C LANGUAGE AND PIC18 C COMPILER

PIC Microcontroller: An Introduction to Software & Hardware Interfacing

Han-Way Huang

Thomson Delmar Learning, 2005

Chung-Ping Young
楊中平

Networked Embedded Applications and Technologies Lab

Department of Computer Science and Information Engineering
National Cheng Kung University, TAIWAN

Introduction to C Language

- C language is gradually replacing assembly language in many embedded applications.
- C language allows the user to work on the program logic at a higher level than the assembly language.
- A C program consists of functions and variables.
- A function contains statements that specify the operations to be performed.
- A C statement could be declaration, assignment, function call, control, and null.
- A simple C program is as follows:

```
#include  <stdio.h>
main (void)
{
    int  a, b, c;
    a = 3;
    b = 5;
    c = a + b;
    printf(“ a + b = %d \n”, c);
    return 0;
}
```

Types, Operators, and Expressions

- Variables and constants are the basic objects in C.
- Variables must be declared before they can be used.
- Both the name and the type of the variable must be included in the declaration.
- A variable name may start with a letter (**A** through **Z** or **a** through **z**) or underscore character followed by zero or more letters, digits, or underscore characters.
- The variable name is case-sensitive.

Basic Data Types in C

1. **void**: used most commonly with functions
2. **char**: can hold a single byte of data
3. **int**: integer. The size of type **int** is 16 bits for PIC18.
4. **float:** 32-bit, single-precision, floating-point number
5. **double:** 64-bit double-precision, floating-point number (32-bit for PIC18)

- The PIC18 C compiler also supports the 24-bit **short long** integer data type.
- Several modifiers can be applied to integer type including **short**, **long**, and **unsigned**.
- **Int** and **char** types are signed by default.

Variable Declarations

A declaration specifies a type and contains a list of one or more variables of that type.

```
int   i, j, k;
char  cx, cy;
```

A variable may also be initialized when it is declared:

```
int   i = 0;
char  echo = ‘y’;    /* ASCII of letter y is assigned */
```

Constants

1. integer
2. character
3. floating-point:
4. character: enclosed by single quotes
5. string: enclosed by double quotes

Radix of Numbers

1. decimal: default radix
2. octal: add prefix **0.** e.g., 04321
3. hexadecimal: add prefix **0x.** e.g., 0x3A4B

Arithmetic Operators

+, -, *, /, %, ++, --

Bitwise Operators

&	AND	PORTD = PORTD & 0x7F;
|	OR	PORTD = PORTD | 0x80;
^	XOR	PORTD = PORTD ^ 0x0F;
~	NOT	xyz = ~xyz;
>>	right shift	abc = abc >> 2;
<<	left shift	abc = abc << 4;

Arithmetic and logic operators are often used together with the **=** operator.

For example,

```
abc = abc + 3;          can be written as  abc += 3;
abc = abc >> 4;         can be written as  abc >>= 4;
xy  = xy & 0x7F;        can be written as  xy &= 0x7F;
```

Relational and Logical Operators

==	equal to (two ‘=’ characters)
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
&&	and
||	or
!	not (one’s complement)

In C language,

- A statement is terminated by a colon.
- A compound statement is a group of statements enclosed by braces **{** and **}**.
- A compound statement is not terminated by a semicolon.
- A control-statement specify the order in which computations are performed.

Control Flow Statements

If statement

if (expression)
    statement;

```
if (a > b)
        PORTA |= 0x48;
```

If-else Statement

if (expression)
    statement$_1$;
else
    statement$_2$;

```
if (ax == 0)
        abc = 3;
else
        abc = 5;
```

if-else statement can be replaced by abc = (ax == 0)? 3:5;
the **?:** operator.

Multiway Conditional Statement

```
if (expression_1)
    statement_1;
else if (expression_2)
    statement_2;
else if (expression_3)
    statement_3;
…
else
    statement_n;
```

```
if (k == 1)
    return  2;
else if (k == 4)
    return  5;
else
    return  0;
```

Switch Statement

```
switch (expression) {
    case const_expr_1:
        statement_1;
        break;
    case const_expr_2:
        statement_2;
        break;
    …
    default:
        statement_n;
}
```

```
switch (i) {
    case 1:  set_temp (50);
             break;
    case 2:  set_pressure (30);
             break;
    …
    default:
             set_temp (20);
             set_pressure(28);
             break;
}
```

for-loop Statement

```
for (expr1; expr2; expr3)
        statement;
where,
        expr1 & expr3 are assignments or function calls
        expr2 is a relational statement

Examples
        count1 = 0;
        for (i = 0; i < 30; i++)
                if (arr[i] & 0x03)        /* is arr[i] dividable by 4? */
                        count1 ++;

        count2 = 0;
        for (j = 30; j > 0; j--)
                if ((arr[i] > 5) && (arr[j] < 20))
                        count2++;
```

while statement

```
while (expression)
    statement;
```

```
while (!(ADCON1 & 0x80)); /* null statement */

while (1) {               /* infinite loop */
    …
}
```

do-while statement

do

statement$_x$

while (expression$_y$)

```
i = sum = 0;
do
    sum = sum + i;
    i++;
while (i < 50);
```

goto statement

```
goto  label;
```

The use of **goto** statement is not considered a good programming style.

```
if (temperature > 100)
    goto  alarm;
…

alarm:
    set_alarm();      /* call a function to
                         turn on alarm; */
```

Input and Output

- Not part of C language.
- **ANSI** standard defines a set of library functions for input and output that must be supported by the C compiler. Among them

The **getchar ( )** function returns a character when it is called. The character is received from the serial communication port.

The **putchar (int)** outputs a character on the standard output device.

The **puts (const char *s)** function outputs the string pointed to by **s** to the standard output device.

The **printf (formatting string, arg$_1$, arg$_2$, …, arg$_n$)** function outputs the arguments to the standard output device using the supplied formatting string).

The Microchip PIC18 C compiler does not support the **printf()** function.

Functions and Program Structure

- Every C program consists of one or more functions.
- The definition of a function cannot be embedded within another function.
- Values can be passed to a function through arguments.
- A function may return a value to the caller using the **return** statement.
- The syntax of a function definition is as follows:

```
return_type  function_name (declarations of arguments)
{
        declarations and statements
}
```

The following function converts a lowercase letter to uppercase:

```
char  lower2upper (char cx)
{
        if (cx >= ‘a’ && cx <= ‘z’) return (cx – (‘a’ – ‘A’));
}
```

- A function cannot be called before it has been defined.
- This dilemma is solved by using **function prototype statement**.
- The syntax for a function prototype statement is

```
        return_type  function_name (declarations of arguments);
```

Example 5.1 Write a C function to compute the average of an array of integers. Both the starting address of the array and the array count are passed to this function.

Solution:

```
int  array_ave (int arr[ ], int ar_cnt)
{
        short long int sum;
        int i;
        sum = 0;
        for (i = 0; i < ar_cnt; i++)
                sum += arr[i];
        return (sum / ar_cnt);
}
```

Example 5.2 Write a function to compute the square root of a 32-bit number using the successive approximation method.

Solution:

```
unsigned sq_root (unsigned long int xz)
{
        unsigned int sar, guess_mask, rest_mask;
        unsigned int i;
        sar = 0;            /* successive approximation register is initialized to 0 */
        guess_mask = 0x8000;  /* this mask is used to guess the ith bit to be 1 */
        i = 16;
        do {
                rest_mask = ~guess_mask; /* rest_mask is used to cancel the incorrect
guess */
                sar |= guess_mask; /* guess the ith bit to be 1 */
                if (sar * sar > xz)
                        sar &= rest_mask;           // change the bit to 0
                guess_mask >> 1;
                i--;
        } while (i > 0);
        if ((xy - sar * sar) < ((sar + 1)*(sar + 1) – xy))
                return sar;
        else return (sar + 1);
}
```

Example 5.3 Write a function to test whether an integer is a prime number.
Solution:

```
unsigned sq_root (unsigned long int xz);

/* this function returns a 1 if ka is prime. Otherwise, it returns a 0. */
char  test_prime (unsigned long int ka)
{
            unsigned int i, limit;
            if (ka == 1) return 0;
            else if (k2 == 2) return 1;
            limit = sq_root (ka);           /* find the square root of ka */
            for (i = 2; i <= limit; i++)
                        if ((a % i) == 0) return 0;
            return 1;
}
```

Example 5.4 Write a program to find out the number of prime numbers between 1000 and 10000.

Solution:

```
#include <stdio.h>
char test_prime (unsigned int a);              /* prototype declaration of test_prime () */
unsigned sq_root (unsigned long int xz); /* prototype of sq_root () */

main ( )
{
        unsigned int  i, prime_count;
        prime_count = 0;
        for (i = 1000; i <= 10000; i++) {
                if (test_prime(i))
                        prime_count ++;
        }
        printf(“\n The total prime numbers between 1000 and 10000 is %d\n”,
prime_count);
}
/* include the functions sq_root () and test_prime () here */
```

Pointers and Addresses

- A pointer is a variable that holds the address of a variable.
- Pointers can be used to pass information back and forth between a function and its calling point.
- Pointers provide a way to return multiple data items from a function via its arguments.
- The syntax for declaring a pointer type:

```
type_name *pointer_name;
```

- One can use the **&** operator to assign the address of a variable to a pointer variable.
- One can use the ***** operator to access the value pointed to by a pointer variable.

For example,

```
int  *ax;
char *cp;
int  x, y;

ax  = &x;          /* assign the address of x to ax */
y   = *ax;         /* y gets the value of x */
```

Example 5.5 Write the bubble sort function to sort an array of integers.
Solution: A swap function is needed by the bubble sort.

```
void  swap (int *, int *);
void  bubble (int a[], int n)        /* n is the array count */
{
        int i, j;
        for (i = 0; i < n – 1; i++)
                for (j = 0; j < n – (i+1); j++)
                        if (a[j ] > a[j+1])
                                swap (&a[j], &a[j+1]);
}
void swap (int *px, int *py)
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

Arrays

- An array holds one or multiple data items of common characteristics.
- Each array element is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in brackets.
- Each subscript must be a nonnegative number.
- The number of subscripts determines the dimensionality of the array.
- High dimensional arrays are not used very often in 8- and 16-bit microcontroller applications.
- A one-dimensional array can be expressed as

```
data_type  array_name [expression];
```

- A two-dimensional array is defined as

```
data_type  array_name [expr1][expr2];
```

- An array can be initialized when it is defined:

```
unsigned char led_pattern [10] =
            {0x7E, 0x30, 0x6D, 0x79, 0x33, 0x5B, 0x5F, 0x70, 0x7F, 0x7B};

char prompt [24] = “Please enter an integer:”;
```

Passing Arrays to a Function

- An array name can be used as an argument to a function.
- To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within a function call.
- An example is as follows:

```
int     average (int n, int arr[]);
void main ( )
{
        int n, avg;                  /* variable declaration */
        int arr[50];                   /* array definition */
        …
        avg = average(n, arr);         /* function call */
        …
}
int average (int k, int brr[])  /* function definition */
{
        …
}
```

Structures

- A structure is a group of related variables that can be accessed through a common name.
- Each item within a structure has its own data type.
- The syntax of a structure declaration:

```
struct  struct_name {           /* struct_name is optional */
      type1  member1;
      type2  member2;
      …
}
```

- The **struct_name** is optional, if it exists, defines a **structure tag**.
- A struct declaration defines a type.
- The right brace terminates the list of members and may be followed by a list of variables.
- A structure definition not followed by a list of variables does not reserve any space.
- An example,

```
struct catalog_tag {
        char        author [40];
        char        title [40];
        char        pub [40];
        unsigned  int date;
        unsigned  char rev;
} card;
```

Union

- May hold objects of different types and sizes with the compiler keeping track of size and alignment requirements.
- Allow one to manipulate different kinds of data in a single area of storage without embedding any machine dependent information in the program.
- Syntax of the union is

```
union  union_name {
        type-name1 $element_1$;
        type-name2 $element_2$;
        …
        type-namen $element_n$;
};
```

- The **union_name** field is optional, when exists, is called a **union tag**.
- Union variables can be declared at the same time when a union type is declared.

An example of union

```
union  u_tag {
int    i;
char         c[4];
} temp;
```

The access of union members is similar to structure type:

1. union-name.member
2. union-pointer→member

Automatic/External/Static/Volatile

- A variable defined inside a function is an **internal variable**.
- **Internal variables** are **automatic**. They come into existence when a function is entered and disappear when it is left.
- **External variables** are defined outside of any function and may be accessible by many functions.
- An internal variable can be made into **static** by adding the keyword **static** when it is declared.
- A static variable will maintain its value over function calls.
- When a variable is declared static outside all the functions in a file, its scope is limited to the file in which it is declared.
- A **volatile** variable has a value that can be changed by something other than the user code.
- I/O ports and timer registers are examples of volatile variables.
- Compiler makes no assumption on the value of a volatile variable and won't perform any optimization on the volatile variable.

Project Build Process of the Microchip PIC18 C Compiler

1. Source code entering and editing
2. Object code generation
3. Library files creation and maintenance (optional). The user can optionally put related reusable function modules in a single library file.
4. Program linking and executable code generation
5. The whole process is shown in Figure 5.2.
6. The file **output.hex** can be downloaded into the target hardware for execution.
7. The fle **output.cod** provides information needed in debugging process.
8. The file **output.lst** contains the source code side by side with final binary code and line numbers.
9. The file **output.out** is an intermediate file used by the linker to generate **cod** file, hex file, and listing file.
10. The file **output.map** shows the memory layout after linking.

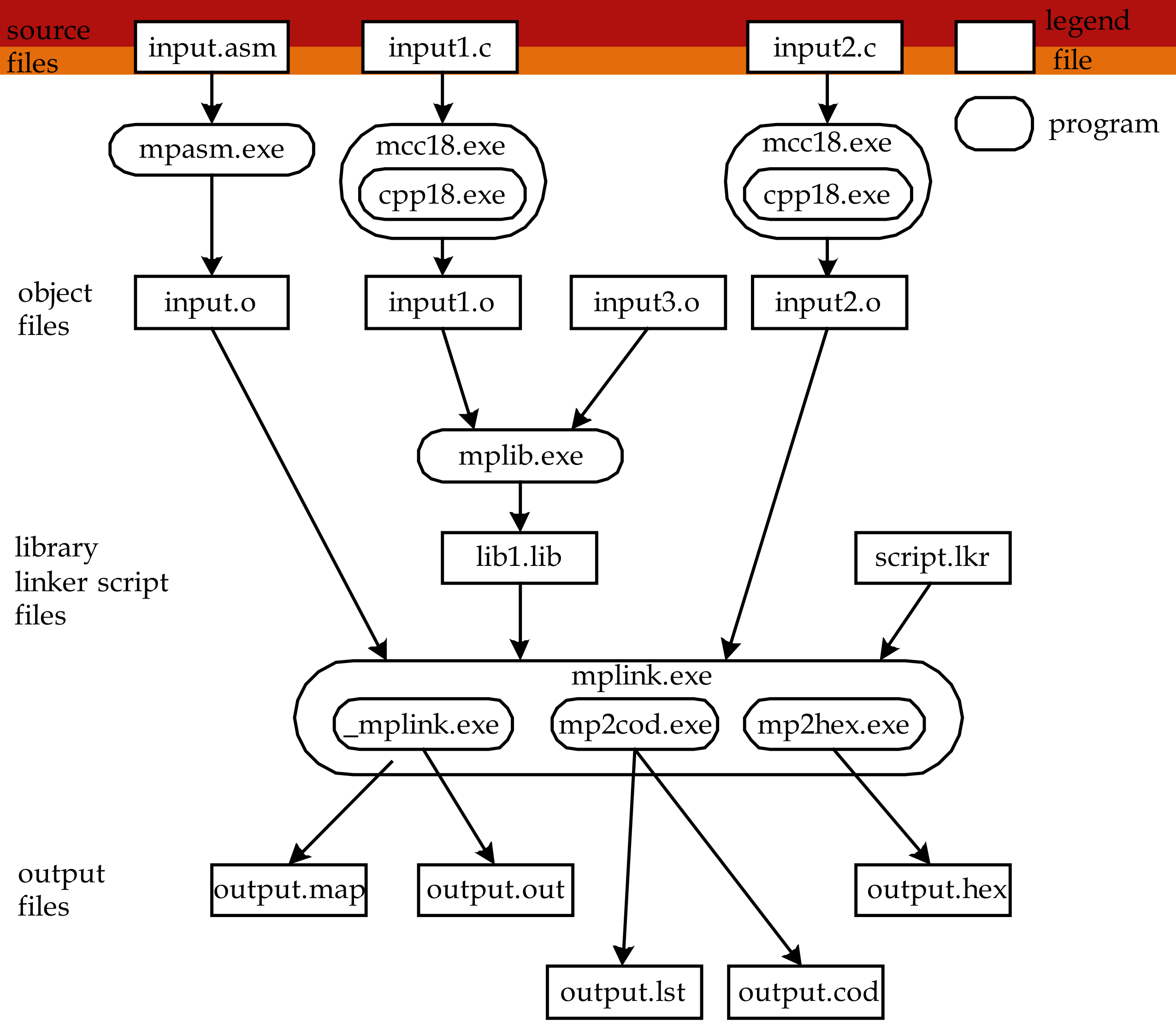


Figure 5.2 Project build process (reprint with permission of Microchip)

The MPLINK Linker Functions

1. Locating code and data
2. Resolving addresses
3. Generating an executable
4. Configuring stack size and location
5. Identifying address conflicts
6. Providing symbolic debug information

A linker file (with a suffix **.lkr** to the file name) must be added to the project to provide the above function.

The linker file without debug support for the PIC18F8720 is shown in Figure 5.3a.
The linker file with debug support for the PIC18F8720 is shown in Figure 5.3b.

```
// Sample linker command file for 18F8720
// $Id: 18f8720.lkr,v 1.4 2002/08/22 20:53:50 sealep Exp $
LIBPATH .
FILES c018i.o
FILES clib.lib
FILES  p18f8720.lib

CODEPAGE      NAME=vectors     START=0x0         END=0x29        PROTECTED
CODEPAGE      NAME=page        START=0x2A        END=0x1FFFFF
CODEPAGE      NAME=idlocs      START=0x200000    END=0x200007    PROTECTED
CODEPAGE      NAME=config      START=0x300000    END=0x30000D    PROTECTED
CODEPAGE      NAME=devid       START=0x3FFFFE    END=0x3FFFFF    PROTECTED
CODEPAGE      NAME=eedata      START=0xF00000    END=0xF000FF    PROTECTED
ACCESSBANK    NAME=accessram   START=0x0         END=0x5F
DATABANK      NAME=gpr0        START=0x60        END=0xFF
DATABANK      NAME=gpr1        START=0x100       END=0x1FF
DATABANK      NAME=gpr2        START=0x200       END=0x2FF
DATABANK      NAME=gpr3        START=0x300       END=0x3FF
DATABANK      NAME=gpr4        START=0x400       END=0x4FF
DATABANK      NAME=gpr5        START=0x500       END=0x5FF
DATABANK      NAME=gpr6        START=0x600       END=0x6FF
DATABANK      NAME=gpr7        START=0x700       END=0x7FF
DATABANK      NAME=gpr8        START=0x800       END=0x8FF
DATABANK      NAME=gpr9        START=0x900       END=0x9FF
DATABANK      NAME=gpr10       START=0xA00       END=0xAFF
DATABANK      NAME=gpr11       START=0xB00       END=0xBFF
DATABANK      NAME=gpr12       START=0xC00       END=0xCFF
DATABANK      NAME=gpr13       START=0xD00       END=0xDFF
DATABANK      NAME=gpr14       START=0xE00       END=0xEFF
ACCESSBANK    NAME=accesssfr   START=0xF60       END=0xFFF       PROTECTED
SECTION       NAME=CONFIG      ROM=config
STACK         RAM=gpr14
SIZE=0x100
```

Figure 5.3a P18F8720 linker file without debugging support (reprint with permission of Microchip)

```
// Sample linker command file for 18F8720i for MPLAB ICD2
// $Id: 18f8720i.lkr,v 1.3 2002/11/07 23:23:51 sealep Exp $
LIBPATH .
FILES c018i.o
FILESclib.lib
FILES p18f8720.lib

CODEPAGE      NAME=vectors    START=0x0        END=0x29       PROTECTED
CODEPAGE      NAME=page       START=0x2A       END=0x1FDBF
CODEPAGE      NAME=debug      START=0x1FDC0    END=0x1FFFF    PROTECTED
CODEPAGE      NAME=idlocs     START=0x200000   END=0x200007   PROTECTED
CODEPAGE      NAME=config     START=0x300000   END=0x30000D   PROTECTED
CODEPAGE      NAME=devid      START=0x3FFFFE   END=0x3FFFFF   PROTECTED
CODEPAGE      NAME=eedata     START=0xF00000   END=0xF000FF   PROTECTED
ACCESSBANK    NAME=accessram  START=0x0        END=0x5F
DATABANK      NAME=gpr0       START=0x60       END=0xFF
DATABANK      NAME=gpr1       START=0x100      END=0x1FF
DATABANK      NAME=gpr2       START=0x200      END=0x2FF
DATABANK      NAME=gpr3       START=0x300      END=0x3FF
DATABANK      NAME=gpr4       START=0x400      END=0x4FF
DATABANK      NAME=gpr5       START=0x500      END=0x5FF
DATABANK      NAME=gpr6       START=0x600      END=0x6FF
DATABANK      NAME=gpr7       START=0x700      END=0x7FF
DATABANK      NAME=gpr8       START=0x800      END=0x8FF
DATABANK      NAME=gpr9       START=0x900      END=0x9FF
DATABANK      NAME=gpr10      START=0xA00      END=0xAFF
DATABANK      NAME=gpr11      START=0xB00      END=0xBFF
DATABANK      NAME=gpr12      START=0xC00      END=0xCFF
DATABANK      NAME=gpr13      START=0xD00      END=0xDFF
DATABANK      NAME=gpr14      START=0xE00      END=0xEF3
DATABANK      NAME=dbgspr     START=0xEF4      END=0xEFF      PROTECTED
ACCESSBANK    NAME=accesssfr  START=0xF60      END=0xFFF      PROTECTED
SECTION       NAME=CONFIG     ROM=config
STACK         RAM=gpr13
SIZE=0x100
```

Figure 5.3b P18F8720 linker file with debugging support (reprint with permission of Microchip)

Inline Assembly

- The MPLAB C compiler provides an internal assembler that allows the user to enter a block of assembly instructions into the C program.
- The block of instructions must begin with _**asm** and end with _**endasm**.
- Some restrictions to inline assembly apply:
  1. No directive support
  2. Comments must be C or C++ notation
  3. Full text mnemonics must be used for table reads/writes, that is

- TBLRD	(not TBLRD*)
- TBLRDPOSTDEC	(not TBLRD*-)
- TBLRDPOSTINC	(not TBLRD*+)
- TBLRDPREINC	(not TBLRD+*)
- TBLWT	(not TBLWT*)
- TBLWTPOSTDEC	(not TBLWT*-)
- TBLWTPOSTINC	(not TBLWT*+)
- TBLWTPREINC	(not TBLWT+*)

4. No defaults for instruction operands—all operands must be specified.
5. Default radix is decimal.
6. Label must include colon.

Example of in-line assembly

```
_asm
         clrf      count,0
loop:
         movlw     0x20              // check loop count
         cpfseq    count,0           //          “
         goto      doit
         goto      done
doit:
         movf      sum_lo,0,0        // move sum_lo to WREG
         addwf     count,0,0         // add count to sum_lo
         movwf     sum_lo,0          // update sum_lo
         movlw     0                 // add carry to high byte of sum
         addwf     sum_hi,1,0        //          “
         incf      count,1,0
         goto      loop
done:
         nop
_endasm
```

Bit Field Manipulation

The processor-specific header file includes a structure definition that allows the user to access individual bits of a register by

1. appending **bits** to the register name
2. appending a period to the symbol resulted in step 1
3. specifying the bit name after the period

For example,

```
PORTBbits.RB0          = 1;  /* pull PORTB bit 0 to high */
PORTBbits.RB1          = 0;  /* pull PORTB bit 1 to low */
STATUSbits.C           = 0;  /* clear the C flag to 0 */
```

PIC18 Instructions Provided as Macros

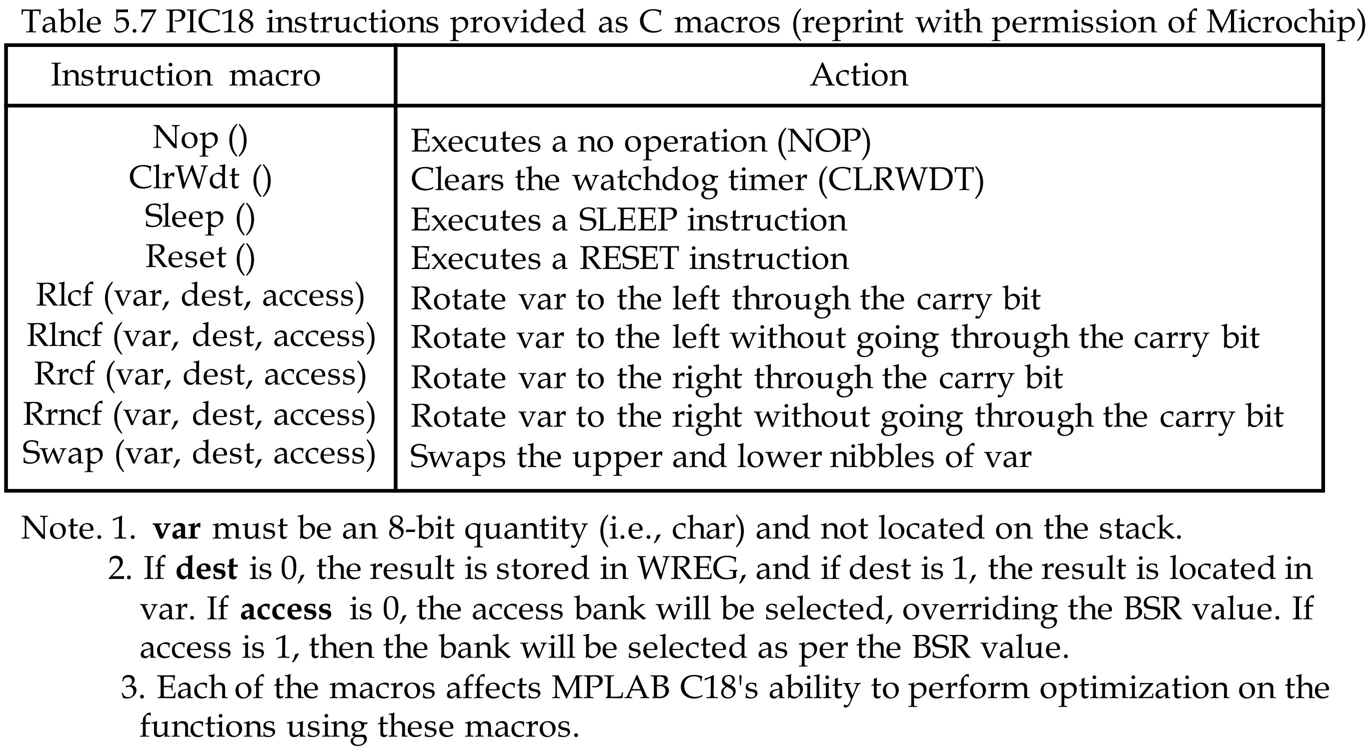


Table 5.7 PIC18 instructions provided as C macros (reprint with permission of Microchip)

Instruction macro	Action
Nop ()	Executes a no operation (NOP)
ClrWdt ()	Clears the watchdog timer (CLRWDT)
Sleep ()	Executes a SLEEP instruction
Reset ()	Executes a RESET instruction
Rlcf (var, dest, access)	Rotate var to the left through the carry bit
Rlncf (var, dest, access)	Rotate var to the left without going through the carry bit
Rrcf (var, dest, access)	Rotate var to the right through the carry bit
Rrncf (var, dest, access)	Rotate var to the right without going through the carry bit
Swap (var, dest, access)	Swaps the upper and lower nibbles of var

Note. 1. **var** must be an 8-bit quantity (i.e., char) and not located on the stack.

2. If **dest** is 0, the result is stored in WREG, and if dest is 1, the result is located in var. If **access** is 0, the access bank will be selected, overriding the BSR value. If access is 1, then the bank will be selected as per the BSR value.

3. Each of the macros affects MPLAB C18's ability to perform optimization on the functions using these macros.

The #pragma Statement

A compiler writer can add implementation-dependent options to the language by using the **#pragma** statement including

1. declare data or code sections
2. declare section attributes
3. locate code or data
4. declare interrupt vectors
5. declare interrupt service routines (to be discussed in Chapter 6)
6. other implementation-dependent features

MPLAB C18 Library Functions

- A library is a collection of functions grouped for reference and ease of linking.
- C18 libraries are included in the **lib** subdirectory of the installation.
- C libraries are divided into two groups: processor-independent libraries and processor-specific libraries.

Processor-Specific Libraries

- Files contain definitions that may vary across different members of the PIC18 family.
- These libraries include all the peripheral routines and the special-function definitions.
- The processor-specific libraries are named p**processor.**lib. For example, the library for the PIC18F8720 is named p18F8720.lib.

Processor-Independent Libraries

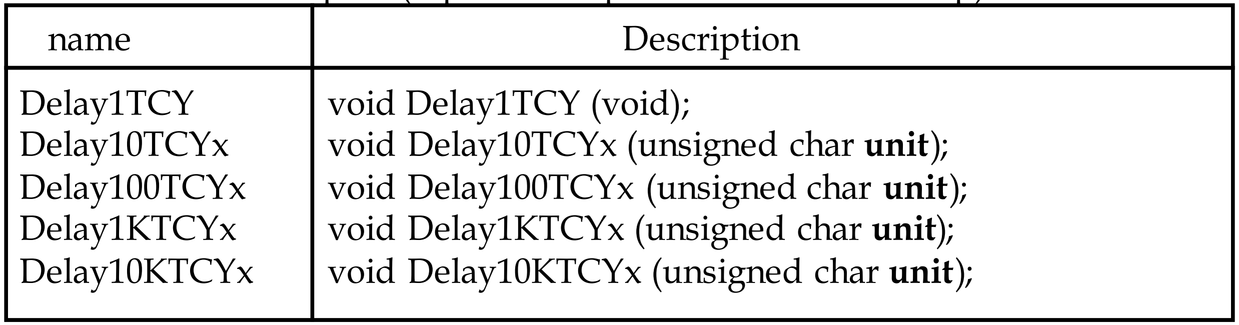
- General functions and math functions are in this category.
- These functions are contained in **clib.lib**.

MPLAB C18 general library supports the following categories of routines:

- Character classification functions
- Data conversion functions
- Delay functions
- Memory and string manipulations

These library functions are listed in Table 5.10a, 5.10b, 5.11a, 5.11b, 5.12a, and 5.12b. The prototype definitions of delay functions are listed in Table 5.12b.

Table 5.12b Names and prototype declarations of C delay routines in MPLAB C18 compiler (reprint with permission of Microchip)



name	Description
Delay1TCY	void Delay1TCY (void);
Delay10TCYx	void Delay10TCYx (unsigned char **unit**);
Delay100TCYx	void Delay100TCYx (unsigned char **unit**);
Delay1KTCYx	void Delay1KTCYx (unsigned char **unit**);
Delay10KTCYx	void Delay10KTCYx (unsigned char **unit**);

Creating time delay can easily be done by calling the appropriate delay functions. For example, with f_{OSC} = 32 MHz, instruction cycle time = 1/8 μs,

The following two statements can create 100 μs and 1 ms delays, respectively:

```
    Delay100TCYx(8);

    Delay1KTCYx(8);
```