el308.s explained

.bss → assemble statements onto the end of the uninitialized data (.bss) section.

```
.bss
    B1: .space 4   ; 4 bytes
    B2: .space 1   ; 1 byte → avoid! Brings data alignment to odd addr. boundary
```

.section .const, psv → places values to the PSV space in data memory

```
line1: .ascii "PIC 24 Board v1.0"   ; use even # of bytes!
       .byte  0x45, 15, "A"
       .word  0xFEA0, handle(function1)
```

← this will be discussed in detail later

.text → marks the begining of code

Numbers returned by the keypad interface range from 0 to 15. (or 0000 to 1111). In order to display them on the LCD, they need to be converted to the ASCII code that corresponds to the character representing the digit.

0 → "0" = 48 = 0x30
1 → "1" = 49 = 0x31
⋮
9 → "9" = 57 = 0x39
A → "A" = 65 = 0x41
B → "B" = 66 = 0x42
⋮
F → "F" = 70 = 0x46

The code for A, does not follow the code for 9, hence a simple addition will not do the trick. One solution, then, is to write the following code: (Suppose W0 contains the keypad scan code)

```
        cp.b  W0, #9
        bra   GU, hex_digit
        add.b W0, #0x30, W0
        bra   done
hex_digit:
        mov.b #55, W1
        add.b W0, W1, W0
done:
```

This code would convert the number in W0 to its ASCII representation.

This code has two disadvantages:

① Its execution time changes with the value in W0.

② Its long compared to a "table lookup".

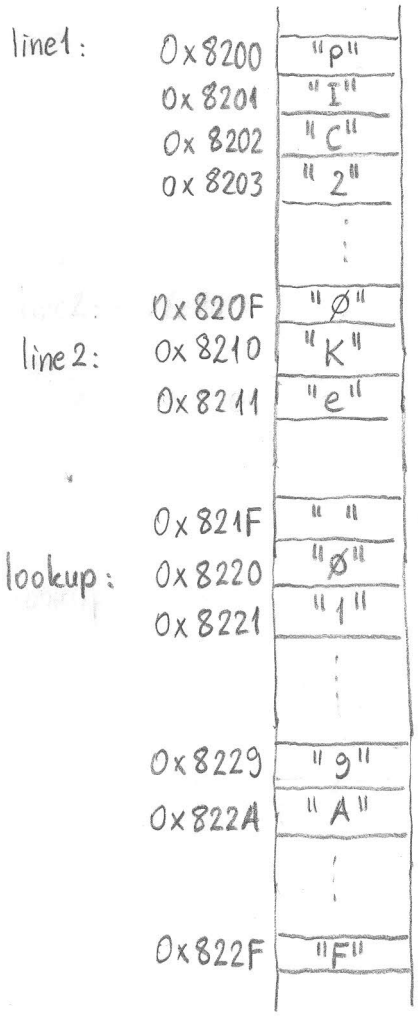
The second item above suggests that there is an alternative solution to the problem. As it will be explained, the solution involves a lookup table for the conversion, and will be quite neat:

The disassembly listing of el308.s reveals the following fact:

mov #psvoffset(lookup), W1 → mov.w #0x8220, W1

hence, the array "lookup" appears at address 8220h in data memory (this is accomplished by the "Program Space Visibility" feature of the microcontroller, Constants declared in the .section .const, psv are mapped to the lower half of the data memory, i.e. addresses larger than 0x8000).

A similar analysis reveals that psvoffset(line1) = 0x8200, where line1 is the first label appearing in PSV. Consequently,



→ space " " = 0x20 = 32, last char of line2

⚠ "p" means "ASCII code of P", a convenience provided by the assembler and linker

Now, let us trace the lines in the code section of e1308.s :

```
mov PORTD, W0   ;  → W0 gets keypad scan code from the port
mov #0x0F, W1
and W0, W1, W0  ; unmask unrelated bits. Now, 0 ≤ W0 ≤ 15
mov #psvoffset(lookup), W1   ; W1 ← 0x8220, a pointer to
                             ; the first byte of the lookup table
mov.b [W0+W1], W0   ; the sum W0+W1 points to the relevant
                    ; entry in the lookup table
```

Suppose W0 = 0, i.e., the user has pressed zero on the keypad. In that case, W0 + W1 = 0x8220, hence [W0+W1] points to the memory cell that contains "0". Consequently, W0 gets "0". Likewise, pressing one produces "1", two produces "2" and so on. The result for nine is "9", ten is "A", .. , and fifteen is "F".

Here, W1 is a pointer to mark the starting point of the character array "lookup", while W0 moves us within the array, because of which it is called an OFFSET. This is somewhat similar to the index used in referring elements of arrays in high level languages, e.g. inC

```
char lookup[10] = "0123456789ABCDEF";
```

would declare the array, while elements might be referred to as

```
lookup[i]
```

where i is an integer that would assume the role of W0 in the assembly language version of the code

Once we have the correct character code in W0, it is time to display it. As mentioned in class, LCD interfacing is not trivial. As a convenience, you are provided with a simplified display interface. The arrays (in .bss)

```
LCD_line1 : .space 16
LCD-line2 : .space 16
```

are declared as buffers for the LCD display, which is capable of displaying two lines of text, each of which has 16 characters. Whatever data is written to those arrays are transferred to the LCD by the T1Interrupt routine. The mechanism will be described later.

Consequently, to display the keypad scan code at the lower right corner of the LCD, one needs to refer to the last element in LCD_line2.

The line

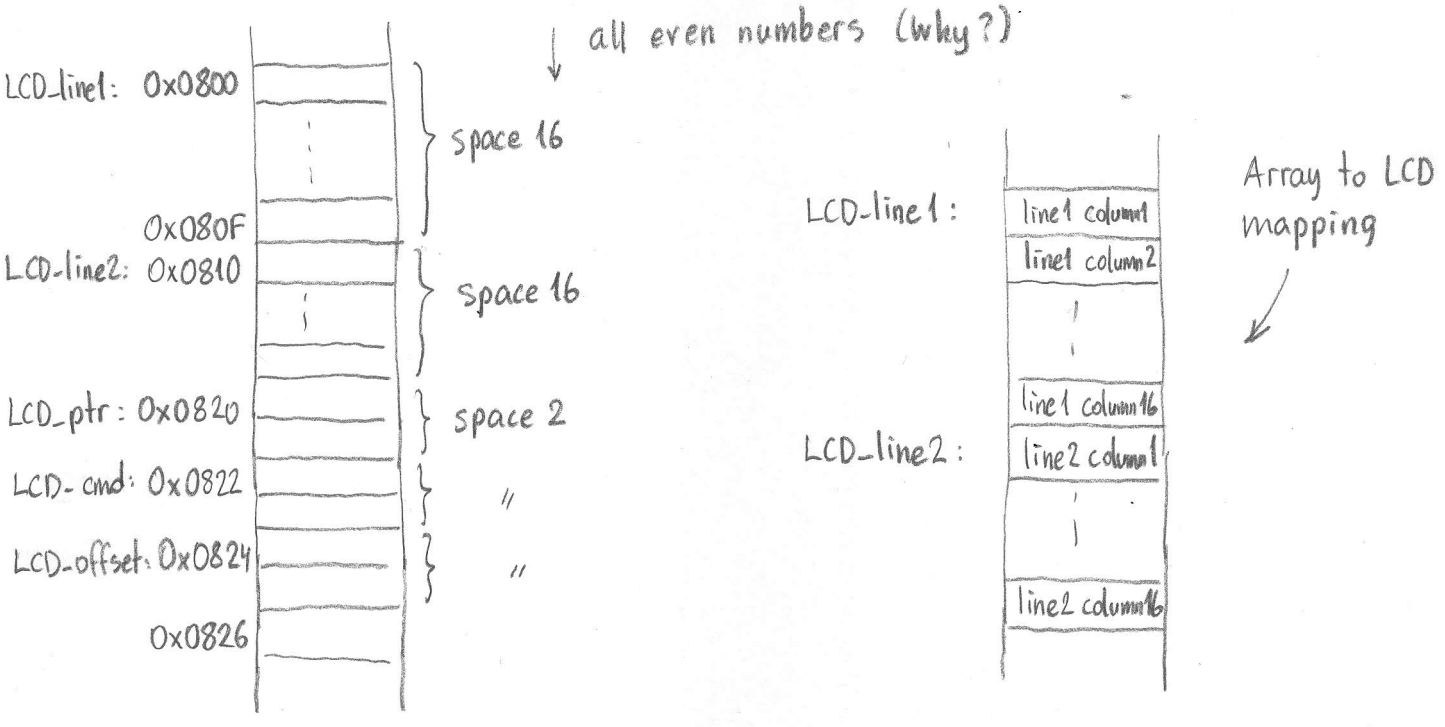
```
mov  # LCD_line2, W1
```

returns a pointer the first element of LCD_line2, hence the pound (#) sign acts as the "Address of" operator (& or ampersand) in C.

The disassembly listing of the above line is

```
mov.w  # 0x0810, W1
```

which is no surprise as the user RAM area starts at 0x0800, so



Hence, [W1 + 15] points to 0x081F, which is the last byte in LCD_line2.

Also note that

```
mov  # SLSP_init, W15   ⟶   mov.w  # 0x0826, W15
```

i.e. the rest of user RAM is used as stack.