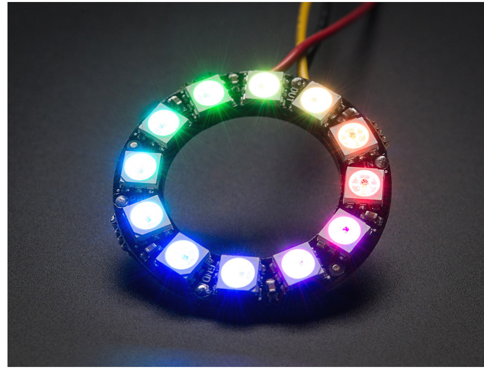
NeoPixels & In-Line Assembler



www.AdaFruit.com NeoPixel LED

Previously, we looked at driving a NeoPixel in assembler. Assembler has some advantages:

- It lets you control the I/O pins
- It allows you to precisely set the timing by counting instructions

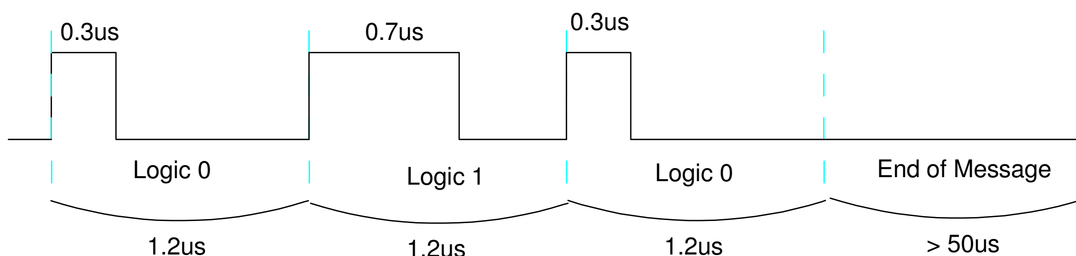
On the down side, it is very difficult to do anything in assembler: it's much easier to use a higher-level language like C.

The problem with using C for driving a NeoPixel relates to timing: each line of C can compile into multiple lines of assembler. To send a RGB message to the NeoPixel, the data needs to be send as 24 bits:

Green (byte 1)								Red (byte 2)								Blue (byte 3)							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

Timing is critical for each bit:

- Each bit needs to be 1.2us long (12 clocks)
- Logic level 0 is a 300ns pulse (3 clocks)
- Logic level 1 is a 700ns pulse (7 clocks)
- A pause of 50us or more (500 clocks) signifies a new message



Such precise timing is difficulty to obtain in C. Instead, it would be nice if you could incorporate assembler within a C program. You can.

In-Line Assembler

Hi-Tech C offers two ways to incorporate assembler within a C program. If you want to insert a single assembler command, use the command

```
asm("   nop");
```

This inserts the assembler code within the double quotes within the resulting C code.

A second option is to use the compiler directive

```
#asm
      nop
      nop
      nop
#endasm
```

This tells the compiler to insert the assembler code in the middle within the C program.

Global Variables

One problem with in-line assembler is how to pass data from the C program to the assembler program. This is where global variables come into play.

In ECE 173 Introduction to C Programming, you were told *never never use global variables.* This is because if three different routines are using the same variable, it gets really hard to debug the code.

In this class, we modify that to say *given a choice, never use global variables.* Here, we're trying to pass data from a C program to an assembler program. Everyone can see a global variable. That makes global variables a way that both variables can share data.

In assembler, the way you define a global variable is

```
PIXEL equ 0x0000
```

This places the variable PIXEL at memory address 0x0000. In C, you do this with the following code:

```
// Global Variables
unsigned char PIXEL @ 0x000;
```

Both the C and assembler routines can now access the same variable, PIXEL.

In-Line Assembler and Bottom Up Programming

From before, we had an assembler routine which could drive a NeoPixel. Let's reuse those routines and add a C subroutine which is passed {RED, GREEN, BLUE} and then sends this data to the assembler low-level routine.

This routine receives three bytes: RED, GREEN, BLUE. It then

- Stores the color in the global variable, PIXEL (located at 0x0000), and
- Calls the assembler routine Pixel_8

After all three colors are sent, it returns from subroutine using the assembler command *return*.

- The subroutine Pixel_8 is identical to what we used back in assembler
- The subroutine Pixel_1 is also identical - only using the phycisal address for PORTD (c:3971).

note: This code was found by compiling the code

RD0 = 1;

and then looking at the resulting assembler code in the .lst file.

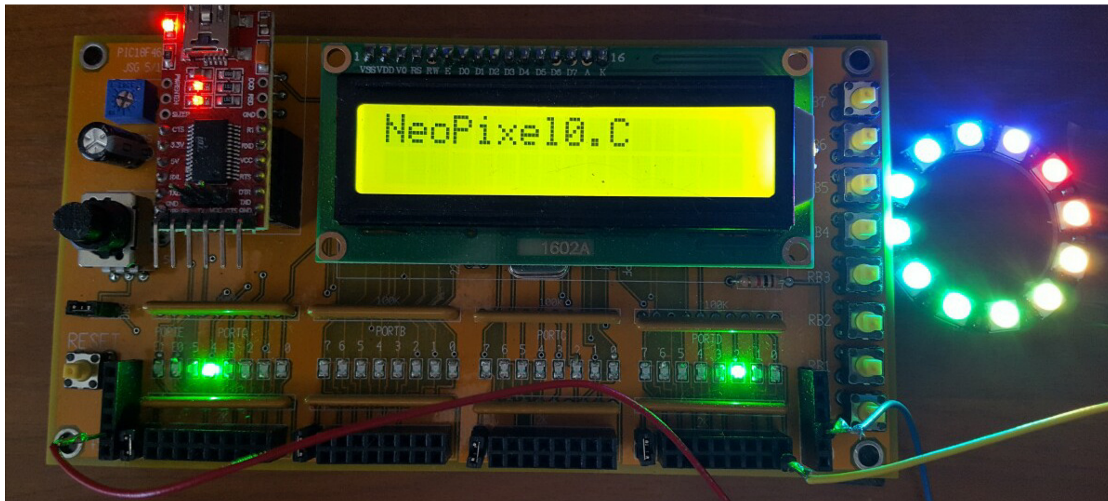
```
void NeoPixel_Display(char RED, char GREEN, char BLUE)
{
   PIXEL = GREEN;   asm(" call Pixel_8 ");
   PIXEL = RED;     asm(" call Pixel_8 ");
   PIXEL = BLUE;    asm(" call Pixel_8 ");
   asm(" return");

#asm
Pixel_8:
   call Pixel_1
   call Pixel_1
   call Pixel_1
   call Pixel_1
   call Pixel_1
   call Pixel_1
   call Pixel_1
   call Pixel_1
   return
Pixel_1:
   bsf ((c:3971)),0 ; PORTD,0
   nop
   btfss ((c:0000)),7
   bcf ((c:3971)),0
   rlncf ((c:0000)),F
   nop
   nop
   bcf    ((c:3971)),0
   return
#endasm

}
```

To test this code, make the first pixel red:

C-Code:

```
while(1) {
    NeoPixel_Display(200, 0, 50);
    Wait(100);
    }
```

NeoPixel0.C



Display a color wheel on a 16-segment Neo Pixel (Program NeoPixel0.C)

```
while(1) {
    NeoPixel_Display(100, 0, 0);
    NeoPixel_Display(80, 20, 0);
    NeoPixel_Display(60, 40, 0);
    NeoPixel_Display(40, 60, 0);
    NeoPixel_Display(20, 80, 0);
    NeoPixel_Display(0, 100, 0);
    NeoPixel_Display(0, 80, 20);
    NeoPixel_Display(0, 60, 40);
    NeoPixel_Display(0, 40, 60);
    NeoPixel_Display(0, 20, 80);
    NeoPixel_Display(0, 0,100);
    NeoPixel_Display(16, 0, 84);
    NeoPixel_Display(32, 0, 68);
    NeoPixel_Display(48, 0, 52);
    NeoPixel_Display(64, 0, 36);
    NeoPixel_Display(80, 0, 20);
    Wait(100);
    }
```

NeoPixel1.C

Use the push buttons to vary the color

- RB5: Red gets brighter (+1)
- RB4: Red gets dimmer (-1)
- RB3: Green gets brighter (+1)
- RB2: Green gets dimmer (-1)
- RB1: Blue gets brighter (+1)
- RB0: Blye gets dimmer (-1)

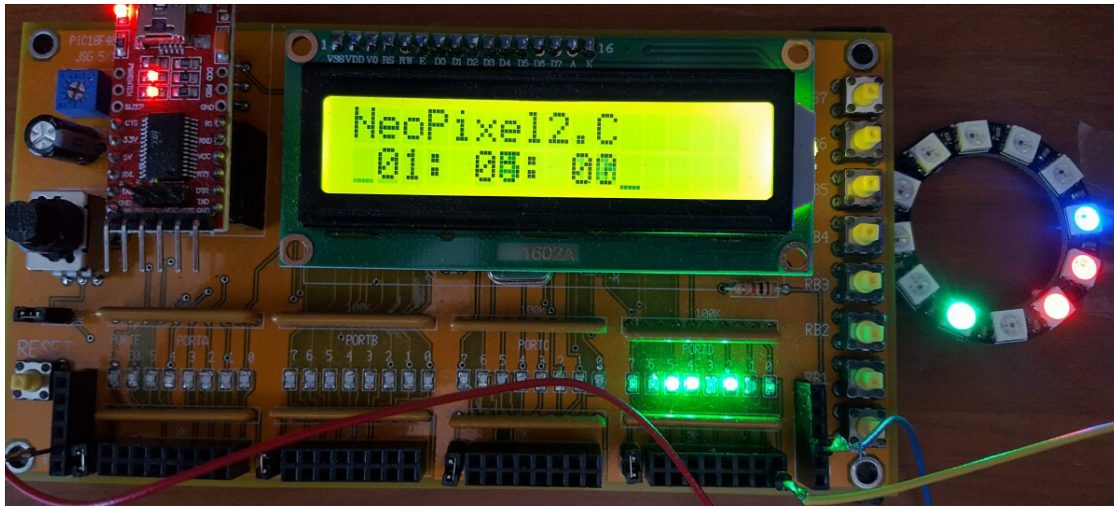
Code:

```
RED = 0;
GREEN = 0;
BLUE = 0;

while(1) {
   if(RB5) RED += 1;
   if(RB4) RED -= 1;
   if(RB3) GREEN += 1;
   if(RB2) GREEN -= 1;
   if(RB1) BLUE += 1;
   if(RB0) BLUE -= 1;
   LCD_Move(1,0);   LCD_Out(RED, 3, 0);
   LCD_Move(6,0);   LCD_Out(GREEN, 3, 0);
   LCD_Move(11,0);  LCD_Out(BLUE, 3, 0);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   NeoPixel_Display(RED, GREEN, BLUE);
   Wait(50);
   }
```

NeoPixel2.C

For more complicated displays, you can use an array. For example, suppose you want to have one light turn on a 16-segment NeoPixel display with the color spinning around the display Red spins around every second

- When red makes one rotation, green steps one
- When green makes one rotation, blue steps one



NeoPixel2.C The red light spins around and around.
Each cycle for red, green rotates one step. Each cycle for greem blue rotates one step.

To do this, create global variables that are arrays

```
// Global Variables

unsigned char PIXEL @ 0x000;
const unsigned char MSG0[20] = "NeoPixel2.C ";


unsigned char RED[16];
unsigned char GREEN[16];
unsigned char BLUE[16];
```

Define a subroutine which is passed which of the 16 lights is to be turned on. All elements of each array should be off (0) except for the one passed to the display routine:

```
void Update_RGB(char r, char g, char b)
{
   unsigned char i;
   for (i=0; i<16; i++) {
      RED[i] = 0;
      GREEN[i] = 0;
      BLUE[i] = 0;
      }
   RED[r] = 50;
   GREEN[g] = 50;
   BLUE[b] = 50;
   }
```

Change the display routine so that it sends out all 16 RGB colors to the NeoPixel

```
void NeoPixel_Display(void)

{
   PIXEL = GREEN[0]; asm(" call Pixel_8 ");
   PIXEL = RED[0]; asm(" call Pixel_8 ");
   PIXEL = BLUE[0]; asm(" call Pixel_8 ");

   PIXEL = GREEN[1]; asm(" call Pixel_8 ");
   PIXEL = RED[1]; asm(" call Pixel_8 ");
   PIXEL = BLUE[1]; asm(" call Pixel_8 ");

   (etc)
   }
```

Finally, set up the main routine so that it spins the colors at the desired rate:

```
while(1) {

   C1 = (C1 + 1) % 16;

   if (C1 == 0) {
      C16 = (C16 + 1) % 16;
      if (C16 == 0) {
         C256 = (C256 + 1) % 16;
         }
      }

   LCD_Move(1,0);
   LCD_Out(C256, 0, 2);
   LCD_Write(':');
   LCD_Out(C16, 0, 2);
   LCD_Write(':');
   LCD_Out(C1, 0, 2);

   Update_RGB(C1, C16, C256);
   NeoPixel_Display();
   Wait(62);
   }
```

Final Results: NeoPixel2.C compiles into 2468 bytes (1234 lines of assembler) That's a lot more assembler than I would like to debug

That's also only 3.8% of program memory. A PIC can do a lot more.

```
Memory Summary:
Program space used 9A4h ( 2468) of 10000h bytes ( 3.8%)
Data space used 4Bh ( 75) of F80h bytes ( 1.9%)
EEPROM space used 0h ( 0) of 400h bytes ( 0.0%)
ID Location space used 0h ( 0) of 8h nibbles ( 0.0%)
Configuration bits used 0h ( 0) of 7h words ( 0.0%)
```