Chapter 9 Introduction to PIC18 C Programming

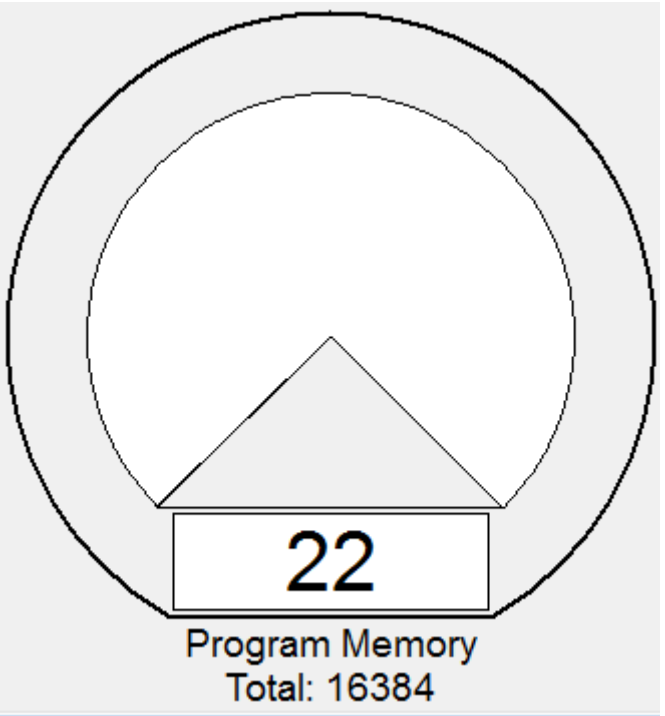
Why program the PIC18 in C?

- C programming is easier and less time consuming
- C is easier to modify and update
- Existing libraries available (e.g., delays)
- C code is portable to other microcontroller with little or no modification
- We will be using Microchip C18 compiler

Example – Delay Program in Lab 1

```
;-------------------------------------------------------------------------------
;Reset vector
;
                ORG     0x0000
                goto    Main                ;go to start of main code

;-------------------------------------------------------------------------------
;Start of main program
;
Main:           movlw   0x0F                ;Set all Ports digital I/O
                movwf   ADCON1
                clrf    TRISD
                clrf    PORTD

MainLoop:       incf    PORTD
                call    Delay
                bra     MainLoop

;---------------------------------------
Delay:          clrf    DELAY_H
                clrf    DELAY_L
DelayLoop:      decfsz  DELAY_L
                bra     DelayLoop
                decfsz  DELAY_H
                bra     DelayLoop
                return

;*******************************************************************************
;End of program
;
                END
```

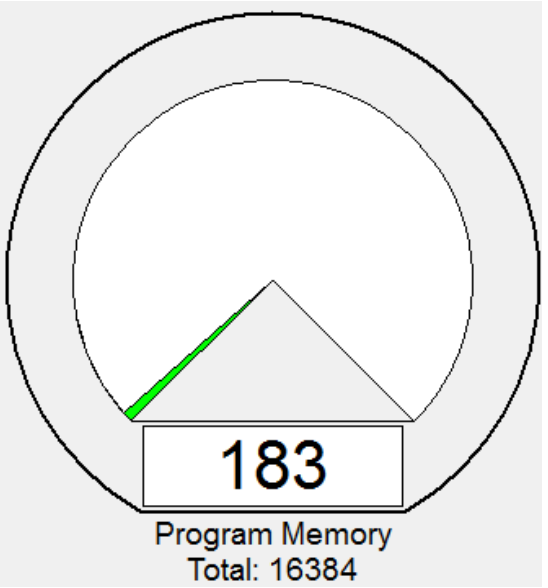
Memory Usage Gauge



Same Program Coded in C

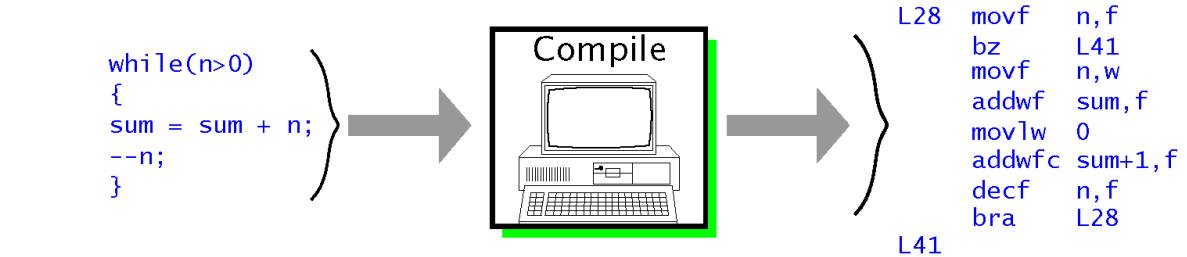
```
void    Delay(int cnt);

//----------------------------------------
void main(void)
{
    ADCON1 = 0x0F;              //Set All Port Digit I/O
    TRISD  = 0b00000000;        //PortD Output
    PORTD = 0;

    while(1) {
        PORTD++;
        Delay(10000);
    }
}

//----------------------------------------
void Delay(int cnt)
{
    unsigned int i;

    for(i=0; i<cnt; i++);
    return;
}
```

Memory Usage Gauge

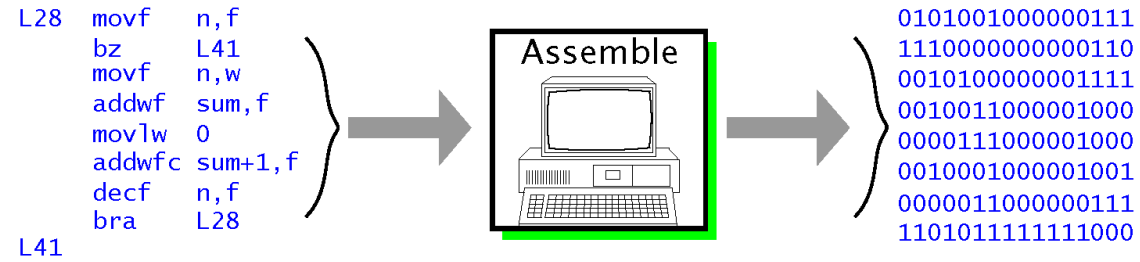


Disadvantages of C

- The code produced is less space-efficient and runs more slowly than native assembly code.
- A compiler is much more expensive than an assembler.



(a) First, compile to assembly-level code.



(b) Second, assemble-link to machine code.

Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

Data Types, Sizes and Ranges

Data Type	Size in Bits	Data Range
unsigned char	8	0 to 255
(signed) char	8	-128 to +127
unsigned int	16	0 to 65535
(signed) int	16	-32768 to 32767

Data Types, Sizes and Ranges

`char`

- Because PIC18 is an 8-bit microcontroller, `char` data type is most commonly used.
- C compliers use the signed type as default unless the qualifier unsigned is put in front.

`int`

- `int` variables are stored in two 8-bit registers.
- Don’t use `int` unless we have to. If one 8-bit register is enough to store a variable, we don’t want to use 2 registers.

Use of C to generate time delay

- Using assembly language, we can control the exact instructions executed in a time delay subroutine and thus be able to control the exact time delay.

```
DelayLoop: decfsz   DELAY_L
           bra DelayLoop
           decfsz  DELAY_H
           bra DelayLoop
```

Use of C to generate time delay

- C compliers convert C statements to assembly instructions.
- Different compliers produce assembly code of different length.
- The actual time delay generated by the following function depends on the complier used.

```
void Delay
{
        unsigned int i;
        for(i=0; i<10000; i++);
        return;
}
```

- Need to measure the exact time delay using MPLAB StopWatch tool.

I/O Programming in C

- Recall: I/O Programming involves PORTx and TRISx registers.
- Byte I/O Programming: Change the whole byte stored in PORTx or TRISx.
- e.g., `PORTB = 0x18`

  `TRISB = 0X20`

I/O Programming in C

```
;---------------------------------------
;Reset vector
;
                ORG     0x0000
                goto    Main

;---------------------------------------
;Start of main program
;
Main:           movlw   0x0F
                movwf   ADCON1
                clrf    TRISD
                clrf    PORTD

MainLoop:       incf    PORTD
                call    Delay
                bra     MainLoop

;.....................................
Delay:          clrf    DELAY_H
                clrf    DELAY_L
DelayLoop:      decfsz  DELAY_L
                bra     DelayLoop
                decfsz  DELAY_H
                bra     DelayLoop
                return

;*************************************
;End of program
;
                END
```

```
void    Delay(int cnt);

//-----------------------------
void main(void)
{
    ADCON1 = 0x0F;
    TRISD  = 0b00000000;
    PORTD = 0;

    while(1) {
        PORTD++;
        Delay(10000);
    }
}

//-----------------------------
void Delay(int cnt)
{
    unsigned int i;

    for(i=0; i<cnt; i++);
    return;
}
```

I/O Programming in C

- Bit-addressable I/O programming: Change a single bit without disturbing the rest of the PORTx or TRISx registers.
- PORTBbits.RB7 = 7th bit of PORTB
- TRISBbits.TRISB7 = 7th bit of TRISB
- Same function as bcf or bsf in assembly language
- e.g., `bcf PORTB, 5` is expressed as `PORTBbits.RB5 = 0` in C
- e.g., `bcf TRISB, 5` is expressed as `TRISBbits.TRISB5 = 0` in C

Logic Operations in C

Bit-wise operators:

1. AND (&)
   - Extract lower nibble: PORTB & 0x0F
2. OR (|)
   - e.g., SPI_VALUE = 0x30 | SPI_HI
3. Exclusive OR (^): 1^1 = 0
4. Inverter (~)
   - e.g., Toggle PORTB: `PORTB = ~PORTB`

`if` statement: Conditional branching

```
if (CONDITION) {
        Statement
}
```

- `CONDITION:` The condition in which `statement` would be executed (if 1, execute `statement`; if 0, skip statement)

`for` loop

```
for(INITIALIZATION; CONDITION; INC/DEC){
                    Statement
}
```

- `INITIALIZATION:` Initialize the “COUNT” variable
- `CONDITION:` The condition in which this loop will continue (if 1, continue; if 0, exit)
- `INC/DEC:` Increment/decrement the “COUNT” variable
- Used when you know how many times the loop should run

An example :

```
//Using for loops to add numbers 1 - 5
unsigned  char sum = 0;
for(int i = 1; i<6; i++)
{ sum += i; }
```

`while` loop

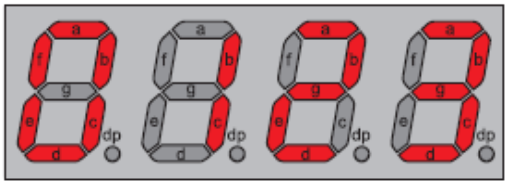
```
while(CONDITION){
        Statement
}
```

- `CONDITION:` The condition in which this loop will continue (if 1, continue; if 0, exit)
- Used when you DO NOT know how many times the loop should run or if the loop should run infinitely many times (use `while(1)`).

Simplicity of C: An Example

Lab 2 Task 2

Task: Display your group number of the 4-digit 7-segment LED.



Remember how much work you have done to make it work in assembly?

Very simple coding in C.

Simplicity of C: An Example

```
void Delay(unsigned int cnt)
{
          while(cnt != 0) {
                    cnt--;
          }
}

void main(void)
{
          unsigned char Segment[10] = {0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07,
0x7f, 0x6f};
          unsigned char DispBuf[4]  = {0, 0, 1, 8};
          unsigned char i;

          ADCON1 = 0x0f;                              //Set All Port Digit I/O
          TRISB  = 0b00000000;                        //Set PortB Output
          TRISD  = 0b00000000;                        //Set PortD Output

          while(1) {
                    for(i=0; i<=3; i++) {
                              PORTD = 0;
                              PORTB = i;
                              PORTD = Segment[DispBuf[i]];
                              Delay(400);
                    }
          }
}
```

C18 Timers Library

- C18 Timers Functions:

Function	Description
OpenTimerx()	Configure and enable timer x.
ReadTimerx()	Read the value of timer x.
WriteTimerx()	Write a value into timer x.
CloseTimerx()	Disable timer x.

To use the Timers library, simply put following statement before use

```
#include <timers.h>
```

Our interrupt program revisited

```
#include <p18F4520.h>
#include <timers.h>
#pragma config  OSC = HS, WDT = OFF, LVP = OFF

void timer_isr_internal(void);
//-----------------------------------------------------------------------------
#pragma code timer_isr = 0x08 // Store the below code at address 0x08

void timer_isr(void)
{
          _asm GOTO timer_isr_internal _endasm   // allowed to write part of your code in ASM.
}

//-----------------------------------------------------------------------------
#pragma code
void main (void)
{
    TRISBbits.RB5 = 0;                                  //set RB5 output
    PORTBbits.RB5 = 0;

    //T0CON = 0x08;                             // Timer0, 16-bit, no prescale, internal ck
    //TMR0H = 0xD8;
    //TMR0L = 0xF0;
    WriteTimer0(0xD8F0);

    RCONbits.IPEN = 0;                          //disable priority levels
    INTCONbits.TMR0IF = 0;
    //INTCONbits.TMR0IE = 1;     // Interrupt enabled by the TIMER_INT_ON option in OpenTimer0
    INTCONbits.GIE = 1;

    OpenTimer0(TIMER_INT_ON & T0_16BIT & T0_SOURCE_INT & T0_PS_1_1); //T0CONbits.TMR0ON = 1;

    while(1);
}
```

Our interrupt program revisited

```
#pragma interrupt timer_isr_internal
void timer_isr_internal (void)
{
      if (INTCONbits.TMR0IF)
      {
            INTCONbits.TMR0IF=0;
            PORTBbits.RB5 = ~PORTBbits.RB5;//toggle
PortB.5 to create sq. wave
            //TMR0H = 0xD8;
      //TMR0L = 0xF0;
      WriteTimer0(0xD8F0);
    }
}
```

interrupt, interruptlow

- `#pragma interrupt fname`
  - `retfie 1` ends the ISR.
  - WREG, BSR and STATUS registers are restored from the shadow registers.
- `#pragma interruptlow fname`
  - `retfie` ends the ISR.
  - WREG, BSR and STATUS registers are restored from temporary registers.

Just like in assembly language

```
        CBLOCK 0x7D
        w_temp, status_temp, bsr_temp
        endc
        org 0x008
         goto  isr_high_priority
        org 0x0018
         goto  isr_low_priority
        org 0x????
isr_high_priority
        ;;; ISR high priority code
        retfie 1   ;; use shadow reg

isr_low_priority
        movwf w_temp          ; context
        movff  STATUS,status_temp
        movff BSR, bsr_temp
        ;;....ISR CODE ...
        ;;.... . . .
        movff bsr_temp,bsr   ; restore context
        movf  w_temp,w
        movff  status_temp,STATUS
        retfie
```

ISR Assembly

space for w, status, bsr

high priority ISR can use shadow registers (fast stack)

low priority must explicitly save the processor context

restore context, do not use shadow registers on ‘retfie’

C18 ADC Library

- C18 ADC Library Functions:

Function	Description
OpenADC()	Configure the A/D convertor.
SetChanADC()	Select A/D channel to be used.
ConvertADC()	Start an A/D conversion.
BusyADC()	Is A/D converter currently performing a conversion?
ReadADC()	Read the results of an A/D conversion.
CloseADC()	Disable the A/D converter.

To use the Timers library, simply put following statement before use

```
#include <adc.h>
```

ADC Example in Assembly

```
Main:     movlw   b'00001110'     ;Set AN0 Analog Port, others Digital I/O
          movwf   ADCON1
          movlw   b'00000001'     ;Select ADC Channel 0, Enable ADC
          movwf   ADCON0
          movlw   b'00010100'     ; ADFM Left justified, ACQT 4TAD,
          movwf   ADCON2          FOSC/4

          clrf TRISD              ; set PORTD output

MainLoop: bsf ADCON0, GO          ; start Conversion
adc_wait: btfsc   ADCON0, GO      ; adc_wait waits for ADC to be done
             bra     adc_wait

          movff ADRESH, PORTD     ;display Top 8 bit

          bra MainLoop
```

Equivalent Operation in C

```
#include <p18f4520.h>
#include <adc.h>

#pragma config OSC=HS, WDT=OFF, LVP=OFF

void main( void )
{
        static int result;

        TRISD = 0x00;

    OpenADC(ADC_LEFT_JUST & ADC_FOSC_4 & ADC_4_TAD,
            ADC_CH0 & ADC_REF_VDD_VSS & ADC_INT_OFF,
            ADC_1ANA);
    while(1)
    {
        ConvertADC();               // Start conversion
        while(BusyADC());           // Wait for completion
        result = ReadADC();
        PORTD = result >> 8;
    }
}
```

I^2C Functions

Function	Description
EEByteWrite	Write a single byte.
EEPageWrite	Write a string of data.
EERandomRead	Read a single byte from an arbitrary address.
EESequentialRead	Read a string of data.
EEAckPolling	Generate acknowledgement polling sequence: Send the control byte repeatedly to test whether the EEPROM has completed the internal reading cycle.

I²C Example

```
#include <p18F4520.h>
#include <i2c.h>
#pragma config  OSC = HS, WDT = OFF, LVP = OFF

#pragma code
void main (void)
{
        unsigned char i, WordAddress;
        unsigned char DigitsToI2C[4]  = {1, 2, 3, 4};
        unsigned char DigitsReadFromI2C[4];
        unsigned int DigitsToI2Cint;
        unsigned char err;

        ADCON1 = 0x0F;

        //TRISCbits.RC3 = TRISCbits.RC4 = 1; Done in OpenI2C

        OpenI2C(MASTER, SLEW_OFF);

        SSPADD = 0x09;
```

I²C Example

```
        WordAddress = 0;

        // Byte Write:
        for (i = 0; i < 4; i++)
        {
                EEByteWrite(0xA0, WordAddress, DigitsToI2C[i]);
                WordAddress++;
                EEAckPolling(0xA0);
        }

        WordAddress = 0;

        // Byte Read:
        for (i = 0; i < 4; i++)
        {
                DigitsToI2Cint = EERandomRead(0xA0, WordAddress);
                //This function return a 16-bit int. The byte read is
stored in the LSB.
                DigitsToI2C[i] = DigitsToI2Cint & 0x00FF;
                WordAddress++;
        }
```

I²C Example

```
        //Page Write:
        EEPageWrite(0xA0, 0x00, DigitsToI2C);
        EEAckPolling(0xA0);

        // Sequential Read:
        EESequentialRead(0xA0, 0x00, DigitsReadFromI2C, 4);

}
```