Microprocessors and Microcontrollers

Embedded C Programming

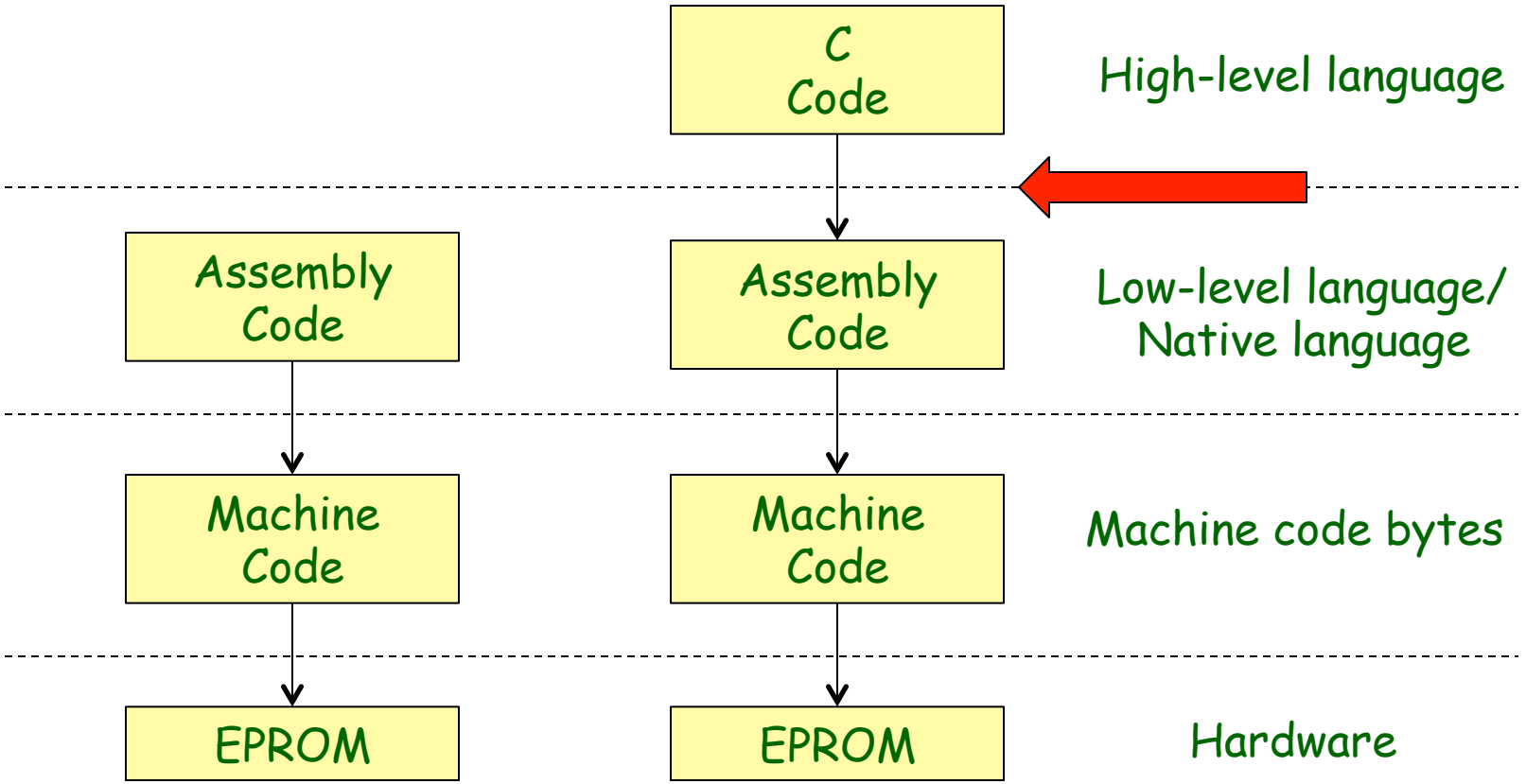
EE3954

by
Maarten Uijt de Haag, Tim Bambeck

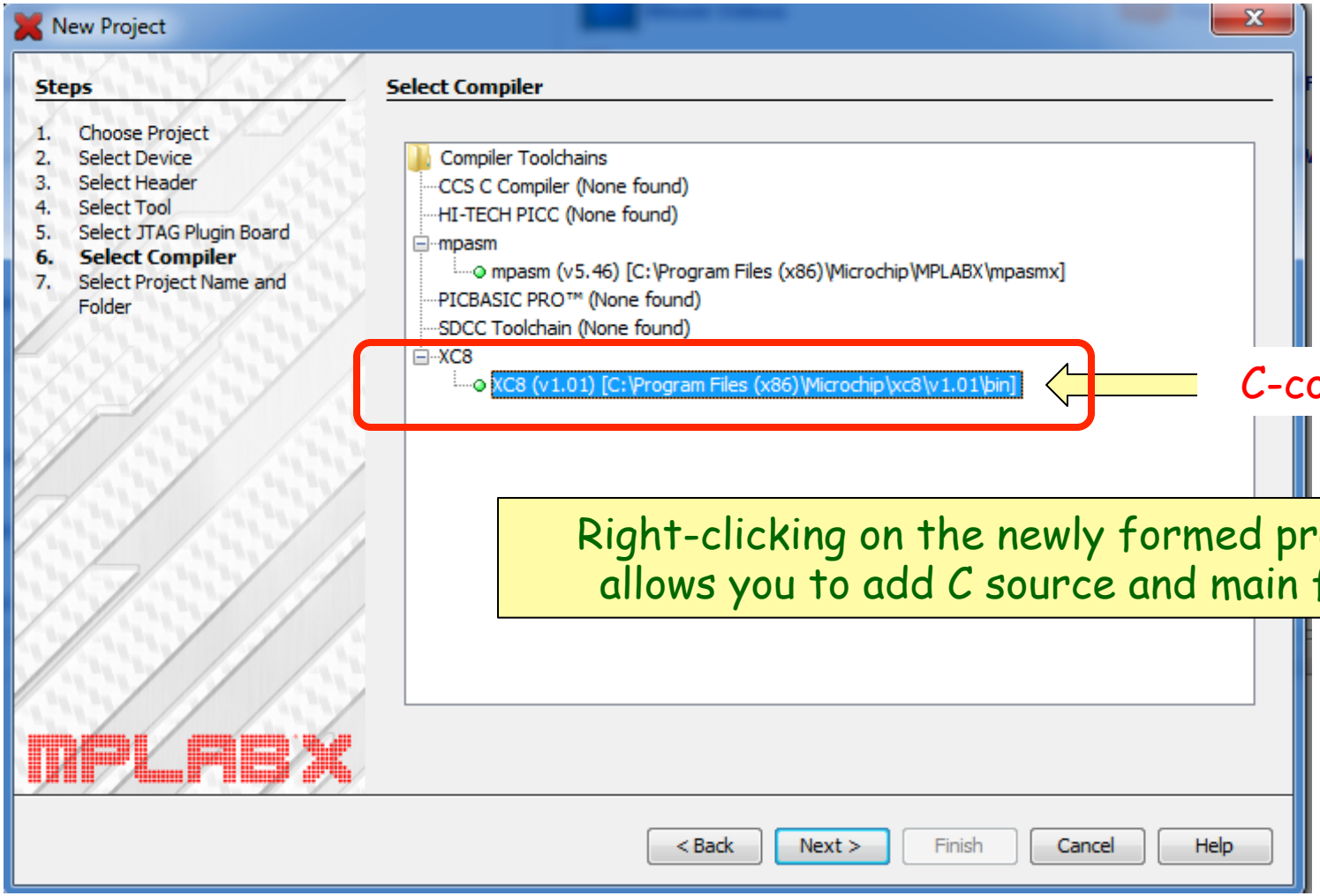
References

- MPLAB® XC8 C Compiler User's Guide

Assembly versus C



Tools

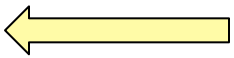


C-compiler tools

Right-clicking on the newly formed project allows you to add C source and main files

Main Template

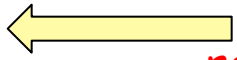
```
/*
 * File:   newmain.c
 * Author: Maarten Uijt de Haag
 *
 * Created on November 6, 2012, 4:21 PM
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *
 */
int main(int argc, char** argv) {

    return (EXIT_SUCCESS);
}
```



Can be replaced by void main(void) as well

Basic Template

```
#include <xc.h>
// Configuration bits
// oscillator frequency for delay
// Global variables
// Functions
// Main Program
void main(void) {
    // C code for the main program
}
```



Defines all registers and pins

Configuration Bits

- Assembly:

```
list p=16f877        ;

__CONFIG   0x3F39
```

- C:

```
/*
 *  Insert your commentary here
 */

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>

// Configuration bits
__CONFIG(0x3F39);

// oscillator frequency for delay
#define _XTAL_FREQ 4000000
```

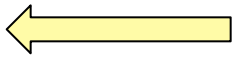
Defines all registers and pins

How to Access SFRs and Pins

Most Special Function Registers (SFRs) and pins are defined in "xc.h" by their official name as specified in the reference manual and datasheet.

For example: PORTA, TRISA, RA0, RA1, RCIF, TXSTA, TXIF, GIE, PEIE, SYNC, TXEN, etc. etc.

```
void main(void) {

        // Configure the I/O ports
        TRISB = 0b11100001;
        TRISD = 0b00000000;
}
```



NO BANK SELECTION REQUIRED

TABLE 5-7: RADIX FORMATS

Radix	Format	Example
binary	0b *number* or 0B *number*	0b10011010
octal	0 *number*	0763
decimal	*number*	129
hexadecimal	0x *number* or 0X *number*	0x2F

Accessing Bits in SFRs

TABLE 11-2: REGISTERS/BITS ASSOCIATED WITH A/D

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on MCLR, WDT
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PS[illegible](1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PS[illegible](1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
1Eh	ADRESH	A/D Result Re[illegible]ter High Byte								xxxx xxxx	uuuu uuuu
9Eh	ADRESL	A/D Result Re[illegible]ter Low Byte								xxxx xxxx	uuuu uuuu
1Fh	ADCON0	ADCS1	AD[illegible]0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	—	ADON	0000 00-0	0000 00-0
9Fh	ADCON1	ADFM	—	—	[illegible]	PCFG3	PCFG2	PCFG1	PCFG0	--0- 0000	--0- 0000
85h	TRISA	[illegible]	—	PORTA Data[illegible]ection Register						--11 1111	--11 1111
05h	PORTA	[illegible]	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	--0u 0000
89h(1)	TRISE	[illegible]	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	0000 -111
09h(1)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	---- -uuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used for A/D conversion.

Note 1: These registers/bits are not available on the 28-pin devices.

How to Access Pins

Multiple options are available to access a PORT pin:

Example PORTA pin 1: RA1

PORTAbits.RA1

```
void main(void) {

        // Configure the I/O ports
        TRISA = 0x00;
        TRISB = 0xFF;

        RA1 = 1;                    <-- Outputs
        PORTAbits.RA2 = 1;          <-- Outputs

Input --> if(RB3 == 1)      RA3 = 1;   <-- Outputs
          else              RA3 = 0;
}
```

Register Usage

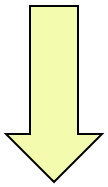
The assembly code generated from the C source code will use certain registers in the PIC MCU register set and assumes that nothing other than code it generates can alter the contents of these registers.

TABLE 5-9: REGISTERS USED BY THE COMPILER

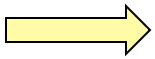
Applicable devices	Register name
All 8-bit devices	W
All 8-bit devices	STATUS
All mid-range devices	PCLATH
All PIC18 devices	PCLATH, PCLATU
Enhanced mid-range and PIC18 devices	BSR
Non-enhanced mid-range devices	FSR
Enhanced mid-range and PIC18 devices	FSR0L, FSR0H, FSR1L, FSR1H
All PIC18 devices	FSR2L, FSR2H
All PIC18 devices	TBLPTRL, TBLPTRH, TBLPTRU, TABLAT
All PIC18 devices	PRODL, PRODH

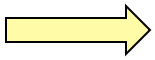
Variables

Only integer arithmetic



Non-standard types





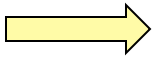
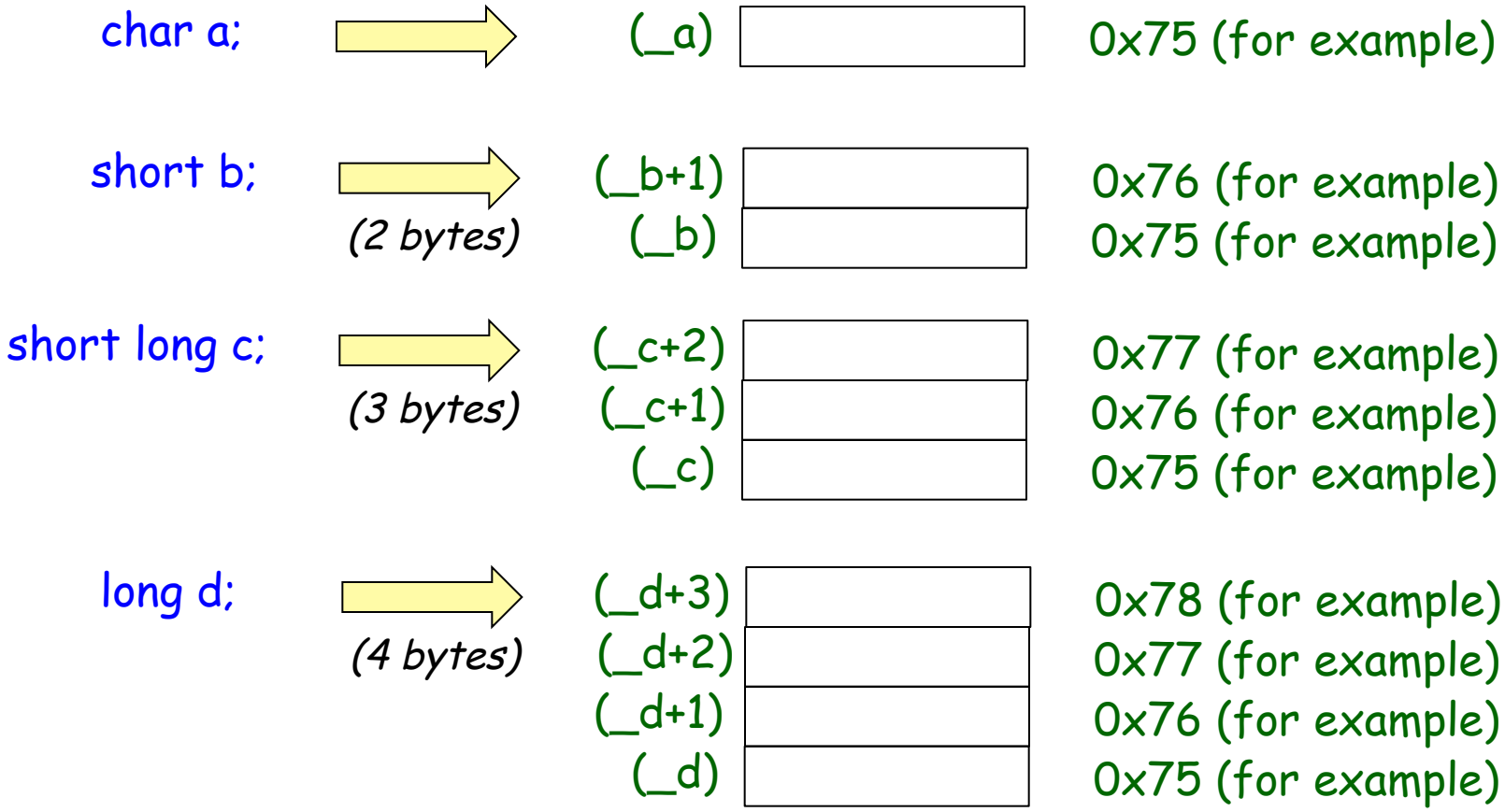


TABLE 5-1: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
`bit`	1	Unsigned integer
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`signed short long`	24	Signed integer
`unsigned short long`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32	Signed integer
`unsigned long long`	32	Unsigned integer

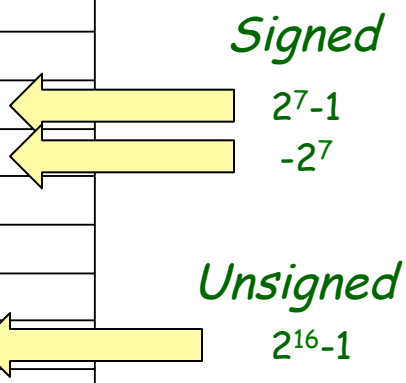
Variables


char a;
(_a)
0x75 (for example)
short b;
(2 bytes)
(_b+1)
(_b)
0x76 (for example)
0x75 (for example)
short long c;
(3 bytes)
(_c+2)
(_c+1)
(_c)
0x77 (for example)
0x76 (for example)
0x75 (for example)
long d;
(4 bytes)
(_d+3)
(_d+2)
(_d+1)
(_d)
0x78 (for example)
0x77 (for example)
0x76 (for example)
0x75 (for example)

Type Ranges

TABLE 5-2: RANGES OF INTEGER TYPE VALUES

Symbol	Meaning	Value
CHAR_BIT	Bits per char	8
CHAR_MAX	Max. value of a char	127
CHAR_MIN	Min. value of a char	-128
SCHAR_MAX	Max. value of a signed char	127
SCHAR_MIN	Min. value of a signed char	-128
UCHAR_MAX	Max. value of an unsigned char	255
SHRT_MAX	Max. value of a short	32767
SHRT_MIN	Min. value of a short	-32768
USHRT_MAX	Max. value of an unsigned short	65535
INT_MAX	Max. value of an int	32767
INT_MIN	Min. value of a int	-32768
UINT_MAX	Max. value of an unsigned int	65535
SHRTLONG_MAX	Max. value of a short long	8388607
SHRTLONG_MIN	Min. value of a short long	-8388608
USHRTLONG_MAX	Max. value of an unsigned short long	16777215
LONG_MAX	Max. value of a long	2147483647
LONG_MIN	Min. value of a long	-2147483648
ULONG_MAX	Max. value of an unsigned long	4294967295
LLONG_MAX	Max. value of a long long	2147483647
LLONG_MIN	Min. value of a long long	-2147483648
ULLONG_MAX	Max. value of an unsigned long long	4294967295



Use of Functions

```
// Functions
char AddTwoNumbersReturnLowByte(int a, int b);

//  Main Program

void main(void) {

        TRISA = 0x00;
        PORTA = AddTwoNumbersReturnLowByte(23456, 12456);

}

char AddTwoNumbersReturnLowByte(int a, int b);
{
        char ret;
        int c;

        c = a + b;
        ret  = (char)(0x00FF & c);

        return(ret);
}
```

16 bits = 2 bytes

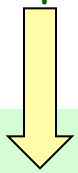
Logic AND operation

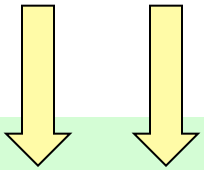
Mask most significant byte

Use of Functions – *Pass by Value*

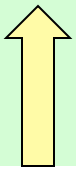
Arguments are values:
Inputs (two bytes each)

Output (byte)



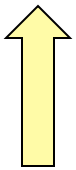


```
char AddTwoNumbersReturnLowByte(int a, int b);
{
        char ret;
        int c;

        c = a + b;
        ret  = (char)(0x00FF & c);

        return(ret);
}
```



Return the output (byte)

"When passing arguments by value, the only way to return a value back to the caller is via the function's return value"



Alternative:
"Pass by reference" and
"Pass by address"

'bit'

bit flag; ⟸ holds the values 0 or 1

```
bit func(void) {

        return(RA0);
}
```

⟸ The 1 or 0 value will be returned in the carry flag in the STATUS register

Eight bit objects are packed into each byte of memory storage, so they don't consume large amounts of internal RAM.

Operations on bit objects are performed using the single bit instructions (bsf and bcf) wherever possible, thus the generated code to access bit objects is very efficient.

Arithmetic Operations

Operator	Description	Example
+	Addition	a = b + c;
-	Subtraction	a = b – c;
+=	Addition and assignment	a += 16;
-=	Subtraction and assignment	b -= 32;
<<	Shift left by 'n' bits	a << 4;
>>	Shift right by 'n' bits	c >> 2;
&	Logical AND	a = b & c;
|	Logical OR	c = a | b;

Time tag example:

Computed Goto's

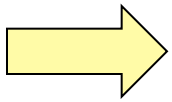
- Remember the computer goto for the 7-segment display:

```
NUM:    addwf   PCL,F           ; 1
        retlw   B'00111111'     ; 2 - return the code for a '0'
        retlw   B'00000110'     ; 2 - return the code for a '1'
        retlw   B'01011011'     ; 2 - return the code for a '2'
        retlw   B'01001111'     ; 2 - return the code for a '3'
        retlw   B'01100110'     ; 2 - return the code for a '4'
        retlw   B'01101101'     ; 2 - return the code for a '5'
        retlw   B'01111100'     ; 2 - return the code for a '6'
        retlw   B'00000111'     ; 2 - return the code for a '7'
        retlw   B'01111111'     ; 2 - return the code for a '8'
        retlw   B'01100111'     ; 2 - return the code for a '9'
```

Computed Goto's

- In C we use a 'switch' statement;
- So, the following function/subroutine would return the 7-segment code for each number (0,…,9)

```
unsigned char Get7SegmentCode(unsigned char num)
{
    switch(num)
    {
      case 0: return(0b00111111);
      case 1: return(0b00000110);
      case 2: return(0b01011011);
      case 3: return(0b01001111);
      case 4: return(0b01100110);
      case 5: return(0b01101101);
      case 6: return(0b01111100);
      case 7: return(0b00000111);
      case 8: return(0b01111111);
      case 9: return(0b01100111);
      default: return(0b00111111);
    }
}
```



With the full version of the XC8 compiler (not free), this code will be optimized to a computed goto.

Timing Loops

- Remember the 1-second delay loop:

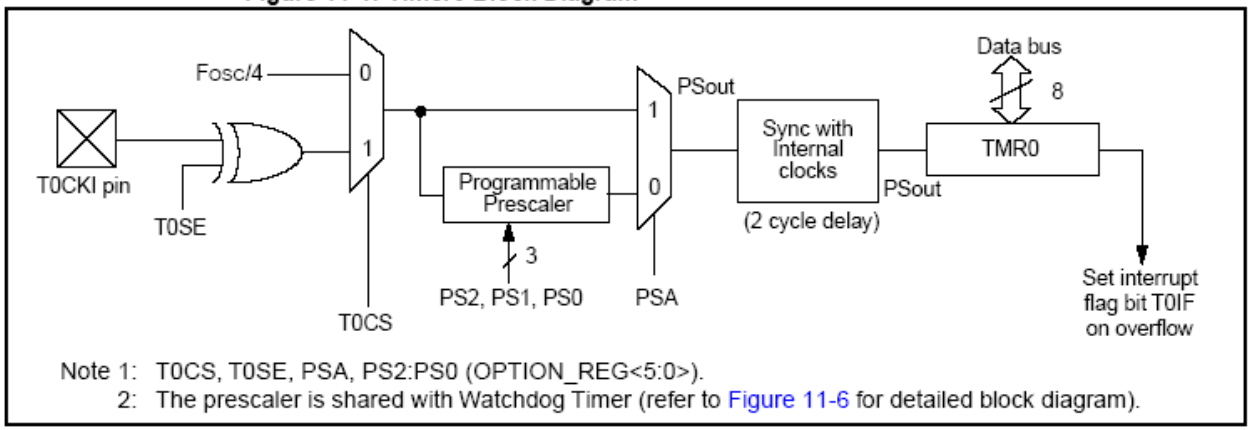
```
DELAY:      movlw   d'167'    ; this is the OUT_CNT starting value.
            movwf   out_cnt   ; store it in OUT_CNT register.
mid_agn     movlw   d'176'    ; this is the MID_CNT starting value.
            movwf   mid_cnt   ; store it in MID_CNT register.
in_agn      movlw   d'10'     ; this is the IN_CNT starting value.
            movwf   in_cnt    ; store it in IN_CNT location.
in_nxt      decfsz  in_cnt,f  ; decrement the inner counter value.
            goto    in_nxt    ; if not zero, go decrement again.
            decfsz  mid_cnt,f           ; decrement the middle counter.
            goto    in_agn    ; if middle cntr is not zero, do inner again.
            decfsz  out_cnt,f; decrement the outer counter.
            goto    mid_agn   ; if outer cntr is not zero, do middle again.
            return            ; if outer counter = 0 then done
```

- Now is:

```
// Delay for 1000ms = 1second
 __delay_ms(1000);
```

Interrupts

Figure 11-1: Timer0 Block Diagram



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 11-6 for detailed block diagram).

- Now in C:

```
// Initialize the option
register bits
PS2     = 1;
PS1     = 0;
PS0     = 1;
PSA     = 0;
T0SE    = 1;
T0CS    = 0;
GIE     = 1;
T0IE    = 1;
```

⟸ Bits defined in 'xc.h' header file

⟹

```
OPTION_REG = 0b11010101;
GIE = 1;
T0IE = 1;
```

Interrupts

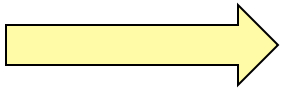
- Can be easily enabled by using enable interrupt function: ei()
- Can be easily disabled by using diable interrupt function: di()

```
ADIE    = 1;        // A/D interrupts will be used
PEIE    = 1;        // peripheral interrupts are enabled
ei();               // enable interrupts
...
...
...
di();               // disable interrupts
```

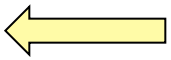
Like setting GIE = 0;

Use of Mixed Code

- Various methods exist for mixing assembly code in with the C source code.

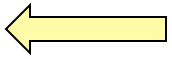


Use #asm and #endasm directives

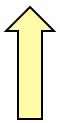


Suppose variable 'var' is at data memory location 0x74 (chosen by C compiler)

```
unsigned int var;

void main(void)
{
        var = 1;
#asm
        BANKSEL(_var)
        RLF             (_var)
        RLF             (_var+1)
#endasm
}
```



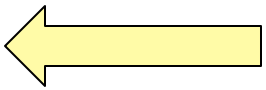
Selects the correct data memory bank for the variable



(_var) = memory location 0x74, (_var+1) is memory location 0x75

Lab #2 - C Version

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>

// Configuration bit
__CONFIG(0x3F39);

// oscillator frequency for delay
#define _XTAL_FREQ 4000000

// Global variables
unsigned char Number;

// Functions
unsigned char Get7SegmentCode(unsigned char);
```



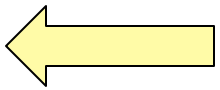
Must be included for the delay to work correctly

Lab #2 - C Version

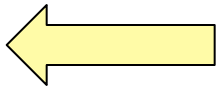
```
// Main loop of the program
int main(int argc, char** argv) {

   // Local variables
   unsigned char NumberCode;

   // Configure the I/O ports
   TRISB = 0b11100001;    // b'11100001';
   TRISD = 0b00000000;

   // Initialize the "Number"
   Number = 0;

   // Select the left-most 7-segment display by setting the corresponding bit to '1'
   RB1 = 1;

   // Infinite loop
   while(1)
   {
      // Get the code for the current nuber
      NumberCode = Get7SegmentCode(Number);

      // Write the number code to teh 7-segment display
      PORTD = NumberCode;

      // Delay for 1000ms = 1second
      __delay_ms(1000);

      // Increment the number
      Number = Number + 1;

      // If teh number equals 10, wrap around
      if(Number > 9) Number = 0;
   }

   return (EXIT_SUCCESS);
}
```

Loops

```
int         ii;
char        array[10];

for(ii=0;ii<10;ii++)
{
            array[ii] = ii;
            ...
            ...
}
```



10-byte character array



Check MPLAB X IDE -> Window -> Output-> Disassembly Listing File

When defining an array of characters, integers, etc., remember the size limitations of the data memory.

For example, int array[100] represents 200 bytes, and will be filing up a large portion of your data memory; actually, the compiler will probably not let you define an array that size.

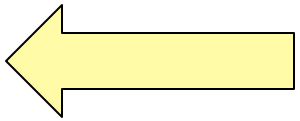
Loops

```
bit flag;

…

flag = determineFlag()

while(flag)
{
        …
        …
        …
        flag = determineFlag();
        …
}
```

```
bit flag;

…

…

do
{
        …
        …
        …
        flag = determineFlag();
        …
} while(flag);
```

Infinite Loop

```
while(1)
{
        ...
        ...
        ...
        ...
        ...
}
```

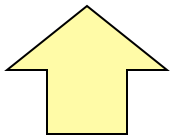


Condition is always true, program within loops keeps getting executed.

User-definable Functions

- Some functions are defined in the library, but are user-definable.
- Example:

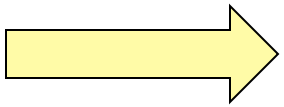
```
printf("\r\n\r\n\tM\tMain Menu\r\n\r\n");
```

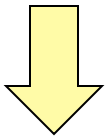


Would print out the first line of the Main menu for lab #5 on a usual program

```
\r      : carriage return (ASCII code 0x0d)
\n      : line feed (ASCII code 0x0a)
\t      : tab (ASCII code 0x09)
```

User-definable Functions

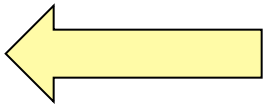
printf  Relies on the user's definition of function putch



In other words, for each character in the string which is in the argument of the printf, the function putch is called.

For example, when using UART serial communication, putch could be:

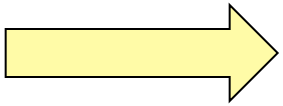
```
void putch(char byte)
{
    while(!TXIF) continue;
    TXREG = byte;
}
```



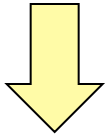
Wait until the TXIF flag is set (= 1)

User-definable Functions

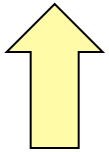
```
void putch(char byte)
{
    while(!TXIF) continue;
    TXREG = byte;
}
```



Can use it by itself now as well



putch(0x0c);



To send a 'clear screen' ASCII character

User-definable Functions

getch* → For getting characters from an I/O device

Empty stub

For example, when using UART serial communication, getch could be:

Wait until the RCIF flag is set (= 1)

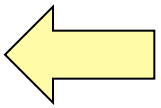
```
char getch()
{
    while(!RCIF) continue;

    return RCREG;
}
```

**could be used with C scanf function.*

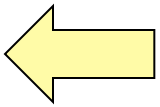
User-definable Functions

Example:

```
char rxbyte;

void interrupt my_isr(void)
{
        if(RCIE && RCIF)
        {
                rxbyte = getch();
        }
}
```



Only for UART receive interrupts

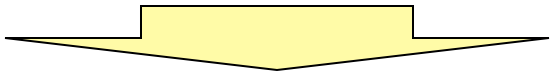


Reading clears the flag

IMPORTANT: ONE SHOULD PERFORM OVERRRUN AND FRAME ERROR CHECK AS WELL FOR ROBUSTNESS

Pointers

printf("Hello PIC programmers");



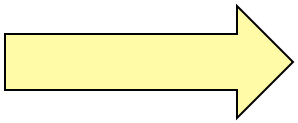
String of characters

Can be defined as well by the following declaration:

const char *str = "Hello PIC programmers";

So, string is stored in program memory

'pointer' to a constant character; str is the program memory address of this string.

So now  printf(str);