Chapter 6 Programming for i8086

This chapter describes how to use the functions depending on the type of hardware, internal data expressions, and function calls on the machine.
This chapter is indented for programmers who are familiar with the machine language of i8086. Programmers who wish to write C language subroutines in the assembly language must read this chapter thoroughly.

6.1 Memory models

The i8086 processor has a memory area of 1M bytes. However, the program can refer to only an area of 64 Kbytes in this memory area. Therefore, it is preferable to express the pointer type using 16 bits. In the C language, whether the pointer points the program or data can be determined based on program sentences. Therefore, it is possible to allocate 64 Kbytes to each of the program and data areas.

However, this is not enough to write an application program. In some cases, it is required to refer to a large data area even though the efficiency may lower. To achieve this, this compiler provides several memory models to allow programmers to select optimal one suitable for the purpose.

Four types of memory models, S, P, D, and L, are available.

S model — Both the program and data are limited to 64 Kbytes. Normally, this model is used. The S model is the most efficient memory model in the program size and speed.

P model — Up to 64 Kbytes is allowed for the data, but up to 1 Mbytes is allowed for the program. However, the size of the program in one module (file) is limited to 64 Kbytes. This model is efficient next to the S model.

D model — Up to 64 K bytes is allowed for the program, but up to 1 M bytes is allowed for the total data capacity. However, the size of each data (variable) is limited to 64 Kbytes. The efficiency of this memory model is low when compared to the S and P models.

L model — Both the program and data can use up to 1 Mbytes. However, the size of the program in each module (file) is limited to 64 Kbytes and the size of each data (variable) is also limited to 64 Kbytes. The efficiency of this memory model is also low, the same as the D model.

6.2 far, near

In the LSI C, it is possible to handle large data using the S model or P model. To achieve this, the keyword “far” is provided. Additionally, to improve the efficiency of the D model and L model, the keyword “near” is provided. There are three kinds of methods available to use the keywords “far” and “near”.

- `far` function and `near` function

```
int far foo ();
```

The above example shows that the function `foo` is called by the `CALLF (far CALL)` command. In the P and L models, all functions become `far` functions automatically.

Using this format, far function can be defined in the S and D models. At this time, the far function `foo` is normally placed in the `TEXT` segment in the same manner as normal functions, but returned to the function, which has called, by the `RETF` command instead of the `RET` command.

```
static void near bar ();
```

The function `bar` is called by the `CALL (near CALL)` command. In the S and D models, all functions become near functions automatically.

Using this format, near function can be defined in the P and L models. At this time, the function `bar` is placed in the segment generated from the file name in the same manner as other functions in the file where the function `bar` is defined (see section 6.3, “Segment name”), but the `RET` command is used instead of the `RETF` to exit the function.
Since a pair of the near `CALL` and `RET` is used to call the function `bar`, the efficiency of code size and execution speed is improved.

However, the near function cannot be called from other modules (files). Since other modules are different modules, that is, different segments, the near function cannot be called using the `CALL` command and returned using the `RET` command.
Therefore, the near function must be defined as the static function.

- `far` variable

```
int far x;
```

The above example shows that the variable x is placed in the individual segment. All variables are handled as near unless far is put. The size of each far variable is limited to 64 Kbytes.

- `far` pointer

```
int far *fp;
```

The above example shows that the pointer variable `fp` can point any segment. In the S and P models, all pointers become near pointers unless otherwise specified. In the D and L models, all pointers become far pointers unless otherwise specified. The far pointer can be used in any model. If large data is handled locally, only necessary pointers are explicitly declared as far and the S model is used to obtain higher efficiency.

If `int` type data is added or subtracted to/from the far pointer or address of the far variable, the offset of the address is only changed. To make calculations including segments, long type data must be added or subtracted. As a result of calculation, the offset becomes 15 or less in the normalization format.

The following program shows an example using the far pointer.

```
clear_graphic_memory ()
{
        unsigned i;
        char far *p;

        p = (char far *) 0xA8000000;
        for (i = 0; i < 32768; i++)
        p [i] =0;
}
```

When comparing far pointers, they are handled as 32-bit unsigned positive number.
At this time, note that pointers cannot be compared correctly if both pointers are not normalized. When OL is added to the pointer, the normalized address can be obtained.

6.3 Segment name

The compiler places programs and data in segments shown in Table 6.1. `x` in the Table shows the name of a file containing sources. For example, when the program in the file foo.c is compiled using the P model, it is placed in the segment named `foo_TEXT`. Additionally, `n` in the Table shows a segment appearing number and becomes like `foo_1_DATA` or `foo_2_BSS`.

	Segment name	Group name	Class name
Program using S or D	TEXT	CGROUP	CODE
Program using P or L	x_TEXT		CODE
Initialized data	DATA	DGROUP	DATA
Uninitialized data	BSS	DGROUP	DATA
Initialized far data	x_n_DATA		FAR_DATA
Uninitialized data	x_n_BSS		FAR_BSS

Table 6.1 Segment name

6.4 Internal data expression

Table 6.2 shows the internal expressions of basic type data.

Data type	Number of bits
`char, signed char, unsigned char`	`8`
`short, unsigned short`	`16`
`int, unsigned int`	`16`
`long, unsigned long`	`32`
`float`	`32 (= 1 + 8 + 23)`
`double`	`64 (= 1 + 11 + 52)`
`long double`	`80 (= 1 + 15 + 64)`
`near pointer`	`16`
`far pointer`	`32`

Table 6.2 Number of bits for basic type data

All data types are expressed in decimal and the lowest byte is placed at the smallest address (little endian). Address boundaries are not adjusted. The standard char is the same as unsigned char. However, it can be changed to signed char using the `-cs` option in the compiler. A negative number of singed `int` type data is expressed using two's complement.

The floating-point type data is composed of three fields, sign bit, exponent part, and mantissa part. The Table below shows the number of bits for each field. All types of data contain a sign bit of 1 bit. If the sign bit is 0 and 1, it shows positive and negative data, respectively. The exponent part is normally biased. A bit pattern of 0111…111 shows that the exponent part is 0. In the float and double types, first 1 is omitted from the mantissa part and it is considered that the decimal point is located at the leftmost position. In the long double type, first 1 is not omitted and it is considered that the decimal point is located between bit_{63} and bit_{62}.

The following shows internal expressions of several floating-point type data in hexadecimal.

	float	double	long double
0	00000000	0000000000000000	00000000000000000000
1	3F800000	3FF0000000000000	3FFF8000000000000000
3	40400000	4008000000000000	4000C000000000000000
-3	C0400000	C008000000000000	C000C000000000000000
0.25	3E800000	3FD0000000000000	3FFD8000000000000000

There are two types of pointer type data, near and far. near pointer is used in the S and P models while far pointer is used in the D and L models.
The near pointer points data in the data segment implicitly, and has only an offset of 16 bits. The far pointer can point any address and have 32 bits in total, a segment of 16 bits and an offset of 16 bits. The segment and offset are at the higher and lower positions, respectively.

6.5 Register allocation

The compiler allocates variables, the memory class of which is declared as register, to the registers. In addition to this, variables declared as auto are also to be allocated to the registers in the LSI C. Since local variables without the memory class are considered as auto, they are allocated to the registers.

If you do not wish to allocate a variable to the registers, you must put the volatile qualifier. Variables, for which volatile is specified, are not allocated to the registers.

The compiler reads a function onto the memory, checks usage of each variable, and excludes variables which refer to addresses (variables, to which & unary operator is applied) from the register allocation. After that, the compiler allocates the variables to optimal registers selected from registers AX, BX, CX, DX, SI, and DI. If the variables are not allocated to the registers completely, excess variables are located on the memory.

Normally, whether or not the variables are allocated to the registers does not affect the execution results of the program. However, if the register variables are used in the function which calls the `setjmp ()` function, this register may contain an unexpected value when the function is returned by the `longjump ()` function.

For example, if the following program is run, it is expected to display 3142.

```
jmp_buf env;

bar ()
{
        longjmp (env, i);
}

foo ()
{
        int x;
        x = 1234;
        if (setjump (env) == 0) {
        x = 3142;
        bar ();
        }

        printf ("%d\n", x);
}
```

As a result of program execution, however, 1234 is displayed actually.
If the compiler allocates the variable x to the register automatically, the value in the register (value x) at calling of `setjmp ()` is registered and it is retrieved by the `longjmp ()` function.
To solve such problem, local variables must be declared as volatile in the function which calls `setjmp ()`.
Generally, there are no limitations on use of register variables other than the above problem.

6.6 Function call protocol

This section describes how the function parameters are transferred and the function values are set. If you wish to make a subroutine using the assembly language, you must read this section thoroughly.

To call a function, the near CALL command is used for the S and D models while the far CALL command is used for the P and L models.
The compiler adds an underscore (_) following the name of the function or external variable. This may prevent the same spelling as that used for the assembler reserved words. Therefore, to define a function or external variable using the assembly language, an underscore (_) is added following its name.

Regardless of the memory model, the function, which is called up, does not break values in registers SS, CS, SP, BP, DI, SI, DX, and CX. If necessary, the data in the registers is pushed to the stack area at the beginning of the function, and then popped at the last to restore the data in the registers.

When the parameters are stacked in the stack area, the data in the stack area is restored by the function which calls a function instead of the function which is called up. Therefore, the RET n command (command which returns with n added to SP) cannot be used. The normal RET command must be used.

To exchange the function parameters, the registers AX, BX, CX, and DX, and stack are used. The following rules are applied to each parameter from the left the parameters to determine a register where the parameter is put.

1. If the parameter type is char, unsigned char, signed char, 1-byte structure, or 1-byte enumeration, that parameter is allocated to AL, BL, CL, or DL which is not used. If all registers are not empty, one word is kept on the stack.
2. If the parameter type is `int`, unsigned `int`, short, unsigned short, near pointer, 2-byte structure, or 2-byte enumeration, that parameter is allocated to AX, BX, CX, or DX which is not used. If all registers are not empty, one word is kept on the stack.
3. If the parameter type is long, unsigned long, far pointer, 4-byte structure, or 4-byte enumeration, that parameter is allocated to BXAX or DXCX register pair which is not used. If all registers are not empty, two words are kept on the stack.
   At this time, BX and DX show an upper word while AX and CX show a lower word.
4. If all above rules are not applied to a parameter, this parameter is placed on the stack. However, if the number of the data bytes is an odd number, additional one byte and the number of even bytes are kept.

If the function uses variable parameters, the above rules are not applied. At this time, the stack area is used to exchange the parameters. That is, it is the same as that the above rules are applied assuming that all registers are in use.

The function value is allocated to the register or stack area according to the following rules.

1. If the function type is `char`, `unsigned char`, `signed char`, 1-byte structure, or 1-byte enumeration, the function value is allocated to the AL register.
2. If the function type is `int`, `unsigned int`, `short`, `unsigned short`, `near pointer`, 2-byte structure, or 2-byte enumeration, the function value is allocated to the AX register.
3. If the function type is `long`, `unsigned long`, `far pointer`, 4-byte structure, or 4-byte enumeration, the function value is allocated to the BXAX register pair. At this time, BX and AX show an upper word and lower word, respectively.
4. If the above registers are not available, the function value is put at an address next to the address at which the last parameter on the stack is put. This area must be kept by the function calling side.

Since the current compiler does not transfer the address at which the function value is to be stored, the variable parameter function cannot return `float`, `double`, `long double`, and structure type data. (This may be corrected in the future.)

The following shows a program example. In this example, SP means a value of the stack pointer at the inlet of the function, and it is assumed that all the functions are near functions which are to be called by the near `CALL` command. Therefore, if you use a far function, you need to add 2 to deviation against SP.

```
char x (int a, char b, char c, char near *d, int e);
       a      = AX
       b      = BL
       c      = CL
       d      = DX
       e      = [SP + 2…SP + 3]
       function value      = AL

long y (int a, long b, char c);
       a      = AX
       b      = DXCX
       c      = BL
       function value      = BXAX

float z (double a, int b, float c);
       a      = [SP + 2…SP + 9]
       b      = AX
       c      = [SP + 10…SP + 13]
       function value      = [SP + 14…SP + 17]

int printf (FILE *fp, char *fmt, …);
       fp     = [SP + 2…SP + 3]
       fmt    = [SP + 4…SP + 5]
       function value      = AX
```

6.7 Assembly in-line function

In this compiler, it is possible to embed the assembly language in the C program. For function containing few commands, the overhead due to execution of the CALL or RET command cannot be disregarded. Therefore, the in-line development may improve the efficiency. This is called an assembly in-line function that has the following function call format.

```
_asm_XXX ("\n …. assembly language command string\n", parameter list)
```

Where, XXX is a desired alphanumeric character string and null can also be used. Assembly language command to be generated is put at the first parameter. To write commands with more than 2 lines, "\n" is put halfway. At this time, the compiler generates the character string at the first parameter directly instead of generation of the CALL command. Other portions are handled in the same manner as function call. Therefore, the parameters can be transferred and the function values can also be returned. Portions from the second parameter of the in-line function are handled as normal function parameters.

It is not possible to write assembler pseudo commands and commands that change the stack pointer in the in-line function. Additionally, it is thought that few commands are to be written in the in-line function. If many commands are written, compilation may not be made correctly.

As for a typical example, the following shows a program that handles the port input/output.

```
char_asm_c (char *);
char_asm_cc (char *, char);
#define inp10()           _asm_c("\n\tIN\tAL,16\n")
#define outp11(c)  _asm_cc("\n\tOUT\t17,AL\n", c)

void put (char c)
{
      while ((inp10 () & 1) == 0)
      ;
      outp11(c);
}
```

The above program generates the following codes.

```
put_::
       PUSH   CX
       MOV    CL, AL
_2:
       IN     AL, 16
       TEST   AL, 1
       JE     _2
       MOV    AL, CL
       OUT    17, AL
       POP    CX
       RET
```

As shown in the above example, it is recommended to define the in-line function as macro for easy reading.
In the above example, the port numbers are fixed. The program is modified to transfer the port numbers using the parameters.
To achieve this, two types of methods are available as described below.

1. A part of the character string is changed to embed the port number.
2. Port number is transferred through the DX register.

6.7.1 Embedding the port number in the character string

This is achieved using the preprocessor function.

```
char_asm_c (char *);
char_asm_cc (char *, char);
#define _qq_(s)           #s
#define _q_(s)            _qq_(s)
#define inp (p)           _asm_c      ("\n\tIN\tAL,      "
_q_(__eval__(p)))
#define outp (p, c) _asm_cc  ("\n\tOUT\t"  _q_(__eval__(p))
", AL", (c))


void put (char c)
{
       while ((inp (0x10) & 1) == 0)
       ;
       outp (0x11, c);
}
```

If the definition is made as shown above, `outp (0x11, c)` is developed by the preprocessor as shown below.

```
_asm_c ("\n\tOUT\t" "17" ",AL", c)
```

Additionally, it is further developed as shown below if character strings placed in parallel are connected.

```
_asm_c ("\n\tOUT\t17,AL", c)
```

`inp ()` is also developed in the same manner.

Note that data needs to be converted into decimal data using the `__eval__` macro since the hexadecimal and octal expressions of the assembly language and C language are different from each other.

6.7.2 Transferring the port number through the register

Forth parameter in the function is allocated to the DX register. This is used to transfer the port number. At this time, however, second and third parameters must be skipped. In the LSI C-86, parameters can be omitted by writing the name of the in-line function in the function parameter.

```
char _asm_ci ();
char _asm_cci ();
#define inp (p)          _asm_ci    ("\n\tIN\tAL,    DX",
_asm_ci, _asm_ci, _asm_ci, (p))
#define outp (p, c) _asm_cci   ("\n\tOUT\tDX,   AL",   (c),
_asm_ci, _asm_ci, (p))

void put (char c)
{
      while ((inp (0x10) & 1) == 0)
      ;
      outp (0x11, c);
}
```

6.8 #pragma

`#pragma` is a command that gives the instruction to the compiler and has the following format.

```
#pragma command
```

The following commands are applied.

`checksp`	This command generates the code that inspects the stack overflow at the inlet of the function.
`nochecksp`	On the contrary to checksp, this command does not generate the code that inspects the stack overflow at the inlet of the function.
`optimize time`	This command instructs the compiler to focus on the object code execution speed. That is, this command instructs that an object code with the faster speed is needed even though the object code is large.
`optimize space`	This command instructs the compiler to focus on the small object code. That is, this command instructs that a small object code is needed even though the speed is slow.
`recursive`	This command sets the default function mode to recursive.
`nonrec`	This command sets the default function mode to nonrec.
`regalo`	This command allocates the variables to the registers.
`noregalo`	This command prohibits to allocate the variables to the registers.

The default settings are `nochecksp`, `optimize space`, `recursive`, and `regalo`.

6.9 Program example of connection between the C and assembly languages

At the end of this chapter, a simple program example to connect the C and assembly languages is shown. This program converts the decimal number into hexadecimal number, and displays it.

`bar.c:`

```
#include <stdio.h>
#include <stdlib.h>
extern char foo (char);
char   upperflag;
int    main (int argc, char *argv [])
{
       int    n;
       n = atoi (argv [i]);
       upperflag = argv [2][0] - '1';
       putchar (foo (n >> 12));
       putchar (foo (n >> 8));
       putchar (foo (n >> 4));
       putchar (foo (n));
}
```

```
foo.a86:
       CGROUP GROUP  TEXT
       TEXT   CSEG

       EXTERN       upperflag_

foo_::
       AND    AL, 00FH
       ADD    AL, 090H
       DAA
       ADC    AL, 040H
       DAA

       CMP    [upperflag_].B, 0
       JNE    _1
       CMP    AL, 'A'
       JB     _1
              ADD    AL, 'a' - 'A'
```

```
_1:
RET
```

If a decimal number is given from the command line in this program, it is converted into the hexadecimal number. This number is then displayed.
The following shows a program in which the assembly language part is written using MASH.

foo.asm:

```
CGROUP GROUP  TEXT

              PUBLIC foo_
              EXTERN upperflag_:BYTE

TEXT          SEGMENT BYTE PUBLIC 'CODE'
              ASSUME CS: TEXT

foo_          PROC          NEAR
              AND           AL, 00FH
              ADD           AL, 090H
              DAA
              ADC           AL, 040H
              DAA

              CMP           upperflag_, 0
              JNE           _1
              CMP           AL, 'A'
              JB            _1
                            ADD   AL, 'a' - 'A'
              _1:

              RET
foo_          ENDP
TEXT          ENDS
              END
```