
C Start-Up Module and Simple Digital I/O

Lecture 7

In these notes . . .

C Start-Up Module

- Why is it needed?
- How is it done?
- MCPM, section 2.2

Simple Digital I/O

- Port
 - Data Direction
 - Data
 - Drive Capacity
- Use
 - Initialization
 - Reading
 - Writing
- Example
 - Echo LEDs



C Start-Up Module – Why?

MCU must be configured to run in correct mode

- Are memory wait states needed?

Program sections must be allocated to specific parts of memory

- Code, data, interrupt vectors

Some C constructs must be initialized *before* the program starts running

- Where does the stack start?
- What about initialized static variables?

Other C constructs are created or managed “on the fly”

- Building a stack frame (activation record) to call a subroutine
- Dynamically allocating blocks of memory
- *Don't have to do anything about these*

Main() function must be called

C Start-Up Module Functions – ncrt0.a30

Include sect30.inc – provides supporting information

Initialize stack pointer to top of stack area

Configure MCU to use external clock without frequency division

Load pointer for relocatable (variable) interrupt vector table

Zero out bss section

- use a macro to write zeroes

Initialize data section

- use a macro to copy data

Initialize heap, if it exists

Call main as a subroutine

Execute infinite loop if main ends and control returns here

sect30.inc

Define sizes and macros

- heap size
- user stack size (if RTOS used)
- interrupt stack size
- interrupt vector address
- define macros for
 - zeroing data (N_BZERO)
 - copying data (N_BCOPY)
 - initializing heap control variables (HEAPINIT)
 - special page interrupt vectors

Arrange memory (sub)sections

- SBDATA (static base data)
 - data_SE, bss_SE
 - data_SO, bss_SO

– Near RAM

- data_NE, bss_NE
- data_NO, bss_NO
- stack
- interrupt stack
- heap

– Near ROM

- rom_NE, rom_NO

– Far ROM

- rom_FE, rom_FO
- data_SEI, data_SOI, data_NEI, data_NOI, data_FEI, data_FOI
- switch table

Define interrupt vectors

- variable vectors
- fixed vectors

Setting Stack Size in Sect30.inc

```
STACKSIZE .equ 0h
```

```
ISTACKSIZE .equ 80h
```

Stack is used for calling subroutines, handling interrupts, and temporary local storage

Two stack pointers available: USP and ISP

- Which pointer is active depends upon the stack pointer select bit (“U”) in the flag register. See M16C Software Manual, p. 5
- U is cleared on reset, so ISP is selected initially
- USP is used if an RTOS (real-time operating system) is present
- We will use ISP only

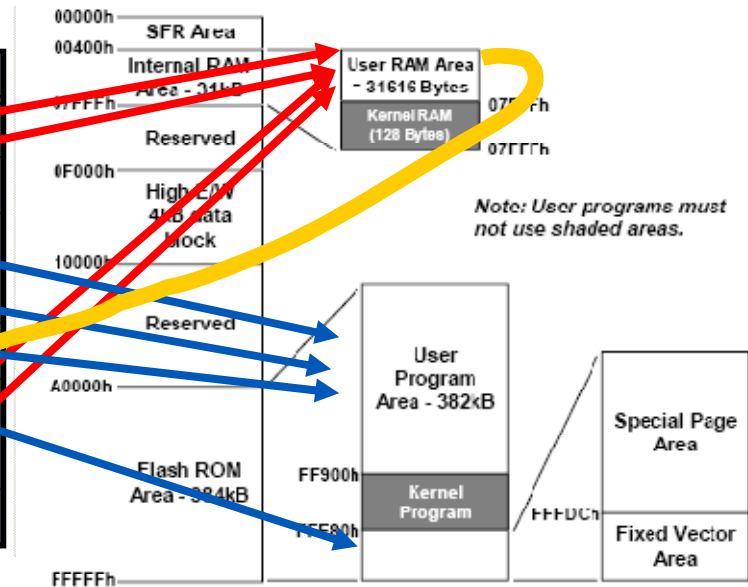
How big of a stack do we need? Coming up later...

- Too small and the program will crash as stack overwrites static variables, and static variables overwrite stack frame
- Too large and we waste RAM

Allocating Sections to RAM or ROM

Table 2.1.1 Sections types Managed by NC30

Section base name	Content
data	Contains static variables with initial values.
bss	Contains static variables without initial values.
rom	Contains character strings and constants.
program	Contains programs.
vector	Variable vector area (compiler does not generate)
fvector	Fixed vector area (compiler does not generate)
stack	Stack area (compiler does not generate)
heap	Heap area (compiler does not generate)



Note: User programs must not use shaded areas.

data_SE
bss_SE
data_SO
bss_SO
data_NE
bss_NE
data_NO
bss_NO
stack
istack
heap

data and bss sections are subdivided for efficiency at run-time

- Addressable from static base register? (within 64K of SB?)
 - yes, put in S subsection
 - no, if address in range 0-0FFFFh put in N subsection (near)
 - no, else put in F subsection (far)
- Even data size? (can ensure all accesses are aligned with bus)
 - yes, put in E subsection
 - no, put in O subsection

Also have I subsection (in ROM) for initial data values

Directives for Arranging Sections

sect30.inc defines where each section goes in memory

Section directive has two formats

- Allocate section to a memory area – *this version is used here*
 - `.section name, area_type`
`[CODE|ROMDATA|DATA],`
`[ALIGN]`
- Place the following code in a certain section
 - `.section name`

```
; Near RAM data area
; SBADATA area
        .section data_SE,DATA
        .org      400H
data_SE_top:
        .section bss_SE,DATA,ALIGN
bss_SE_top:
        .section data_SO,DATA
data_SO_top:
        .section bss_SO,DATA
bss_SO_top:
; near RAM area
        .section data_NE,DATA,ALIGN
data_NE_top:
        .section bss_NE,DATA,ALIGN
. . .
        .blkb     STACKSIZE
stack_top:
        .blkb     ISTACKSIZE
istack_top:
        .section heap,DATA
heap_top:
        .blkb     HEAPSIZE

        .section rom_FE,ROMDATA
        .org      0A0000H
rom_FE_top:

        .section rom_FO,ROMDATA
rom_FO_top:
```

Macros to Initialize Memory Sections – sect30.inc

Define two assembly-language macros

N_BZERO – zero out bytes

- Two arguments
 - **TOP_**: start address of memory section
 - **SECT_**: name of memory section
- Uses instruction **sstr.b** (string store - byte) to store R0L at addresses A1 to A1+R3
- **sizeof** is assembler directive

N_BCOPY

- Three arguments
 - **FROM_**: start address of source data
 - **TO_**: start address of destination
 - **SECT_**: name of memory section
- Uses instruction **smovf.b** (string move forward - byte) to copy R3 bytes of data from address A0 to address A1

```
N_BZERO .macro TOP_ ,SECT_
    mov.b    #00H, R0L
    mov.w    #(TOP_ & 0FFFFH), A1
    mov.w    #sizeof SECT_ , R3
   sstr.b
    .endm
```

```
N_BCOPY .macro FROM_,TO_,SECT_
    mov.w    #(FROM_ & 0FFFFH),A0
    mov.b    #(FROM_ >>16),R1H
    mov.w    #TO_ ,A1
    mov.w    #sizeof SECT_ , R3
    smovf.b
    .endm
```

Initialize Memory Sections – ncr30.a30

ncr30.a30 calls macros defined in sect30.inc

Fill bss sections with zeros

- N_BZERO

Copy initialized data from ROM

- N_BCOPY

```
=====
; Variable area initialize. This code uses
; the macros in "sect30.inc" for initializing
; C variables. Clears global variables, sets
; initialized variables, etc.
;=====
; NEAR area initialize.
;-----
; bss zero clear
;-----
      N_BZERO  bss_SE_top,bss_SE
      N_BZERO  bss_SO_top,bss_SO
      N_BZERO  bss_NE_top,bss_NE
      N_BZERO  bss_NO_top,bss_NO

;-----
; initialize data section
;-----
      N_BCOPY  data_SEI_top,data_SE_top,data_SE
      N_BCOPY  data_SOI_top,data_SO_top,data_SO
      N_BCOPY  data_NEI_top,data_NE_top,data_NE
      N_BCOPY  data_NOI_top,data_NO_top,data_NO
```

Define Fixed Interrupt Vector Table - sect30.inc

First locate it in the fixed vector section (MCPM p.85)

- Hardware expects it to be at 0FFFDCh, so put it there

```
.section fvector  
.org 0FFFDCh
```

Then fill in entries

- insert name of ISR at appropriate vector location

```
RESET:
```

```
.lword start
```

- (Note that labels (e.g. RESET) are for the coder's convenience)
- Use dummy_int for unused interrupts. dummy_int (in ncr0_26skp.a30) immediately returns from the interrupt without doing anything

```
UDI:
```

```
.lword dummy_int
```

```
OVER_FLOW:
```

```
.lword dummy_int
```

Don't delete unused vectors! The hardware expects the vector to be at a **specific address**, so deleting vectors will break the system

M16C also has a variable vector table for other interrupts (examine sect30.inc for details)

Start the main() function – ncrt0.a30

Stack pointer ISP has been set up to point to valid location

Now we just perform a subroutine call to the main function

- no arguments to main for embedded systems

Main should never return

- If it does return, go into infinite loop **_exit** (or could trigger debug mode)

```
=====
; Call main() function
;-----
        .glb          _main
        jsr.a        _main

=====
; exit() function. This function is
; used in case of accidental return
; from main() or debugging code could
; be placed here.
;-----
        .glb          _exit
        .glb          $exit
_exit:          ; End program
$exit:
        jmp          _exit
```



Digital Input/Output (I/O) Ports

The fundamental interfacing subsystem

- Port bits can be inputs or outputs
- M30626 has fourteen *Programmable I/O Ports* (total of 106 digital I/O bits) (P0 to P13 = 8 bits each, P14 = 2)
- For some other MCUs some ports may be limited to only input or output

Direction register sets bit direction

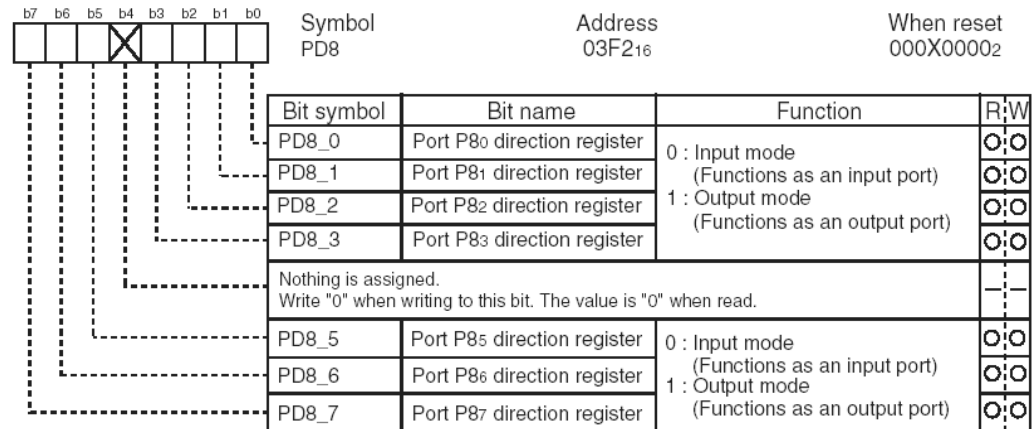
- Port Direction register names PDx
 - 1: Output
 - 0: Input

Data register holds actual data

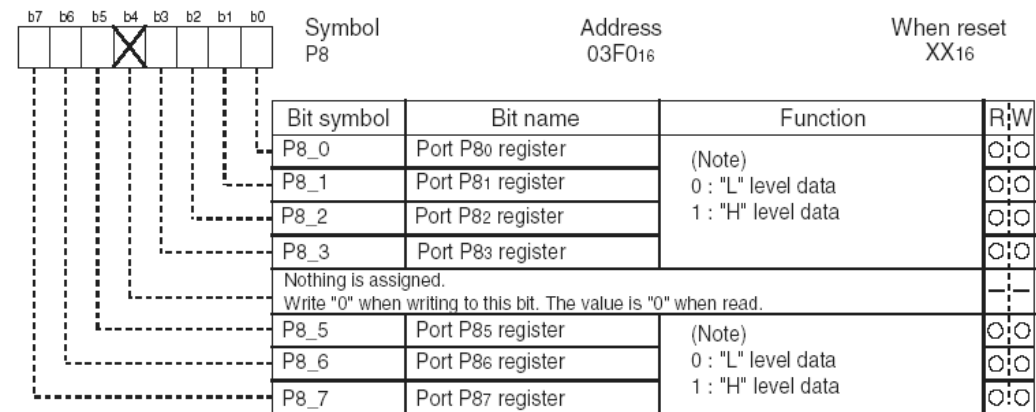
- Port data register names: Px

M16C62P Hardware Manual

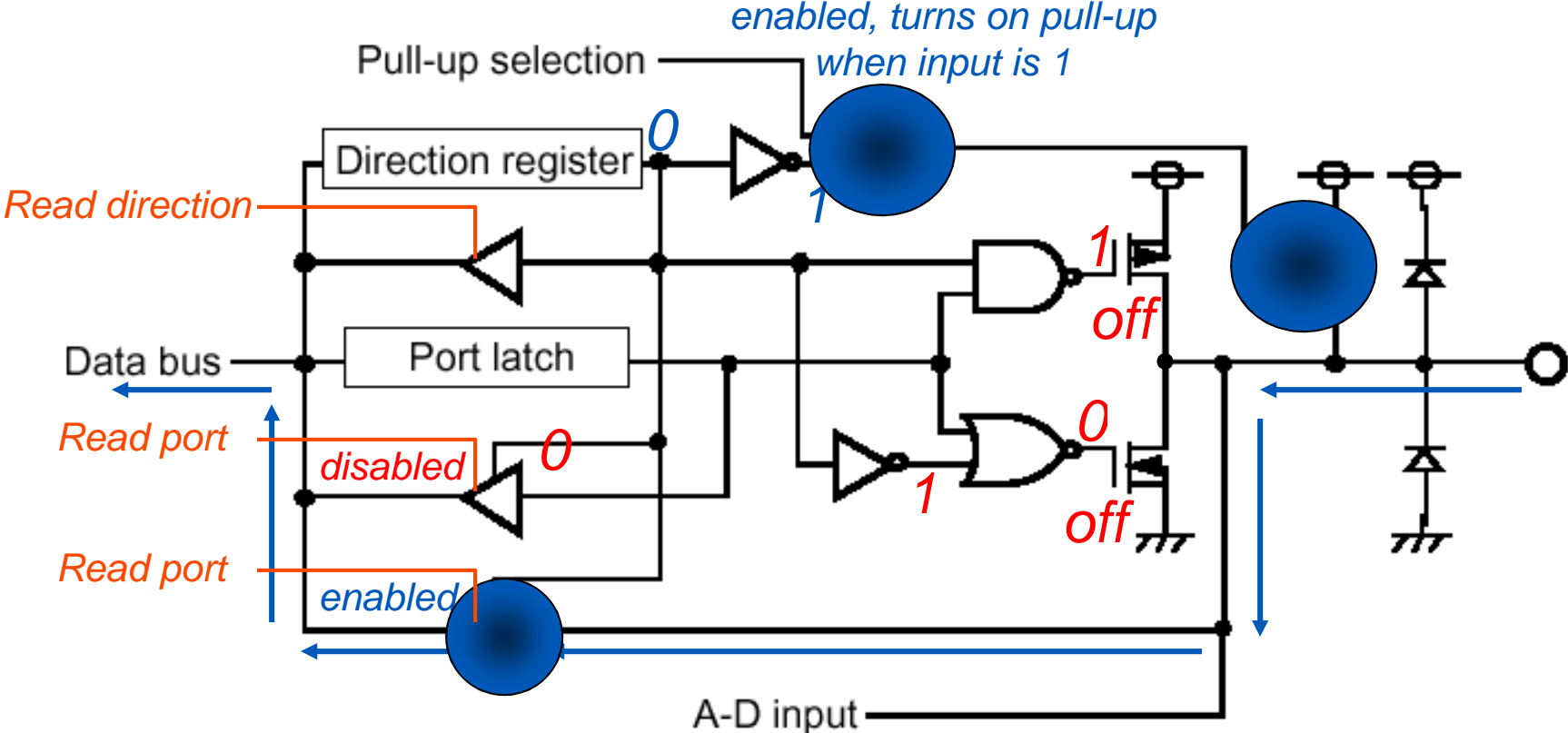
Port P8 direction register



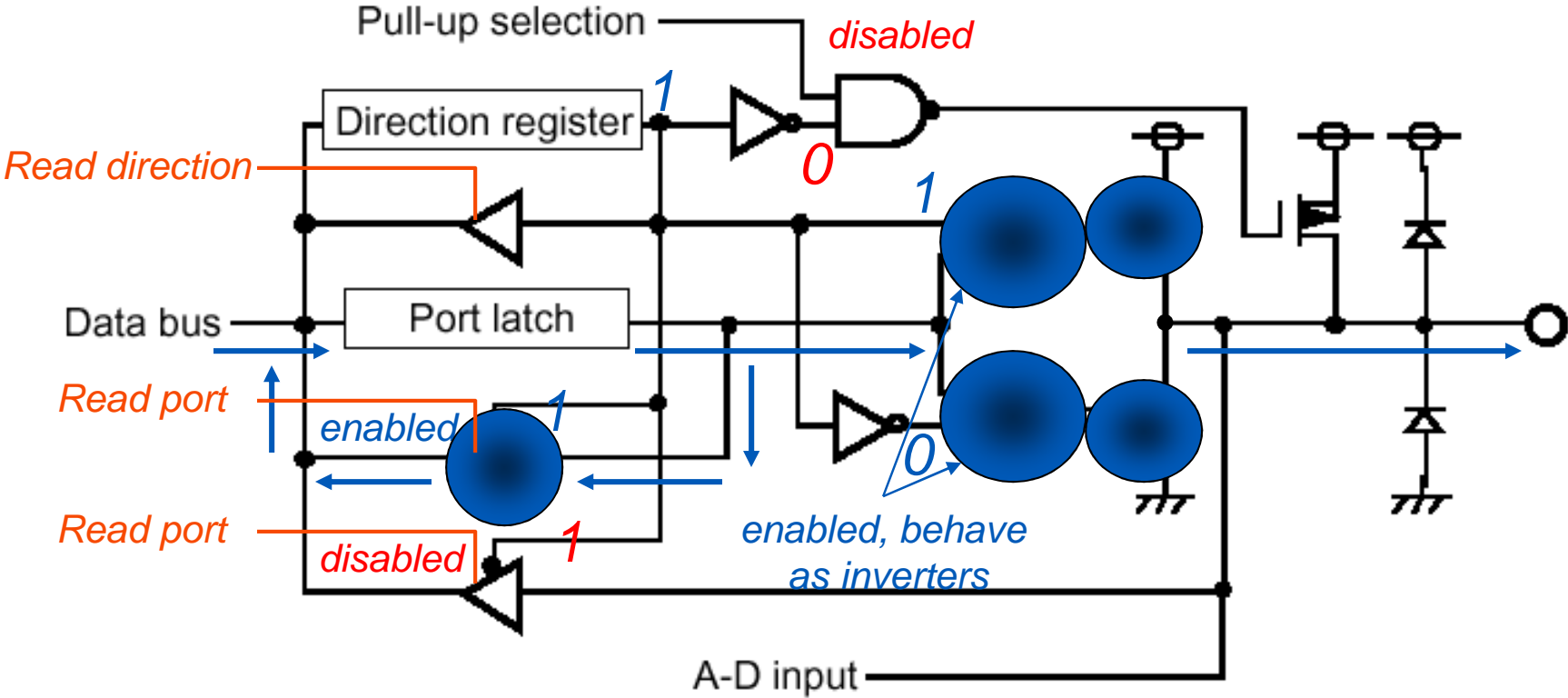
Port P8 register



Digital I/O Port as Input



Digital I/O Port as Output



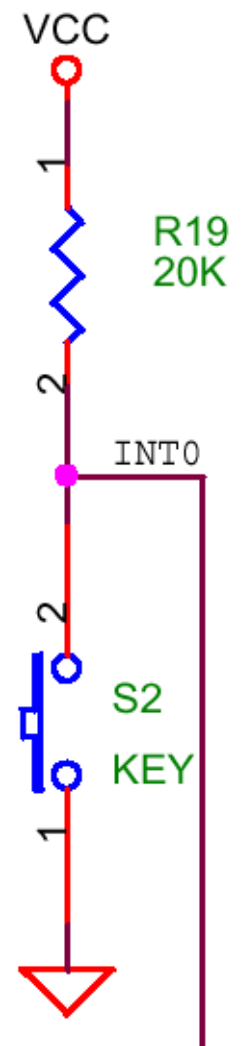
Pull-Up Resistors for Inputs

Used to simplify interfacing with devices with limited signal swing

- M30626 is digital CMOS and is only designed to operate correctly with valid input voltages (*Everything else is illegal and is not guaranteed*)
 - Logic 1: $0.8 * V_{CC}$ to V_{CC} , Logic 0: $0V$ to $0.2 * V_{CC}$
- Resistor is used to pull up voltage of signal from device with two states
 - Low resistance to ground
 - High resistance (essentially open circuit)
- Pull-up resistor is built into microcontroller to simplify circuit design and eliminate external components (save money, size, assembly effort)

M30626 Pull-Up Resistors

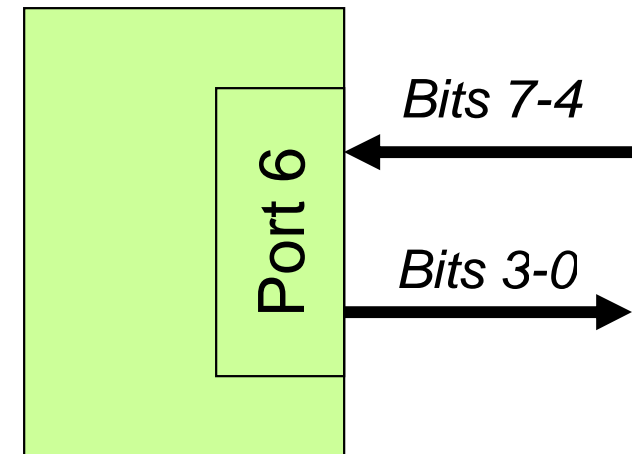
- Controlled in blocks of 4 (upper and lower halves of each port)
- Each block is enabled by a bit in PUR0, PUR1 or PUR2
- Pull-ups disabled if a port bit is configured as an output
- Value typically $120k\Omega$, min $66 k\Omega$, max $500k\Omega$
- *M16C626 Hardware Manual, p. 169*



Example: P6 Echoes Nibble Data

Configuring port to desired structure

- Top 4 bits (4-7) of P6 are inputs
 - Clear bits 4-7 of PD6
- These inputs need pull-up resistors
 - Set bit PU15 of special function register (SFR) PUR1 to enable pull-ups
- Bottom 4 bits of P6 are outputs
 - Set bits 0-3 of PD6



```
Init:  or.b  #PU15, PUR1
       mov.b #00001111b, PD6
```

```
Loop: mov.b P6, R0      ; read inputs
       shl.b #-4, R0   ; move bits 7-4 into 3-0
       mov.b R0, P6    ; write outputs
       jmp  Loop       ; repeat forever
```

Do I really have to write assembly code?

No! Can instead use C variables, structs and unions to access SFRs and their bits as needed

How to access a byte-wide I/O register P6 (at 003ecH) and still be able to access bits cleanly

- Define **structure** called `bit_def` which holds eight bits
- Define **union** called `byte_def` which provides bit and byte views *of the same data*
- Declare a data structure variable of the type `byte_def`
union `byte_def p6_addr;`
- Tell the compiler that `p6_addr` must be located at address 0x03ec, **#pragma** `ADDRESS p6_addr 0x03ec`
- Can now access the port easily
 - byte: `p6_addr.byte = 31;`
 - individual bits: `p6_addr.bit.b3 = 1;`

Renesas has done all the hard work: `sfr62p.h` in `MTools\SKP16C62P\Sample_Code\Common` defines names for all the SFRs and relevant bits

Struct allocates space for each field

```
struct bit_def {
    char    b0:1;
    char    b1:1;
    char    b2:1;
    char    b3:1;
    char    b4:1;
    char    b5:1;
    char    b6:1;
    char    b7:1;
};
union byte_def{
    struct bit_def bit;
    char    byte;
};
```

*Union defines different field names for **same space***

What about..

... that #pragma?

- Pragma directives tell the compiler to do something special (if it is pragmatic or practical)
- Typically used to extend C, or allow better control of what compiler does with your C code
- #pragma ADDRESS
 - specifies absolute address of variable
 - MCPM p. 94 has a description

... things like “p6_0” in the Renesas example code?

- Renesas went a step further and put in shortcuts to access the byte and bits more easily
 - `#define p6 p6_addr.byte /* Port P6 entire byte */`
 - `#define p6_0 p6_addr.bit.b0 /* Port P6 bit0 */`
 - `#define p6_1 p6_addr.bit.b1 /* Port P6 bit1 */`
- Now you see why they chose p6_addr as the name of the data structure
- If you do this kind of thing it's easy to get confused, leading to buggy code or code which the compiler rejects

Example in C: P6 Echoes Nibble Data

Reading and writing data

- Load data from input port
- Move top nibble to bottom
- Write data to output port
- Jump back to start

Now let's invert the data before writing it out

- Load data from input port
- Move top nibble to bottom
- Invert it (complement)
- Write data to output port
- Jump back to start

```
#include "sfr62p.h"
#define DIR_OUT (1)
#define DIR_IN (0)
unsigned char a;
pu15 = 1;
/* pd6 = 0xf0; */
pd6_0 = pd6_1 = DIR_OUT;
pd6_2 = pd6_3 = DIR_OUT;
pd6_4 = pd6_5 = DIR_IN;
pd6_6 = pd6_7 = DIR_IN;
while (1) {
    a = p6;
    a >>= 4;
    p6 = a;
}
```

`a = ~a;`



C Definitions for our Two Nibble Ports

Output nibble comes first
(bits 0-3)

Input nibble comes second
(bits 4-7)

Don't need a union unless
we want a different view of
same data

```
struct nibble_port_T {
    char Out:4;
    char In:4;
};

struct nibble_port_T my_ports;
...
void main() {
    char a;
    ...
    while (1) {
        a = my_ports.In;
        my_ports.Out = a;
        /* my_ports.Out = ~a; */
    }
}
```

Sample Code from Demo

```
#include "stdio.h"    /* sprintf */
#include "sfr62p.h"
#include "LCD.h"
#include "string.h"

#define LED0 (p8_0) /* LEDs */
#define LED1 (p7_4)
#define LED2 (p7_2)

#define LED_ON (0) /* 0 is ON for LEDs */
#define LED_OFF (1)
#define DIR_IN (0)
#define DIR_OUT (1)

#define SW1 (p8_3) /* Switches */
#define SW2 (p8_1)
#define SW3 (p8_2)

void init_switches() {
    pd8_1 = pd8_2 = pd8_3 = DIR_IN;
}

void init_LEDs() {
    pd8_0 = pd7_4 = pd7_2 = DIR_OUT;
    LED0 = LED1 = LED2 = LED_ON;
    LED0 = LED1 = LED2 = LED_OFF;
}

void test_switches(void) {
    while (1) {
        LED0 = (!SW1)? LED_ON : LED_OFF;
        LED1 = (!SW2)? LED_ON : LED_OFF;
        LED2 = (!SW3)? LED_ON : LED_OFF;
    }
}

void main () {
    char buf[9];
    long int i, r=12345;

    MCUInit();
    init_switches();
    init_LEDs();
    InitDisplay("\tSample \n");

    test_switches();
}
}
```



Sample Code from Demo

```
#include "stdio.h"    /* sprintf */
#include "sfr62p.h"
#include "LCD.h"
#include "string.h"

#define LED0 (p8_0) /* LEDs */
#define LED1 (p7_4)
#define LED2 (p7_2)

#define LED_ON (0) /* 0 is ON for LEDs */
#define LED_OFF (1)
#define DIR_IN (0)
#define DIR_OUT (1)

#define SW1 (p8_3) /* Switches */
#define SW2 (p8_1)
#define SW3 (p8_2)

void init_switches() {
    pd8_1 = pd8_2 = pd8_3 = DIR_IN;
}

void init_LEDs() {
    pd8_0 = pd7_4 = pd7_2 = DIR_OUT;
    LED0 = LED1 = LED2 = LED_ON;
    LED0 = LED1 = LED2 = LED_OFF;
}

void test_switches(void) {
    while (1) {
        LED0 = (!SW1)? LED_ON : LED_OFF;
        LED1 = (!SW2)? LED_ON : LED_OFF;
        LED2 = (!SW3)? LED_ON : LED_OFF;
    }
}

void main () {
    char buf[9];
    long int i, r=12345;

    MCUInit();
    init_switches();
    init_LEDs();
    InitDisplay("\tSample \n");

    // test_switches();

    DisplayString(LCD_LINE1, "Response");
    DisplayString(LCD_LINE2, " Timer ");
    while(1) {
        for (i=0; i<200000+(r%50000); i++)
            ;
        i=0;
        LED0 = LED1 = LED2 = LED_ON;
        while (SW1) i++;
    #if (0)
        sprintf(buf, "%8ld", i);
        DisplayString(LCD_LINE1, buf);
        DisplayString(LCD_LINE2, "iters. ");
    #else
        sprintf(buf, "%8.3f", i*250.1/287674);
        DisplayString(LCD_LINE1, buf);
        DisplayString(LCD_LINE2, "millisec");
    #endif
        LED0 = LED1 = LED2 = LED_OFF;
        r=0;
        while (!SW1) /* wait for sw to come up */
            r++;
    }
}
```

