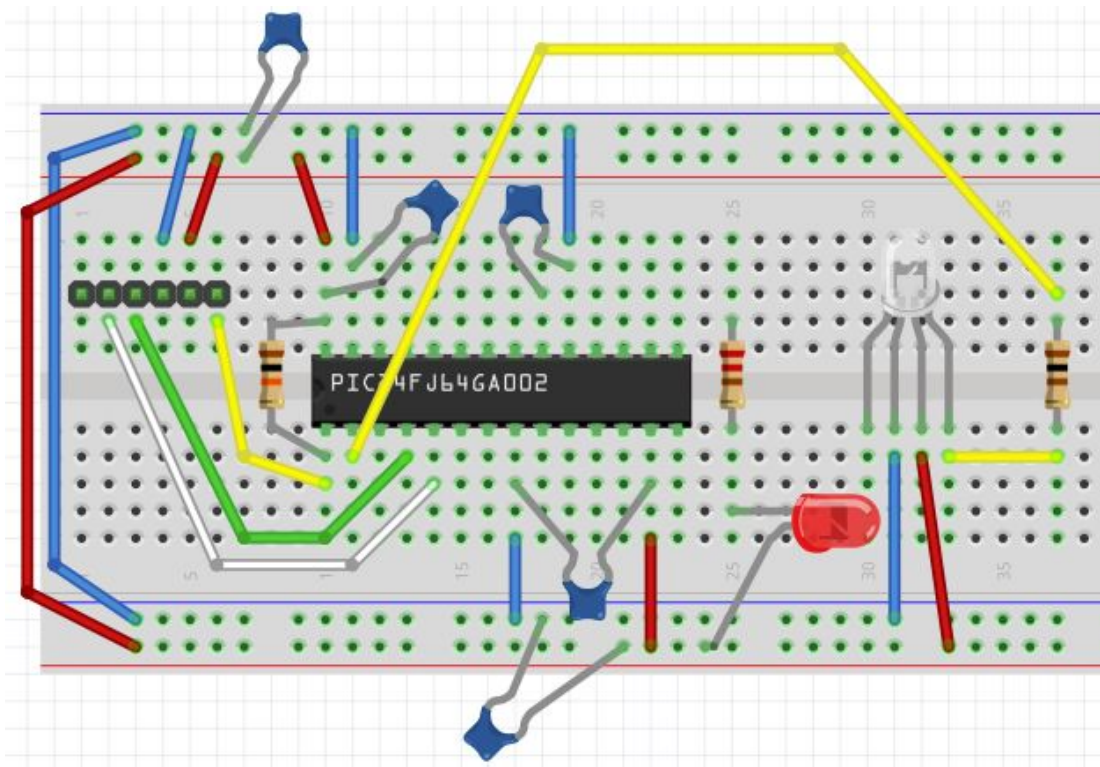
Verify PIC24 hardware operation

In this section you will connect your pre-lab program up to an oscilloscope and verify the output pulse train is of the correct duration. First setup your PIC, something like this:



iLED pins






Schematic and Wiring Diagram for Test and Debug of WS2812-based iLED

- 1.) Load your program onto your PIC24

Remember: You can do this by:

- a.) Right-clicking on the project → Select Properties
 - b.) If the “Hardware Tool” box is greyed out, select the UNLOCK button
 - c.) Scroll down to the bottom of the Hardware Tool: Box
 - d.) Select “chipKitProgrammer - SN xxxxxx”
 - e.) Click OK
 - f.) Click “Make and Program” ()
- 2.) Connect your output (RA0) to scope (CH1), don't forget to connect GND too.
 - 3.) Configure your scope to trigger on CH1, rising edges @ 1.65V
 - 4.) Configure your scope to measure positive pulse width and negative pulse width

You should see a continuous pulse train with a positive pulse width of 1.5us and a negative pulse width of 2.0us (plus or minus <5%).

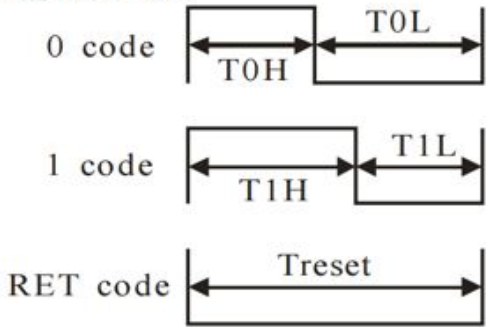
Create a fixed serial bit pattern for WS2812 chip

The WS2812 is the chip at the heart of our iLEDs. The chip specification should be closely reviewed before this lab begins. When looking at this specification, it is easy to be confused. After you've read the specification (trust us this is important), review the details below.

The iLED specification, page 4, shows that a narrow pulse width is used for transmitting a "0", wide pulse width for transmitting a "1" and finally a long period of low as a RESet signal. Each of these patterns are outlined in the iLED specification and replicated below:

Data transfer time(TH+TL=1.25μs±600ns)			
T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

Sequence chart:



0 code: T0H, T0L

1 code: T1H, T1L

RET code: Treset

WS2812 - Original Specification

This specification is a bit confusing at first. For example, you might deduct from the table above that adding T0H + T0L = 1.15μs, yet the text at the top says "TH+TL=1.25μs", which looks like a discrepancy. But what the table is telling us is the *range* of pulse widths that would be interpreted as a 0 code or a 1 code. In the example we just considered, 1.15μs is still a legitimate value within the range of 1.25μs ± 0.6us.

Perhaps it would be easier to see the big picture if we translate the above table to the following ones that spell out the min/max values for the parameters:

Data Transfer Time (TH+TL)

0.65μs .. 1.85μs

T0H	T0L	T0 Period
0.2μs .. 0.5μs	0.65μs .. 0.95μs	0.85μs .. 1.45μs

T1H	T1L	T1 Period
0.55μs .. 0.85μs	0.45μs .. 0.75μs	1μs .. 1.5μs

A long LOW value of > 50μs would be considered a reset signal. We highly recommend that you translate the above tables to the number of cycles your PIC24 should waste to create these delay ranges.

Looking at page 5 of the specification, the required sequence of bits to fully programing the iLED is 24-bits in total, followed by a long low (RESET/Latch) command. This should load the iLED with the data and cause the color to change.

Follow the procedure below:

- 1.) Create two new assembly functions modeled after your `write_bit_stream` function:

```
void write_0(void) {} // Create a 0.35us pulse in a 1.25us
period
void write_1(void) {} // Create a 0.70us pulse in a 1.25us
period
```

Note that we picked 1.25μs because it conforms to the specs (satisfies both the T0 and T1 Period constraints). Why did we pick the same period for both 0 and 1? Because you want these functions to be called interchangeably, thus it would be a plus if they take exactly the same number of cycles to operate (including all REPEAT, CALL, RETURN overhead). That way we know exactly how many cycles it will take to change colors on the iLED regardless of the RGB color value.

We suggest you do NOT call delay functions inside the `write_0` and `write_1` functions. You can use `repeat` and `nop` to create the right amount of delay directly.

When these functions are complete, test them out in hardware before proceeding:

- 2.) Re-write the “foreverLoop”, calling each of `write_0` and `write_1` twice like so:

<pre>foreverLoop: call write_0 call write_1 call write_0</pre>
--

```
call write_1  
bra      foreverLoop
```

- 3.) “Clean and Build” your program to check for errors
- 4.) “Make and Program” your device
- 5.) View the output on the oscilloscope, you should see a stuttering waveform
- 6.) Press the “Single Seq” or “Single Acquisition” button on your scope to view a snapshot in time of the waveform coming out of your PIC. You should see an alternating pattern of skinny pulses and fat pulses on a period of 1.2us.
- 7.) Measure the pulse widths, and period to verify they are correct

Once we are certain that all the timing is correct, write 24-bits to the iLED!

- 8.) Since we are going to show only one color, we don’t have to send the color data in a forever loop. You can delete instructions from your forever loop (don’t remove the bra forever Loop instruction!!!), and add serial transmission of RGB colors before the forever Loop label. Rewrite your code such that it contains 24 write_0 or write_1 calls (any combination that you choose). Remember that 24-bits = a color. Precede transmission of colors by a long reset signal (longer than 50μs), and remember that you have to make RA0 LOW for the reset.
Remember: You wrote a “Delay 100us” for the pre-lab?

Take some time now and play with the output bit stream. Can you determine a major error in the WS2812 datasheet? (HINT: It is on page 5.)

- 9.) When you are ready, prepare the HTML-based color tag that your TA has requested and demonstrate it to them.

Hint: You will need to determine the correct color order and bit endianness of the serial protocol. You can’t always depend on the accuracy of the datasheet!

Lab Procedure Checklist

- ☐ Determine the color order and endianness of the iLED
- ☐ Have a configurable assembly program that sets the iLED to any of 16M colors
- ☐ Configure and Demonstrate your program and iLED to your TA

Lab Report

This lab will have little in the way of report requirements, the full report will be at the end of Lab2b. But as an interim report, submit your assembly code, and show your delay calculations

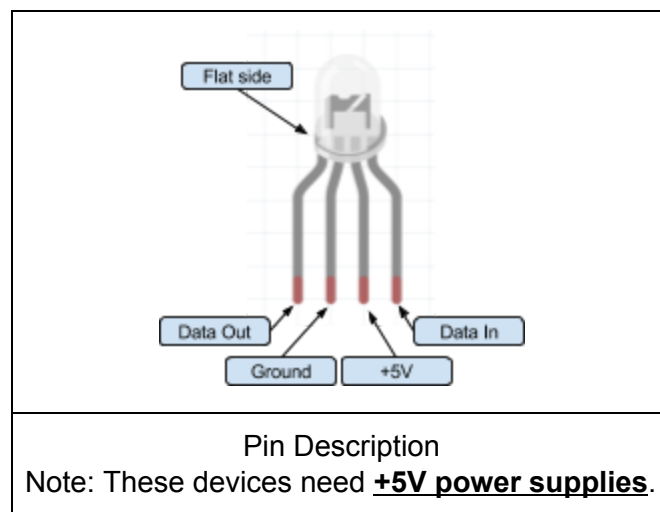
(number of cycles) that you used in your code to implement `write_0` and `write_1`. List any bugs that you had to fix for the iLED to show the colors you wanted. Also list what test patterns you used to test your program (i.e., what combination of `write_0` and `write_1` you used).

Appendix: Reminders about iLEDs

Note this is an abbreviated introduction, please see the WS2812 specification, Adafruit page, and EE1301 Device Description for more details.

These “iLEDs” are part of a class of devices that are called individually addressable RGB LEDs. This means that contained in each packaged iLED there are three LEDs and a very small LED driver chip with a serial data input. When wired up in a chain, every iLED can be individually set to a different color. Each iLED requires a shared power supply and ground pin. Additionally each iLED has a Data_In and Data_Out pin. When connected in series, the first iLED in the string grabs the first 24-bits (color setting) and then passes the rest of the data to the next iLED down the chain, which grabs the next 24-bits (second color setting) passing the rest of the data on, etc.

iLED Pinout Diagram



WARNING: Plugging these iLEDs in backwards (even for a second) will destroy them!!!