FUJITSU

富士通 16 位微控制器

C 语言手册

提高篇

富士通复旦应用研究中心

前言

说明：

假设读者已经掌握了使用 C 语言编写程序的技术。如果读者没有掌握这一技术，可以参考市面上的一些书目，还可以在原来的一些 Dos 下的一些 C 开发环境下进行练习，如 Borland 公司的 Dos 下的 TC 或者 BC 编译器。

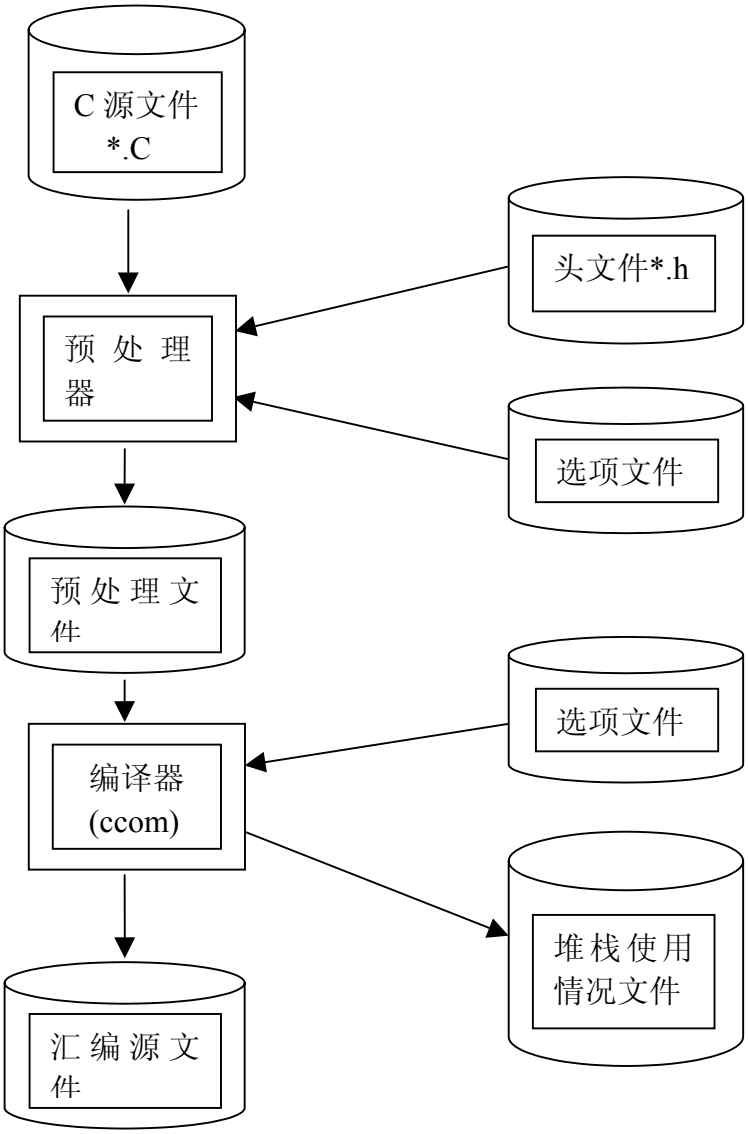
手册中的例子都经过验证，相应的编译器的版本是：V30L02，汇编器的版本是：V30L04，链接器的版本是：V30L05。要注意的是，有的例子直接使用在编译时会有警告提示，有的例子直接使用在链接时会有出错提示。如果读者对照英文的 C 语言手册来看，会发现两者有些差别，那是因为英文手册所对应的编译器/汇编器/链接器的版本号较低，为此在中文手册中作了修正。

目录

第一章 C 预处理器

C 预处理器是 fcc907s C 编译器(fcc907s 是富士通 16 位 MCU FFMC-16 的编译器)的一个组成部分。在 C 语言中，通过预处理指令，可以为 C 语言本身提供很多功能和符号等方面的扩充，可增强其灵活性和方便性。预处理指令只在程序编译时起作用，且通常是按行进行处理的，因此常又称为编译控制行。编译器在对整个程序进行编译之前，先对程序中的编译控制行进行预处理，然后再将预处理的结果与整个 C 语言源程序一起进行编译，产生汇编文件。常用的预处理指令有: 宏定义、文件包含和条件编译。预处理命令以符号“#”开头。

第一节　预处理和编译的结构



如上图所示，为 C 预处理与编译的关系的结构图。

第二节 宏定义

一. 不带参数的宏定义

不带参数的宏定义的一般格式为:

```
#define 标识符 待替换的字符串
```

其中，“标识符”是所定义的宏符号名(或称宏名)。宏定义的作用是，在程序中用指定的宏名来替代指定的字符串。

宏定义又称宏替换。预处理器每次在程序行中扫描到宏名，就把宏名替换为指定的字符串，但如果宏名出现在注释中和字符串中是例外，如果宏名为预处理器的保留字也不能被替换。如下例所示:

[例子]

```
#define BYTE unsigned char
#define ABC define
#ABC NUM 10              /* #ABC 被替换为 #define */
#define MSG Fujitsu
/* MSG */                /* MSG 不被替换为 Fujitsu */
#define MY_MSG“MSG”/* 字符串”MSG”不能替换为“Fujitsu” */
```

宏定义也称为符号常量定义。可以使用一些有一定意义的标识符来替换常数，提高程序的可读性。也可以用一个简单的符号名来替换一个很长的字符串。如下例所示:

[例子]

```
#define BUFFER_SIZE 1024          /* 定义缓冲区大小 */
#define MAX_LENGTH 80             /* 定义最大长度 */
#define FLAG 0xaa                 /* 定义标志字符 */
#define NL printf(“\n”)           /* 定义回车换行 */
```

采用这些定义可以使人们对程序中的这些常数的意义一目了然，有助于编写、修改程序和阅读程序。通常程序中的所有宏定义都集中放在程序的开始处，便于检查和修改，提高程序的可靠性。如果需要修改程序中的某个常量，可以不必在整个程序去查找然后修改，而只要修改一下相应的宏定义行即可。

按习惯，通常将宏符号名用大写字母表示，以区别于其它的变量名。宏定义不是 C 语言的语句，因此在宏定义行的末尾不要加分号，否则在编译时将连同分号一起进行替换，可能导致语法错误。如果在一个宏定义中包含另一个宏符号名，那么就形成宏定义嵌套。宏定义的嵌套深度最大可达 255 级。一般宏定义指令#define 的行放在文件的开头，其作用范围是从被定义的地方开始，至本源文件结束。

可以在其前面加反斜杠(\)来明确指出一个宏符号名。如果未指定待替换的字符串，则宏符号名将替换空字符。

■ #undef 指令

可以用预处理指令#undef 宏符号名，来删除一个宏符号名。如下例所示:

[例子]

```
#define MAX_LEN 80                      /*  定义最大长度为80 */
#define BUFFER_SIZE   \MAX_LEN*20  /*  定义缓冲区大小为 80*20 */
.
.
.
#undef MAX_LEN                          /*  删除上面定义的宏定义: 最大长度 */
#define MAX_LEN 120                     /*  重新定义最大长度为120 */
```

■ 预定义的宏符号名

C 编译器中已经预先定义了下列宏符号名(注意首尾都有双下划线):

__LINE__ : 被编译文件的当前行号

__FILE__ : 被编译文件的文件名

__DATE__ : 编译日期

__TIME__ : 编译时间

__STDC__ : 当编译选项是-Ja 时，为 0；当编译选项是-Jc 时，为 1

以上是 ANSI 标准所规定的预定义宏符号名。以下是 fcc907s C 编译器所预定义的宏符号名:

__COMPILER_FCC907__ : 始终为"1"

__CPU_MB number__ : 当 MB number 等于编译选项所指定的 MB 型号(即 Project->Setup…->Base->Target MCU)时，为 1，否则就出错(编译器认为没有定义过该宏符号名)

__CPU_16L__ : 当 Project->Setup…->Base->Chip Classification 指定的 CPU 类型与之相符时，为 1；否则就出错(编译器认为没有定义过该宏符号名)

__CPU_16LX__ : 同上

__CPU_16F__ : 同上

需要注意的是，以上预定义的宏符号名不能用#undef 和#define 指令来重新进行定义。

二. 带参数的宏定义

带参数的宏定义的一般格式为:

```
#define  宏符号名(参数列表)  表达式
```

宏符号名和左括号必须紧紧相连，其间不能有空格、注释及诸如此类的字符串。括号中，参数表里的参数被称为形式参数，在以后的程序中它们将被实际参数所替代。实际参数的数目必须与形式参数的数目一样。如果未指定任何

表达式，则宏符号名将替换空字符。带参数的宏定义也允许宏定义嵌套，宏定义的嵌套深度最多为 255 级。

[例子]

```
#define eq(a, b) a==b
#define ne(a, b) a!=b
.
.
.
int x,y,z;
x=y=1;
if( eq(x,y) )
    z=10;          /*  这一行得到执行  */
else
    z=20;          /*  这一行未得到执行  */
```

宏定义指令#define 要求在一行内写完，如一行内写不下时可在行末加反斜杠"\"进行续行。

第三节　文件包含(#include 指令)

文件包含是指一个程序文件将另一个指定文件的全部内容包含进来。比如经常使用的文件包含命令 #include <stdio.h>，就是将 C 编译器提供的输入输出库函数的说明文件 stdio.h 包含到自己的程序中去。文件包含命令的一般格式为：

#include <文件名> 或 #include “文件名”

文件包含命令#include 的功能是用指定文件的全部内容替换该预处理行。在进行较大规模程序设计时，文件包含命令是十分有用的。为了适应模块化编程的需要，可以将组成 C 语言程序的各个功能函数分散到多个程序文件中，分别由若干人员完成编程，最后再用#include 命令将它们嵌入到一个总的程序文件中去。需要注意的是，一个#include 命令只能指定一个被包含文件，如果程序中需要包含多个文件则需要使用多个包含命令。还可以将一些常用的符号常量、带参数的宏以及构造类型的变量等定义在一个独立的文件中，当某个程序需要时再将其包含进来。这样将可以减少重复劳动，提高程序的编制效率。

文件包含命令#include 通常放在 C 语言程序的开头，被包含的文件一般是一些公用的宏定义和外部变量说明，当它们出错，或是由于某种原因需要修改其内容时，只需对相应的包含文件进行修改，而不必对使用它们的各个程序文件都作修改，这样有利于程序的维护和更新。当程序中需要调用 C 编译器提供的各种库函数的时候，必须在程序的开头使用#include 命令将相应函数的说明文件包含进来。经常在程序开头使用的命令#include <stdio.h>就是为了这个目的。

第四节 条件编译

一般情况下对 C 语言程序进行编译时所有的程序行都参加编译，但有时希望对其中一部分内容只在满足一定条件时才进行编译，这就是所谓的条件编译。条件编译可以选择不同的编译范围，从而产生不同的代码。fcc907s C 编译器的预处理器提供以下编译命令：#if、#elif、#else、#endif、#ifdef、#ifndef，这些命令有三种使用格式，如下陈述。

■ **条件编译命令格式一**

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该命令格式的功能是：如果指定的标识符已被定义，则程序段 1 参加编译并产生有效代码，而忽略掉程序段 2，否则程序段 2 参加编译并产生有效代码而忽略程序段 1。其中#else 和程序段 2 可以没有。这里的程序段可以是单行或多行的 C 语言语句。这种条件编译对于提高 C 语言源程序的通用性是很有好处的。

[例子]

```
#define DEBUG
.
.
.
#ifdef DEBUG
    printf("We are debugging");
#endif
```

象这样一段源程序不作任何修改就可以用于正常运行的状态和调试状态下。当然还可以仿此设计出其它多种条件编译。

■ **条件编译命令格式二**

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该命令格式与第一种命令格式只在第一行上不同，它的作用与第一种刚好相反，即：如果指定的标识符未被定义，则程序段 1 参加编译并产生有效代码，而忽略程序段 2，否则程序段 2 参加编译并产生有效代码而忽略程序段 1。

以上两种格式的用法也很相似，可根据实际情况视需要而定。

■ **条件编译命令格式三**

```
#if 常量表达式 1
    程序段 1
#elif 常量表达式 2
```

```
        程序段 2
... ...
#elif  常量表达式 n-1
        程序段 n-1
#else
        程序段 n
#endif
```

这种格式条件编译的功能是：如果常量表达式 1 的值为真(非 0)，则程序段 1 参加编译，然后将控制传递给匹配的#endif 命令，结束本次条件编译，继续下面的编译处理。否则，如果常量表达式 1 的值为假(0)，则忽略掉程序段 1(不参加编译)而将控制传递给下面的一个#elif 命令，对常量表达式 2 的值进行判断。如果常量表达式 2 的值为假(0)，则将控制再传递给下一个#elif 命令。如此进行直到遇到#else 或#endif 命令为止。使用这种条件编译格式可以事先给定某一个条件，使程序在不同的条件下完成不同的功能。

[例子]

```
#define __MB_90560
#define TRUE 1
#define FALSE 0
#define DEBUG 1

#pragma      section   CODE=main, attr=CODE

void main(void)
{
   int _90560,_90570,debug;
   #ifdef __MB_90560
      _90560 = TRUE;                    /*  这一行得到执行  */
   #endif
   #ifndef __MB_90570
      _90570 = FALSE;                   /*  这一行得到执行  */
   #endif
   #if DEBUG
      debug = TRUE;                     /*  这一行得到执行  */
   #else
      debug = FALSE;                    /*  这一行未得到执行  */
   #endif
   #if defined(__MB_90560)
      _90560 = TRUE;                    /*  这一行得到执行  */
   #elif defined(__MB_90570)
      _90570 = TRUE;                    /*  这一行未得到执行  */
   #endif
}
```

第五节 其它预处理指令

除了上面介绍的宏定义、文件包含和条件编译预处理命令之外，C 编译器还支持#pragma、#error 和#line 预处理命令。#line 命令一般很少使用，下面介绍#error 和#pragma 命令的功能和使用方法。

#error 命令通常嵌入在条件编译中，以便捕捉到一些不可预料的编译条件。正常情况下该条件的值应为假，若条件的值为真，则输出一条由#error 命令后面的字符串所给出的错误信息并停止编译。例如，如果有#define MYVAL,它的值必须为 0 或 1，为了测试 MYVAL 的值是否正确，可在程序中安排下一段条件编译：

[例子]

```
#if (MYVAL!=0 && MYVAL!=1)
#error MYVAL must be defined to either 0 or 1
#endif
```

当 MYVAL 的值出错时将输出出错信息“MYVAL must be defined to either 0 or 1”并停止编译。

#pragma 命令通常用在源程序来实现 C 编译器的各种特殊扩充功能，其一般格式为：

```
#pragma 字符串序列
```

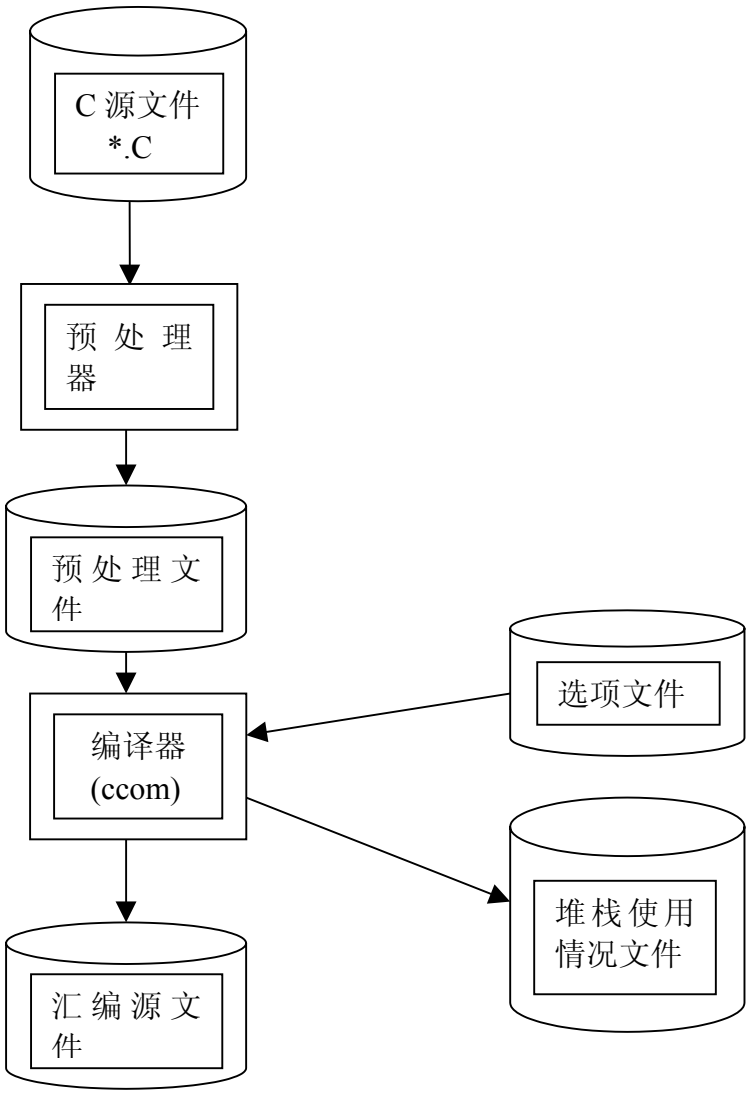
[例子]

```
sample(){
#pragma asm
    MOVN    A, #1
    MOVW    _temp, A
#pragma endasm
}
```

通过#pragma 命令，可以实现在 C 语言源程序中嵌入汇编指令，给 SECTION 改名，设置寄存器段，设置中断级别等等 C 编译器的各种特殊扩充功能。在下一章将对#pragma 命令作详细介绍。

第二章　C 编译器

第一节 编译的结构



如上图所示，为 C 编译器进行编译的结构图。

第二节　C 编译器的数据调用协议

一. 内存模式和数据在内存中的存储格式

■ 内存模式(编译模式)

内存模式是指如何在存储器中放置程序代码和数据，它们允许占用的存储器大小，及如何存取它们。

与 PC 机上的编译器相似，富士通 fcc907s C 编译器允许用户使用 4 种内存模式来进行 C 语言编程。4 种可用的内存模式及其含义如下面的表格所示。

表格　内存模式列表

内存模式	代码地址空间	数据地址空间	编译选项*
小模式	16 位	16 位	-model small
中模式	24 位	16 位	-model medium
紧凑模式	16 位	24 位	-model compact
大模式	24 位	24 位	-model large

*注：在富士通的 MCU 集成开发环境 Softune Workbench 中，编译选项在 Project->Setup Tool Option…->C Compiler->Category->Target Depend 的 Memory Model 这一位置上进行设置。

❍ 小模式

在该模式下，程序中的代码放在 16 位地址空间(64k)的代码段内，数据放在 16 位地址空间(64k)的数据段内。在该模式下，指针都是近程的（near）。因为代码和数据都不超出 16 位的地址空间，所以编译器可以生成紧凑的且执行速度快的程序代码。

如果给变量或函数加上类型限定符(如__near 或__far)，那么编译器将按照限定符的要求来对相应的变量或函数进行编译。

[例子]

输入：C 源代码(给变量加上类型限定符)

```
__far int i;
.
.
.
void main(void)
{
```

```
    i = 123;
}
```

以“小模式”来编译，输出：汇编代码(节选)

```
        .SECTION        DATA_test, DATA, ALIGN=2
FAR_DATA_S:

        .ALIGN  2
        .GLOBAL _i
_i:                                             /* i 为远程数据 */
        .RES.B  2
FAR_DATA_E:
    .
    .
    .
    MOV     A, #bnksym _i
    MOV     ADB, A                              /* 设好远程数据 i
的段寄存器(附加段寄存器 ADB) */
    MOV     A, #123
    MOVW    ADB:_i, A                           /* i 为远程数据 */
```

输入：C 源代码(给函数加上类型限定符)

```
__far void func2(void)
{
}
```

以“小模式”来编译，输出：汇编代码(节选)

```
;-------begin_of_function
        .GLOBAL _func2
_func2:
        LINK    #0
        UNLINK
        RETP                                    /* 远程返回 */
```

如果不使用 ROM 镜像功能，那么为了访问 ROM 中的数据，则必须选中“-ramconst”选项(该编译选项位于 Project->Setup Tool Option…->C Compiler-> Target Depend->Const variable in RAM)。

❍ 中模式

在该模式下，程序中的代码放在 24 位地址空间(16M)的代码段内，数据放在 16 位地址空间(64k)的数据段内。在该模式下，代码段使用远（far）指针；比如，函数调用和返回对应的汇编代码是 CALLP 和 RETP（二者都使用的是 24 位的地址指针，可参见 16LX Programming Manual 的第 9 章）；而数据段使用近程指针（near）。该种编译模式适用于大代码量，小数据量的大程序。

❍ 紧凑模式

在该模式下，程序中的代码放在 16 位地址空间(64k)的代码段内，数据放在 24 位地址空间(16M)的数据段内。在该模式下，代码段使用近指针（near）；而数据段内的指针是远程的（far）；比如，对一个变量（未加上类型限定符__near 或__far）的引用，编译器生成的汇编代码一般是，先设置好该远程数据使用的段寄存器（一般是附加段寄存器 ADB），然后在对该远程数据引用时要加上段寄存器的前缀。

[例子]

```
MOV     A, #bnksym _i
MOV     ADB, A                          /* 设好远程数据 i 的段寄存器(附加段寄存器 ADB) */
MOV     A, #123
MOVW    ADB:_i, A                       /* i 为远程数据，引用 i 时加上了段寄存器的前缀 */
```

❍ 大模式

在该模式下，程序中的代码放在 24 位地址空间(16M)的代码段内，数据放在 24 位地址空间(16M)的数据段内。在该模式下，指针都是远程的（far）。

对于其它内存模式，象小模式一样，如果给变量或函数加上类型限定符(如__near 或__far)，那么编译器将按照限定符的要求来对相应的变量或函数进行编译。

■ 边界对齐

下面表格中说明编译器支持的各种数据类型占用的存储空间及边界对齐的情况。

变量类型	占用空间[字节]	边界对齐[字节]
Char	1	1
Signed char	1	1
Unsigned char	1	1
Short	2	2
Unsigned short	2	2
Int	2	2
Unsigned int	2	2
Long	4	2
Unsigned long	4	2
Float	4	2
Double	8	2

Long double	8	2
Near pointer/address	2	2
Far pointer/address	4	2
Structure/union	在后面解释	在后面解释

■ 简单数据在内存中的存储格式

如上所述，“char”类型数据占用 1 字节的存储器长度。

“int”和“short”类型数据占用存储器长度为 2 个字节（16 位）。举例说明数据的存放顺序，一个值为 0x1234 的“int”类型数据，存放格式如下：

地址	+0	+1
内容	0x34	0x12

“long”类型数据占用存储器长度为 4 个字节（32 位）。举例说明数据的存放顺序，一个值为 0x12345678 的“long”类型数据，存放格式如下：

地址	+0	+1	+2	+3
内容	0x78	0x56	0x34	0x12

“float”类型数据占用存储器长度为 4 个字节（32 位），它在存储器中按照 ANSI/IEEE Standard 754-1985 标准存放。举例说明数据的存放顺序，一个值为-1.35（用十六进制表示为 0xBFACCCCD）的"float"类型数据，存放格式如下：

地址	+0	+1	+2	+3
内容	0xCD	0xCC	0xAC	0xBF

“double”类型数据占用存储器长度为 8 个字节（64 位），它在存储器中按照 ANSI/IEEE Standard 754-1985 标准存放。举例说明数据的存放顺序，一个值为 24.6（用十六进制表示为 0x403899999999999A）的“double”类型数据，存放格式如下：

地址	+0	+1	+2	+3	+4	+5	+6	+7
内容	0x9A	0x99	0x99	0x99	0x99	0x99	0x38	0x40

注：浮点数的具体数据格式在下面解说。

■ 位组数据在内存中的存储格式

下面说明位组数据占用的存储空间和其边界的对齐。

连续的位组数据被紧挨着存放在相同存储单元，而不管给位组数据作限定的数据类型，从 LSB（最低有效位，Least Significant Bit）向 MSB（最高有效位，Most Significant Bit）存放。举例如下：

[例子]

```
struct tag1 {
    int A:10;
```

```
    short B:3;
    char C:2;
};
```

15(MSB)　　13　　10　　0(LSB)

Unoccupied	C	B	A

编译器在编译时会试图使位组数据连续存放，但是如果位组数据连续存放跨越了位组数据类型的边界，那么编译器就把位组数据存放在适合位组数据类型的下一个区域。就下面的例子来说明：如果使变量 B 紧挨着变量 A 连续存放，存放的位置是从 位 12－位 17，也就是说从字节偏移 1 的 bit4 开始存放到字节偏移 2 的 bit0，因为变量 B 的边界是 2 字节（字节偏移 0 和字节偏移 1 对于变量 B 来说，属于一个边界区域，字节偏移 2 和字节偏移 3 属于下一个边界区域），所以编译器不能把变量 B 紧挨着变量 A 存放。同样道理，编译器不能把变量 C 紧挨着变量 B 存放。

[例子]

```
struct tag2 {
    long int A:12;     /* 4-byte boundary data */
    short B:5;         /* 2-byte boundary data */
    char C:5;          /* 1-byte boundary data */
};
```

31(MSB)　28　24　　21　　16　　12　　0(LSB)

Unoccupied	C	Unoccupied	B	Unoccupied	A

如果一个位组变量的位长度为 0，那么特别地编译器就把它存放在下一个适合该位组变量所属的数据类型的下一个区域。就下面的例子来说明：变量 B 之后的那个位组变量的位长度为 0，同时其数据类型是整型（以 2 字节为边界），因此该变量被放在偏移为 2 的变量位置上（即变量 C 的位置上）；然后，变量 C 可以紧挨着位长度为 0 的那一变量存放，相当于变量 C 和长度为 0 的那一变量的存放位置重叠起来了。

[例子]

```
struct tag3 {
    int A:5;
    int B:5;
    int :0;
    int C:6;
};
```

15(MSB)　　10　　6　5　　0(LSB)

Unoccupied	B	A
Unoccupied		C

■ 结构/联合数据在内存中的存储格式

下面说明结构/联合变量的数据长度（占用的存储空间大小）和其边界的对齐。

[例子]

struct st1 { char a; } t1;　　→　sizeof(t1) = 1 BYTE
struct st2 { short a; } t2;　　→　sizeof(t2) = 2 BYTE
struct st3 { int a; } t3;　　→　sizeof(t3) = 2 BYTE
struct st4 { char A;　short B; } t4;　　→　sizeof(t4) = 4 BYTE
struct st5 { char A;　int B; } t5;　　→　sizeof(t5) = 4 BYTE

结构/联合的成员变量的数据存放顺序举例如下：

[例子]

对于 struct st4 { char a;　short b; } t4;
或　struct st5 { char a;　int b; } t5;　　两者的数据存放顺序是一样的：

15(MSB)　　8　　0(LSB)

Unoccupied	A
B	

[例子]

对于嵌套的结构/联合变量的数据长度和数据存放顺序是一样的：

```
struct st6 {
    char A;
    struct st7 {
        short A;
        char B;
    } t7;
} t6;
```

sizeof(t7) = 4 BYTE
sizeof(t6) = 6 BYTE

15(MSB)　　8　　0(LSB)

Unoccupied	A
t6.A	
Unoccupied	t6.B

二. 与汇编语言程序的接口

■ 编译器对函数名和变量名的处理

为了使汇编语言的源程序能够调用 C 语言源程序，编译器必须按照某种规则来对 C 语言源程序进行编译，生成的汇编输出文件中的变量名标号和函数名标号才能被其它汇编源程序引用，以达到汇编程序调用 C 源程序的目的 。

C 源程序中的对应值	C 编译器生成的标号
函数名	_函数名
外部变量名	_外部变量名
静态变量名	LI_no
局部变量名	-
形式参数名	-
字符串，及其派生类型	LS_no
自动变量初始值	LS_no
（跳转的）目标地址标号	L_no

注意：“no”是 C 编译器内部产生的数字。

[例子]

函数名	void func1(void)	_func1

外部变量名	int i;
	void main(void)
	{
	}
		_i

静态变量名	void main(void)
	{
	　static int i;
	}
		LI_1（注 1）

字符串，及其派生类型	void main(void)
	{
	　char str[]="hello";
	}
		LS_0（注 2）

自动变量初始值		

（跳转的）目标地址标号	int i,j;
	i=1;
	if(i!=0)
	
	j=10;
	
	else
	j=20;
		MOVW A, @RW3+-4
		BEQ L_23
		
		MOVN A, #10
		MOVW @RW3+-2, A
		BRA L_24
		
		L_23:
		MOV A, #20
		MOVW @RW3+-2, A
		
		L_24:
		（注 3）

注 1：在这里静态变量“i”编译后对应的标号为“LI_1”，即 C 编译器内部产生的数字为“1”，在其它场合下，编译器内部产生的可能是其它数字。

注 2：标号“LS_0”处存放的是字符串"hello"，而字符串变量“str”是局部变量，因而是不产生标号的（局部变量实际上是存放在堆栈中）。

注 3：C 源代码中有判断语句（或其它引起分支/跳转语句）时，编译器要生成跳转的目标地址标号。在本例中是“L_23”和“L_24”，其中“L_”后面的数字是 C 编译器内部产生的数字。

说明：如果一个变量是外部静态变量，则它适用于静态变量的规则。

三. SECTION

■ SECTION 结构

下面的表格中说明 C 编译器可能生成的各类 SECTION 及其含义。当一个 section 名是用 24 位地址的方式来存取时，则其 section 名改为（下表所示的）section 名+“_模块名”。C 编译器把源文件名当作模块名。如果编译时用编译选项“-s”来更改 section 名，那么编译选项“-s”规定的 section 名得到应用。

序号	Section 类型	Section 名	类型	边界对齐[字节]	写入	初始值
1	Code section	CODE	CODE	2	禁止	有
2	Initialized section	INIT	DATA	2	允许	无
3	Initial value of INIT	DCONST	CONST	2	禁止	有
4	Constant section	CONST	CONST	2	禁止	有
5	RAM area of CONST	CINIT	DATA	2	禁止	无
6	Data section	DATA	DATA	2	允许	无
7	Initialized direct section	DIRINIT	DIR	2	允许	无
8	Initial value of DIRINIT	DIRCONST	DIRCONST	2	允许	有
9	Direct section	DIRDATA	DIR	2	允许	无
10	I/O section	IO	IO	2	允许	无
11	Vector section	INTVECT	CONST	2	禁止	有

各种 section 的用途和其与 C 语言的关系解说如下：

(1) Code section

该 section 存放机器代码。C 语言的函数体对应于该 section。其缺省的 section 名是“CODE”。

[例子]
输入：C 源代码，源文件名为 test.c

```
void main(void)
{
}
```

以“小模式”或“紧凑模式”来编译，输出：汇编代码(节选)

```
        .PROGRAM    test
        .SECTION    CODE, CODE, ALIGN=1
                    /* section 名是“CODE”  */
_main:
        LINK    #0
        UNLINK
```

```
        RET
```

以“中模式”或“大模式”来编译（），输出：汇编代码(节选)

```
        .PROGRAM        test

        .SECTION        CODE_test, CODE, ALIGN=1
                        /* section 名是 CODE_test */
                        /*  此处模块名是“test”  */
_main:
        LINK    #0
        UNLINK
        RETP            /*  远程返回：24 位地址的方式  */
```

(2) Initialized section

该 section 存放一些声明时就赋予初值的变量。在 C 语言中，该 section 对应于在声明时赋予初值的外部变量、外部静态变量和内部静态变量，且这些变量未加“CONST”限定符。其缺省的 section 名是“INIT”。

[例子]
输入：C 源代码

```
int i=0x11;
static int j=0x22;

void main(void)
{
   static int k=0x33;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
        .SECTION        INIT, DATA, ALIGN=2
                        /* section 名是“INIT”  */
        .ALIGN  2
LI_2:                                   /*  对应于变量 k */
        .RES.H  1
        .ALIGN  2
LI_1:                                   /*  对应于变量 j */
        .RES.H  1
        .ALIGN  2
        .GLOBAL _i
_i:                                     /*  对应于变量 i */
        .RES.H  1
```

(3) Initial value of INIT

一些声明时就赋予初值的变量，该 section 存放其初值（存放“INIT” section 中变量对应的初值）。该 section 被放置在 ROM 里面。必须使用 startup 文件（比如，使用随 C 编译器附带的 start905s.asm）里的初始化例程，把该

如，使用随 C 编译器附带的 start905s.asm）里的初始化例程，把该 section 里的初值传送到“INIT”section 中相应的变量里去。其缺省的 section 名是“DCONST”。

[例子]
（与上一例子相同）输入：C 源代码

```
int i=0x11;
static int j=0x22;

void main(void)
{
  static int k=0x33;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
.SECTION        DCONST, CONST, ALIGN=2
                /* section 名是“DCONST” */
.ALIGN  2
.DATA.H 51              /*  变量 k 的初值 0x33=51 */

.ALIGN  2
.DATA.H 34              /*  变量 j 的初值 0x22=34 */

.ALIGN  2
.DATA.H 17              /*  变量 i 的初值 0x11=17 */
```

(4) Constant section

该 section 存放一些声明时就赋予初值的“常量” 变量。在 C 语言中，该 section 对应于在声明时赋予初值的外部变量、外部静态变量和内部静态变量，且这些变量加有“CONST”限定符。其缺省的 section 名是“CONST”。

[例子]
输入：C 源代码

```
const int i=0x11;
const static int j=0x22;

void main(void)
{
  const static int k=0x33;
}
```

以“小模式”或“中模式”来编译，编译时未使用“**-ramconst**”编译选项；输出：汇编代码(节选)

```
        .SECTION        CONST, CONST, ALIGN=2
                        /* section 名是“CONST” */
        .ALIGN  2
```

```
LI_2:
        .DATA.H 51                      /* 变量 k   0x33=51 */
        .ALIGN  2
LI_1:
        .DATA.H 34                      /* 变量 j   0x22=34 */
        .ALIGN  2
        .GLOBAL _I
_i:
        .DATA.H 17                      /* 变量 i   0x11=17 */
```

(5) RAM area of CCONST

该 section 存放一些声明时就赋予初值的“常量” 变量，其前提是：如果使用的 CPU 类型不支持 ROM 镜像的功能，编译时使用“**-ramconst**”编译选项才会生成该 section。必须使用 startup 文件（比如，使用随 C 编译器附带的 start905s.asm）里的初始化例程，把 “CONST”section 里的初值传送到该 section 中相应的变量里去。其缺省的 section 名是“CINIT”。要注意的是：虽然对“常量” 变量（加“CONST”限定符）进行赋值，在编译时会有警告提示，但缺省情况下进行软件模拟（“Simulator Debugger” ），不会发生运行错误[例子]

输入：C 源代码

```
const int i=0x11;
const static int j=0x22;

void main(void)
{
  const static int k=0x33;
  i =0xff;                          /* 这一行，在编译时有警告提示，
                                       但在软件模拟时不发生运行错误 */
}
```

以“小模式”或“中模式”来编译，编译时使用“**-ramconst**”编译选项；输出：汇编代码(节选)

```
        .SECTION        CONST, CONST, ALIGN=2

        .ALIGN  2
        .DATA.H 51
        .ALIGN  2
        .DATA.H 34
        .ALIGN  2
        .DATA.H 17
```

```
                .SECTION        CINIT, DATA, ALIGN=2
                                /* section 名是“CINIT”  */
                .ALIGN  2
LI_2:
                .RES.H  1
                .ALIGN  2
LI_1:
                .RES.H  1
                .ALIGN  2
                .GLOBAL _i
    _i:
                .RES.H  1
```

(6) Data section

该 section 存放一些声明时不赋予初值的变量。在 C 语言中，该 section 对应于在声明时不赋予初值的外部变量、外部静态变量和内部静态变量，不管这些变量上是否加“CONST”限定符。其缺省的 section 名是“DATA”。

[例子]
输入：C 源代码

```
int i;
static int j;
void main(void)
{
   static int k;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
                .SECTION        DATA, DATA, ALIGN=2
                                /* section 名是“DATA”  */
                .ALIGN  2
LI_2:
                .RES.B  2
                .ALIGN  2
LI_1:
                .RES.B  2
                .ALIGN  2
                .GLOBAL _i
    _i:
                .RES.B  2
```

(7) Initialized direct section

该 section 存放一些声明时赋予初值的变量，且这些变量加有“**__direct**”类型限定符。在 C 语言中，该 section 对应于在声明时赋予初值的外部变量、外

部静态变量和内部静态变量，这些变量加有“__direct”类型限定符但未加“CONST”类型限定符。其缺省的 section 名是“DIRINIT”。

[例子]
输入：C 源代码

```
__direct int i=0x12;
__direct static int j=0x34;

void main(void)
{
  __direct static int k=0x56;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
        .SECTION        DIRINIT, DIR, ALIGN=2
                        /* section 名是“DIRINIT” */
        .ALIGN  2
LI_2:
        .RES.H  1
        .ALIGN  2
LI_1:
        .RES.H  1
        .ALIGN  2
        .GLOBAL _i
_i:
        .RES.H  1
```

(8) Initial value of DIRINIT

该 section 存放“DIRINIT”section 中变量对应的初值。该 section 被放置在 ROM 里面。必须使用 startup 文件（比如，使用随 C 编译器附带的 start905s.asm）里的初始化例程，把该 section 里的初值传送到“DIRINIT” section 中相应的变量里去。其缺省的 section 名是“DIRCONST”。

[例子]
输入：C 源代码

```
__direct int i=0x12;
__direct static int j=0x34;

void main(void)
{
  __direct static int k=0x56;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
        .SECTION        DIRCONST, DIRCONST, ALIGN=2
```

```
                                /* section 名是“DIRCONST” */
        .ALIGN  2
        .DATA.H 86              /* 变量 k 的初值 0x56=86 */
        .ALIGN  2
        .DATA.H 52              /* 变量 j 的初值 0x34=52 */
        .ALIGN  2
        .DATA.H 18              /* 变量 i 的初值 0x12=18 */
```

(9) Direct section

该 section 存放一些声明时没有赋予初值的变量，且这些变量加有“**__direct**”类型限定符。在 C 语言中，该 section 对应于在声明时未赋予初值的外部变量、外部静态变量和内部静态变量，这些变量加有“**__direct**”类型限定符而不管这些变量上是否加有“CONST”限定符。其缺省的 section 名是“DIRDATA”。

[例子]
输入：C 源代码

```
__direct int i;
__direct static int j;

void main(void)
{
  __direct static int k;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
        .SECTION        DIRDATA, DIR, ALIGN=2
                        /* section 名是“DIRDATA” */
        .ALIGN  2
LI_2:
        .RES.B  2
        .ALIGN  2
LI_1:
        .RES.B  2
        .ALIGN  2
        .GLOBAL _i
_i:
        .RES.B  2
```

(10) I/O section

该 section 存放那些加有“**__io**”类型限定符的变量。在 C 语言中，该 section 对应于那些加有“**__io**”类型限定符的外部变量、外部静态变量和内部静态变量，不管这些变量上是否加有“CONST”限定符。其缺省的 section 名是“IO”。

[例子]
输入：C 源代码

```
__io int i;
__io static int j;

void main(void)
{
  __io static int k;
}
```

以“小模式”或“中模式”来编译，输出：汇编代码(节选)

```
        .SECTION        IO, IO, ALIGN=2
                        /* section 名是“IO” */
        .ALIGN  2
        .GLOBAL _i
_i:
        .RES.B  2
        .ALIGN  2
LI_1:
        .RES.B  2
        .ALIGN  2
LI_2:
        .RES.B  2
```

(11) Vector section

该 section 存放中断向量表。使用富士通 C 语言中的扩展指令“**#pragma intvect**”，就可以产生该 section。其缺省的 section 名是“INTVECT”。

[例子]

输入：C 源代码

```
#pragma intvect timerbase_int 36

void main(void)
{
}

__interrupt void timerbase_int(void)
{
}
```

以“小模式”或“紧凑模式”来编译，输出：汇编代码(节选)

```
        .SECTION        INTVECT, DATA, LOCATE=H'FFFF6C
        .ORG    H'FFFF6C
        .DATA.L _timerbase_int
```

四. 函数调用接口

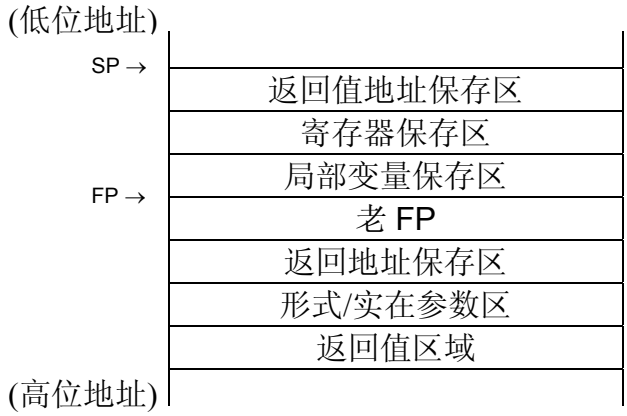
函数间调用的堆栈、参数和寄存器所遵循的规则称为标准链接规则。当使用标准链接规则时，一个以 C 语言编写的模块可以与一个以其它语言编写的模块（如以汇编语言编写）进行互相调用。

■ 函数调用接口有如下几方面内容

- 堆栈结构
- 函数参数
- 函数参数扩展格式
- 调用过程
- 寄存器的使用
- 函数返回值

■ 堆栈结构

下图显示标准链接规则下的堆栈结构



(1) 返回值地址保存区

对于返回值类型是结构/联合/double 或 long double 的函数，其返回值区域的首地址存放在此处。当返回值类型是如上所述类型时，调用函数把返回值区域的首地址存放在累加器 AL 里，然后把 AL 传递给被调用函数。被调用函数知道 AL 中存放着返回值区域的首地址，如果被调用函数认为必须把 AL 保存起来（比如，在被调用函数里又把 AL 派作了其它用途），则把 AL 的数值保存在该处，否则该区域不存在。

(2) 寄存器保存区

如果在被调用函数里用到了某些寄存器，并且寄存器要保护起来，则

相应的寄存器值被保存在该区域；否则该区域不存在。

(3) 局部变量保存区

被调用函数里定义的局部变量使用该区域。

(4) 老 FP

调用函数的帧指针(RW3，在引用局部变量时要用到帧指针)保存在此处。

(5) 返回地址保存区

被调用函数的返回地址保存在此。被调用函数运行结束后，CPU 从该区域把返回地址读出并赋值给 PC，然后程序从返回地址开始继续运行。

(6) 形式/实在参数区

此区域用作参数的传递。从调用函数的角度来说，调用函数把实在参数放入该区域；从被调用函数的角度来说，对其形式参数的引用将引用到该区域。

形式参数简称为形参，实在参数简称为实参。关于函数参数将在下面的章节作更详细说明。

(7) 返回值区域

对于返回值类型是结构/联合/double 或 long double 的函数，其返回值区域由调用函数保护起来。被调用函数假定该区域被保护在堆栈中，因此，如果该区域被保护在堆栈外，则造成的结果将难以预料。

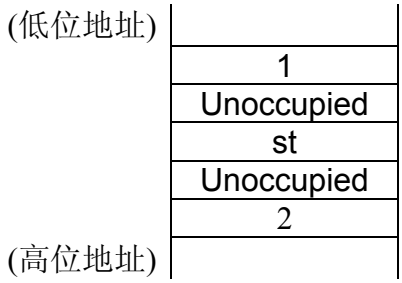
编译器把该区域和实在参数区域进行重叠放置，其目的是为了提高在某些情况下目标代码的效率。因此，返回值为该类型的被调用函数，在该区域存放返回值时，必须先存放高位地址的数据，然后再存放低位地址的数据；并且，要在对所有的形式参数都引用过之后，才能对该区域执行写入操作。

■ 函数参数

向被调用函数传递参数是通过堆栈来进行的。不管对于一个小于 2 字节长的参数或者参数的长度不是 2 的倍数，则这些参数在堆栈中所占用的数据长度，都以 2 的倍数向上取整。形参/实参由调用函数来分配和释放。

下面的例子说明函数参数的具体放置情况。

```
struct A{char a; }st;
extern void sub(char,struct A,int};
sub(1,st,2);
```



■ 函数参数扩展格式

当把函数参数存放到堆栈时，其类型要先转换为相应的扩展类型。当从被调用函数返回后，由调用函数来把函数参数释放掉。

下面的表格说明函数参数的扩展格式。

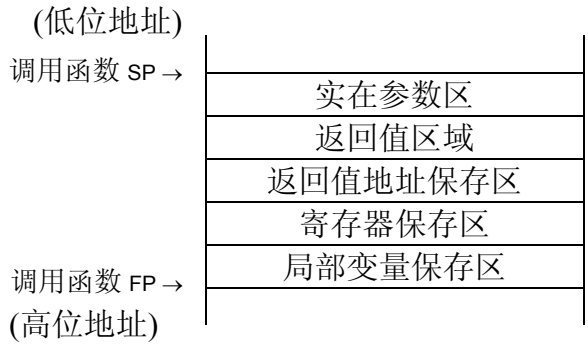
实在参数类型	扩展类型 *1	占用堆栈大小 [字节]
char	int	2
signed char	int	2
unsigned char	int	2
short	无扩展	2
unsigned short	无扩展	2
int	无扩展	2
unsigned int	无扩展	2
long	无扩展	4
unsigned long	无扩展	4
float	double	8
double	无扩展	8
long double	无扩展	8
near pointer/address	无扩展	2
far pointer/address	无扩展	4
structure/union	*2	*2

*1：函数参数的扩展类型是指，当没有指明形式参数的类型时的扩展；如果在函数声明时指明了函数参数类型，则指明的形式参数类型得到应用。

*2：对于一个小于 2 字节长的参数或者参数的长度不是 2 的倍数，这些参数在堆栈中所占用的数据长度，都以 2 的倍数向上取整。

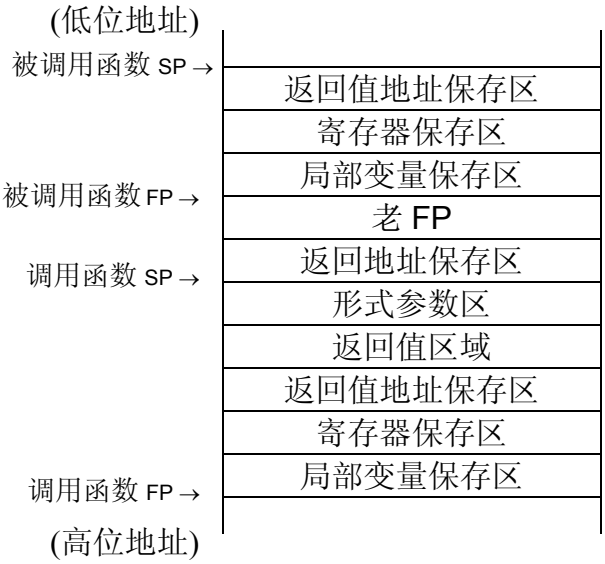
■ 调用过程

下图说明调用时的堆栈情况（按照标准链接规则进行编译）。



被调用函数把调用函数的帧指针（RW3）保存在堆栈中，然后把堆栈指针赋值给帧指针，该新的帧指针作为被调用函数的帧指针。接着，在堆栈中保留出被调用函数的局部变量的占用空间，和往堆栈中保存调用函数的寄存器以及返回值地址。

下图说明由被调用函数创建的堆栈情况（按照标准链接规则进行编译）。



■ 寄存器的使用

在被调用函数里，可能使用了一些寄存器，但被调用函数保证在返回时，把下列寄存器恢复到调用函数原来的值上去。

• 通用寄存器 RW0-RW3，RW6，RW7 和 USP（SSP）

上述功能的实现途径是，被调用函数把相应的寄存器保存到寄存器保存区里。对于在被调用函数里没有使用的寄存器，是不会被保存的。此外，如果在被调用函数里使用嵌入汇编语句而改变了上述寄存器，就无法保证上述寄存器的值能够恢复。

下表说明在函数调用和返回时寄存器的的使用规则

寄存器	在调用时（用途）	在返回时（用途）
A	返回值区域的首地址	返回值*
RW0-RW2	未规定	保持调用时的数值
RW3	帧指针	保持调用时的数值
RW4 和 RW5	未规定	未规定
RW6 和 RW7	未规定	保持调用时的数值
USP（SSP）	堆栈指针	保持调用时的数值

说明：对于没有返回值的被调用函数或者返回值类型是 结构/联合/double/long double 的情况，没有规定寄存器“A”的用途。

■ 函数返回值

下表说明在标准链接规则下的函数返回值接口。

返回值类型	返回值接口
void	无
char	AL
signed char	AL
unsigned char	AL
short	AL
unsigned short	AL
int	AL
unsigned int	AL
long	A
unsigned long	A
float	A
near pointer/address	AL
far pointer/address	A
double	AL*
long double	AL*
structure/union	AL*

注意：累加器“A”的字长是 32 位（4 字节）的，当函数的返回值类型是结构/联合/double/long double 时，要么有可能超过累加器的字长，要么必定超过累加器的字长，因此不能通过累加器来传递返回值。调用函数把返回值区域的首地址存放在累加器 AL 里，然后把 AL 传递给被调用函数。被调用函数知道 AL 中存放着返回值区域的首地址，如果被调用函数认为必须把 AL 保存起来（比如，在被调用函数里又把 AL 派作了其它用途），则把 AL 的数值保存在返回值地址保存区；因而相当于通过 AL 来传递参数。

五. 中断函数调用接口

通过给函数加上“__interrupt”类型限定符，就成为中断函数。如果用普通的方法直接调用中断函数，而不是由于发生中断导致中断函数被调用，则无法保证这种直接调用（中断函数）的正确性和安全性。在中断函数里进行（普通）函数调用遵循前述说明的标准链接规则。

■ 中断函数调用接口有如下几方面内容

- 堆栈结构

  中断发生后，堆栈被切换为中断堆栈。
- 函数参数

  中断函数不带任何参数。
- 调用过程

  中断发生后，通过查中断向量表，而触发中断程序运行。
- 寄存器的使用

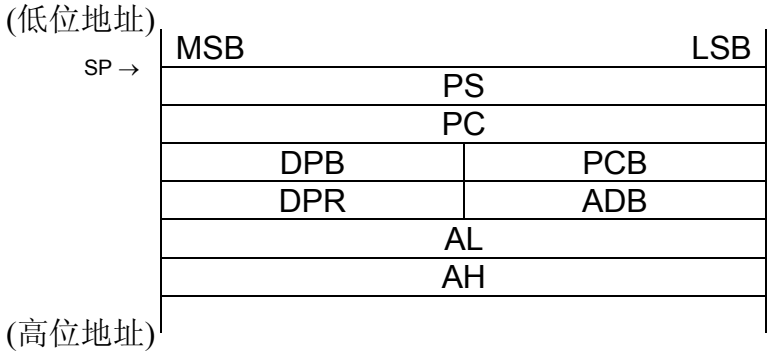
  编译器保证，在中断函数里所有（使用到的）寄存器都被保护起来的；除非，在中断函数里使用嵌入汇编语句而改变了寄存器的值。
- 函数返回值

  中断函数没有返回值。

■ 中断调用的堆栈结构

中断发生后，堆栈指针被切换为中断堆栈指针（中断函数以系统堆栈指针“SSP”作为堆栈指针）。

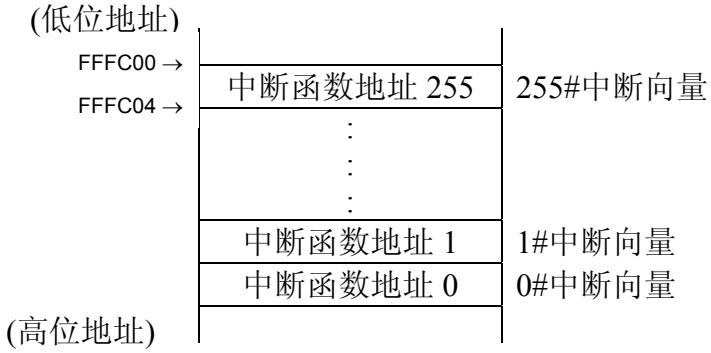
下图说明中断发生后，中断调用的堆栈结构。



■ 中断调用过程

中断发生后，通过查中断向量表，而触发中断程序运行。如果用普通的方法直接调用中断函数，而不是由于发生中断导致中断函数被调用，则无法保证这种直接调用（中断函数）的正确性和安全性。

下图是中断向量表的示例说明。



六. C 编译器的限制

下表说明富士通 C 编译器在编译时受到的限制。其中还把 ANSI 标准的最低要求罗列出来以作对照。

序号	项目	ANSI 标准	编译器
1	一个复杂声明，如自嵌套的结构（结构的成员变量类型是该结构自身）或一般嵌套结构，最大的嵌套深度 （Count of nesting levels for a compound statement, repetition control structure, and selection control structure）	15	∞
2	预处理器的条件编译最大嵌套深度	8	∞
3	结构/联合声明中，指针、数组和函数以及这些的任意组合，允许的最大数目 （ Count of pointers, arrays, and function declarators (any combinations of these) for qualifying one arithmetic type, structure type, union type, or incomplete type in a declaration）	12	∞
4	在一个声明中（变量或函数声明）加括号“(...)”进行嵌套的最大深度 （ Count of nests provided by parentheses for one complete declarator）	31	∞
5	一个表达式中可嵌套（加括号“(...)” ）的最大深度 （Count of nest expressions provided by parentheses for one complete expression）	32	∞
6	有效的局部变量名和宏名 字符长度	31	∞
7	有效的外部变量名 字符长度	6	254
8	一个源程序中的外部变量的最大数目	511	∞
9	功能块（{...}）里可定义的（局部）变量的最大数目 （Count of identifiers having the block valid range in one block）	127	∞
10	一个源程序中可定义的宏的最大数目	1024	∞
11	函数声明（定义）时形式参数的最大数目	31	∞
12	函数调用时实在参数的最大数目	31	∞
13	宏声明（定义）时形式参数的最大数目	31	∞
14	宏调用时实在参数的最大数目	31	∞
15	C 语言语句一行中最多能写的字符数	509	∞
16	字符串中字符的最大数目	509	∞
17	算术表达式中（被运算符分隔开的）算子的最大数目 （Count of bytes of one arithmetic unit）	32767	65535
18	嵌套包含头文件的最大深度	8	252
19	“switch”语句中“case”分支的最大数目（不包括嵌套的“switch”语句）	257	∞
20	结构或联合中成员变量的最大数目	127	∞
21	枚举变量中枚举常数的最大数目	127	∞
22	结构或联合嵌套的最大深度	15	∞

说明：虽然编译器可以辨识的外部变量最大字符数是没有限制的，但编译器仅仅输出了 255 个字符到中间汇编文件中（注意：第一个字符必定为下划线“_”）；所以，如果两个外部变量的前 254 个字符是相同的，在汇编时将发生错误。

注意：上述表格中的∞表示，富士通 C 编译器 fcc907s 仅受到可用内存大小的限制。

第三节　C 编译器的特殊扩充

一. 嵌入汇编

在 C 源程序中可通过下列两种方式来实现嵌入汇编：

- 一个是通过使用 **asm** 表达式
- 另一个是通过 **Pragma** 预处理指令

■ 使用 asm 表达式

一般格式：

```
__asm(字符串序列);
```

说明：**asm** 表达式，可以在函数内和函数外使用。当在函数内使用 **asm** 表达式时，就在表达式的位置上直接展开为汇编语句。当在函数外使用 **asm** 表达式时，它被扩展成独立的 SECTION。因此，如果在函数外使用 **asm** 表达式，一定要用 SECTION 的预处理指令来为其单独定义一个 SECTION，否则难以保证相应操作的正确性。

在函数内使用 **asm** 表达式，必须由用户自己来保存和恢复寄存器。对于累加器是个例外，因为编译器会对累加器进行保存和恢复，所以可以随意地使用累加器。

“-O”的编译优化选项仍然适用于使用 **asm** 的表达式。

[例子]
输入：扩充的 C 源代码

```
/* When written inside the function */
extern int temp;
void sample(){
    __asm(" MOVN A, #1");        /* 注意：第一个双引号右边有一个空格 */
    __asm(" MOVW _temp, A");
}
/* When written outside the function */
__asm(" .SECTION DATA, DATA, ALIGN=2");
__asm(" .ALIGN 2");
```

```
__asm(" .GLOBAL _a");
__asm(" _a: .RES.B 2");
```

输出：汇编代码(节选)

```
    .SECTION   CODE, CODE, ALIGN=1
;-------begin_of_function
    .GLOBAL    _sample
_sample:
    LINK #0
    MOVN A, #1
    MOVW _temp, A
    UNLINK
    RET
    .SECTION   DATA, DATA, ALIGN=2
    .ALIGN 2
    .GLOBAL _a
_a: .RES.B 2
```

■ 使用 pragma 预处理指令

一般格式：

```
#pragma asm
      汇编语句
#pragma endasm
```

说明：**pragma** 预处理指令可以在函数内和函数外使用。当在函数内使用时，在**#Pragma asm** 和**#Pragma endasm** 预处理指令之间的汇编语句，直接被转换成（拷贝）汇编代码。当在函数外使用时，它被扩展成独立的 SECTION。因此，如果在函数外使用 **pragma** 预处理指令，一定要用 SECTION 的预处理指令来为其单独定义一个 SECTION，否则难以保证相应操作的正确性。在函数内使用 **pragma** 预处理指令，必须由用户自己来保存和恢复寄存器。对于累加器是个例外，因为编译器会对累加器进行保存和恢复，所以可以随意地使用累加器。

“-O”的编译优化选项仍然适用于使用 **pragma** 的预处理指令。

[例子]

输入：扩充的 C 源代码

```
/* When written inside the function */
void sample(){
#pragma asm
    MOVN A, #1
```

```
    MOVW _temp, A
#pragma endasm
}
/* When written outside the function */
#pragma asm
    .SECTION DATA, DATA, ALIGN=2
    .ALIGN 2
    .GLOBAL _a
_a: .RES.B 2
#pragma endasm
```

输出：汇编代码(节选)

```
    .SECTION    CODE, CODE, ALIGN=1
;-------begin_of_function
    .GLOBAL    _sample
_sample:
    LINK #0
    MOVN A, #1
    MOVW _temp, A
    UNLINK
    RET
    .SECTION DATA, DATA, ALIGN=2
    .ALIGN 2
    .GLOBAL _a
_a: .RES.B 2
```

二. 中断相关的函数

有下列 5 个中断控制函数/伪指令：

- **禁止中断函数**
- **允许中断函数**
- **中断级别设置函数**
- **中断函数的声明/定义**
- **中断向量表生成伪指令**

■ 禁止中断函数

一般格式：

```
void __DI(void);
```

说明：清零中断允许标志位，从而禁止中断。

[例子]
输入：扩充的 C 源代码

```
__DI();
```

输出：汇编代码

```
AND   CCR, #191
```

■ 允许中断函数

一般格式：

```
void __EI(void);
```

说明：置位中断允许标志位，从而允许中断。

[例子]
输入：扩充的 C 源代码

```
__EI();
```

输出：汇编代码

```
OR     CCR, #64
```

■ 中断级别设置函数

一般格式：

```
void __set_il(int level);
```

说明：把中断级别设置为“level”级别

[例子]
输入：扩充的 C 源代码

```
__set_il(2);
```

输出：汇编代码

```
MOV   ILM, #2
```

■ 中断函数的声明/定义

一般格式：

```
__interrupt void Interrupt function (void) { ... }
和
extern __interrupt void Interrupt function (void);
```

说明：中断函数的声明/定义是通过在函数前面加“__interrupt”类型限定符来实现。

因为中断函数的运行是由中断来触发的，所以，中断函数是不带参数的，也不返回任何值。如果用普通的方法直接调用中断函数，而不是由于发生中断导致中断函数被调用，则无法保证这种直接调用（中断函数）的正确性和安全性。

[例子]
输入：扩充的 C 源代码

```
__interrupt void sample(void){ ... }
```

输出：汇编代码(节选)

```
_sample:
   LINK #0
   ...
   UNLINK
   RETI
```

■ 中断向量表生成伪指令

一般格式：

#pragma intvect Interrupt function name Vector number [Mode value] 和 #pragma defvect Interrupt function name

说明：#pragma intvect 伪指令为中断函数生成相应的中断向量表。

#pragma defvect 伪指令为，没有使用#pragma intvect 伪指令的其它中断向量，生成缺省的中断向量表。

中断向量表被放置在一个单独的名为“INTVECT”的 SECTION 里。

当使用#pragma defvect 伪指令时，就产生所有的中断向量，而无一遗漏。因此，如果使用了#pragma defvect 伪指令，所有的中断向量都必须在这一源程序里定义，否则中断向量会产生冲突，编译时会有警告。反之，如果没有使用#pragma defvect 伪指令，就可以在几个（两个或者更多）源程序里使用#pragma intvect 伪指令。

不能使用该功能在同一中断向量号上，定义不同的中断向量/中断函数。但是，在同一源程序里，在同一中断向量号上，分多次定义相同的中断向量/中断函数，并不会报错。

在中断向量号的位置上，只能使用整数常量（使用整数常量表达式，如“1+1”，都不行）。中断向量号的数目必须介于 0 到 255 之间。同样，在“mode”的位置上，也只能使用整数常量。

[例子]

输入：扩充的 C 源代码

```
#pragma intvect sample 1 0       /* 中断向量号为 1
                                    Mode 为 0 */
__interrupt void sample(void)
{
}
```

输出：汇编代码(节选)

```
        .SECTION        INTVECT, DATA, LOCATE=H'FFFFF8
        .ORG    H'FFFFF8
        .DATA.E _sample
        .DATA.B 0

        .SECTION        CODE, CODE, ALIGN=1
;-------begin_of_function
```

```
        .GLOBAL _sample
_sample:
        LINK    #0
        UNLINK
        RETI
```

三. I/O 变量的定义

一般格式：

```
__io Variable definition;
```

说明：一个引用 I/O 地址范围从 0x00 到 0xff 的 I/O 变量的声明/定义，可以通过在变量前面加“__io”类型限定符来实现。

由于为 I/O 区域（指 I/O 范围从 0x00 到 0xff）的存取设计了专用的机器指令，所以相应的源程序可以编译生成紧凑和高速的目标代码。为了定义一个地址范围高于 0xff 的变量，要使用“volatile”类型限定符。

加“__io”类型限定符而定义的 I/O 变量（在定义时）是不能赋予初值的。

如果定义的 I/O 变量类型是结构或联合，那么就认为其所有的成员变量都位于 I/O 区域中（成员变量不能超越出 I/O 区域的边界）。同时，不能单独把结构或联合的某个成员变量定义为 I/O 变量。

对于加“__io”类型限定符而定义的 I/O 变量，编译时总是假定其也加有“volatile”类型限定符（加了“__io”类型限定符的变量，就相当于也加了“volatile”类型限定符）。其例外是，编译时使用了“-K NOVOLATILE”编译选项，则加了“__io”类型限定符的变量，不能等效于也加了“volatile”类型限定符

[例子]

输入：扩充的 C 源代码

```
#pragma section IO=IOA,attr=IO, locate=0x10
__io int a;
void func(void)
{
  a=1;
}
```

输出：汇编代码(节选)

```
            .SECTION        IOA, IO, LOCATE=H'10
            .ALIGN  2
            .GLOBAL _a
_a:
            .RES.B  2
            .SECTION        CODE, CODE, ALIGN=1
;-------begin_of_function
            .GLOBAL _func
_func:
            LINK    #0
            MOVN    A, #1
            MOVW    I:_a, A
            UNLINK
            RET
```

四. direct 变量的定义

一般格式：

```
__direct Variable definition;
```

说明：。

[例子]
输入：扩充的 C 源代码

```
__direct int p;
void sample(void)
{
  p=1;
}
```

输出：汇编代码(节选)

```
        .SECTION        DIRDATA, DIR, ALIGN=2
        .ALIGN  2
        .GLOBAL _p
_p:
        .RES.B  2
        .SECTION        CODE, CODE, ALIGN=1
;-------begin_of_function
        .GLOBAL _sample
_sample:
        LINK    #0
        MOVN    A, #1
        MOVW    S:_p, A
        UNLINK
        RET
```

五. 近程/远程变量(或函数)的定义

一般格式：

```
__near Variable definition;
__far Variable definition;
```

说明：要给变量/函数设定一定的地址空间范围，需要给变量/函数加上“__near/__far”类型限定符。给变量/函数加上“__near”类型限定符，变量/函数就成为近程变量/函数，被放置在 16 位的地址空间里。给变量/函数加上“__far”类型限定符，变量/函数就成为远程变量/函数，可放置在 24 位的地址空间里里的任何位置。

如果在变量上未加“__near/__far”类型限定符，则在编译时，根据编译所选的内存模式（小模式、中模式、紧凑模式或大模式）来决定变量对应的地址空间。

局部变量不能加这些类型限定符。

当远指针(“far”)被强制类型转换为近指针(“near”)，其高 8 位被抛弃。当近指针(“near”)被强制类型转换为远指针(“far”)，段寄存器“DTB”的值被赋予远指针的高 8 位。

当局部变量的地址被赋予远指针，段寄存器“USB”或“SSB”的值被赋予远指针的高 8 位。但是，如果局部变量的地址先被赋予近指针，然后近指针再被强制类型转换成远指针，则段寄存器“DTB”的值被赋予远指针的高 8 位，这样就出错了。

如果在远程函数（加“__far”类型限定符）里调用了近程函数（加“__near”类型限定符），则这两个函数必须位于同一 SECTION 里。其原因是，调用远程函数时要设置“PCB”段寄存器，而调用近程函数时不设置“PCB”段寄存器。

变量段寄存器不能跨越段的边界，否则，编译生成的代码无法对其正确存取。

[例子]

输入：扩充的 C 源代码，源文件名为 test.c

```
__near int p;
__far int q;
void sample(void){
  p=1;
  q=2;
}
```

输出：汇编代码(节选)

```
        .SECTION        DATA_test, DATA, ALIGN=2
FAR_DATA_S:
        .ALIGN  2
        .GLOBAL _q
_q:
```

```
        .RES.B  2
        .SECTION        DATA, DATA, ALIGN=2
        .ALIGN  2
        .GLOBAL _p
_p:
        .RES.B  2
        .SECTION        DATA_test, DATA, ALIGN=2
FAR_DATA_E:
        .SECTION        CODE, CODE, ALIGN=1
;-------begin_of_function
        .GLOBAL _sample
_sample:
        LINK    #0
        MOVN    A, #1
        MOVW    _p, A
        MOV     A, #bnksym _q
        MOV     ADB, A
        MOVN    A, #2
        MOVW    ADB:_q, A
        UNLINK
        RET
```

六. 函数的内嵌扩展

一般格式：

```
#pragma inline Function name [, Function name ...]
```

说明：一般的函数调用都编译成“CALL”调用；但如果一个函数被象上面一样定义成内嵌函数，则对它的调用将编译成，原来的“CALL”调用换成该函数的函数体。相当于，把该函数的函数体（多行代码）嵌入到原来“CALL”调用处，即内嵌；也可看成，原来的“CALL”调用扩展成了多行代码的函数体，即扩展。

递归调用的函数不能使用内嵌扩展功能（实际上，用了该功能并不会出错，只是没有效果）。具有如下特点的函数也不能使用内嵌扩展功能：使用了嵌入汇编语句，具有结构/联合类型的函数参数，**setjmp** 函数（参见头文件 setjmp.h）及其它…

在一个源文件里可以使用多行内嵌扩展语句，内嵌扩展语句也可以用在条件编译里面，这里面所有的函数都是有效的内嵌函数（在条件编译的条件成立的前提下）。

内嵌扩展功能也可以通过使用“-x”编译选项来实现。

如果在编译时，没有使用“-O”优化编译选项，则内嵌扩展功能不起作用。

[例子]

输入：扩充的 C 源代码，编译时使用“-O 1”的优化编译选项。

```
#pragma inline func
int p;
void func(void);
void main(void)
{
  func();
}
void func(void)
{
  p=1;
}
```

输出：汇编代码（节选）

```
        .SECTION        CODE, CODE, ALIGN=1
;-------begin_of_function
        .GLOBAL _main
_main:

        MOVN    A, #1           /* 注意：这里不是 CALL    _func
*/
        MOVW    _p, A           /* 而是把函数_func 内嵌进来了 */
```

```
        RET
;-------begin_of_function
        .GLOBAL _func
_func:
        MOVN    A, #1
        MOVW    _p, A
        RET
```

七. 更改 SECTION 名

一般格式：

```
#pragma section DEFSECT[=NEWNAME][,attr=SECTATTR][,locate=ADDR]
```

说明：其功能是，把缺省 SECTION 名“DEFSECT”改名为“NEWNAME”，把 SECTION 类型改为“SECTATTR”，把该 SECTION 的首地址定位在 ADDR 上。

对于内存模式为大模式、紧凑模式(变量)和中模式(函数)下的变量和函数，以及远程变量和函数（加__far 类型限定符），可以给其对应的 SECTION 名前加上“FAR_”。

关于缺省 SECTION 名，可参见前面的章节(SECTION 结构)。关于各 SECTION 类型，还可参见汇编语言手册。

如果指定了 SECTION 的首地址，则在链接时不能再对其配置地址。

应该对同一个 SECTION 使用该功能最多改名一次；如果使用该功能对同一个 SECTION 改名多次，则只有最后一次改名有效。如果使用“-s”编译选项对同一个 SECTION 进行了改名，则“-s”编译选项进行的改名是有效的。

[例子]

输入：扩充的 C 源代码

```
#pragma section CODE=program,attr=CODE,locate=0xff1000
void main(void){}
```

输出：汇编代码

```
       .SECTION       program, CODE, LOCATE=H'FF1000

                      /* 缺省的 SECTION 名“CODE”改成了 */

                      /* “program” */

;-------begin_of_function
       .GLOBAL _main
_main:
       LINK    #0
       UNLINK
       RET
```

八. 寄存器区的设置

一般格式：

```
#pragma register(NUM)
#pragma noregister
```

说明：其功能是给函数设定一个寄存器区。“#pragma register”给其随后的函数设定一个寄存器区，“#pragma noregister”清除相应的设定。

“NUM”表示寄存器区号，只可以是 0 到 31 的整数常量，该整数常量可以用 16 进制、10 进制和 8 进制来表示。

虽然在函数的开始处对寄存器区号进行设置，但是在函数结束时并不会把寄存器区号恢复回去。

“#pragma register”和“#pragma noregister”必须成对出现，且不允许嵌套使用。

[例子]
输入：扩充的 C 源代码

```
#pragma register(2)
void func(void){}
#pragma noregister
```

输出：汇编代码

```
_func:
       MOV     RP, #2
       LINK    #0
       UNLINK
       RET
```

九. 中断级别的设置

一般格式：

```
#pragma ilm(NUM)
#pragma noilm
```

说明：其功能是给函数设定一个中断级别。“#pragma ilm”给其随后的函数设定一个中断级别，“#pragma noilm”清除相应的设定。

“NUM”表示中断级别号，只可以是 0 到 7 的整数常量，该整数常量可以用 16 进制、10 进制和 8 进制来表示。

虽然在函数的开始处对中断级别号进行设置，但是在函数结束时并不会把中断级别号恢复回去。

“# pragma ilm”和“#pragma noilm”必须成对出现，且不允许嵌套使用。

[例子]
输入：扩充的 C 源代码

```
#pragma ilm(1)
void func(void){}
#pragma noilm
```

输出：汇编代码

```
_func:
       MOV     ILM, #1
       LINK    #0
       UNLINK
       RET
```

十. 是否使用系统堆栈的设置

一般格式：

```
#pragma ssb
#pragma nossb
```

说明：其功能是告诉编译器，其中的函数将使用系统堆栈。“#pragma ssb”使其随后的函数使用系统堆栈，“#pragma nossb”清除相应的设定。

“# pragma ssb”和“#pragma nossb”必须成对出现，且不允许嵌套使用。并且，“# pragma ssb”和“#pragma nossb”对不能出现在“# pragma except”和“#pragma noexcept”（“# pragma except”和“#pragma noexcept”在下面解说）对中间。

[例子]
输入：扩充的 C 源代码

```
__far int *p;
#pragma ssb
void func(void){
  int a;
  p=&a;
}
#pragma nossb
```

输出：汇编代码

```
_func:
        LINK    #2
        MOV     A, SSB
        MOVEA   A, @RW3+-2
        MOVL    _p, A
        UNLINK
        RET
```

十一. 系统堆栈/用户堆栈都可用的设置

一般格式：

```
#pragma except
#pragma noexcept
```

说明：其功能是告诉编译器，其中的函数既可以使用系统堆栈也可以使用用户堆栈。“#pragma except” 告诉编译器，其随后的函数既可以使用系统堆栈也可以使用用户堆栈，“#pragma noexcept”取消这种假定。

“# pragma except”和“#pragma noexcept”必须成对出现，且不允许嵌套使用。并且，“# pragma except”和“#pragma noexcept”对不能出现在“# pragma ssb”和“#pragma nossb”对中间，因为这两种设置是矛盾的。

[例子]
输入：扩充的 C 源代码

```
__far int *p;
#pragma except
void func(void){
  int a;
  p=&a;
}
#pragma noexcept
```

输出：汇编代码(节选)

```
_func:
   LINK     #2
   CALLP    LOADSPB
   MOVEA    A, @RW3+-2
   MOVL     _p, A
   UNLINK
   RET
```

十二. 不保存寄存器的中断函数

一般格式：

```
__nosavereg Function definition
```

说明：当不需要对寄存器进行保存时，可以使用该功能来禁止寄存器保存操作。比如，中断函数要改变寄存器区号，就没有必要保存寄存器，从而使用该功能，这样可以生成运行速度快的代码。寄存器区号的改变，可以使用“#pragma register”这一 C 语言扩充功能来实现。

[例子]
输入：扩充的 C 源代码

```
extern void sub(void);
#pragma register(5)
  __nosavereg __interrupt void func(void){sub();}
#pragma noregister
```

输出：汇编代码

```
_func:
   MOV   RP, #5
   LINK  #0
   CALL  _sub
   UNLINK
   RETI
```

十三. 内置的函数

定义了下列几个内置函数：

- **__wait_nop**
- **__mul**
- **__div**
- **__mod**
- **__mulu**
- **__divu**
- **__modu**

■ __wait_nop 内置函数

一般格式：

```
void __wait_nop(void);
```

说明：有时候，在进行 I/O 操作和等待中断时，需要进行短暂的延时，这时候一般使用“NOP”指令，在 C 源程序中可用嵌入汇编的方式来实现。但，使用了嵌入汇编后，就使优化编译功能(和其它一些功能)失效了，结果会使生成的目标代码效率降低。

__wait_nop 内置函数就是针对这一需求而设计的。这一内置函数等效于“NOP”指令，但它不会使优化编译功能(和其它一些功能)失效。

[例子]

输入：扩充的 C 源代码

```
void sample(void)
{
  __wait_nop();
}
```

输出：汇编代码

```
_sample:
   LINK #0
   NOP
   UNLINK
   RET
```

■ __mul 内置函数

一般格式：

```
signed long __mul(signed int, signed int);
```

说明：其功能是，把两个 16 位的有符号数相乘，并返回 32 位的有符号数---结果。使用该内置函数可以避免 16 位运算的溢出，还可以提高运算的效率。

该内置函数仅适用于富士通 16 位的 F2MC-16LX/16F 微控制器系列。

[例子]

输入：扩充的 C 源代码

```
extern signed int arg1,arg2;
extern signed long ans;
void sample(void){
  ans = __mul(arg1, arg2);
}
```

输出：汇编代码(节选)

```
MOVW    A, _arg2
PUSHW   A
MOVW    A, _arg1
PUSHW   A
CALL    ___mul
ADDSP   #4
EXTW
MOVL    _ans, A
```

■ __div 内置函数

一般格式：

```
signed int __div(signed long, signed int);
```

说明：其功能是，把一个 32 位的有符号数除以 16 位的有符号数，并把其结果---16 位的有符号数返回。使用该内置函数可以提高运算的效率。

该内置函数仅适用于富士通 16 位的 F2MC-16LX/16F 微控制器系列。

[例子]

输入：扩充的 C 源代码

```
extern signed int arg2,ans;
extern signed long arg1;
```

```
void sample(void){
  ans = __div(arg1, arg2);
}
```

输出：汇编代码

```
MOVW    A, _arg2
PUSHW   A
MOVL    A, _arg1
PUSHW   AH
PUSHW   A
CALL    ___div
ADDSP   #6
MOVW    _ans, A
```

■ __mod 内置函数

一般格式：

```
signed int __mod(signed long, signed int);
```

说明：其功能是，把一个 32 位的有符号数模上(取模)16 位的有符号数，并把其结果---16 位的有符号数返回。使用该内置函数可以提高运算的效率。

该内置函数仅适用于富士通 16 位的 F2MC-16LX/16F 微控制器系列。

[例子]

输入：扩充的 C 源代码

```
extern signed int arg2,ans;
extern signed long arg1;
void sample(void){
  ans = __mod(arg1, arg2);
}
```

输出：汇编代码

```
MOVW    A, _arg2
PUSHW   A
MOVL    A, _arg1
PUSHW   AH
PUSHW   A
CALL    ___mod
ADDSP   #6
MOVW    _ans, A
```

■ __mulu 内置函数

一般格式：

```
unsigned long __mulu(unsigned int, unsigned int);
```

说明：其功能是，把两个 16 位的无符号数相乘，并返回 32 位的无符号数---结果。使用该内置函数可以避免 16 位运算的溢出，还可以提高运算的效率。

[例子]
输入：扩充的 C 源代码

```
extern unsigned int arg1,arg2;
extern unsigned long ans;
void sample(void){
  ans = __mulu(arg1, arg2);
}
```

输出：汇编代码

```
MOVW    A, _arg1
MULUW   A, _arg2
MOVL    _ans, A
```

■ __divu 内置函数

一般格式：

```
unsigned int __divu(unsigned long, unsigned int);
```

说明：其功能是，把一个 32 位的无符号数除以 16 位的无符号数，并把其结果---16 位的无符号数返回。使用该内置函数可以提高运算的效率。

[例子]
输入：扩充的 C 源代码

```
extern unsigned int arg2,ans;
extern unsigned long arg1;
void sample(void){
  ans = __divu(arg1, arg2);
}
```

输出：汇编代码

```
MOVL    A, _arg1
MOVW    RW0, _arg2
DIVUW   A, RW0
MOVW    RW0, A
MOVW    A, RW0
MOVW    _ans, A
```

■ __modu 内置函数

一般格式：

```
unsigned int __modu(unsigned long, unsigned int);
```

说明：其功能是，把一个 32 位的无符号数模上(取模)16 位的无符号数，并把其结果---16 位的无符号数返回。使用该内置函数可以提高运算的效率。

[例子]

输入：扩充的 C 源代码

```
extern unsigned int arg2,ans;
extern unsigned long arg1;
void sample(void){
  ans = __modu(arg1, arg2);
}
```

输出：汇编代码

```
MOVL    A, _arg1
MOVW    RW0, _arg2
DIVUW   A, RW0
MOVW    A, RW0
MOVW    _ans, A
```

十四. 预定义的宏

■ ANSI 标准规定的预定义宏

宏名	描述
__LINE__	源文件中当前行的行号
__FILE__	源文件的文件名
__DATA__	（对源文件）编译时的日期
__TIME__	（对源文件）编译时的时间
__STDC__	该宏指示编译过程是否符合规范。 当使用编译选项“-Ja”（此编译选项允许使用 C 的扩充功能）时，它为 0；当使用编译选项“-Jc”（此编译选项指示，编译严格按照 ANSI 规范进行，不允许使用 C 的扩充功能）时，它为 1。

■ 富士通 fcc907s C 编译器预定义的宏

除了 ANSI 标准中规定的预定义宏外，还定义了下列其独有的宏。

宏名	描述
__COMPILER_FCC907__	为 1
__CPU_MB number__	为 1。但如果使用的不是本型号的 CPU，则会编译出错：该标识符未定义*1
__CPU_16L__	为 1。但如果使用的不是本系列的 CPU（这里指：FMC16L 系列），则会编译出错：该标识符未定义*2
__CPU_16LX__	为 1。但如果使用的不是本系列的 CPU（这里指：FMC16LX 系列），则会编译出错：该标识符未定义*2
__CPU_16F__	为 1。但如果使用的不是本系列的 CPU（这里指：FMC16F 系列），则会编译出错：该标识符未定义*2

注 1：使用何种型号的 CPU，在此处设置，Project->Setup…->Base->Target MCU。比如，如果使用型号 MB90V560 的 CPU，则宏“__CPU_MB90V560__”为 1。

注 2：使用何种系列的 CPU，在此处设置，Project->Setup…->Base->Chip Classification。

第三章　C 库函数

第一节 库函数概述

随富士通 C 编译系统提供了 18 个库文件和 14 个头文件。

一. 库文件和内存模式

下表说明，在不同的编译模式下所使用的库文件名。

文件名	内存模式
lib907s.lib lib905s.lib lib902s.lib	小模式
lib907m.lib lib905m.lib lib902m.lib	中模式
lib907c.lib lib905c.lib lib902c.lib	紧凑模式
lib907l.lib lib905l.lib lib902l.lib	大模式
lib907sr.lib lib905sr.lib lib902sr.lib	小模式和 “-ramconst”编译选项*
lib907mr.lib lib905mr.lib lib902mr.lib	中模式和 “-ramconst”编译选项*

下表说明，对应不同的编译模式下，模拟调试器所使用的低级库文件名。

文件名	内存模式
lib907sif.lib lib905sif.lib lib902sif.lib	小模式
lib907mif.lib lib905mif.lib lib902mif.lib	中模式
lib907cif.lib lib905cif.lib lib902cif.lib	紧凑模式
lib907lif.lib lib905lif.lib lib902lif.lib	大模式
lib907srif.lib lib905srif.lib lib902srif.lib	小模式和 “-ramconst”编译选项*
lib907mrif.lib lib905mrif.lib lib902mrif.lib	中模式和 “-ramconst”编译选项*

说明：*是指，在编译时使用“-ramconst”编译选项。关于此编译选项的详细描述可见于前面的章节。

以“lib907”打头的库文件对应 F^2MC16、F^2MC16H 和 F^2MC16L 的 CPU，以“lib905”打头的库文件对应 $F^2MC16LX$ 的 CPU，以“lib902”打头的库文件对应 F^2MC16F 的 CPU。

二. 头文件

有如下 14 个头文件：

assert.h	ctype.h
float.h	limits.h
math.h	setjmp.h
stdarg.h	stddef.h
stdio.h	stdlib.h
string.h	fcntl.h
unistd.h	sys/types.h

其中，这 3 个头文件声明了进行低级函数调用时所需的宏和(变量和函数)类型：fcntl.h　unistd.h 和 sys/types.h

三. 库文件对应的 SECTION 和内存模式

在不同的内存模式下，库文件对应的 SECTION 名也不同。其对应关系见下表：

Section 类型	小模式	中模式	紧凑模式	大模式
Code section	CODE	LIBCODE	CODE	LIBCODE
Data section	DATA	DATA	LIBDATA	LIBDATA
Initial value of DINIT	DCONST	DCONST	LIBDCONST	LIBDCONST
Initialized section	INIT	INIT	LIBINIT	LIBINIT
Constant section	CONST	CONST	LIBCONST	LIBCONST
RAM area of CCONST	CINIT	CINIT		

四. 依赖于系统的库函数

文件输入输出，内存管理和程序退出这些函数，在不同的编译系统下，其实现方法很不一样(与之比较，其它函数如数学函数，在不同的编译系统下的实现方法比较相近)，把它们称为依赖于系统的函数。在进行这样的函数调用时，这些函数又要调用一些被称为低级函数调用的库函数。

低级库函数的类型简介(其详细介绍在下面的章节)如下：

- **open**　：打开一个文件
- **close**　：关闭一个文件
- **read**　：从一个文件中读字符
- **write**　：往一个文件中写字符

- **lseek** ：改变进行文件操作(读/写)时的文件指针位置
- **isatty** ：检查已打开的文件的文件指针是否指向文件末尾
- **sbrk** ：动态分配和改变内存
- **_exit** ：程序正常退出
- **_abort** ：程序异常退出

第二节　库函数的协作

一. 库函数协作

库函数可以分为标准库函数和低级库函数。一般而言，用户只需调用标准库函数，而不需要直接调用低级库函数；当调用一些依赖于系统的标准库函数时，这些标准库函数会自动去调用低级库函数，库函数的协作就是指这一过程。

为了进行一些标准库函数调用，需要(先)调用下列函数：

- 流初始化
- 标准输入/输出文件和标准错误输出文件的打开和关闭
- 低级函数的创建(Low-level function creation)

二. 初始化和退出函数

■ 流初始化

_stream_init 函数对流进行初始化，必须在初始化文件(startup)里调用该函数，这样才能确保在 C 语言的 **main** 主函数里进行正确的流操作。

void _stream_init(void);

■ 标准输入/输出文件和标准错误输出文件的打开和关闭

三. 低级库函数类型

- 文件的打开和关闭

当用户调用 **fopen** 和其它打开文件的函数时，低级库函数 **open** 会被自动调用。

同样，**fclose** 和其它关闭文件的函数会调用低级库函数 **close**。

- 文件的输入和输出(读和写)

scanf,**printf** 和其它文件输入/输出函数会调用低级库函数 **read** 和 **write**。

• 文件指针位置的移动

fseek 和移动文件指针的函数会调用低级库函数 **lseek**。

• 文件指针的检查

检查已打开的文件的文件指针是否指向文件末尾(**isatty**)。

• 内存的动态分配

malloc 和其它动态分配内存函数会调用低级库函数 **sbrk**。

• 程序的正常退出和异常退出

exit 和 **abort** 这两个程序的正常退出和异常退出函数，会调用低级库函数 **_exit** 和**_abort** 使程序退出。

四. 标准库函数和其所需调用的低级库函数

标准库函数			初始化和退出
assert () abort ()*	open () read () lseek () sbrk ()	close () write () isatty () _abort ()	
所有的 stdio.h 函数	open () read () lseek () sbrk ()	close () write () isatty ()	
calloc () malloc () realloc () free ()	sbrk ()		
exit ()*	open () read () lseek () sbrk ()	close () write () isatty () _exit ()	

说明*：当函数 **abort** 和 **exit** 被调用时，它们会对打开的文件进行关闭；因此，文件操作相关的低级函数(**open**, **close**, **read**, **write**, **lseek** 和 **sbrk**)流初始化函数是其必需的。

如果程序中不需要使用文件，那么可以直接调用**_abort** 函数而不必调用 **abort** 函数。

如果程序中没有使用文件，而且程序中使用 **atexit** 函数而使函数登记(function registration)没有完成，那么可以直接调用**_exit** 函数而不必调用 **exit** 函数。

第三节　低级库函数的详细说明

对低级库函数的调用必须按照下面的详细说明来使用。

一. open 函数

一般格式：

```
#include <fcntl.h>
int open( char *fname, int fmode, int p );
```

[说明]

该函数按照 **fmode** 指定的方式，打开一个名为 **fname** 的文件。对于 fmode，可以使用下面所说的位标志的组合(逻辑或 OR)。**p** 的值通常为 0777(八进制)。对于 fmode，可用的位标志如下：

– O_RDONLY:

以只读的方式打开一个文件

– O_WRONLY:

以只写的方式打开一个文件

– O_RDWR:

以可读、可写的方式打开一个文件

上面三个标志是互相排斥的。

– O_CREAT:

以创建的方式打开一个文件。如果指定的文件已经存在，则忽略该标志。

– O_TRUNC:

以截断的方式打开一个文件，即如果文件中原来存在数据，则原来的数据被清除。

– O_APPEND:

以追加的方式打开一个文件。在这种打开方式下对文件写入，则不管文件指针在哪里，都是往文件的最后进行写入。

– O_BINARY:

以二进制的方式打开一个文件。如果未使用该标志来打开文件，则把打开的文件按文本文件的方式进行处理。

[返回值]

如果文件打开成功，返回(整型的)文件号；否则，返回-1。

二. close 函数

一般格式：

```
#include <unistd.h>
int close( int fileno);
```

[说明]

该函数关闭文件号为 **fileno** 的文件。该文件号 **fileno** 应该是打开文件(open)时所返回的。

[返回值]

如果文件关闭成功，返回 0；否则，返回-1。

三. read 函数

一般格式：

```
#include <unistd.h>
int read( int fileno, char *buf, unsigned int size);
```

[说明]

该函数从文件号为 **fileno** 的已打开的文件中，读出 **size** 字节数据，放入名为 **buf** 的缓冲区中。

如果在系统环境(system environment)中的换行符不是**\n**，则可以通过 **read** 函数来把从文件中读到的换行符转换为**\n**。

[返回值]

如果读取文件成功，返回读到的字符数目。如果读取文件失败，则返回-1。如果在读取文件的中途，到达了文件末尾，则返回值比 **size** 指定的数值小(仍然返回读到的字符数目)。

四. write 函数

一般格式：

```
#include <unistd.h>
int write (int fileno, char *buf, unsigned int size);
```

[说明]

该函数从名为 **buf** 的缓冲区中，读出 **size** 字节数据，写入文件号为 **fileno** 的已打开的文件中。如果文件是以追加的方式打开的，则写入是往文件的末尾进行的。

如果在系统环境(system environment)中的换行符不是**\n**，则可以通过 **write** 函数来把换行符转换为**\n** 再往文件中写入。

[返回值]

如果写入文件成功，返回写入文件的字符数目。如果写入文件失败，则返回-1。

五. lseek 函数

一般格式：

```
#include <unistd.h>
long int lseek( int fileno, long int offset, int whence);
```

[说明]

该函数把文件号为 **fileno** 的已打开的文件的文件指针，从 **whence** 指定的起始位置，移动 **offset** 字节。文件指针的起始位置 **whence**，可以使用下列三个数值：

– SEEK_CUR:

文件指针的当前位置

– SEEK_END:

文件的末尾

– SEEK_SET:

文件的开头

[返回值]

如果移动文件指针成功，返回新的文件指针位置；否则返回-1L。

六. isatty 函数

一般格式：

```
#include <unistd.h>
int isatty( int fileno);
```

[说明]

该函数检查已打开的文件的文件指针是否指向文件末尾。如果文件指针指向文件末尾，就返回 **true**，否则就返回 **false**。

[返回值]

如果文件指针指向文件末尾，就返回 **true**，否则就返回 **false**。

七. sbrk 函数

一般格式：

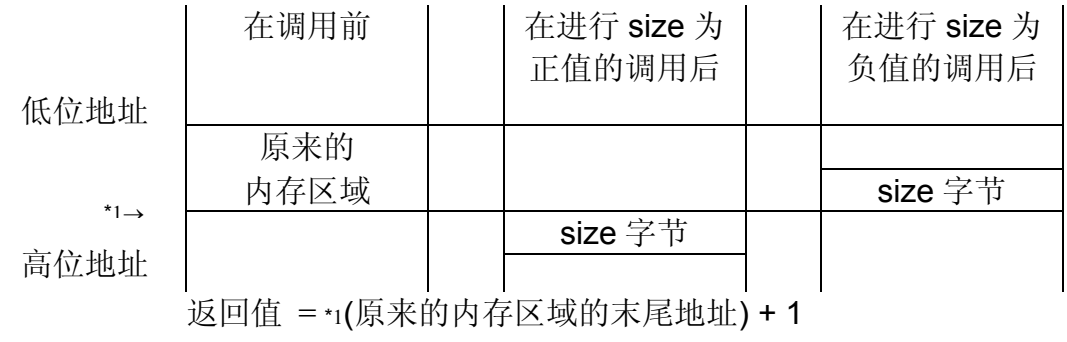
```
char *sbrk( int size);
```

[说明]

该函数在原来的内存区域的基础上，增加 **size** 字节。如果 **size** 的数值为负值，则实际上内存区域被减少了。

该函数第一次被调用时，得到大小为 **size** 字节的内存区域。

下图说明调用 sbrk 函数前后，内存区域的变化情况。



[返回值]

如果该函数调用成功，返回值等于 原来的内存区域的末尾地址+1。如果该函数是第一次被调用，则返回所获得的内存区域的起始地址。如果函数调用失败，则返回(char)-1。

八. _exit 函数

一般格式：

```
#include <stdlib.h>
void _exit( int status);
```

[说明]
该函数使程序正常退出/结束。当 **status** 的值为 0 或为 **EXIT_SUCCESS**，把正常结束状态(successful end state)返回给系统；当 **status** 的值为 **EXIT_FAILURE**，则把非正常结束状态(unsuccessful end state)返回给系统

[返回值]
该函数不会返回到被调用函数，因而也没有返回值。

九. _abort 函数

一般格式：

```
void _abort( void);
```

[说明]
该函数使程序正常退出/结束。

[返回值]
该函数不会返回到被调用函数，因而也没有返回值。

第四章 嵌入式 C 语言的特殊之处

第一节　StartUp 启动文件

通常，在 PC 机环境下使用各种 C 编译器进行 C 语言编程，用不到 StartUp 启动文件，该环境下的 C 语言用户的思维里不会有 StartUp 启动文件的概念。一般而言，与此相对照，在微控制器/单片机环境下，进行嵌入式 C 语言的编程，就需要用到 StartUp 启动文件（当然，也有少数编译器不使用 StartUp 启动文件）。对于富士通的 C 语言开发环境来说，就必须使用 StartUp 启动文件。

一. 如果（强行）不使用 StartUp 启动文件

[例子]
输入：扩充的 C 源代码

```
#pragma     section  CODE=main,  attr=CODE
void main(void)
{
  int i,j;
  i=0;
  j=1;
  i=j*2;
}
#pragma asm
   .SECTION reset, data, LOCATE=0xFFFFDC
   .data.l  _main
   .END  _main
#pragma  endasm
```

输出：汇编代码(附有代码的绝对地址)

```
_main:
FF0000:       LINK    #4
FF0002:       MOVN    A, #0
FF0003:       MOVW    @RW3+-4, A
FF0005:       MOVN    A, #1
FF0006:       MOVW    @RW3+-2, A
FF0008:       MOVW    A, @RW3+-2
FF000A:       LSLW    A
FF000B:       MOVW    @RW3+-4, A
FF000D:       UNLINK
FF000E:       RET
```

说明: 复位后，从地址 0FF0000H 开始执行，执行的第一行代码是”LINK　　#4”；由于在执行该行代码之前，没有设置过堆栈指针 SP(复位后系统堆栈指针 SSP，和用户堆栈指针 USP 的初始值都是不确定的)；因此，执行第一行代码”LINK　　#4”，就会因为堆栈指针 SP 值的不确定而使代码运行有错；之后，在遇到”UNLINK”和”RET”这样的指令时，很可能造成程序运行飞掉。打个比喻，就象一列火车开在没有铁轨的路上，将会造成什么后果，读者可以想象得出来(对，的确太危险了)。

二. 使用一个简单的 StartUp 启动文件

[例子]

1. 汇编的 StartUp 文件

```
        .SECTION   ALLSTACK, STACK, LOCATE=400H
        .RES.B     100H
    stack_top:
        .SECTION   START_CODE, CODE, LOCATE=0FF0000H
        .import    _main
    start:
        and     ccr, #0
        mov     ilm, #0
        mov     rp, #0
        mov     a, #bnksym stack_top
        mov     ssb, a
        mov     usb, a
        movw    a, #stack_top
        movw    sp, a
        call    _main
    end:
        jmpp    end
        .SECTION   RESET_VECT, const, locate=0FFFFDCH
        .DATA.E         start
        .DATA.B         0          ;mode data(byte)
        .END    start
```

2. C 源代码

```
    void main(void)
    {
      int i,j;
      i=0;
      j=1;
      i=j*2;
    }
```

说明: 复位后，从汇编语言的 StartUp 文件中标号为”start”处(地址 0FF0000H)开始执行，

三. 随富士通 C 编译器附带的 StartUp 启动文件

例如: start905s.asm
节选 1(后面有说明)

```
        .SECTION  CODE,     CODE,          ALIGN=1
__start:
;--------------------------------------------------------------------------
; set register bank is 0
;--------------------------------------------------------------------------
        MOV     RP, #0
;--------------------------------------------------------------------------
; set ILM to the lowest level
;--------------------------------------------------------------------------
        MOV     ILM, #7
;--------------------------------------------------------------------------
; set direct page register
;--------------------------------------------------------------------------
        MOV     A, #PAGE DIRDATA_S
        MOV     DPR, A
;--------------------------------------------------------------------------
; set system stack
;--------------------------------------------------------------------------
        AND     CCR, #0x20
        MOV     A, #BNKSYM SSTACK_TOP
        MOV     SSB, A
        MOVW A, #SSTACK_TOP
        MOVW SP, A
        AND     CCR, #0x00DF
```

说明: 这一段代码对一些寄存器进行初始化，如堆栈段寄存器 SSB、堆栈指针 SP(实际上是系统堆栈指针 SSP，因为复位后寄存器 CCR 的 S 标志位为 1)、寄存器区指针 RP、中断级别屏蔽寄存器 ILM、直接分页寄存器 DPR 和条件码寄存器 CCR。这些初始化对于编写一个功能比较全的实际应用来说，是必需的。

节选 2(后面有说明)

```
;--------------------------------------------------------------------------
; copy initial value *CONST section to *INIT section
;--------------------------------------------------------------------------
#macro      ICOPY  src_addr, dest_addr, src_segment
```

```
        MOV     A, #BNKSYM \src_addr
        MOV     ADB, A
        MOV     A, #BNKSYM \dest_addr
        MOV     DTB, A
        MOVW RW0, #SIZEOF (\src_segment)
        MOVW A, #\dest_addr
        MOVW A, #\src_addr
        MOVSI DTB, ADB
#endm
        ICOPY  DIRCONST_S, DIRINIT_S, DIRCONST
        ICOPY  DCONST_S,     INIT_S,       DCONST
        ICOPY  LIBDCONST_S,LIBINIT_S, LIBDCONST
;------------------------------------------------------------------------
; zero clear of *VAR section
;------------------------------------------------------------------------
#macro      FILL0    src_addr, src_segment
        MOV     A, #BNKSYM \src_addr
        MOV     DTB, A
        MOVW RW0, #SIZEOF (\src_segment)
        MOVW A, #\src_addr
        MOVN  A, #0
        FILSI    DTB
#endm
        FILL0   DIRDATA_S, DIRDATA
        FILL0   DATA_S,          DATA
        FILL0   LIBDATA_S, LIBDATA
```

说明: 这一段代码，对声明时赋予初值的变量(参见前面的章节)进行初值传送(从初值区域传送到相应的变量区域)；对声明时没有赋予初值的变量(如外部变量，参见前面的章节)进行清 0 操作；并且，还对函数库中的变量进行初始化(LIBDCONST)和清零(LIBDATA)。

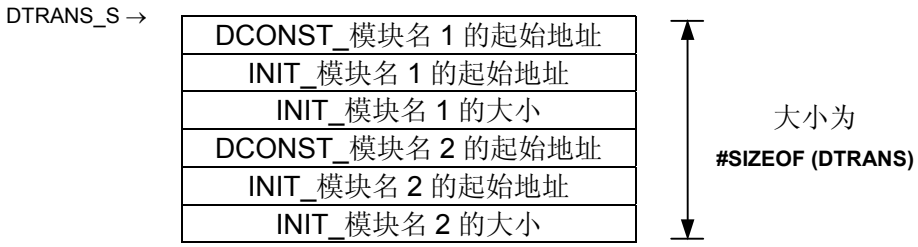
节选 3(后面有说明)

```
;------------------------------------------------------------------------
; copy initial value DCONST_module section to INIT_module section
;------------------------------------------------------------------------
        MOV     A, #BNKSYM DTRANS_S
        MOV     DTB, A
        MOVW RW1, #DTRANS_S
        BRA     LABEL2
LABEL1:
        MOVW A, @RW1+6
        MOV     USB, A
        MOVW A, @RW1+2
```

```
        MOV     ADB, A
        MOVW A, @RW1+4
        MOVW A, @RW1
        MOVW RW0, @RW1+8
        MOVSI SPB, ADB
        MOVN  A, #10
        ADDW  RW1, A
LABEL2:
        MOVW A, RW1
        SUBW   A, #DTRANS_S
        CMPW   A, #SIZEOF (DTRANS)
        BNE     LABEL1
;-------------------------------------------------------------------------
; zero clear of DATA_module section
;-------------------------------------------------------------------------
        MOV     A, #BNKSYM DCLEAR_S
        MOV     DTB, A
        MOVW RW1, #DCLEAR_S
        BRA     LABEL4
LABEL3:
        MOV     A, @RW1+2
        MOV     ADB, A
        MOVW RW0, @RW1+4
        MOVW A, @RW1
        MOVN  A, #0
        FILSI    ADB
        MOVN  A, #6
        ADDW  RW1, A
LABEL4:
        MOVW A, RW1
        SUBW   A, #DCLEAR_S
        CMPW   A, #SIZEOF (DCLEAR)
        BNE     LABEL3
```

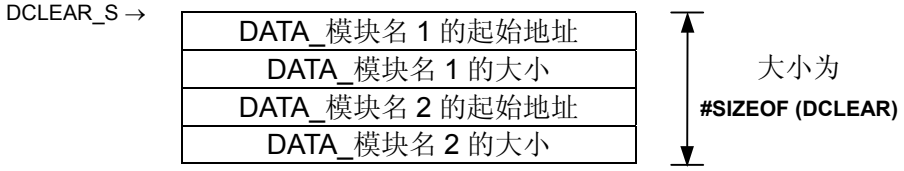
说明: 这一段代码，对声明时赋予初值的变量，这些变量所属的 section 是用 24 位地址的方式来存取的(以"紧凑模式"或"大模式"来编译，或者变量前加了__far 类型限定符，可以参见第二章的关于 SECTION 结构的那部分内)，进行初值传送(从初值区域传送到相应的变量区域)；对声明时没有赋予初值的变量，比如外部变量和静态变量，这些变量所属的 section 名也是用 24 位地址的方式来存取的，进行清 0 操作。其实，该节选与上一个节选，节选 2，很类似，只不过该节选中，变量所属的 section 都是用 24 位地址的方式来存取的。

关于 section **DTRANS**，示例说明如下:



声明时赋予初值的变量，其初始值放在”DCONST_+模块名”的模块里，初始值将被传送到”INIT_+模块名”的模块中。

关于 section **DCLEAR**，示例说明如下:



将对”DATA_+模块名”的模块进行清 0。

第二节　C 语言与汇编语言互相调用以及嵌入汇编

下面举一个例子来说明：

[接口情况]

PORT0 为输出口，接 8 个 LED；PORT1 作为输入口，接 DIP 开关

[注意]

1 C 语言中的变量要放到适当的地址空间里。

2 重要的地方都有中文注释。

3 读者可以用软件模拟的调试方式，来观察其情况。

[源代码]

1. C 源程序，test.c

```
#include "io90560.h"
#pragma     section   CODE=main, attr=CODE
void delay();
BYTE port_check;

void main(void)
{
  PORT0_DIR=0xff;     // PORT0 方向寄存器赋值  0ffh,成为输出引脚
  PORT1_DIR=0;        // PORT1 方向寄存器赋值  0,成为输入引脚
  PORT0_DATA=0;
  while(1){
    delay();
    PORT0_DATA=PORT0_DATA^PORT1_DATA;
    /*  下面一段是嵌入汇编的例子  */
    #pragma asm
          mov   A, _PDR1_
          bne    port1_not_zero
          mov   _PDR0_, #0
       port1_not_zero:
    #pragma endasm
  }
}

void delay(void)
{
  long int i;
  for(i=100000;i>0;i--)
```

```
    ;
}
```

2. 包含文件，声明 IO 接口，io90560.h

```
typedef unsigned char BYTE;
typedef unsigned short WORD;

union PORT0_DATA_TAG{
    BYTE abyte;
    struct{
        BYTE P00 :1;
        BYTE P01 :1;
        BYTE P02 :1;
        BYTE P03 :1;
        BYTE P04 :1;
        BYTE P05 :1;
        BYTE P06 :1;
        BYTE P07 :1;
    }bits;
};
union PORT1_DATA_TAG{
    BYTE abyte;
    struct{
        BYTE P10 :1;
        BYTE P11 :1;
        BYTE P12 :1;
        BYTE P13 :1;
        BYTE P14 :1;
        BYTE P15 :1;
        BYTE P16 :1;
        BYTE P17 :1;
    }bits;
};
union PORT0_DIR_TAG{
    BYTE abyte;
    struct{
        BYTE P00 :1;
        BYTE P01 :1;
        BYTE P02 :1;
        BYTE P03 :1;
        BYTE P04 :1;
        BYTE P05 :1;
        BYTE P06 :1;
```

```
        BYTE P07 :1;
    }bits;
};
union PORT1_DIR_TAG{
    BYTE abyte;
    struct{
        BYTE P10 :1;
        BYTE P11 :1;
        BYTE P12 :1;
        BYTE P13 :1;
        BYTE P14 :1;
        BYTE P15 :1;
        BYTE P16 :1;
        BYTE P17 :1;
    }bits;
};

extern union PORT0_DATA_TAG PDR0_;
#define PORT0_DATA PDR0_.abyte
#define PORT0_DATA_0 PDR0_.bits.P00
#define PORT0_DATA_1 PDR0_.bits.P01
#define PORT0_DATA_2 PDR0_.bits.P02
#define PORT0_DATA_3 PDR0_.bits.P03
#define PORT0_DATA_4 PDR0_.bits.P04
#define PORT0_DATA_5 PDR0_.bits.P05
#define PORT0_DATA_6 PDR0_.bits.P06
#define PORT0_DATA_7 PDR0_.bits.P07

extern union PORT1_DATA_TAG PDR1_;
#define PORT1_DATA PDR1_.abyte
#define PORT1_DATA_0 PDR1_.bits.P10
#define PORT1_DATA_1 PDR1_.bits.P11
#define PORT1_DATA_2 PDR1_.bits.P12
#define PORT1_DATA_3 PDR1_.bits.P13
#define PORT1_DATA_4 PDR1_.bits.P14
#define PORT1_DATA_5 PDR1_.bits.P15
#define PORT1_DATA_6 PDR1_.bits.P16
#define PORT1_DATA_7 PDR1_.bits.P17

extern union PORT0_DIR_TAG DDR0_;
#define PORT0_DIR DDR0_.abyte
#define PORT0_DIR_0 DDR0_.bits.P00
#define PORT0_DIR_1 DDR0_.bits.P01
#define PORT0_DIR_2 DDR0_.bits.P02
```

```
#define PORT0_DIR_3 DDR0_.bits.P03
#define PORT0_DIR_4 DDR0_.bits.P04
#define PORT0_DIR_5 DDR0_.bits.P05
#define PORT0_DIR_6 DDR0_.bits.P06
#define PORT0_DIR_7 DDR0_.bits.P07

extern union PORT1_DIR_TAG DDR1_;
#define PORT1_DIR DDR1_.abyte
#define PORT1_DIR_0 DDR1_.bits.P10
#define PORT1_DIR_1 DDR1_.bits.P11
#define PORT1_DIR_2 DDR1_.bits.P12
#define PORT1_DIR_3 DDR1_.bits.P13
#define PORT1_DIR_4 DDR1_.bits.P14
#define PORT1_DIR_5 DDR1_.bits.P15
#define PORT1_DIR_6 DDR1_.bits.P16
#define PORT1_DIR_7 DDR1_.bits.P17
```

3. 汇编程序，作为启动文件，start.asm

```
    .SECTION    ALLDATA, DATA, LOCATE=200H
    .RES.B      80H
    .SECTION    ALLSTACK, STACK, LOCATE=400H
    .RES.B      100H
stack_top:
    .RES.B      2
    .SECTION    START_CODE, CODE;, LOCATE=0FF0000H
    .IMPORT     _main, _port_check, _delay
    .IMPORT     _DDR0_, _DDR1_

start:
    and     ccr, #0
    mov     ilm, #0
    mov     rp, #0
    mov     a, #bnksym stack_top
    mov     ssb, a
    mov     usb, a
    movw    a, #stack_top
    movw    sp, a
    mov     _DDR0_, #0ffh   ;PORT0 方向寄存器赋值 0ffh,成为输出引脚
    mov     _DDR1_, #0      ;PORT1 方向寄存器赋值  0,成为输入引脚
    mov     i:0, #0ffh      ;PORT0 数据寄存器  输出高电平
    call    _delay          ;延时,调用 C 语言的子程序
    mov     i:0, #0         ;PORT0 数据寄存器  输出高电平
    call    _delay
```

```
    mov     a, i:1              ;读 PORT1 数据寄存器
    mov     _port_check, a      ;存放到 C 语言定义的变量里
    call    _main
end:
    jmp     $
    jmpp    end

    .SECTION        RESET_VECT, const, locate=0FFFFDCH
    .DATA.E         start
    .DATA.B         0                   ;mode data(byte)

    .END    start
```

4. 汇编程序，定义 IO 接口，io90560.asm

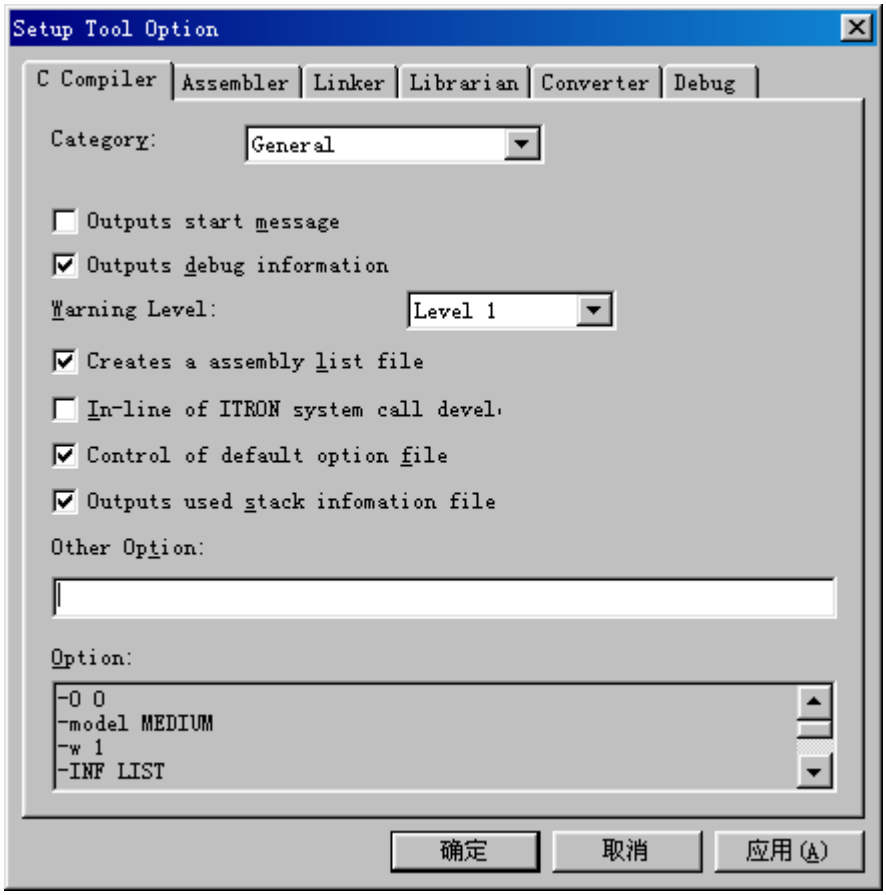
```
    .SECTION   ALL_IO,     IO
    .GLOBAL    _PDR0_,     _PDR1_
    .GLOBAL    _DDR0_,     _DDR1_

_PDR0_     .RES.B  1
_PDR1_     .RES.B  1
    .ORG    10H
_DDR0_     .RES.B  1
_DDR1_     .RES.B  1
    .END
```

附录一

1 编译选项与集成开发环境

在富士通集成开发环境 Softune Workbench 中，编译选项位于菜单项 Project->Setup Tool Option…->C Compiler。

下图是编译选项的界面：



注意: 如果改变了编译选项，一般会在 Option 下面的列表框中反映出来。

■ Category: General

- Outputs start message

相当于编译选项”-V”。

该编译选项的作用是，把编译器的版本信息输出到标准输出，和集成开发环境的输出窗口(Output,菜单项 View->Output)上去。

[例子]

输出: 集成开发环境的输出窗口(节选)

Now building...

start.asm

test.c

FFMC-16 Family Softune C Compile V30L02.

ALL RIGHTS RESERVED, COPYRIGHT (C) FUJITSU LIMITED 1986-1999

LICENSED MATERIAL - PROGRAM PROPERTY OF FUJITSU LIMITED

FFMC-16 Family Softune cpp V30L02.

ALL RIGHTS RESERVED, COPYRIGHT (C) FUJITSU LIMITED 1986-1999

LICENSED MATERIAL - PROGRAM PROPERTY OF FUJITSU LIMITED

FFMC-16 Family Softune ccom V30L02.

ALL RIGHTS RESERVED, COPYRIGHT (C) FUJITSU LIMITED 1986-1999

LICENSED MATERIAL - PROGRAM PROPERTY OF FUJITSU LIMITED

FFMC-16 Family Softune Assembler V30L04

ALL RIGHTS RESERVED, COPYRIGHT (C) FUJITSU LIMITED 1992-1999

LICENSED MATERIAL - PROGRAM PROPERTY OF FUJITSU LIMITED

是否需要使用该编译选项: 一般不需要。

缺省情况下，不选中该编译选项。所谓缺省情况是指，用户建立一个新的工程，未作任何改变，编译选项的初始状态。

• Outputs debug information

相当于编译选项”-g”。

该编译选项的作用是，在 obj 文件中加入所需的调试信息。

[例如]

在未使用(选中)该编译选项时，生成的某个 obj 文件的大小为 284 字节。

使用(选中)该编译选项时，生成的同一个 obj 文件的大小为 702 字节。由此看出，在 obj 文件中加入调试信息后，obj 文件的大小变大了。

是否需要使用该编译选项: 一般来说需要。如果用户想进行源代码级的调试，则必须使用该编译选项(这一点对于汇编的类似汇编选项，和链接的类似链接选项也同样适用)。如果不使用该编译选项，可能会有链接的警告提示。

缺省情况下，不选中该编译选项。

• Warning Level

相当于编译选项”-w num”,其中 num 为 0-8 的数字。

该编译选项的作用是，设置警告提示信息的输出级别。如果警告级别为 Level 0，则不会输出警告提示信息；警告级别越高，则警告信息越多也越详细。缺省情况下，为 Level 1，即对应”-w 1”。

• Creates a assembly list file

相当于编译选项”-INF LIST”。

该编译选项的作用是，生成汇编列表文件。

[例如]

源文件为 test.c，必须在使用(选中)该编译选项时，才会生成相应的汇编列表文件 test.lst。

是否需要使用该编译选项: 如果用户对编写的代码感到不够放心，或者用户对 C 源代码－编译－汇编代码这一过程感兴趣，可以使用该编译选项来生成汇编列表文件，进行一些检查工作。在正常情况下，可以不使用该选项。

缺省情况下，不选中该编译选项。

(软件制作者们把单词拼错了，应该为 Creates an assembly list file)

• In-line of ITRON system call development

相当于编译选项"-K REALOS"。

该编译选项的作用是，如果用户使用了富士通的 REALOS(富士通的实时操作系统，符合 ITRON 规范)，则需要使用该编译选项，以实现 REALOS 的内嵌扩展。

是否需要使用该编译选项: 一般来说，用户并没有使用富士通的 REALOS，因而一般不需要使用该编译选项。

缺省情况下，不选中该编译选项。

• Control of default option file

该编译选项的作用是：其一，按照英文的 C 语言手册和从字面理解，该编译选项使编译器从缺省的编译选项文件(缺省的编译选项文件为 fcc907.opt，编译器会到环境变量" OPT907"指定的目录中查找该文件，环境变量" OPT907"位于此处：Setup->Development…->Environment Variable->OPT907)中，读出其中的编译选项，从而影响编译过程。实际上，经试验发现，该编译选项没有起作用。

缺省情况下，选中该编译选项。

• Outputs used stack infomation file

该编译选项的作用是，把堆栈的使用情况输出到.stk 文件中。

[例如]

源文件为 test.c，在使用(选中)该编译选项时，生成相应的堆栈的使用情况文件 test.stk；如果不使用该编译选项，就不会生成 test.stk 文件。

缺省情况下，选中该编译选项。

(软件制作者们把单词拼错了，应该为 Outputs used stack information file)

• Other Option

其它编译选项，可以在此处的编辑框中输入。

[例如]

为了使用 C++类型的注释(//)，可以在此处的编辑框中输入-B。

■ Category: Define Macro

一般来说，用户没有必要在此处定义宏；而可以在 C 源文件中来定义宏，这样更加方便一些。

• 定义宏(Set)

[例子]

在 Macro Name 编辑框中输入"TRUE"，在 Value 编辑框中输入"1"。

按 Set 按钮

则在 Macro Name List 列表框中得到"TRUE=1"，该宏被定义了。

它相当于编译选项" -D TRUE=1"。

在 Macro Name 编辑框中输入"FALSE"，在 Value 编辑框中输入"0"。

按 Set 按钮

则在 Macro Name List 列表框中得到"FALSE=0"，该宏被定义了。

它相当于编译选项" -D FALSE=0"。

• 删除宏(Delete)

[例子]

在 Macro Name List 列表框中定义了 2 个宏，"TRUE=1"和"FALSE=0"；选中其中任意一个宏

按 Delete 按钮

则相应的宏被删除。

■ Category: Include Path

其作用是，设定搜索路径，搜索路径供查找包含文件用。

一般来说，对于较简单的工程，包含文件和 C 源文件在同一目录下，用户不需要使用该功能。对于较复杂的工程，包含文件和 C 源文件不在同一目录下，用户可以使用该功能；但用户也可以在包含文件时(#include)直接把路径写上，而不使用该功能。

• 设定搜索路径(Add)

[例子]

C 源文件中包含了头文件 io90560.h，且该头文件与 C 源文件不在同一个目录下，而是在"D:\Fujitsu\SofTune\Work\test"路径下。在 Include Path 编辑框中，输入"D:\Fujitsu\SofTune\Work\test"

按 Add 按钮

则在 Macro Name List 列表框中得到"D:\Fujitsu\SofTune\Work\test"，该搜索路径被定义了。

它相当于编译选项"-I "D:\Fujitsu\SofTune\Work\test""。

• 删除搜索路径(Delete)
[例子]
在 Macro Name List 列表框中选中”D:\Fujitsu\SofTune\Work\test”，
按 Delete 按钮
则相应的搜索路径被删除。

■ Category: Optimize

其作用是，设定编译优化的级别。
[例子]
在 General-purpose Optimization Level 下拉框中，选择”Level 1”
它相当于编译选项"-O 1"。

点击 Set…按钮，弹出 Optimization 对话框：
(要注意的是，如果选择不进行编译优化，即在 General-purpose Optimization Level 下拉框中，选择”None”，则 Optimization 对话框中，很多选项不能选---变灰了。)

• Loop unrolling
其作用是，当存在循环语句时，它试图通过增加代码的长度来解除循环。其结果是，代码的执行速度可能更快。
它相当于编译选项"-K UNROLL"。
不选中该选项，则对应编译选项"-K NOUNROLL"。
[例子]
不使用该选项:

```
for(i=0;i<3;i++){ a[i]=0;}
```

使用该选项:

```
a[0]=0;
a[1]=0;
a[2]=0;
```

• In-line expansion of standard library functions, or replacement to equivalent function
其作用是，它试图对标准库函数进行内嵌扩展或用等效的函数来替代。其结果是，代码的长度可能增加，但执行速度会更快。
它相当于编译选项"-K LIB"。
不选中该选项，则对应编译选项"-K NOLIB"。
[例子]
输入:

```
extern int i;
```

```
void func(void){
  i=strlen("ABC");
}
```

输出:

```
MOVN   A, #3 ; Processing equivalent to strlen expanded
MOVW   _i, A
```

• Optimization of changing the evaluation method of aithmetr

其作用是，它试图改变算术运算的顺序从而对算术表达式进行优化。其结果会使代码的执行速度更快。

它相当于编译选项"-K EOPT"。

不选中该选项，则对应编译选项"-K NOEOPT"。

[例子]

输入:

```
extern int i;
void func(int a, int b){
  i=a-100+b+100;
}
```

使用该选项，输出:

```
MOVW   A, @RW3+4
ADDW   A, @RW3+6 ; Order of arithmetic operation replaced
MOVW   _i, A
```

未使用该选项，输出:

```
MOVW   A, @RW3+4
SUBW   A, #100
ADDW   A, @RW3+6
ADDW   A, #100
MOVW   _i, A
```

(软件制作者们把单词拼错了，应该为 Optimization of changing the evaluation method of arithmetic)

• Optimization of argument area on stack

其作用是，它试图把实在参数在一起同时释放掉以达到优化的目的，从而可能生成较短的和执行速度较快的目标

它相当于编译选项"-K ADDSP"。

不选中该选项，则对应编译选项"-K NOADDSP"。

[例子]

输入:

```
extern int i;
extern void sub(int);
void func(void){
  sub(i);
  sub(i);
}
```

输出:

```
MOVW   A, _i
PUSHW A
CALL     _sub
MOVW   A, _i
PUSHW A
CALL     _sub
ADDSP  #4; Releasing argument areas synthesized
```

上面的例子来自英文的 C 语言手册。虽然该例子能较好体现该编译选项的作用；但实际上，经试验发现，得不到上面的输出，实际的输出如下，从下面的实际输出来看，似乎该编译选项没有起作用。

实际输出:

```
MOVW   A, _i
PUSHW A
CALL     _sub
POPW    AH
MOVW   A, _i
PUSHW A
CALL     _sub
POPW    AH
```

• Control Optimization of pointer aliasing

它相当于编译选项"-K ALIAS"。

不选中该选项，则对应编译选项"-K NOALIAS"。

编译选项"-K NOALIAS"的作用是，它假定指针指向的数据域与其它变量和指针(指向的数据域)不相同，从而对指针指向的数据作优化。

[例子]

输入:

```
extern int i;
extern int j;
void func(int *p){
   *p=i+1;
   j=i+1;
}
```

输出:

```
MOVW   A, _i
MOVN    A, #1
ADDW    A
MOVW   RW4, A
MOVW   A, @RW3+4
MOVW   @AL, AH
MOVW   A, RW4
MOVW    _j, A ; Value of *p=i+1 reused
```

上面的例子来自英文的 C 语言手册。虽然该例子能体现该编译选项的作用；但实际上，经试验发现，得不到上面的输出，实际的输出如下，从下面的

但实际上，经试验发现，得不到上面的输出，实际的输出如下，从下面的实际输出来看，似乎该编译选项没有起作用。

实际输出:

```
MOVW   A, _i
MOVN   A, #1
ADDW   A
MOVW   A, @RW3+4
MOVW   @AL, AH
MOVW   A, _i
MOVN   A, #1
ADDW   A
MOVW   _j, A
```

• In-line expansion of function below the specified number of lines

它相当于编译选项"-xauto size"，size 为逻辑行的行数目。

其作用是，对 C 源文件中逻辑行的行数目小于和等于 size 的函数，进行内嵌扩展。

[例子]

输入:

```
int p,q,r;
void func1(int);
void func2(int);
void main(void)
{
   func1(1);
   func2(2);
}
void func1(int i)
{
   p=i;
   if(i>=2)
      q=2*i;
}
void func2(int i)
{
   r=i;
}
```

编译时，在编辑框中输入 2，即对应 size(逻辑行的行数目)为 2

输出: (节选)

```
  _main:
        MOVN      A, #3
        PUSHW     A
        CALL      _func1
        POPW      AH
```

```
MOVN    A, #2
MOVW    _r, A
RET
```

■ Category: Target Depend

其作用是，设定编译的内存模式。关于编译的内存模式的详细说明可见于前面的章节。

• Small
其作用是，设定编译的内存模式为小模式。
它相当于编译选项"-model SMALL"。

• Medium
其作用是，设定编译的内存模式为中模式。
它相当于编译选项"-model MEDIUM"。

• Compact
其作用是，设定编译的内存模式为紧凑模式。
它相当于编译选项"-model COMPACT"。

• Large
其作用是，设定编译的内存模式为大模式。
它相当于编译选项"-model LARGE"。

附录二

1 库函数定义的类型,宏和函数

■ **assert.h**

• Function

assert

■ **ctype.h**

• Macros

isalnum	**isalpha**	**iscntrl**	**isdigit**	**isgraph**
islower	**isprint**	**ispunct**	**isspace**	**isupper**
isxdigit	**tolower**	**toupper**		

■ **float.h**

定义了一些浮点数相关的宏(如 单精度/双精度浮点数的最小值,最大值)

• Macros

FLT_RADIX	**FLT_ROUNDS**	**FLT_MANT_DIGT**
DBL_MANT_DIG	**LDBL_MANT_DIG**	**FLT_DIG**
DBL_DIG	**LDBL_DIG**	**FLT_MIN_EXP**
DBL_MIN_EXP	**LDBL_MIN_EXP**	**FLT_MIN_10_EXP**
DBL_MIN_10_EXP	**LDBL_MIN_10_EXP**	**FLT_MAX_EXP**
DBL_MAX_EXP	**LDBL_MAX_EXP**	**FLT_MAX_10_EXP**
DBL_MAX_10_EXP	**LDBL_MAX_10_EXP**	**FLT_MAX**
DBL_MAX	**LDBL_MAX**	**FLT_EPSILON**
DBL_EPSILON	**LDBL_EPSILON**	**FLT_MIN**
DBL_MIN	**LDBL_MIN**	

■ **limits.h**

定义了一些变量表示范围相关的宏(如 整型变量的最小值,最大值)

• Macros

MB_LEN_MAX	**CHAR_BIT**	**SCHAR_MIN**	**SCHAR_MAX**
UCHAR_MAX	**CHAR_MIN**	**CHAR_MAX**	**INT_MIN**
INT_MAX	**UINT_MAX**	**SHRT_MIN**	**SHRT_MAX**
USHRT_MAX	**LONG_MIN**	**LONG_MAX**	**ULONG_MAX**

■ math.h

定义了一些数学运算相关的函数(如 正弦,余弦, 正切,指数函数等等)

• Macros

HUGE_VAL **EDOM** **ERANGE**

• Function

acos	**asin**	**atan**	**atan2**	**cos**
sin	**tan**	**cosh**	**sinh**	**tanh**
exp	**frexp**	**ldexp**	**log**	**log10**
modf	**pow**	**sqrt**	**ceil**	**fabs**
floor	**fmod**			

■ stdarg.h

• Type

va_list

• Macros

va_start **va_arg** **va_end**

■ stddef.h

• Type

ptrdiff_t **size_t**

• Macros

NULL **offsetof**

■ stdio.h

定义了一些标准输入/输出相关的函数(如 打开文件/关闭文件,格式化输入/格式化输出函数等等)

• Type

ptrdiff_t **size_t** **FILE** **fpos_t**

• Macros

NULL	**EOF**	**SEEK_SET**	**SEEK_CUR**	**SEEK_END**
_IONBF	**_IOLBF**	**_IOFBF**	**BUFSIZ**	**stdin**
stdout	**stderr**	**putchar**	**putc**	**getchar**
getc	**offsetof**			

• Function

putchar	**putc**	**getchar**	**getc**	**fclose**
fflush	**fopen**	**freopen**	**setbuf**	**setvbut**
fprintf	**fscanf**	**printf**	**scanf**	**sprintf**
sscanf	**vfprintf**	**vprintf**	**vsprintf**	**fgetc**
fgets	**fputc**	**fputs**	**gets**	**puts**
ungetc	**fred**	**fwrite**	**fgetpos**	**fseek**
fsetpos	**ftell**	**rewind**	**clearerr**	**feof**
ferror				

■ **stdlib.h**

• Type

ptrdiff_t **size_t** **div_t** **ldiv_t**

• Macros

NULL **offsetof** **EXIT_FAILURE** **EXIT_SUCCESS**
RAND_MAX

• Function

atof	**atoi**	**atol**	**strtod**	**strtol**
strtoul	**rand**	**srand**	**calloc**	**free**
malloc	**realloc**	**abort**	**atexit**	**exit**
bsearch	**qsort**	**abs**	**div**	**labs**
ldiv				

■ **string.h**

定义了一些内存操作和字符串操作相关的函数(如 内存块的拷贝/移动/比较,字符串的拷贝/移动/比较 函数等等)

• Type

ptrdiff_t **size_t**

• Macros

NULL **offsetof**

• Function

memcpy	**memmove**	**strcpy**	**strncpy**	**strcat**
strncat	**memcmp**	**strcmp**	**strncmp**	**memchr**
strchr	**strcspn**	**strpbrk**	**strrchr**	**strspn**
strstr	**strtok**	**memset**	**strlen**	

■ **fcntl.h**

定义了一些打开文件的属性相关的宏(如 只读,只写,可读写,追加 的文件打开属性)

• Macros

O_RDONLY	**O_WRONLY**	**O_RDWR**	**O_APPEND**	**O_CREAT**
O_TRUNC	**O_BINARY**			

■ **unistd.h**

定义了移动文件指针相关的宏(如 文件的开头,文件指针的当前位置,文件的末尾)

• Macros

SEEK_SET **SEEK_CUR** **SEEK_END**

■ **sys/types.h**

• Type

off_t