To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

用户手册

CubeSuite Ver. 1.00

集成开发环境

78K0R 编码

目标设备

78K0R 微控制器

文档编号 U19382CA1V0UM00 (第 1 版)
发行日期 2009 年 1 月

日本印刷

[备忘录]

总目录

- **本文档所登载的内容有效期截止至 2009 年 01 月，信息先于产品的生产周期发布。将来可能未经预先通知而更改。在实际进行生产设计时，请参阅各产品最新的数据表或数据手册等相关资料以获取本公司产品的最新规格。**
- 并非所有的产品和/或型号都向每个国家供应。请向本公司销售代表查询产品供应及其他信息。
- 未经本公司事先书面许可，禁止复制或转载本文件中的内容。否则因本文档所登载内容引发的错误，本公司概不负责。
- 本公司对于因使用本文件中列明的本公司产品而引起的，对第三者的专利、版权以及其它知识产权的侵权行为概不负责。本文件登载的内容不应视为本公司对本公司或其他人所有的专利、版权以及其它知识产权作出任何明示或默示的许可及授权。
- 本文件中的电路、软件以及相关信息仅用以说明半导体产品的运作和应用实例。用户如在设备设计中应用本文件中的电路、软件以及相关信息，应自行负责。对于用户或其他人因使用了上述电路、软件以及相关信息而引起的任何损失，本公司概不负责。
- 虽然本公司致力于提高半导体产品的质量及可靠性，但用户应同意并知晓，我们仍然无法完全消除出现产品缺陷的可能。为了最大限度地减少因本公司半导体产品故障而引起的对人身、财产造成损害（包括死亡）的危险，用户务必在其设计中采用必要的安全措施，如冗余度、防火和防故障等安全设计。
- 本公司产品质量分为：

  “标准等级”、“专业等级”以及“特殊等级”三种质量等级。

  “特殊等级”仅适用于为特定用途而根据用户指定的质量保证程序所开发的日电电子产品。另外，各种日电电子产品的推荐用途取决于其质量等级，详见如下。用户在选用本公司的产品时，请事先确认产品的质量等级。

  “标准等级”： 计算机，办公自动化设备，通信设备，测试和测量设备，音频·视频设备，家电，加工机械以及产业用机器人。

  “专业等级”： 运输设备（汽车、火车、船舶等），交通用信号控制设备，防灾装置，防止犯罪装置，各种安全装置以及医疗设备（不包括专门为维持生命而设计的设备）。

  “特殊等级： 航空器械，宇航设备，海底中继设备，原子能控制系统，为了维持生命的医疗设备、用于维持生命的装置或系统等。

  除在本公司半导体产品的数据表或数据手册等资料中另有特别规定以外，本公司半导体产品的质量等级均为“标准等级”。如果用户希望在本公司设计意图以外使用本公司半导体产品，务必事先与本公司销售代表联系以确认本公司是否同意为该项应用提供支持。

（注）

（1） 本声明中的“本公司”是指日本电气电子株式会社（NEC Electronics Corporation）及其控股公司。

（2） 本声明中的“本公司产品”是指所有由日本电气电子株式会社开发或制造的产品或为日本电气电子株式会社（定义如上）开发或制造的产品。

前言

本手册描述了在 78K0R 微控制器的开发应用和系统中所使用的 CubeSuite 综合开发环境，并介绍该软件的特点.

CubeSuite 是 78K0R 微控制器的综合开发环境(IDE), 它集合了在软件开发阶段(如设计, 执行和调试)所必须的工具在一个平台上。

通过这个综合环境, 使得通过它来实现所有开发成为可能，而不需要再分开使用各种不同工具。

读者

本手册适用于想了解 CubeSuite 功能和设计软件及硬件应用系统的用户。

目的

本手册旨在硬件或软件系统开发使用这些设备，给用户在使用 Cubesuite 功能作为参考。

组成

本手册由如下几部分组成

第 1 章 通用
第 2 章 功能
第 3 章 构建输出列表
第 4 章 示例程序
第 5 章 注意事项
第 6 章 函数说明
第 7 章 启动程序
第 8 章 ROM 化
第 9 章 编译程序和汇编程序的引用
第 10 章 注意事项
附录 A 索引

怎样阅读本手册

在阅读本手册前，读者应掌握电子工程、逻辑电路和微控制器等电子工程方面的基础知识。

规定

数据的意义:	数据的高位部分在左边，地位部分在右边
低态有效表示:	$\overline{XXX}$ (在引脚或信号名称上划线表示)
注:	用脚标“注”来表示手册中需要注解的条目
注意:	表示需要特别注意的信息提示
备注:	补充信息
数字表示法:	十进制… XXXX
	十六进制 … 0xXXXX

相关文档

本手册中提到的相关文档可能包括有初稿版本。但是，初稿版本没有特别注明。

文档名称		文档编号
CubeSuite Ver.1.00 集成开发环境 用户手册	启动	U19377E
	编程	U19390E
	消息	U19391E
	78K0R 编码	本手册
	78K0R 构建	U19385E
	78K0R 调试	U19388E
	78K0R 设计	U19379E

注意事项　**上面所列相关文档可能会有新的版本，请确认使用最新版本的文档进行设计、开发等。**

[备忘录]

目录

插图列表

插图编号 .标题 页码

表格列表

第 1 章 概述

本章介绍 78K0R C 编译包 ("CA78K0R") 在系统开发中的作用 , 并且提供其功能概要。

1.1 概要

78K0R C 编译器是将传统 C 或 ANSI C 编写的源程序转换为机器语言的转换程序。 78K0R C 编译器可生成目标文件或汇编源文件。

78K0R 汇编程序是语言处理程序，它将汇编语言编写的源程序转换为机器语言。

1.1.1 C 编译器和汇编程序

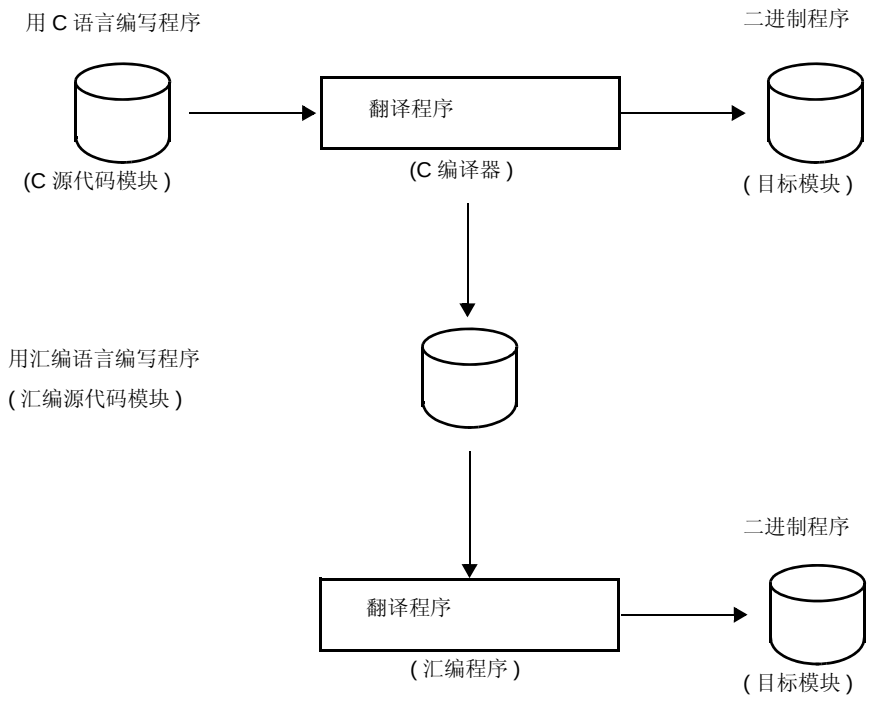
(1) C 语言和汇编语言

C 编译器把 C 源程序模块作为输入，输出目标模块或汇编源代码模块。这意味着能使用 C 语言开发程序，且在需要开发有好的适应性的程序时使用汇编语言。

汇编程序把汇编源代码模块作为输入，且产生目标模块输出。

下图显示使用 C 编译器和汇编程序进行程序开发的流程。

图 1-1. 使用 C 编译器和汇编程序开发流程



(2) 可重定位的汇编程序

在写入单片机的存储器中前，使用汇编程序从汇编源代码文件转换得到机器语言。必须已经决定各机器语言指令写入存储器中的位置。

因此，在汇编程序汇编的机器语言中添加信息，表明各机器语言指令在存储器中的起始位置。

根据存储地址上分配机器语言指令的方法，大体上汇编程序能分为绝对地址汇编程序和可重定位汇编程序。

RA78K0R 是可重定位汇编程序。

- 绝对地址汇编程序

  绝对地址汇编程序通过汇编语言在绝对地址上分配汇编的机器语言指令。

- 可重定位的汇编程序

  在可重定位汇编程序中，地址是通过暂定汇编语言汇编的机器语言指令来决定的。

  绝对地址由随后的链接程序决定。

在过去，当使用绝对汇编程序来创建程序，程序员一般不得不使用单个大块来编写整个程序。然而，当大程序的所有组件包含在单个块中时，程序变得复杂，使它变得很难理解和维护。

要避免这种情况，现在通常大程序的开发分为几个子程序，调用模块，每个功能单元一个。这样的编程技术称为模块化编程。

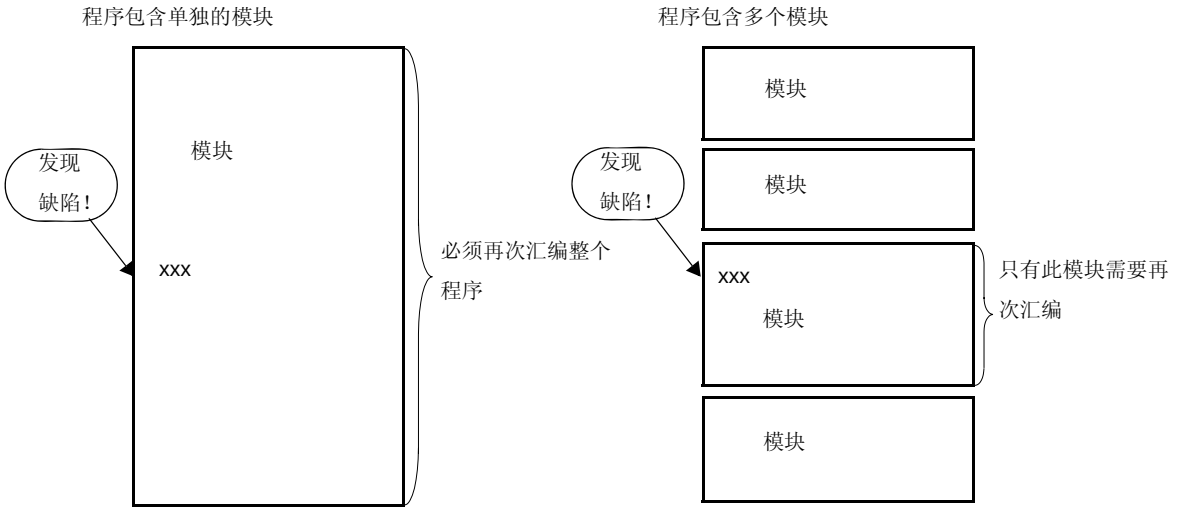
可重定位的汇编程序则非常适合模块化编程，它有以下优点：

(a) 有效的开发效率

同时编写整个大程序比较困难。在这样的情况下，把程序分割为独立功能的模块，能使两个或更多的程序员并行开发子程序，提高了开发的效率。

此外，当发现缺陷时，只需要修正包含缺陷的那个模块，并重新汇编，而不需要汇编整个程序。这样能缩短调试时间。

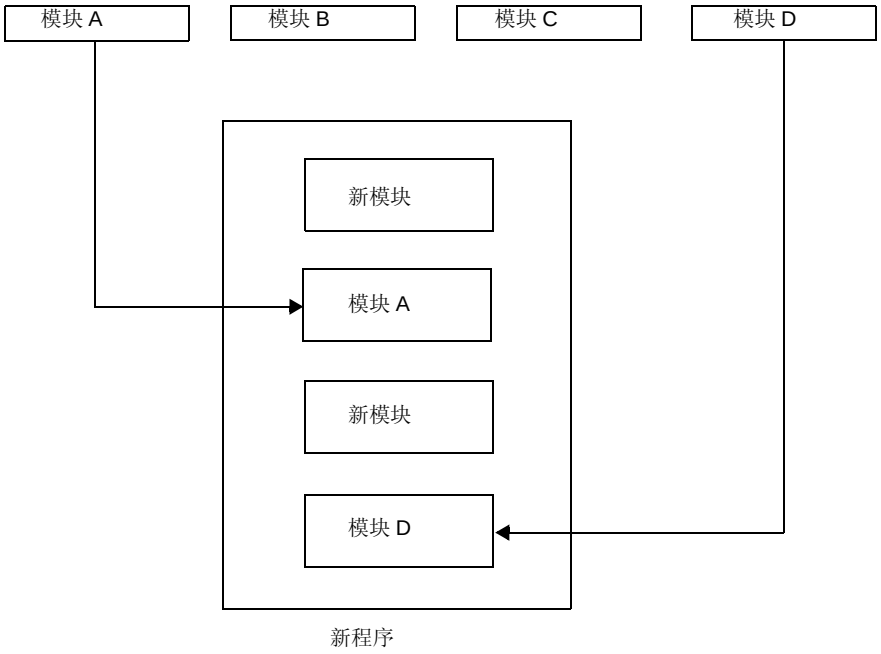
图 1-2. 分割为模块



(b) 资源利用

通过前期有效开发积累的可靠和通用的模块，这些软件资源能在新程序中再使用。当累积了大量资源时，可以在开发新的程序时节省时间和人工。

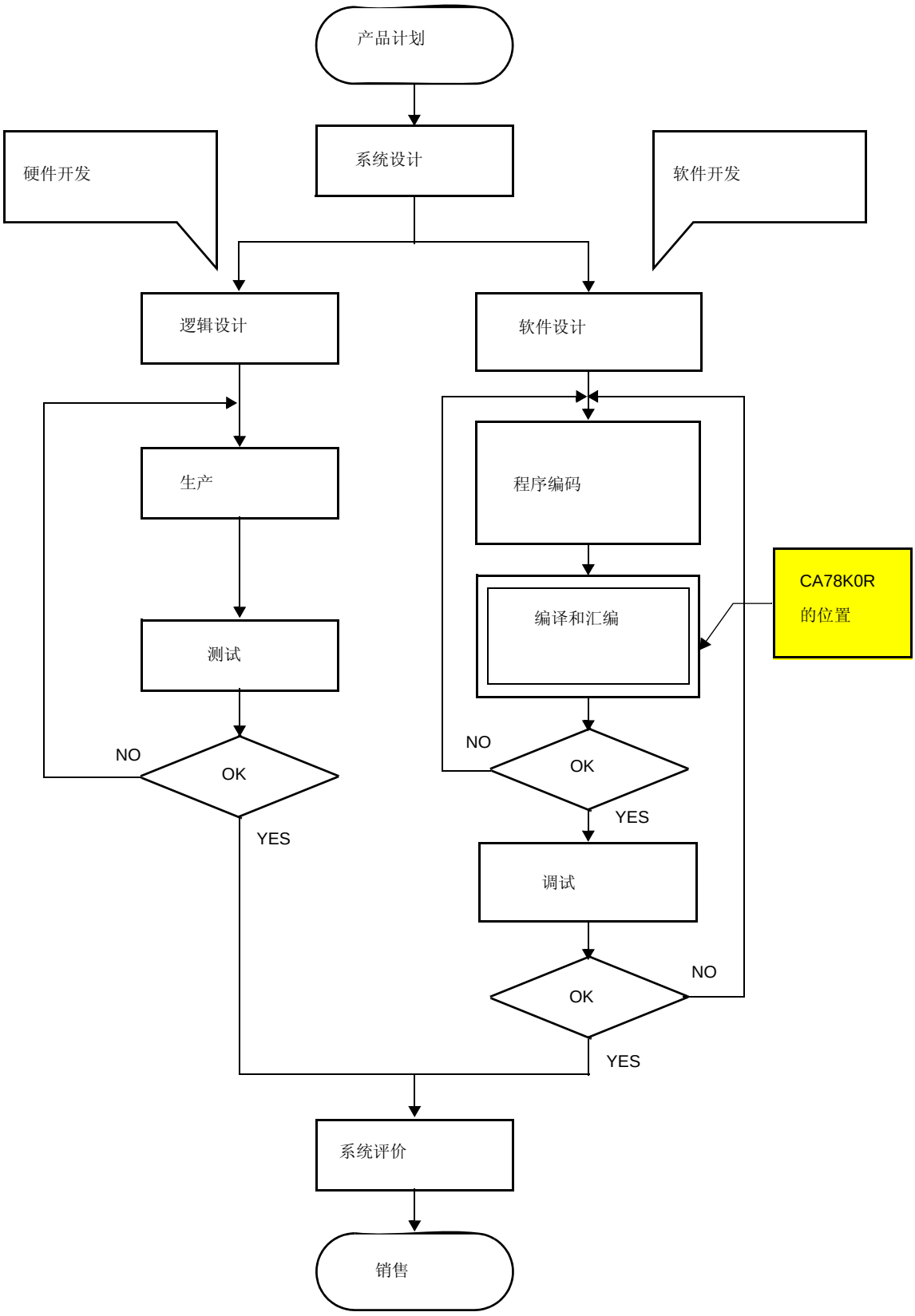
图 1-3. 资源利用



1.1.2 编译器和汇编器位置

下图显示了编译器和汇编程序在产品开发流程中的位置。

图 1-4. 单片机 应用产品开发流程



1.1.3 处理流程

本章介绍程序开发中处理的流程。

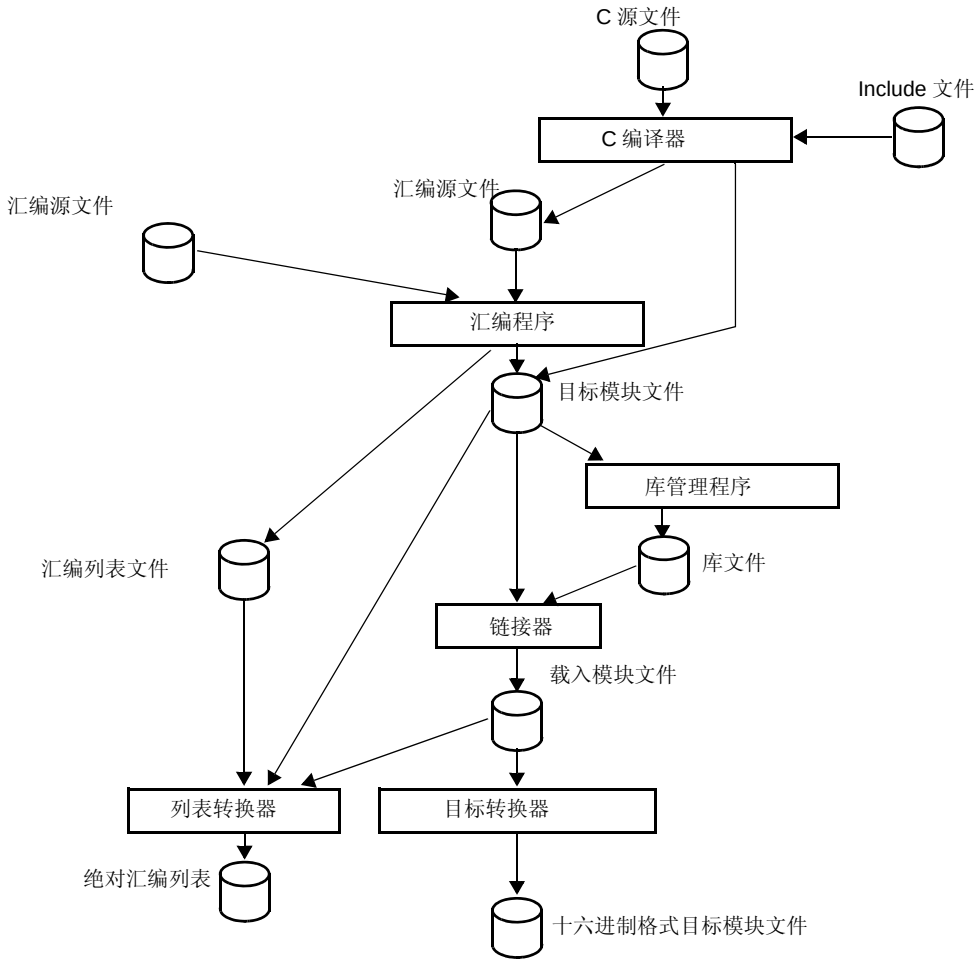
C 编译器编译 C 源模块文件并生成目标文件或汇编源代码模块文件。人工优化生成汇编源模块文件，能创建更有效的目标模块文件。这有利于程序执行高速处理以及需要精简模块。

以下程序包括在处理流程中。

表 1-1. 包括在处理流程中的程序

程序	功能
编译器	编译 C 源代码模块文件
汇编程序	汇编汇编语言的源代码模块文件
链接器	链接目标模块文件 决定可重定位段地址
目标转换器	转换为十六进制格式目标模块文件
库管理程序	创建库文件
列表转换器	生成绝对地址汇编列表文件

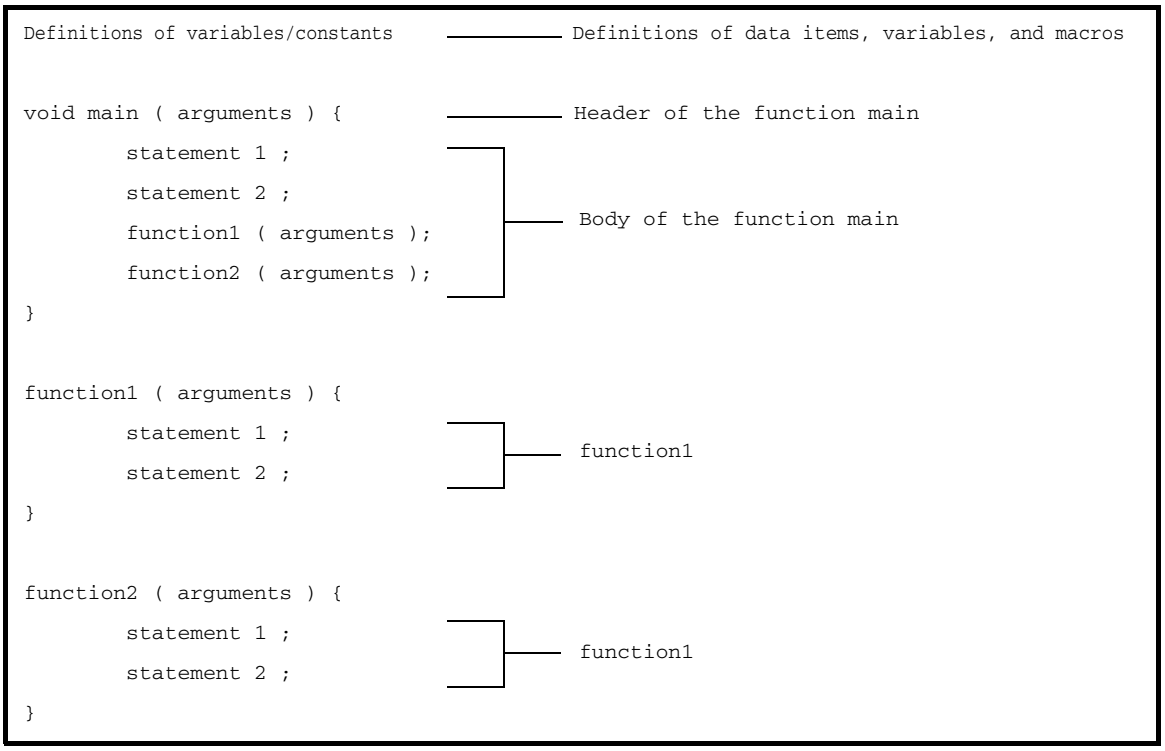
图 1-5. 编译器和汇编程序处理流程



1.1.4 C 源程序的基本结构

用 C 语言编写的程序是函数的集合。函数是独立的代码单位，它执行特定的行为。每个 C 语言程序必须有 "main" 函数，该函数成为程序的主程序且在开始执行时第一个被调用。

每个函数包含头部分和主体部分，头部分定义它的函数名称和参数，主体部分包含声明和语句。 C 程序格式显示如下。



实际 C 源程序看上去像这样。

```
#define TRUE    1       /* #define xxx xxx  Preprocessor directive (macro definition) */
#define FALSE   0       /* #define xxx xxx  Preprocessor directive (macro definition) */
#define SIZE    200     /* #define xxx xxx  Preprocessor directive (macro definition) */

void displaystring ( char*, int ); /* xxx xxxx ( xxx, xxx )
                                         Function prototype declaration */
void displaychar ( char );         /* xxx xxxx ( xxx )
                                     Function prototype declaration */

char    mark[SIZE + 1];               /* char xxx
                                         Type declaration, External definition */
                                      /* xx[xx]           Operator */
void main ( void ) {
    int     i, prime, k, count ;      /* int xxx          Type declaration */

    count = 0 ;                       /* xx = xx          Operator */

    for ( i = 0 ; i <= SIZE ; i ++ )  /* for ( xx ; xx ; xx ) xxx ; Control structure */
        mark[i] = TRUE ;
    for ( i = 0 ; i <= SIZE ; i ++ ) {
```

```
        if ( mark[i] ) {
            prime = i + i + 3 ;              /* xxx = xxx + xxx + xxx   Operator */
            displaystring ( "%6d", prime );  /* xxx ( xxx );            Operator */

            count ++ ;
            if ( ( count%8 ) == 0 ) displaychar ( '\n' );  /* if ( xxx ) xxx ;
                                                                  Control structure */
            for ( k = i + prime ; k <= SIZE ; k += prime )
                mark[k] = FALSE ;
        }
    }
    displaystring ( "\n%d primes found.", count );       /* xxx ( xxx );       Operator */
}

void displaystring ( char *s, int i ) {
    int     j ;
    char    *ss ;

    j = i ;
    ss = s ;
}

void displaychar ( char c ) {
    char    d ;

    d = c ;
}
```

(1) 类型和存储类的声明

声明数据类型和目标标识符的存储类。

(2) 运算符或表达式

执行算术，逻辑，或赋值指令。

(3) 控制结构

设定程序的流程。 C 语言有多个不同控制类型的指令，例如条件控制，循环和转移。

(4) 结构体或联合体

声明结构体或联合体。结构体是数据对象，它包含多个可能不同类型的子对象或成员。联合体类似结构体，但允许两个或更多的变量共享相同的存储空间。

(5) 外部定义

声明函数或外部对象。函数是独立的代码单位，它执行特定的行为。 C 语言程序是函数的集合。

(6) 预处理指令

编译器指令。 #define 指令命令编译器替换第一个操作数中的任何实体，它出现在程序的第二个操作数中。

(7) 函数原型的声明

声明返回值和函数参数的类型。

1.2 特性

本节介绍 CA 的特性。

1.2.1 C 编译器的特性

(1) 符合 ANSI C

编译器符合 C 语言的 ANSI 标准。

备注 ANSI: 美国国家标准化组织

(2) 为有效使用 ROM 和 RAM 存储器而设计

- 外部变量能分配在短寻址存储区内。
  函数参数和自动变量能分配在短寻址存储区内或寄存器中。
- 位指令能以 1 位为单位对数据进行定义和操作。

(3) 中断控制特性

- 78K0R 的外围硬件能直接通过 C 语言来控制。
- 中断处理程序能直接用 C 语言来编写。

(4) 支持 78K0R 的扩展函数

78K0R C 编译器支持以下扩展函数，它们在 ANSI 标准中没有定义。这些函数中某些函数，在 C 语言中可以访问特殊用途寄存器，而其它可以使目标代码更精简且更高速的运行。
以下表格列出了减少目标代码大小以及提高执行速度的扩展函数。

表 1-2. 提高执行速度的方法

方法	扩展函数
分配变量到寄存器中	寄存器变量
分配变量到 saddr 区域	sreg/__sreg
使用 sfr 名称。	sfr 区域
在 C 语言源代码程序中嵌入汇编语言语句。	ASM 语句
能逐位访问 saddr 或 sfr 区域。	位类型变量，布尔 /__boolean 类型变量
使用无符号 char 型来设定位字段。	位字段声明
增加的代码能以内联扩展的方式直接输出。	增加函数
循环的代码能以内联扩展的方式直接输出。	循环函数
特定数据和指令能直接嵌入在代码区域中。	数据插入函数
memcpy 和 memset 能直接扩展内联和输出。	存储函数

有关 78K0R C 编译器扩展函数的详细信息，参阅 "3.3　扩充语言规范 "。

1.2.2 汇编程序特性

78K0R 汇编程序有以下特性。

(1) 宏功能

当在源程序中出现多次相同的指令组，可以定义宏并给该指令组制定一个单独的名称。
宏能提高编码的效率，且让程序更具可读性。

(2) 优化转移指令

78K0R 汇编程序提供 BR 和 CALL (分支指令自动选择指令)。
选择恰当的转移指令可有效利用存储器，只使用转移目标范围需要字节数。但需要为每次转移考虑转移目标，对程序员来说是负担。 BR 和 CALL 指令可自动解决这个问题。它们通过指示汇编程序为转移目标范围生成最恰当的转移指令从而进行有效存储器转移。在转移指令优化时调用这个函数。

(3) 条件汇编

条件汇编允许设定条件来决定是否汇编源程序的特定段。
举个例子，在源代码中包含调试语句时，可设置开关决定其是否应该可以翻译为机器语言。在它们不再需要时，它们能在输出时被排除而不需要在源程序中做大的修改。

(4) 78K0 兼容宏功能

78K0 的汇编程序可以对汇编程序源文件进行汇编。
在汇编 78K0 指令时，若改变源代码的说明之前不能用在 78K0R 上，设定 -compati 选项。

78K0 指令不能使用在 78K0R 上 : DIVUW/ROR4/ROL4/ADJBA/ADJBS/CALLF/DBNZ

1.2.3 限制

(1) 编译器限制

有关编译器的限制，参阅 "(9) 翻译限制 "。

(2) 汇编程序限制

表 1-3. 汇编程序翻译 限制

说明	限制
符号数量 (局部 + 公共)	65,535
交叉引用列表输出符号数量	65,534 注 1
宏引用宏主体的最大容量	1 兆字节
所有宏主体的总大小	10 兆字节
在文件中段的数量	256
在文件中宏和 include 标识的数量	10,000
在 include 文件中宏和 include 标识的数量	10,000
重定位数据项的数量注 2	65,535
数据项的行数	65,535

说明	限制
在文件中 BR/CALL 指令	32,767
源代码行的字符长度	2,048 注 3
符号的字符长度	256
名称定义的字符长度 注 4	1,000
开关定义的字符长度 注 4	31
段名称的字符长度	8
模块名称的字符长度 （NAME 指令）	256
使用 MACRO 指令的参数数量	16
在宏引用中参数的数量	16
在 IRP 指令中参数的数量	16
在宏主体内局部符号的数量	64
在扩展宏中局部符号的总数	65,535
在宏中的嵌套级别 （宏引用， REPT 指令， IRP 指令）	8 级
在 TITLE 控制指令中字符的数量 （-lh 选项）	60 注 5
在 SUBTITLE 控制指令中字符的数量	72
在一个文件中 include 文件的嵌套级别	16 级
有条件的汇编嵌套级别	8 级
通过 -i 选项可设定 include 文件路径的数量	64
通过 -d 选项可定义符号数量	30

注 **1.** 不包括模块名称和段名称的数量。
使用可用存储器。当存储空间不足时，使用文件。

2. 在汇编程序不能决定符号值时，信息传递给链接器。
举例，在 MOV 指令引用外部引用符号，则在 .rel 文件中生成两项重定位信息。

3. 包含 CR 和 LF 码。如果行长超过 2048 字符，则输出警告消息，且忽略第 2049 个字符及以后字符。

4. 通过 SET 和 RESET 指令设置开关名称为真 / 假，且通过构建来使用例如 $lf。

5. 如果字符的最大个数能设定在汇编列表文件的一行中 ("X") 为 119，这个数字将是 "X - 60" 或更小。

(3) 链接器限制

表 1-4. 链接器限制

说明	限制
符号数量 (局部 + 公共)	65,535
在一段中数据项的行号	65,535
段数量	65,535 注
输入模块的数量	1,024
存储区域名称的字符长度	256
存储区域的数量	100 注
通过 -b 选项可设定库文件的数量	64
通过 -i 选项可设定 include 文件路径的数量	64

注 包含那些默认定义

第 2 章 函数

本章介绍了有关更有效地使用 CA78K0R 以及扩展函数的使用的编程技术。

2.1 变量 （汇编语言）

本节介绍在汇编语言中如何使用变量。

2.1.1 定义不带初始值的变量

在数据段分配存储区域。

使用类似 DSEG 指令来定义数据段，并使用 DS 伪指令来分配存储区域。

示例 定义不带初始值的 10 字节变量。

```
        DSEG
_table: DS      10
```

备注 参见 "DSEG" 和 "DS"。

2.1.2 定义带初始值的 const 常量。

在代码段初始化存储区域。

使用 CSEG 伪指令定义代码段 , 并使用 DB (1 个字节), DW (2 个字节), 或 DG (4 个字节) 伪指令来初始化存储区域。

示例 定义带初始值的常量。

```
        CSEG
_val1:  DB      0F0H    ; 1 byte
_val2:  DW      1234H   ; 2 bytes
_val3:  DG      56789H  ; 4 bytes (20 bits)
```

备注 参见 "CSEG", "DB", "DW", 和 "DG"。

2.1.3 定义 1 位变量

在位段中分配 1 位存储区域。

使用 BSEG 伪指令来定义位段，并使用 DBIT 伪指令来分配 1 位存储区域。

示例 定义不带初始值的位变量。

```
        BSEG
_bit1   DBIT
_bit2   DBIT
_bit3   DBIT
```

备注 参见 "BSEG" 和 "DBIT"。

2.1.4 变量的 1/8 位访问

在汇编语言源代码中，在 saddr 区域为地址提供两个符号。符号名称分别用作位存取和字节存取，使用 DSEG 段的重定位属性设定 saddr，为字节存取定义符号的位名，使用 EQU 伪指令为位存取定义符号名。

示例 字节存取符号名称： FLAGBYTE
位存取符号名称： FLAGBIT

- smp1.asm

```
        NAME    SMP1
        PUBLIC  FLAGBYTE, FLAGBIT

FLAGS   DSEG    SADDR           ; The relocation attribute of DSEG is SADDR
FLAGBYTE: DS (1)                ; Define FLAGBYTE
FLAGBIT EQU     FLAGBYTE.0      ; Define FLAGBIT
        END
```

- smp2.asm

```
        NAME    SMP2

        EXTRN   FLAGBYTE
        EXTBIT  FLAGBIT         ; FLAGBIT declared as EXTBIT

        CSEG
C1:
        MOV     FLAGBYTE, #0FFH
        CLR1    FLAGBIT
        END
```

备注 参见 "DSEG" 和 "EQU"。

2.1.5　使用短指令分配可存取段

与其他数据存储区域比较，可以使用短指令存取短寻址区域。有效使用这个区域，可提高程序的有效存储。
在短寻址区域进行分配，用 DSEG 伪指令的重定位属性来设定 saddr 或 saddrp。
以下例子介绍在汇编源代码中的使用。

- 模块 1

```
PUBLIC  TMP1, TMP2
WORK    DSEG saddrp
TMP1:   DS 2 ; word
TMP2:   DS 1 ; byte
```

- 模块 2

```
EXTRN   TMP1, TMP2
SAB     CSEG
MOVW    TMP1, #1234H
MOV     TMP2, #56H
        :
```

备注　　参见 "DSEG"。

2.2 变量 （C 语言）

本节介绍变量 （C 语言）。

2.2.1 分配只在 ROM 中引用的数据

(1) 在 ROM 中分配带初始值的变量

使用 const 标识分配只在 ROM 中引用的带初始值的变量。

示例 分配只在 ROM 中引用的带初始值的变量 "a"

```
const int a = 0x12 ;  /* Allocating ROM */
int b = 0x12 ;        /* Allocating ROM/RAM */
```

变量 "a" 分配在 ROM 中。
变量 "b", 初始值分配在 ROM 中且变量本身分配在 RAM 中 （需要占用 ROM 和 RAM 中的区域）。
启动程序 ROM 化，在 RAM 变量中复制 ROM 中的初始值。
ROM 化需要 ROM 和 RAM 区域。

(2) 在 ROM 中分配表数据。

如果只在 ROM 中分配表数据，定义类型标识 const，如下。

```
const unsigned char table_data[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

2.2.2 使用短指令分配可存取段

与其他数据存储区域比较，可以使用短指令存取短寻址区域。有效使用这个区域，可提高程序的有效存储。
使用示例如下。

定义外部变量为 sreg 或 __sreg, 以及在函数内静态变量 (sreg 变量) 自动分配在可重定位的短需要寻址区域。 [FFE20H 到 FFEB3H]。

```
extern sreg int hsmm0 ;
extern sreg int hsmm1 ;
extern sreg int *hsptr ;

void main ( void ) {
        hsmm0 -= hsmm1 ;
}
```

备注 参见 " 如何使用 saddr 区域 (sreg/__sreg)"。

2.2.3　在 near 区域的分配

使用 small 模式，编译器生成 16 位地址长度的代码。
当预先知道代码和数据在 64KB 内，通过使用小模式而不用大模式获取更多的紧致码。
使用编译选项 (-ms 选项) 设定 small 模式。分配数据和函数在 near 区域内。
或为变量和函数声明添加 __near 类型标识。

```
__near int func ( void );            /* Allocating in near area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__near int func ( void ) {           /* Allocating in near area */
        /* Function processing */
        return 0 ;
}
```

备注　　参考 "near/far 区域规范 "。

2.2.4　分配在 far 区域内

使用 large 模式，编译器生成 20 位地址长度的代码。
如果数据在 64KB 内且代码在 1MB 内，则使用 medium 模式。
使用编译器选项 (-mm 选项) 设定 medium 模式。在 near 区域分配数据且在 far 区域中分配函数。
或为变量和函数声明添加 __near 和 __far 类型标识。

```
__far  int func ( void );            /* Allocating in far area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing */
        return 0 ;
}
```

如果代码和数据在 1MB 内，则使用 large 模式。
使用编译器选项 (-ml 选项) 设定 large 模式。分配数据和函数在 far 区域内。
或为变量和函数声明添加 __far 类型标识。

```
__far  int func ( void );            /* Allocating in far area */
__far const int a = 0 x 12 ;         /* Allocating in far area */
__far int b = 0 x 12 ;               /* Allocating in far area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing */
        return 0 ;
}
```

备注　　参考 "near/far 区域规范 "。

2.2.5　直接分配地址

(1) direcrtmap

声明外部变量 __directmap 以及在函数中静态变量的初始值分配地址，映射变量到指定地址。用整数型数字来设定分配地址。

在 C 源文件中处理 __directmap 变量和静态变量。

在模块中进行 __directmap 声明，定义变量并映射到绝对地址上。

```
__directmap  char c = 0xffe00 ;
__directmap  __sreg char d = 0xffe20 ;
__directmap  __sreg char e = 0xffe21 ;

__directmap  struct x {
        char a ;
        char b ;
} xx = { 0xffe30 };

void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

备注　　参见 " 绝对地址分配规范 (__directmap)"。

(2) 使用段名称

改变编译器输出段名称并设定起始地址。

使用 #pragma 指令改变设定的段名称 , 新的名称 , 以及新段的起始地址。

以下例子改变段名称 @@CODEL 为 CC1，并设定起始地址为 2400H。

```
#pragma section @@CODEL CC1 AT  2400H

void main ( void ) {
        /* Function definition */
}
```

备注 1.　　参见 " 改变编译器输出区段名称 (#pragma section ...)"。

2.2.6　定义 1 位变量

用位和布尔定义变量，且作为 1 位数据处理，分配在短寻址区域。
与处理不带初始值的外部变量相同的方式来处理位和布尔类型变量。
编译器使用以下位操作指令操作位变量。

```
MOV1 , AND1 , OR1 , XOR1 , SET1 , CLR1 , NOT1 , BT , BF
```

C 源代码中可对短寻址区域进行位读写。

```
#define ON      1
#define OFF     0
extern bit data1 ;
extern bit data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while (data1 ) {
                data1 = data2 ;
                testb ( );
        }
        if ( data1 && data2 ) {
                chgb ( );
        }
}
```

备注　　参见 "bit 型变量 (bit), boolean 型变量 (boolean/__boolean)"。

2.2.7　填充结构的空区域

设定 -rc 选项，在 2 字节边界上反选结构体成员排列。
然而，不支持反选非结构体变量排列。

2.3　函数

本节介绍函数。

2.3.1　使用短指令分配可存取段

使用 callt 函数调用，获得的代码要比一般函数调用的代码更紧凑。

调用指令存储调用函数的地址在区域 [80H - 0BFH] 中并调用 callt 表。与直接调用函数相比更可能短线路调用函数。

```
__callt void func1 ( void );
__callt void func1 ( void ) {
            :
        /* Function definition */
            :
}
```

备注　　参阅 "call 函数 (callt/__callt)"。

2.3.2　在 near 区域的分配

使用 small 模式，编译器生成 16 位地址长度的代码。

当预先知道代码和数据在 64KB 内，通过使用小模式而不用大模式获取更多的紧致码。

使用编译选项 (-ms 选项) 设定 small 模式。函数分配在 near 区域。

或为函数声明添加 __near 类型标识。

```
__near int func ( void );              /* Allocating in near area */
__near const int a = 0 x 12 ;          /* Allocating in near area */
__near int b = 0 x 12 ;                /* Allocating in near area */

__near int func ( void ) {             /* Allocating in near area */
        /* Function processing */
        return 0 ;
}

void main ( void ) {
        int     a;
        a = func ( ) ;
}
```

备注　　参考 "near/far 区域规范 "。

2.3.3　分配在 far 区域内

如果数据在 64KB 内且代码在 1MB 内，则使用 medium 模式。
使用编译器选项 (-mm 选项) 设定 medium 模式。函数分配在 far 区域。
如果代码和数据在 1MB 内，则使用 large 模式。
使用编译器选项 (-ml 选项) 设定 large 模式。分配数据和函数在 far 区域内。
或为函数声明添加 __far 类型标识。

```
__far  int func ( void );            /* Allocating in far area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing */
        return 0 ;
}

void main ( void ) {
        int     a;
        a = func ( ) ;
}
```

备注　　参考 "near/far 区域规范 "。

2.3.4　直接分配地址

(1) 使用段名称

改变编译器输出段名称并设定起始地址。
使用 #pragma 指令改变设定的段名称 , 新的名称 , 以及新段的起始地址。

```
#pragma section @@DATA  ??DATA  AT 0FDE00H
int             a1 ;                          // ??DATA
int             a2 ;                          // ??DATA
#pragma section @@DATS  ??DATS  AT 0FFE30H
sreg int        b1 ;                          // ??DATS
sreg int        b2 ;                          // ??DATS
```

备注　　参见 " 改变编译器输出区段名称 (#pragma section ...)"。

2.3.5　函数的行内展开

内嵌 #pragma 指示生成存储操作标准库 memcpy 和 memse 的内联展开代码 , 而不是调用函数。

如果通过展开其他内联函数能使执行更快，没有指令能内联展开每个函数。如果除 memcpy 和 memset 以外的函数能内联展开，用函数格式定义宏，如下所示。

```
#define MEMCOPY ( a, b, c ) \
        { \
                struct st { unsigned char d[ ( c ) ]; }; \
                * ( ( struct st * ) ( a ) ) = * ( ( struct st * ) ( b ) ); \
        }
```

备注　　参见 " 存储器运算函数 (#pragma inline)"。

2.3.6　嵌入汇编指令

在汇编程序源文件中嵌入汇编指令并由编译器输出。

(1) #asm - #endasm

#asm 标记汇编源代码块的起始位置，且 #endasm 标记结束位置。在 #asm 和 #endasm 间编写汇编源代码。

```
#asm
    :   /* Assembly source */
#endasm
```

在属性面板的 [Compile Options] 标签的 [Output assemble file] 中 , 设置为 "Yes"。 (有关设置的方法，参阅 "CubeSuite 78K0R build"。)

备注　　参见 "ASM 语句 (#asm - #endasm/__asm)"。

(2) __asm

在 C 源代码中用下面的格式来描述。

```
__asm ( string literal );
```

在字符串中的字符根据 ANSI 规定逐字进行解释。取消队列，下一行 '\' 字符后继续，且能描述连接字符串。
在属性面板的 [Compile Options] 标签的 [Output assemble file] 中 , 设置为 "Yes"。 (有关设置的方法，参阅 "CubeSuite 78K0R build"。)

备注　　参见 "ASM 语句 (#asm - #endasm/__asm)"。

2.4 使用单片机 函数

本节介绍使用单片机的函数。

2.4.1 从 C 中访问特殊功能寄存器 （SFR)

(1) 设置各 SFR 寄存器

SFR 区域是特殊功能寄存器组区域，例如 78K 单片机外围硬件的模式和控制寄存器 （PM1, P1, TMC80, 等）。
从 C 中使用 SFR 区域，放置 #pragma sfr 在 C 源文件的开头。这样声明各 SFR 寄存器的名称。sfr 关键字可为大写字母或小写字母。

```
#pragma sfr
```

如果在没有声明寄存器名称的情况下企图使用 SFR 区域，则出现以下错误信息。

```
E0711 Undeclared 'variable-name' ; function 'function-name'
```

通过 #pragma sfr 指令使符号可用，符号与特殊功能寄存器列表中的缩写相同。
在 #pragma sfr 之前能描述以下项目：

- 注释
- 预处理指令既不能定义也不能引用变量或函数

在 C 源代码中，目标设备支持使用简单的 sfr 名称。 sfr 名称不需要单独声明。
SFR 名称是没有初始值的外部变量 （不规则）。
如果分配无效常数数据在 SFR 名称中，则编译器产生错误。

备注 参见 " 如何使用 sfr 区域 (sfr)"。

(2) 在 SFR 寄存器中指定位

如下所示，通过使用保留名称或通过使用 " 寄存器名 . 位位置 " 来设定在 SFR 寄存器中的位。

示例 1. 开始 TM1

```
TCE1 = 1 ;
or
TMC1.0 = 1 ;
```

2. 停止 TM1

```
TCE1 = 0 ;
or
TMC1.0 = 0 ;
```

2.4.2　C 编写的中断函数

(1) 中断函数

在设定中断函数时，提供以下两条指令。

- #pragma interrupt
- #pragma vect

都能使用且生成向量表，它能在汇编程序源代码列表中核对输出。

在 C 源文件的开头放置 #pragma 指令。

在 #pragma 之前能描述以下项目：

- 注释
- 预处理指令既不能定义也不能引用变量或函数

示例　　输入到 INTP0 引脚的处理

```
#pragma interrupt INTP0 inter rb1
void inter ( void ) {
        /* Processing for input to INTP0 pin*/
}
```

备注　　参见 "　中断函数 (#pragma vect/#pragma interrupt)"。

(2) RTOS 中断程序

通过使用 #pragma rtos_interrupt 描述 RTOS 中断程序，显示如下。

在 C 源文件的开头放置 #pragma 指令。

在 #pragma 之前能描述以下项目：

- 注释
- 预处理指令既不能定义也不能引用变量或函数

```
#pragma rtos_interrupt INTP0 inthdr1

#include "kernel.h"
#include "kernel_id.h"

void inthdr1 ( void ) {
        /* Handle the interrupt */
        return ;
}
```

备注　　参考 "　RTOS 中断处理 (#pragma rtos_interrupt ...)"。

(3) 分配堆栈区域

当使用中断函数的扩展函数时，不要设定堆栈开关，编译器使用默认堆栈。它不分配任何需要的外部堆栈空间。

2.4.3　在 C 中使用 CPU 控制指令

(1) halt 指令

halt 指令是单片机待机功能之一。要使用它，使用 #pragma HALT 显示如下。
在 C 源文件的开头放置 #pragma 指令。
在 #pragma 之前能描述以下项目：
- 注释
- 预处理指令既不能定义也不能引用变量或函数

在与函数调用相似的表格中，在 C 源代码中用大写字母对它描述如下。

示例　　使用 halt 指令

```
#pragma HALT
    :

void func ( void ) {
    :
        HALT ( );
}
```

备注　　参见 "　CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)"。

(2) stop 指令

stop 指令是单片机待机功能之一。要使用它，使用 #pragma STOP 显示如下。
在 C 源文件的开头放置 #pragma 指令。
在 #pragma 之前能描述以下项目：
- 注释
- 预处理指令既不能定义也不能引用变量或函数

在与函数调用相似的表格中，在 C 源代码中用大写字母对它描述如下。

示例　　使用 stop 指令

```
#pragma STOP
        :

void func ( void ) {
        :
        STOP ( );
}
```

备注　　参见 "　CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)"。

(3) brk 指令

使用单片机的软件中断，使用 #pragma BRK 显示如下。
在 C 源文件的开头放置 #pragma 指令。
在 #pragma 之前能描述以下项目：
- 注释
- 预处理指令既不能定义也不能引用变量或函数

在与函数调用相似的表格中，在 C 源代码中用大写字母对它描述如下。

示例　　使用 brk 指令

```
#pragma BRK
        :

void func ( void ) {
        :
        BRK ( );
}
```

备注　　参见 "　CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)"。

(4) nop 指令

nop 指令在不操作单片机的情况下消耗时钟。要使用它，使用 #pragma NOP 显示如下。
在 C 源文件的开头放置 #pragma 指令。
在 #pragma 之前能描述以下项目：
- 注释
- 预处理指令既不能定义也不能引用变量或函数

在与函数调用相似的表格中，在 C 源代码中用大写字母对它描述如下。

示例　　使用 nop 指令

```
#pragma NOP
        :

void func ( void ) {
        :
        NOP ( );
}
```

备注　　参见 "　CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)"。

2.5　启动程序

本章介绍启动程序。

2.5.1　从启动程序中删除不使用的函数和区域

(1) 删除 exit 函数

通过在启动程序中设置 EQU 符号 EXITSW 为 0，删除 exit （退出）函数

(2) 删除未用区域

有关类似 _ @FNCTBL 区域的未用区域，通过确定使用的库来删除使用标准库，并改变 EQU 符号的值，如启动程序 cstart.asm 中的 EXITSW。

以下表格列出了控制 EQU 符号和受影响的库函数以及符号名称。

EQU 符号	库函数名称	符号名称
BRKSW	brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR
EXITSW	exit	_@FNCTBL _@FNCENT
RANDSW	rand srand	_@SEED
DIVSW	div	_@DIVR
LDIVSW	ldiv	_@LDIVR
STRTOKSW	strtok	_@TOKPTR
FLOATSW	vatof strtod 数学函数 浮点运行时间库	_errno

备注　　参见 "7.4　启动程序 "。

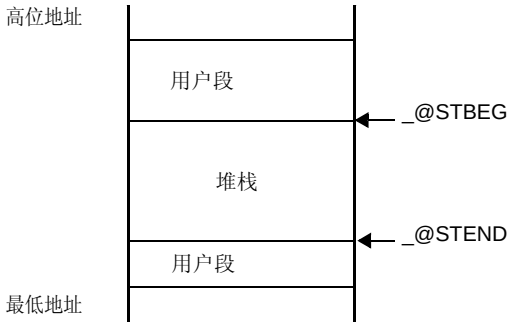
2.5.2　分配堆栈区域

(1) 堆栈设置

在链接时，如果设定堆栈解析符号选项 -s，则生成的符号 _@STEND 标记堆栈最低地址上，且生成的符号 _@STBEG 标记为最高地址 +1。

```
-sSTACK     <-- 通过指令定义堆栈区域
```

图 2-1.　堆栈设置



在这种情况下，设置堆栈指针如下。

```
MOVW    SP, #LOWW       _@STBEG
```

(2) 校对堆栈区域

校对堆栈区域，设定链接器的 -kp 选项，并在链接列表文件中输出公共符号。
堆栈区域是在 _@STEND 符号和 _@STBEG 符号间。

示例　　公用符号列表

```
*** Public symbol list ***

MODULE  ATTR  VALUE NAME

        NUM     0FFE20H  _@STBEG
        NUM     0FFB7EH  _@STEND
```

2.5.3　初始化 RAM

在默认的启动程序中，复制初始值到以下区域。

- @@INIT 段
- @@INITL 段
- @@INITL 段

以下区域清零。

- saddr 区域 (0FFE20H 到 0FFEDFH)
- @@DATA 段
- @@DATAL 段
- @@DATAL 段

如果是除以上以外的初始化区域，则添加合适的初始化处理代码到启动程序中。

备注　　参见 "7.4　启动程序 "。

2.6　链接指令

本节介绍链接指令。

2.6.1　划分默认区域

链接指令允许为定义的存储区域设定名称。然而，有关特殊功能寄存器 （SFR) 区域的位置必须要小心。

例如，如果在 RAM 中定义两个区域并且设定 1）的名称默认定义为 "RAM"，且 2）用户定义的名称 "STACK"，则应该确认 SFR 区域包含在名为 RAM 的区域内。

示例　　链接指令

```
MEMORY  STACK : ( 0FEF00H , 00100H )
MEMORY  RAM :   ( 0FF000H , 01000H )
```

备注　　参见 "5.1.1　链接指令 "。

2.6.2　设定 section 分配

(1) 指定区域

在指定 section 分配时，能设定存储区域。

使用 MERGE 伪类似指令在存储区域内分配目标 section 。

示例　　分配输入段 SEG1 到存储区域 MEM1

```
MEMORY ROM  : ( 0000H, 1000H )
MEMORY  MEM1 : ( 1000H , 2000H )
MERGE   SEG1 : = MEM1
```

备注　　参见 "5.1.1　链接指令 "。

(2) 指定地址

在指定 section 分配时，能设定地址。

使用 MERGE 伪指令设定目标 section 的分配地址。

示例　　分配输入段 SEG1 到地址 500H。

```
MEMORY  ROM : ( 0000H , 10000H )
MERGE  SEG1 : AT ( 500H )
```

备注　　参见 "5.1.1　链接指令 "。

2.7　减少代码 大小

本节介绍减少代码大小的技术。

2.7.1　使用扩展函数生成有效目标代码

在 78K0R 开发应用产品时，78K0R 通过在设备中使用 saddr 和 callt 区域， C 编译器生成有效目标代码。

- 使用外部变量
  - 如果 （saddr 区域可用）
    - 使用 sreg/__sreg 变量 /
      或编译器的 -rd 选项
- 使用 1 位数据
  - 如果 （saddr 区域可用）
    - 使用 位 / 布尔 /__boolean 类型变量
- 函数定义
  - 如果 （反复调用函数）
    - 如果 （callt 区域可用）
      - 定义为 __callt/callt 函数 （更小代码量）
  - 如果 （使用自动变量 && saddr 区域可用）

(1) 使用外部变量

如果当定义的外部变量在 saddr 区域可用，则定义外部变量为 sreg/__sreg 变量。
sreg/__sreg 变量的指令代码要比存储器中的指令代码要短。目标代码将更少且执行速度将更快。（能使用编译器 -rd 选项，而不是 sreg 变量。

```
sreg/__sreg variable define :   extern sreg int    variable-name ;
                                extern __sreg int variable-name ;
```

备注　　参见 " 如何使用 saddr 区域 (sreg/__sreg)"。

(2) 使用 1 位数据

当只使用数据中的 1 位时，定义位类型 (或布尔 /__boolean 类型) 变量。编译器生成位操作指令来操作位 / 布尔 /__boolean 类型变量。比如 sreg 变量，为了更少的代码量和更快的执行速度，存储在 saddr 区域。

```
bit/boolean type variable define :  bit             variable-name ;
                                    boolean         variable-name ;
                                    __boolean       variable-name ;
```

备注　　参见 "bit 型变量 (bit), boolean 型变量 (boolean/__boolean)"。

(3) 函数定义

在使用 callt 区域时，反复调用的函数可以注册在 callt 表中。
通过使用设备 callt 区域来调用 callt 函数，它们能以比正常调用函数更少的代码来完成调用。

```
调用函数定义 : callt    int      tsub ( ) {
                                 :
                             }
```

备注　　参阅 "call 函数 (callt/__callt)".

不改变 C 源代码，使用优化选项编译，且使用 saddr 区域，这样可能会生成高质量目标。

(4) 使用扩展函数

- 函数定义
  - 如果 （使用自动变量 && saddr 区域可用）
    - 寄存器定义
  - 如果 （使用内部静态变量） && （ saddr 区域可用）
    - __sreg 定义

(a) 函数使用自动变量

在使用自动变量的函数可使用 saddr 区域，定义寄存器。寄存器定义分配定义的目标到寄存器中。
程序使用寄存器要比程序使用存储器有更短的目标代码和更高速的执行。

备注　　有关定义寄存器变量 （寄存器 int i;... 参见 " 寄存器变量 (寄存器)"。

(b) 函数使用内部静态变量

在使用内部静态变量的函数可以使用 saddr 区域时，定义 __sreg 或设定 -rs 选项。就像 sreg 变量，可以缩小目标代码和有更高的执行速度。

备注　　参见 " 如何使用 saddr 区域 (sreg/__sreg)"。

(5) 其他函数

其他扩展函数可以生成更高速执行或更紧凑的代码。

(a) 使用 SFR 名称 （或 SFR 位名称）

#pragma sfr

备注　　参见 " 如何使用 sfr 区域 (sfr)"。

(b) __sreg 可为 1 位成员定义位段 (成员也能使用无符号 char 类型）

```
__sreg  struct  bf {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
} bf_1 ;
```

备注　　参见 " 如何使用 saddr 区域 (sreg/__sreg)"。

(c) 使用中断程序的寄存器组开关。

#pragma interrupt INTP0 inter rb1

备注　　参阅 " 中断函数 (#pragma vect/#pragma interrupt)"。

(d) 嵌入式乘法，除法函数的使用

#pragma mul

#pragma div

备注　　参阅 " 乘法函数 (#pragma mul)" 除法函数 (#pragma div)。

(e) 使用汇编语言描述达到更高速

2.7.2 计算复杂表达式

以下例子显示以最合理的方法计算表达式，表达式的结果将总是适合字节类型，即使在需要双字类型的中间结果时。

示例 在 B 在 A 中取整百分比 C。

```
C = ( A x 100 + B  2 )  B
```

在类似以下函数中，结果 C 的变量必须定义为 long int, 在单个字节足够时，需要 4 个字节区域。

```
void _x ( ) {
        c = ( ( unsigned long int ) a * ( unsigned long int ) 100 + ( unsigned long int
) b / ( unsigned long int ) 2 ) / ( unsigned long int ) b ;
}
```

能写成如下，如果为了中间结果使用 double word 类型。

```
#pragma mul
#pragma div
unsigned int    a, b ;
unsigned char   c ;

void _x ( ) {
       c = ( unsigned char ) divux ( ( unsigned long ) ( b / 2 ) + muluw ( a, 100 ) , b
);
}
```

2.8　编译器和汇编程序相互引用

本节介绍编译器和汇编程序的相互引用。

2.8.1　相互引用变量

(1) 在 C 语言中引用定义变量

从汇编语言程序中引用在 C 程序中定义的外部变量，定义成外部 (extern)。
在汇编语言模块中，变量名称前置下划线 (_)。

示例　　C 源代码

```
extern  void subf ( void );
char    c = 0 ;
int     i = 0 ;

void main ( void ) {
        subf ( );
}
```

示例　　汇编源代码

```
$PROCESSOR ( F1166A0 )
        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i
@@CODE CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

备注　　参阅 "9.5　引用在 C 语言中定义的变量 "。

(2) 在汇编语言中引用定义变量

从 C 程序中引用在汇编语言程序中定义的外部变量，定义成外部 (extern)。
在汇编语言程序中，变量名称前置下划线 (_)。

示例　　C 源代码

```
extern char     c ;
extern int      i ;

void subf ( void ) {
        c = ' A ' ;
        i = 4 ;
}
```

示例　　汇编源代码

```
NAME ASMSUB
                PUBLIC  _i
                PUBLIC  _c
ABC DSEG        BASEP
_i : DW         0
_c : DB         0
                END
```

备注　　参见 "9.6　从 C 语言中引用汇编语言中定义的变量 "。

2.8.2 相互引用函数

(1) 在 C 语言中引用定义函数

以下步骤用于从汇编语言程序中调用在 C 中编写的函数。

(a) 保存 C 工作寄存器 （AX,BC,DE)。

(b) 在堆栈中压入参数。

(c) 调用 C 函数

(d) 调整堆栈指针 (SP) 指向参数的字节长度

(e) 引用 C 语言函数的返回值 （BC, 或 DE， BC)。

示例 汇编语言

```
$PROCESSOR ( F1166A0 )
        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2
@@CODE CSEG
_FUNC2 :
        movw    ax, #20H        ; set 2nd argument ( j )
        push    ax              ;
        movw    ax, #21H        ; set 1st argument ( i )
        call    !_CSUB          ; call "CSUB ( i, j )"
        pop     ax              ;
        ret
END
```

参见 "9.4 从汇编语言中调用 C 语言程序 "。

(2) 引用定义在汇编语言中的函数

通过 C 函数进行以下操作来调用在汇编语言中定义的函数。

(a) 为寄存器变量保存基址指针和 saddr 区域。

(b) 复制堆栈指针 （SP) 到基址指针 （HL)。

(c) 对 FUNC 函数进行处理

(d) 设置返回值。

(e) 恢复保存的寄存器。

(f) 返回到 main 函数

示例　　汇编语言

```
$PROCESSOR ( F1166A0 )
        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2
@@DATA          DSEG    BASEP
_DT1 :  DS      ( 2 )
_DT2 :  DS      ( 4 )
@@CODE CSEG
_FUNC :
        PUSH    HL              ; save base pointer
        PUSH    AX
        MOVW    HL, SP          ; copy stack pointer
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; move 1st argument ( i )
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]   ; arg2
        MOVW    !_DT2, AX       ; move 2nd argument ( l )
        MOVW    BC, #0AH        ; set return value
        POP     AX
        POP     HL              ; restore base pointer
        RET                     ;
        END
```

参见 "9.3　从 C 语言中调用汇编语言程序 "。

第 3 章 编译器语言说明

本章解释了 78K0R C 编译器支持的语言规范。

3.1 基本语言规范

C 编译器支持由 ANSI 标准规定的语言规范。这些规范包含
过程定义规定的项目。这章解释了用于 78K0R 微控制器的微处理器处理系统决定的语言规范。
还解释了是否使用严格遵守 ANSI 标准的选项间存在的差异。
参考 "3.3 扩充语言规范 " 以获取由 78K0R C 编译器附加的扩充语言规范。

3.1.1 依赖处理系统的项目

这节讲述了 ANSI 标准中依赖处理系统的项目。

(1) 数据类型和大小

多字节类型的字节顺序是 " 从最低到最高字节 " , 符号由 2 的补码表示。符号加在最高位上 (0 表示正数或 0， 1 表示负数)。

- 一个字节有 8 个位。
- 字节数、字节顺序和目标文件编码按如下规定。

表 3-1. 数据类型和大小

数据类型	大小
char	1 个字节
int, short	2 个字节
long, float, double	4 个字节
pointer	near ： 2 个字节 far ： 4 个字节

(2) 翻译阶段

ANSI 标准指定用于翻译的在语法规则中的 8 个优先翻译等级 （即所谓 " 时期 "）。 " 换行符以外的非空格符 " 的排列定义为处理系统是根据第 3 阶段 ? 分解源文件为预处理标记和空格符 ?，这样安排是保证单个空格符没有被替代。
因而，表格符由带有 -lt 选项的空格符替代。

(3) 诊断信息

当在翻译中发生违反语法规则或违反限制时，编译器输出包含源文件和出错行数 （在能确定时）的错误信息。
这些出错信息通常有三种类型：" 警告 "， " 严重错误 " 和 " 其他错误 "。

(4) 独立环境

(a) 在开始程序处理中调用的函数名字和类型没有规定在独立环境中[注]。因而，它取决于用户自己编码和目标系统。

注　执行 C 语言源程序而不使用操作系统函数的环境。
在 ANSI 标准中，规定了两种执行环境：独立环境和宿主环境。目前， 78K0R C 编译器不支持宿主环境。

(b) 在独立环境中对终止程序的影响没有规定。因而，它取决于用户自己编码和目标系统。

(5) 程序执行

交互单元的配置没有规定。
因而，它取决于用户自己编码和目标系统。

(6) 字符集

执行环境字符集的单元数值是 ASCII 码。

(7) 多字节字符

字符常数和字符串不支持多字节字符。
然而，在注释中支持日文说明。

(8) 字符显示的含义

扩展的符号值如下规定。

表 3-2. 扩展符和含义

扩展符	数值 (ASCII)	含义
\a	07	警报 (警告音)
\b	08	退格
\f	0C	换页 (新页面)
\n	0A	新行 (行移)
\r	0D	回车 (储存)
\t	09	水平表格
\v	0B	垂直表格

(9) 翻译限制

翻译限制值规定如下。

用 * 标记的数值是保证的。在某些情况下这些值会被超出，但是操作不被保证。

表 3-3. 翻译限制值

内容	限制 值
复合语句、重复控制结构和选择控制结构的嵌套级数目 (但是 , 取决于 "case" 标签的数目)	45
条件嵌入的嵌套级数	255
在声明中指针、数组和函数声明的数目 （在任何组合中）限定为，一个声明中包含一个算术类型、结构体类型、联合体类型或不完全类型。	12
嵌套级的数目由一个完整的括号括起来	591*
嵌套级数目的表达式由一个完整的括号括起来	32
宏名中第一个字符的有效数目	256
外部标识符第一个字符的有效数目	249
内部标识符第一个字符的有效数目	249
在翻译单元中具有外部标识符的标识符数目	1024*
声明在基本块中具有有效的标识符数目	255
同时定义在翻译单元的宏数目	32767
在函数定义中的形参数目和在函数调用中的实参数目	39*
在宏定义中的形参数目	31
在宏调用中的实参数目	31
一条源文件逻辑行的字符数目	2048*
链接后一个字符串常数，或一个宽字符串常数的字符数目	509*
一个文件的目标码大小 （数据被指示）	65535
include (#include) 文件的嵌套数目	50
一个 "switch" 语句中的 "case" 标签数目 (包含这些嵌套 , 如果有的话)	257
每个编译单元源文件行的数目	65535*
嵌套函数调用的数目	40*
在单目标模块中的代码 , 数据和堆栈段总大小	取决于存储器模式[注]
单结构体或单联合体的元素数目	256
单枚举类型中的枚举常数的数目	255
定义在单结构声明中的结构体或联合体的嵌套级数目	15
初始化元素的嵌套	15
定义在单个源文件中的函数数目	4095
宏嵌套	200
include 文件路径的数目	64

注 下表列出了不使用扩充功能时的各种存储器模式最大值。

存储器模式	最大值
小型模式	代码 64KB，数据 64KB，总共 128KB
中型模式	代码 1MB，数据 64KB，总共 1MB
大型模式	代码 1MB，数据 1MB，总共 1MB

(10)数量限制

(a) 一般整数类型的限制值 (limits.h 文件)

limits.h 文件指定能表达一般整数类型数值的限制值 (字符类型，有符号 / 无符号整数类型和枚举类型)。由于不支持多字节字符， MB_LEN_MAX 代码没有相应的限制。因此用 MB_LEN_MAX 只能定义为 1。如果指定 -qu 选项， CHAR_MIN 是 0,CHAR_MAX 就取与 UCHAR_MAX 相同的值。由 limits.h 文件定义的限制值如下。

表 3-4. 一般整数类型的限制值 (limits.h 文件)

名称	值	含义
CHAR_BIT	+8	不在位域中的最小代码的位 (= 1 字节) 数目
SCHAR_MIN	-128	有符号 char 型的最小值
SCHAR_MAX	+127	有符号 char 型的最大值
UCHAR_MAX	+255	无符号 char 型的最大值
CHAR_MIN	-128	char 型的最小值
CHAR_MAX	+127	char 型的最大值
SHRT_MIN	-32768	短 int 型的最小值
SHRT_MAX	+32767	短 int 型的最大值
USHRT_MAX	+65535	无符号短 int 型的最大值
INT_MIN	-32768	int 型的最小值
INT_MAX	+32767	int 型的最大值
UINT_MAX	+65535	无符号 int 型的最大值
LONG_MIN	-2147483648	长 int 型的最小值
LONG_MAX	+2147483647	长 int 型的最大值
ULONG_MAX	+4294967295	无符号长 int 型的最大值

(b) 浮点类型的限制值 (float.h 文件)

定义在 float.h 文件中与浮点类型特征有关的限制值。

由 float.h 文件定义的限制值如下。

表 3-5. 浮点类型 (float.h 文件) 定义的限制值

名称	值	含义
FLT_ROUNDS	+1	添加浮点圆整模式。 1 用于 78K0R 微控制器 (以最近方向舍入)。
FLT_RADIX	+2	基指数 (b)
FLT_MANT_DIG	+24	以 FLT_RADIX 的浮点尾数为基础的数字数目 (p)
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	+6	十进制数的位数注 1(q) 能舍入 q 个十进制数到 b 为基数的 p 个浮点数，然后存储 q 十进制数
DBL_DIG		
LDBL_DIG		
FLT_MIN_EXP	-125	当 FLT_RADIX 提高到 FLT_RADIX 平方减一时，规格化到浮点数的最小负整数 (e_{min})。
DBL_MIN_EXP		
LDBL_MIN_EXP		
FLT_MIN_10_EXP	-37	10 的乘方 $_{10}b^{e_{min-1}}$ 最小负对数整数落在规格化的浮点数的范围内。
DBL_MIN_10_EXP		
LDBL_MIN_10_EXP		
FLT_MAX_EXP	+128	当 FLT_RADIX 的指数次方减一 (e_{max}) 时，有限浮点数最大整数能表示的最大整数。
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	+38	能表达的有限浮点数最大值 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP		
LDBL_MAX_10_EXP		
FLT_MAX	3.40282347E + 38F	能表达的有限浮点数最大值 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX		
LDBL_MAX		
FLT_EPSILON	1.19209290E - 07F	注 2 由浮点数类型指定和大于 1 的最小值 1.0 间的不同。 b^{1-p}
DBL_EPSILON		
LDBL_EPSILON		
FLT_MIN	1.17549435E - 38F	规格化的正浮点数的最小值 $b^{e_{min}-1}$
DBL_MIN		
LDBL_MIN		

注 1. 在 ANSI 标准中 DBL_DIG 和 LDBL_DIG 是 10 或大于 10，但是，由于双精度和长双精度类型是 32 位，所以在 78K0R 微控制器中是 6。

2. 在 ANSI 标准中 DBL_EPSILON 和 LDBL_EPSILON 是 1E-9 或小于 1E-9，但是在 78K0R 微控制器中是 1.19209290E-07F。

(11) 标识符

识别初始 249 字符作为标识符。
区分大小写字符。

(12) 字符类型

无指定类型 （有符号，无符号）的字符类型默认处理为有符号的整数。
然而，一个简单的 char 类型可以由 C 编译器 - qi 选项指定为一个无符号整数。
这些不包含在源程序字符集中的类型需要由 ANSI 标准 (转义序列) 转换保存起来，当字符类型以外的类型被替换为字符类型时也是如此。

```
char    c = ?f\777?f;     /* Value of c is -1 */
```

(13) 浮点常数

浮点常数符合 IEEE754 注。

注　IEEE：电气和电子工程师协会
因此， IEEE754 是一个标准，来统一处理浮点操作的数据格式和在系统中数字范围的规范。

(14) 字符常数

(a) 源程序的字符集和执行环境的字符集基本是 ASCII 码以及符合具有相同数值的元素。

(b) 一个整数字符常数值的最后字符含有两个或两个以上字符是无效的。

(c) 如下是不能由基本执行环境字符集表或转义序列的字符。

<1> 八进制或十六进制的转义序列值由八进制或十六进制的标记表示

\077	63

<2> 简化转义序列表达如下。

\'	'
\"	"
\?	?
\\	\

<3> \a, \b, \f, \n, \r, \t, \v 的值同样在 "(8) 字符显示的含义 " 中进行了解释。

(d)　不支持多字节字符常数。

(15) 头文件名

头文件名以两种格式 (< > 和 " ") 反映字符串或将外部源文件名设定在 "(32)　载入头文件 "。

(16) 注释语句

可以用日文注释。日文默认的字符码集是 Shift JIS。

输入源文件的字符码集可以由编译器的 -z 选项或由环境变量来指定。选项指定优先于环境变量指定。因而，当指定 "none" 时，字符码不被保证。

(a)　选项规范

```
-ze | -zn | -zs
```

(b)　环境变量

```
LANG78K [ euc | none | sjis ]
```

为了设置环境变量，使用环境标准过程。

(17) 有符号和无符号常数

如果一般整数型的值转换成较小的有符号整数，最高位截去后取位串。

如果无符号整数转换成相应的有符号整数，内部表征没有改变。

(18) 浮点和一般整数

如果一般整数型值转换成浮点型的值，并且如果被转换的值能被表达但不能精确，其结果舍入到最接近可表达的值。

当结果正好在中间值，它能舍入为偶数 （最后尾数位是 0）。

(19) 双精度型和浮点型

在 78K0R C 编译器中，双精度型以浮点型同样方式表达为浮点数，作为 32- 位 （单精度）数据。

(20) 位单元中的有符号型操作数

字符的移位操作遵守 "(26)　位单元中移位操作 " 的约定。

有符号型的位单元中其它操作 （在位像中那样）按无符号值计算。

(21) 结构体和联合体的元素

如果联合体元素的值以不同元素保存，它是按照对齐条件保存。因此，联合体元素按照对齐条件存取 (见 "(b)　结构体类型 " 和 "(c)　联合体类型 ")。

在联合体包含结构体的情况中，享用公共排列中第一个元素作为元素 , 内部代表是相同的，即使第一个元素公用为结构体也是相同的。

(22) sizeof 运算符

来自 "sizeof" 运算符的结果值遵照有关 "(1)　数据类型和大小 " 中代码字节的约定。
结构体和联合体的字节数目包含填充区的字节。

(23) Cast 运算符

当指针转换成一般整数类型时 , 所需的变量大小与相同，列出在如下的表中。位串作为转换结构保存。
任意整数可以由指针转换。因此，转换整数的结果根据此类型扩展成 int 类型。

near	2 个字节
far	4 个字节

当 near 指针或 int 转换为 far 指针，并且当 near 指针转换成 long 型时，按如下操作。

- 变量指针在最高位加 0xf。 (0 除外。指针零扩展。)
- 函数指针是零扩展。

(24) 除 / 余数操作

当运算对象是负和用整数不能完全除尽时，除操作 ("/") 的结果如下：不管除数或被除数是负，其结果是最小整数大于代数商。
如果除数和被除数都是负，其结果是最大的整数小于代数商。
如果操作数是负， "%" 操作结果取表达式中第一个操作数的符号。

(25) 加减操作

如果两个指针指出同样数组的元素进行相减，结果的类型是 int 类型，其大小是 2 字节。

(26) 位单元中移位操作

如果 "E1 >> E2" 是有符号类型和取负值，执行算术移位。

(27) 存储区域说明

声明存储区域的分级说明符 "register" 可用来最大程度地增加存取速度，但是不始终有效。

(28) 结构体和联合体的说明符

(a) 在 int 型位域符号中，无论有 / 无符号的说明符均简化为无符号。

(b) 为了保留位域，需要分配足够的尺寸来存储保存区单元地址。如果区域不足，遵照类型域的对齐条件那么位域就不能匹配的放入下一个单元。

(c) 在单元中位域的分配顺序是从低到高。
因此， -rb 选项能指定分配顺序从高到低。

(d) 结构体或联合体的非位域中的每个元素按如下界限对齐

cchar 和无符号 char 类型以及 char 数组和无符号 char 类型	字节界限
其它 （包含指针）	双字节界限

(29) 枚举类型说明符

枚举类型是下列能表达所有枚举常数的第一种类型。

- signed char
- unsigned char
- signed int

(30) 类型限定

有类型限定的数据存取的结构是？不稳定？的，其取决于地址 （I/O 端口，等）相对数据的映射。

(31) 条件嵌入

(a) 指定条件嵌入的常数值和出现在其它表达式中的字符常数是等价的。

(b) 有符号字符的字符常数必须没有负值。

(32) 载入头文件

(a) 在 "#include <character string>" 格式中预处理指示

除非 "filename" 是以字符 '\' [注] #include <filename> 预处理器的指令指示预处理器搜索尖括号间 (<..>) 指定的文件，搜索按下面条件：1) 由 -i 选项指定的文件夹， 2) 由 INC78K0R 环境变量指定的文件夹以及，3)inc78k0r 文件夹，此文件夹与 cc78k0r.exe 所在二进制文件夹有关。

如果头文件在指定 "<" 和 ">" 间分隔符的字符串搜索一致，头文件的全部内容被替换。

注　"\" 和 "/" 被视为文件夹的分隔符。

示例

```
#include        <header.h>
```

查找顺序如下。

- 通过 -i 选项设定的文件夹
- 由 INC78K0R 环境变量指定的文件夹
- 标准文件夹

(b) 在 "#include "character string"" 格式中预处理指示

除非 "character string" 是以字符 '\' [注] ,#include "character string" 预处理器的指令指示预处理器搜索引号 ("..") 间指定的文件，搜索按下面条件：1) 包含在源文件的文件夹， 2) 由 -i 选项指定的文件夹， 3) 由 INC78K0R 环境文件变量指定的文件夹，以及 4)..\inc78k0r 文件夹，此文件夹与 cc78k0r.exe 所在二进制文件夹有关。

如果指定在引号分隔符间的文件找到，那么 #include 指示行用文件全部内容替换。

注　"\" 和 "/" 被视为文件夹的分隔符。

示例

```
#include    "CG_ad.h"
```

查找顺序如下。

- 包含源文件的文件夹
- 通过 -i 选项设定的文件夹
- 由 INC78K0R 环境变量指定的文件夹
- 标准文件夹

(c) "#include preprocessing character phrase string" 的格式

"#include preprocessing character phrase string" 视作单个头文件预处理字符惯用语，只有在预处理字符串是一个宏时 , 其替代为 <character string> 或捕 haracter string

(d) 在 "#include <character string>" 格式中预处理指示

在一个 (最终）被分隔字符串和头文件名间 , 字符串中的字母字符的长度被确认，

```
在编译器工作环境中有效的文件名长度是有效的。
```

搜索文件符合上面约定的文件夹

(33)#pragma 指令

#pragma 指令是 ANSI 标准定义的预处理 指令类型。以下 #pragma 的字符串以编译器相关方式翻译。

当 #pragma 指示不被编译器承认时，它被忽略并且翻译继续。如果指令加了关键词且 C 源文件包含此关键词，就发生错误。为了防止错误，从源文件中删除关键词或以 #ifdef 执行它。

(34) 预定义宏名

支持下列所有的宏名。

为了以前 C 语言规范 (K&R 规范)，支持用撞 _ 为了严格遵守 ANSI 标准完成处理，宏的前后带 "_ _ " 来使用。

表 3-6. 支持宏的列表

宏名称	定义
__LINE__	这点上源文件行的行数 （十进制）。
__FILE__	假设的源文件名 （字符串常数）。
__DATE__	翻译源文件的数据 （ "Mmm dd yyyy" 格式中的字符串常数）。 这里，月份的名字与由 ANSI 标准规定建立的 asctime 函数相同 （3 个字母字符，其中仅第一个字符是大写)(如果 dd 的数值小于 10， dd 的第一个字符是空的）。
__TIME__	源文件的翻译时间 （字符串常数具有格式揖 h：mm：ss
__STDC__	十进制常数 1，表示与 ANSI 标准一致。注
__K0R__	十进制常数 1
__K0R_SMALL__	十进制常数 1 （当指定 small 模式时。）
__K0R_MEDIUM__	十进制常数 1 （当指定 large 模式时。）
__K0R_LARGE__	十进制常数 1 （当指定 -qu 选项时。）
__CHAR_UNSIGNED__	十进制常数 1 （当指定 -qu 选项时。）
CPUmacro	宏的十进制常数 1 表示目标 CPU。 在设备文件中 " 产品规格 " 所指出的字符串用 "__" 放在前后来定义。

注 指定 -za 选项时的定义

(35) 特殊数据类型定义

下面是由 stddef.h 文件定义的 NULL, size_t, 和 ptrdiff_t。

表 3-7. NULL， size_t， ptrdiff_t (stddef.h 文件) 的定义

NULL/size_t/ptrdiff_t	定义
NULL	((void *) 0)
size_t	unsigned int
ptrdiff_t	int

3.2 编译期间的环境变量

这节说明了在执行期间 C 编译器是如何处理数据，寄存器，器件规格和环境的。

3.2.1 内部表达和数据的数值区域

这节说明了由 78K0RC 编译器处理的内部表达形式和每种型号的数据数值区域。

(1) 基本类型

基本类型也称算术类型，其包含整数类型和浮点类型。

整数类型可以分成字符类型，有符号整数类型，无符号整数类型和枚举类型。

(a) 整数类型

整数类型可以分成 4 种分类，如下所示。整数类型以二进制 0s 和 1s 表示。

- 字符类型
- 有符号整数类型
- 无符号整数类型
- 枚举类型

<1> 字符类型

字符类型大小足够存储执行字符集的任意元素。

如果将基本执行字符集的元素存储进一个 char 目标，它的值非负的。

字符以外的目标处理为有符号的整数。

如果数值存储时发生溢出，溢出部分被忽略。

<2> 有符号整数类型

如下，有四种有符号的整数类型。

- 有符号 char 型
- 短 int 型
- int 型
- 长 int 型

定义为有符号字符类型的目标占据与 "plain" char 一样大小的区域。

"plain" int 具有由 CPU 执行环境结构建议的大小区域。

对于每种有符号整数类型，具有相对应的无符号整数类型与其使用相同大小的区域。

有符号整数类型的正元素是无符号整数类型的一个子集。

<3> 无符号整数类型

无符号整数类型由关键词摇 nsigned

涉及无符号整数型的计算永远不会溢出，因为减少了不能由无符号整数型表示的结果，比最大值大一的数可由结果类型表示。

<4> 枚举类型

枚举由命名的整型常数值集组成。

不同的枚举构成不同的枚举类型。

每个枚举构成一个枚举类型。

(b) 浮点类型

如下有三种实浮点类型。

- float(浮点型)
- double(双精度型)
- long double(长双精度型)

像浮点类型，78K0R C 编译器的双精度和长双精度支持定义在 ANSI/IEEE 754-1985 的单精度规格化数值的浮点。这意味浮点，双精度和长双精度具有相同的数值范围。

表 3-8. 由类型决定的数值范围

类型	数值范围
(signed) char	-128 ～ +127
unsigned char	0 ～ 255
(signed) short int	-32768 ～ +32767
unsigned short int	0 ～ 65535
(signed) int	-32768 ～ +32767
unsigned int	0 ～ 65535
(signed) long int	-2147483648 ～ +2147483647
unsigned long int	0 ～ 4294967295
float	1.17549435E - 38F ～ 3.40282347E + 38F
double	1.17549435E - 38F ～ 3.40282347E + 38F
long double	1.17549435E - 38F ～ 3.40282347E + 38F

备注 **1.** "signed" 类型指定可以省略。然而，当对字符类型省略时，编译条件 （选项）决定这类型为有符号或无符号的字符类型。

2. short int 和 int 型具有相同的数值范围，但是它们作为不同类型来处理。

3. 无符号短 int 和无符号 int 型具有相同的数值范围，但是它们作为不同类型来处理。

4. 浮点型、双精度型和长双精度型具有相同的数值范围，但是它们作为不同类型来处理。

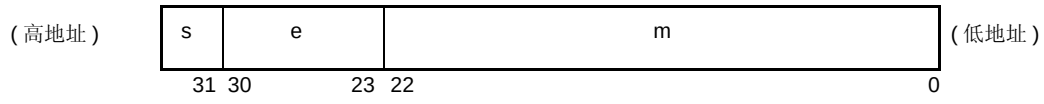
5. 浮点型、双精度型和长双精度型属于绝对数值的范围。

下面显示了浮点数 （浮点型）的规格。

<1> 格式

下面显示了浮点数格式。

图 3-1. 浮点数格式



这个格式的数值如下。

```
          （数值的符号）          （数值的表达式）
（ -1 ）                  *（尾数值）* 2
```

s	Sign (1 bit) 正数是 0，负数是 1.
e	指数 （8 位） 2 的指数表达为 1 字节的整数 （在负的情况中表达为 2 的补码）这些关系如下表所示。
m	尾数 （23 位） 尾数表达了一个绝对数值，位的位置 22 位至 0 位等效于二进制码中的第 1 至第 23 的位置。 但是，当浮点数值是 0 时，指数值始终在调整，以致于尾数在 1 至 2 的范围 （归一化）。这个结果是 1(即 , 数值是 1) 的位置始终是 1，在这个格式中是这样省略表达的。

指数 （16 进制）	指数值
FE	127
:	:
81	2
80	1
7F	0
7E	-1
:	:
01	-126

<2>　0 的表达式

当指数＝ 0 和尾数＝ 0 ， ± 0 是如下表达。

```
          （符号值）
（ -1 ）                  * 0
```

<3>　无穷大指数

当指数＝ＦＦＨ和尾数＝ 0 ， ±？是如下表达的。

```
          （符号值）
（ -1 ）                  *
```

<4>　非规格化数值

当指数 =0，且尾数 0 时，非标准规格化的值表示如下。

```
          （符号值）                  -126
（ -1 ）                  *（尾数值）* 2
```

备注　　这里尾数值是小于 1 的数，所以尾数 2 2 至 0 的位置表示了十进制中第 1 至第 2 3 位置。

<5> 非数字指数 （非数字）

当指数＝ＦＦＨ和尾数？0, 不管符号，表达为 NaN。

<6> 计算结果的舍入

数值向下舍入到最近的偶数。如果计算结果不能在上面浮点格式中表达，那么舍入到最近可表达的数。

如果两个值，这个值能表达预舍入值的差，那么舍入到偶数 （一个最小二进制位是 0 的数）。

<7> 例外

如下表所示有 5 种类型例外。

表 3-9. 数字例外

例外	返回数值
下溢出	非规格化数值
不精确	± 0
溢出	±
被零除	±
无效操作	NaN

当例外发生时，调用 matherr 函数导致警告发生。

(2) 字符类型

有 3 种 char 数据类型。

- char
- signed char
- unsigned char

(3) 不完全类型

有 4 种不完全数据类型。

- 不确定目标大小的数组
- 结构体
- 联合体
- 空类型

(4) 派生类型

有 5 种派生数据类型。

- 数组类型
- 结构体类型
- 联合体类型
- 函数类型
- 指针类型

(a) 数组类型

数组类型描述了连续排列，该排列中成员目标类型特定 , 称为元素类型的目标集。

所有目标成员具有相同大小的区域。能定义数组类型和单独元素。不能建立不完全数组类型。

(b) 结构体类型

一个结构体类型描述为一个目标成员的顺序排列集，每个成员有一个选择指定名和可有区别的类型。

备注　　数组和结构体统称为 汇总类型。在汇总类型中的目标成员顺序排列。

(c) 联合体类型

一个联合体描述了一个成员目标交集。

每个联合体成员具有一个指定名和可有区别的类型。联合体成员可以分别定义。

(d) 函数类型

一个函数类型描述了带有返回指定类型数值的函数。

一个函数由它的返回值，返回成员和返回它的参数类型被字符化。

如果它的返回数值类型是 T，这个函数称为掀 unction returning T

(e) 指针类型

一个指针类型可以从函数类型，目标类型或不完全类型衍生，称为引用类型。

一个指针类型描述了目标类型，其数值为引用类型的实体提供一个关联。

从应用类型 T 衍生的指针类型有时称为 "pointer to T"。

3.2.2　存储器

存储器模式由目标器件的存储空间决定。

(1) 存储器模式

下面的存储器模式是可用的。

表 3-10. 存储器模式

存储器模式	最大值
Small 模式 (-ms 选项)	代码 64KB，数据 64KB，总共 128KB
Medium 模式 (-mm 选项)	代码 1MB，数据 64KB，总共 1MB
Large 模式 (-ml 选项)	代码 1MB，数据 1MB，总共 1MB

包含ＲＯＭ数据的数据段。上表列出了当扩展功能不使用时的最大数值。

(2) 寄存器组

- 现行的寄存器组由 78K0R C 编译器的启动例程设置到揜 B0"。除非它被改变，否则它保持设置到寄存器组 0.
- 它指定设置在中断功能开始已经改变了的寄存器组。

(3) 存储器空间

78K0R C 编译器使用下面存储器空间。

图 3-2. 存储器空间的使用

地址		使用		容量（字节）
00	080 - 0BFH	CALLT 表格		64
FF	E20 - EB3H	sreg 变量，布尔型变量		148
FF	EB4 - EC3H	寄存器变量		1616
FF	EC4 - ED3H	编译器保留区域		16
FF	ED4 - ED7H	段信息		4
FF	ED8 - EDFH	实时运行库的参数		8
FF	EE0 - EF7H	RB3 - RB1	工作寄存器注 1	24
	EF8 - EFFH	RB0	工作寄存器	8
FF	F00 - FFFH	sfr 变量		256
F0	000 - 7FFH	第二 sfr 变量		Max. 2048 注 2

注 **1.** 当寄存器组指定时使用。

2. 取决于所使用器件不同而不同。

3.3　扩充语言规范

这节说明了 78K0R C 编译器独特的扩充，此扩充不被 ANSI(American National Standards Institute) 标准所规定。

78K0R C 编译器的扩展允许产生取得目标器件最有效的使用代码。这些扩展在每种情况下不一定都有用，所以推荐使用在效的情况中。关于 78K0R C 编译器有效使用的更多信息见 " 第 2 章　函数 "。

78K0R C 编译器扩展使用导入微控制器是依赖于 C 源程序，但是仍旧保留 C 语言上兼容。即使在 C 源程序中使用了 78K0R C 编译器的扩展，做少许简单的改变便于用于其它微控制器。

备注　　在这节中 RTOS" 支持 78K0R 实时 OS。

3.3.1　宏名称

78K0R C 编译器定义一个宏名来指示目标器件的微控制器名并且宏名也指示了器件名。这些器件名由编译选项规定产生目标器件的目标代码或在 C 源代码中的器件分类。下面的示例定义宏名 __K0R__ 和 __F1166A0_。

查阅 "(34)　预定义宏名 " 以获取更多关于宏名的信息。

编译选项：

```
>cc78k0r -cf1166a0 prime.c ...
```

3.3.2　关键字

78K0R C 编译器定义下面的关键字以使能扩充功能。类似 ANSI C 关键词，这些关键词不能使用为标号或变量明。

所有这些关键字是小写。任何包含大写字符作为关键字的标识不被认可。

以下由 78K0R C 编译器增加的关键字表中，不是以 “__” 开始的关键词就是能由严谨的 ANSI C 相适应的选项 (-za) 未定义。

表 3-11.　由 78K0RC 编译器增加的关键字

关键字		作用
始终定义	当 -za 选项被指定时未定义	
__callt	callt	通过 callt 表调用函数
__callf	callf	对 78K0 兼容
__sreg	sreg	在 saddr 区域分配变量
-	noauto	对 78K0 兼容
__leaf	norec	对 78K0 兼容
__boolean	boolean	位存取到 saddr 和 sfr 区域
-	位	位存取到 saddr 和 sfr 区域
__interrupt	-	硬件中断
__interrupt_brk	-	软件中断
__asm	-	ASM 语句
__rtos_interrupt	-	RTOS 中断处理器
__pascal	-	对 78K0 兼容

关键字		作用
始终定义	当 -za 选项被指定时未定义	
__flash	-	固化 ROM 函数
__flashf	-	__flashf functions
__directmap	-	绝对地址映射
__temp	-	对 78K0 兼容
__near, __far	-	存储区域说明
__mxcall	-	对 78K0 兼容

(1) 函数

callt, __callt, __interrupt, __interrupt_brk, __rtos_interrupt, __flash, __flashf 关键词是可以增加于函数表述开始的属性描述。

语法显示如下。

```
attribute-qualifier ordinary-declarator function-name (parameter-type-list/identifier-list)
```

下面是示例说明。

```
__callt  int  func  ( int );
```

有效的属性描述限制如下。

注意 callt 和 __callt 被认为是相同的说明。但是，即使当 -za 选项指定时，也要开始以捶 _

- callt
- __interrupt
- __interrupt_brk
- __rtos_interrupt
- __flash
- __flashf

注意事项　当编译器遇到 callf、__callf、noauto、__pascal、__mxcall、norec 和 __leaf 关键字时发出一个警告，否则就忽略它们。

(2) 变量

- sreg 和 __sreg 贯彻与 C 语言的揖 egister(参阅 "　如何使用 saddr 区域 (sreg/__sreg)" 获得关于 sreg 关键字的更多信息。)
- bit、 boolean 和 __boolean 类型的位规定贯彻与 C 语言规范捧 har
  但是，它们仅能应用于外部函数 （外部变量）变量声明。
- __directmap 描述贯彻与 C 语言描述同样的规则。（参阅 ？　绝对地址分配规范 (__directmap)? 获取详细信息。）
- __near 和 __far 描述贯彻与 C 语言描述同样的规则。（参阅 ?near/far 区域规范 ? 获取详细信息。）

注意事项　当编译器遇到 __temp 关键字发出一个警告，否则就忽略它。

3.3.3　#pragma 指令

#pragma 指令是由 ANSI 标准支持的预处理指令类型。 #pragma 指令指示编译器按 #pragma 后面的字符串规定方式翻译。

当编译器遇到不可识别的 #pragma 指令时，忽略这指令并继续编译。如果不可识别的 #pragma 函数定义了一个关键词，在 C 源程序中遇到此关键字就产生一个错误。为了防止这个，非定义的关键字要从 C 源程序中删除或执行 #ifdef。

78K0R C 编译器支持下面的 #pragma 指令，其允许扩充功能。

在 #pragma 后面的关键字可为大写或小写。

参阅 "3.3.4　使用扩展函数 " 获得更多关于使用这些指令以能扩充功能。 #pragma 指令列表

#pragma 指令	作用
#pragma sfr	在 C 源程序文件中使用 SFR 名称。 参见 " 如何使用 sfr 区域 (sfr)"。
#pragma vect #pragma interrupt	在 C 中写中断服务程序。 参见 " 中断函数 (#pragma vect/#pragma interrupt)"。
#pragma di #pragma ei	在 C 中禁止和允许中断。 参见 " 中断函数 (#pragma DI, #pragma EI)"。
#pragma halt #pragma stop #pragma brk #pragma nop	在 C 中写 CPU 控制指令。 参见 "CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)"。
#pragma section	改变编译器输出 section 名和指定 section 分配。 参见 " 改变编译器输出区段名称 (#pragma section ...)"。
#pragma name	改变模块名称。 参见 " 模块名称改变函数 (#pragma name)"。
#pragma rot	使用内联程序功能。 参见 " 循环移位函数 (#pragma rot)"。
#pragma mul	使用内联乘法功能。 参见 " 乘法函数 (#pragma mul)"。
#pragma div	使用最优除法函数。 参见 " 除法函数 (#pragma div)"。
#pragma opc	在当前代码地址插入数据。 参见 " 数据插入函数 (#pragma opc)"。
#pragma rtos_interrupt	在 C 中写 RX78K0R(实时 OS) 中断句柄。 参见 "RTOS 中断处理 (#pragma rtos_interrupt ...)"。
#pragma rtos_task	在 C 中写 RX78K0R(实时 OS) 任务。 参见 "RTOS 任务函数 (#pragma rtos_task)"。
#pragma ext_table	规定闪存区分支表的起始地址。 参见 " 闪存区域跳转表和闪存区域分配 (#pragma ext_table)"。
#pragma ext_func	从引导区调用闪存区函数。 参见 " 从引导区到闪存区的函数调用 #pragma ext_func)"。
#pragma inline	标准库功能内存复制和内存设置的内联扩充。 参见 " 存储器运算函数 (#pragma inline)"。

3.3.4 使用扩展函数

下面列出了 78K0R C 编译器的扩充功能。

表 3-12. 扩充功能列表

扩充功能	说明
call 函数 (callt/__callt)	在 callt 表区中分配调用函数的地址。 使用调用指令相对可减少目标代码。
寄存器变量 (寄存器)	为了加快执行速度，使编译器在寄存器或 saddr 区放置一个变量。 目标代码也更简洁。
如何使用 saddr 区域 (sreg/ __sreg)	分配规定的 sreg 或 specified __sreg 外部变量以及在 saddr 区函数中的静态变量。 saddr 区的变量执行速度能快于一般变量。 目标代码也更简洁。变量能由编译器选项分配进 saddr 区。
外部变量 / 内部静态变量的 saddr 自动分配选项 (-rd) 的使用方法	分配外部变量和外部静态变量到 saddr 区。 saddr 区的变量执行速度能快于一般变量。 目标代码也更简洁。变量能由编译器选项分配进 saddr 区。
内部静态变量的 saddr 自动分配选项的使用方法 (-rs)	分配内部静态变量到 saddr 区。saddr 区的变量执行速度能快于一般变量。 目标代码也更简洁。变量能由编译器选项分配进 saddr 区。
如何使用 sfr 区域 (sfr)	#pragma sfr 指令表示 sfr 名，能用来操作 C 源程序文件的特殊功能寄存器。
bit 型变量 (bit), boolean 型变量 (boolean/__boolean)	产生具有 1 位存储区的变量。 位和 boolean/__boolean 类型变量允许对 saddr 区进行位存取。 boolean 和 __boolean 类型变量在功能上指出了位类型变量并能以同样方式使用。
ASM 语句 (#asm - #endasm/__asm)	#asm 和 __asm 指令允许在 C 源代码中使用汇编语言语句。语句嵌入进由 C 编译器产生的汇编源代码中。
汉字 (2 字节字符) (/* kanji */, // kanji)	C 源程序注释可以包含汉字 （多字节日文字符）。 从 Shift-JIS、 EUC 或不选的选项中选择汉字编码。
中断函数 (#pragma vect/ #pragma interrupt)	产生中断向量表，通过中断输出所需的目标代码。 允许在 C 中写中断函数。
中断函数修饰符 (__interrupt, __interrupt_brk)	可用于描述向量表设置以及在其它文件中的中断函数定义。
中断函数 (#pragma DI, #pragma EI)	在目标代码中嵌入指令 , 禁止 / 允许中断。
CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)	在目标代码中嵌入以下指令。 halt 指令 stop 指令 brk 指令 nop 指令
Bit 域声明 (类型说明符的扩展)	unsigned char, signed char, signed int, unsigned short, signed short 类型的位域定义可节省存储器空间，获得较短目标代码和较快执行速度。
位域声明 （位域的分配方向）	-rb 选项改变位域分配次序。
改变编译器输出区段名称 (#pragma section ...)	允许改变编译器输出 section 名和命令链接器独立分配这块 section。
二进制常量 (0bxxx)	允许在 C 源代码中指定二进制常数。
模块名称改变函数 (#pragma name)	目标代码模块名在 C 源代码中的名称可以任意改变。
循环移位函数 (#pragma rot)	输出对表达值移位的代码到目标代码中，通过直接内联展开。

扩充功能	说明
乘法函数 (#pragma mul)	输出表达值与以直接内联目标相乘的代码。 结果的目标代码较小并且执行速度较快。
除法函数 (#pragma div)	使用除法指令输入 / 输出的数据大小输出指令。 这代码兼容 78K0 编译器。 这目标代码尺寸较小且执行速度快于除法公式。
BCD 运算函数 (#pragma bcd)	出对象中的表达式值执行 BCD 操作的代码，通过直接内联展开。 BCD 操作是由 4 位二进制数代表 1 位十进制数来计算。
数据插入函数 (#pragma opc)	插入常数数据到当前地址中。 专用数据和指令可以不使用 ASM 语句嵌入代码区。
RTOS 中断处理 (#pragma rtos_interrupt ...)	对于 RX78K0R 的中断处理可被描述。
RTOS (__rtos_interrupt) 中断处理修饰符	对于 RX78K0R 的矢量设置和中断处理的说明能在分开的文件中说明。
RTOS 任务函数 (#pragma rtos_task)	对于 RX78K0R 以 #pragma rtos_task 指定的功能名解释为任务。 这允许在 C 中写实时 OS 任务的高效率代码。
闪存区域分配方式 (-zf)	以 -zf 选项编译，允许程序分配在闪存区并且在引导区允许这些程序链接目标代码 （没有 -zf 选项编译）。
闪存区域跳转表和闪存区域分配 (#pragma ext_table)	由 #pragma 指令符规定了闪存区分支表的起始地址，允许启动程序和中断功能分配在闪存区并且允许从引导区调用闪存区函数。
从引导区到闪存区的函数调用 #pragma ext_func)	#pragma 指令规定了从引导区调用闪存区中的函数名和 ID 数值，因而允许从引导区调用闪存区函数。
镜像源程序区说明	以 -mi0/-mi1 选项编译，命令编译器为指定镜像源程序区产生代码。
参数 / 返回值的 int 展开限制方法 (-zb)	以 -zb 选项编译来产生较小的目标代码和较快的执行速度。
存储器运算函数 (#pragma inline)	目标文件由标准库输出函数以直接内联的内存复制和内存设置产生。 结果代码执行速度更快。

扩充功能	说明
绝对地址分配规范 (__directmap)	在模块中 __directmap 的声明，其变量是以绝对地址定义分配定义的。一个或多个变量可以分配于相同的任意地址。
near/far 区域规范	当函数或变量声明时，分配函数或变量的地方通过加到 __near 或 __far 类型描述来特别定义。
存储器模式规范	编译时，分配函数或变量的地点可以通过 -ms、 -mm 或 -ml 选项按存储器模式来指定。
分配 ROM 数据说明	允许分配 ROM 数据进任意 far 或 near 区。

call 函数 (callt/__callt)

被调用函数的地址分配在 callt 表中，用以调用这函数。

[功能]

- callt 指令存放调用的函数地址至 callt 表的 [80H 至 BFH] 区域内，这样将比直接调用函数使用的代码更短。
- 要调用由 callt(或 __callt)(称为 callt 函数) 声明的函数，用？作前缀加在函数名称前。要调用该函数，使用 callt 指令。
- 要被调用的函数与常用普通函数并无不同之处。

[效果]

- 可以缩短目标代码。

[使用方法]

- 如下所示，把 callt/__callt 属性加到要被调用的函数上 （在开始时描述）：

```
callt   extern  type-name       function-name
__callt extern  type-name       function-name
```

[限制]

- callt 函数分配于 [C0H 至 0FFFFH] 的区域 , 而不考虑存储器模块。
- 以 callt/__callt 声明的每一个函数地址，在链接目标模块时将分配在 callt 表中 。基于这个原因 , 在汇编源程序模块中使用 callt 表时 , 必须使用符号建立 " 可重定位 " 的程序。
- 在链接时会检查调用函数的数量。
- 当指定了 -za 选项，则允许 __callt，禁止 callt。
- 指定 -zf 选项时 , 不能定义 callt 函数。如果定义 callt 函数就发生错误。
- callt 表的区域是 80H 至 BFH。
- 当 callt 表使用超过 callt 属性函数的数量允许范围时，将发生编译错误。
- 通过 -ql 选项的指定使用 callt 表。基于那个原因，每个装载的模块允许 callt 属性的数量和链接模块中的总数如下所示。

选项	-ql1	-ql2 至 -ql3
callt 属性函数的数量	32	30

- 不使用 -ql 选项的情况以及默认值如下表所示。

callt 函数	限制值
每个装载模块数量	最大 32
链接模块中的总数	最大 32

[示例]

```
(C source)
============ ca1.c ============
__callt extern  int     tsub ( void );

void main ( void ) {
        int     ret_val ;
        ret_val = tsub ( );
}
```

```
 ============ ca2.c ============


__callt int     tsub ( void ) {
        int val ;
        return val ;
}
```

```
( 编译器的输出目标 )
ca1 module
        EXTRN   ?tsub           ;Declaration
        callt   [?tsub]         ;Call

ca2 module
        PUBLIC  _tsub           ;Declaration
        PUBLIC  ?tsub           ;
@@CALT  CSEG    CALLT0          ;allocation to segment
?tsub : DW      _tsub
@@BASE          CSEG    BASE
_tsub :                         ; Function definition
                :
                                ;Function body
                :
```

对函数 tsub( ) 给定了 call 属性，使其能保存在 callt 表中。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 即使不使用关键字 callt/__callt，也不需要修改 C 源程序。
- 要改变 callt 函数的属性 , 需注意上述使用方法中描述的过程。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 必须使用 #define。详情请参阅 “3.3.5　C 源代码修改 "。

寄存器变量 (寄存器)

变量分配到寄存器和 saddr 区域。

[功能]

- 把描述的变量 (包含函数的参数) 分配到寄存器 (HL) 和 saddr 区域 (_@KREG00 至 _@KREG15)。对描述的寄存器在模块预处理 / 后处理期间保存和恢复寄存器或 saddr 区域。
- 寄存器变量分配详细信息参阅 "3.4　函数调用接口 "。
- 寄存器变量分配进寄存器 HL 或 saddr 区域 (FFEB4H 至 FFEC3H), 根据引用频率的次序分配。只在没有堆栈结构时寄存器变量才分配到寄存器 HL，也仅在指定 -qr 选项时寄存器变量才分配于 saddr 区域。

[效果]

- 分配变量到寄存器或 saddr 区域的指令代码长度通常要比在存储器中的分配指令代码短。这有助于缩短代码和提高程序执行速度。

[使用方法]

- 下面是用寄存器存储分类符描述：

```
callt   extern  type-name       function-name
```

[限制]

- 如果寄存器变量不频繁使用，目标代码可能增加 （取决于源程序的大小和内容）。
- char/int/short/long/float/double/long double 和 pointer 数据类型可以使用寄存器变量声明。
- char 类型使用的区域是 int 类型使用的一半。 long, float, double, long double 和 far pointer 使用的区域是 int 类型使用的两倍。在 chars 型变量之间有字节限制，但是在其它情况中存在字限制。
- 在 int, short 和 near pointer 情况中 , 每个函数最多可使用 8 个变量。第 9 个及其之后的变量分配于普通存储器中。
- 在没有栈帧的函数情况中，对于 int, short 和 near pointer，其每个函数最多可使用 9 个变量。第 10 个及其之后的变量分配于普通存储器中。

[示例]

<C 源代码 >

```
void func ( );

void main ( ) {
        register int    i, j ;
        i = 0 ;
        j = 1 ;
        i += j ;
        func ( );
}
```

(1) 寄存器变量分配至寄存器和 saddr 区域的示例

下面的标号在启动例程中声明 (参阅 "3.5　saddr 区域标签列表 ")。

< 编译器输出的目标代码 >

```
        EXTRN   _@KREG00        ; 将要使用的引用 saddr 区域
@@CODEL CSEG
_main :
        push    hl              ; 在函数初期保存寄存器内容
                                ;
        movw    ax, _@KREG00    ; 在函数初期保存 saddr 初始函数的内容
                                ;
        push    ax
; line  3 :     register int i, j;
; line  4 :     i = 0;
; line  5 :     j = 1;
        movw    hl, #00H        ; 下面的代码在函数的中间输出
                                ;
        onew    ax
        movw    _@KREG00, ax    ; j
; line  6 :     i += j;
        addw    ax, hl
        movw    hl, ax
; line 7 :
        pop     ax              ; 在函数的最后恢复 saddr 的内容
                                ;
        movw    _@KREG00, ax
        pop     hl              ; 在函数的最后恢复寄存器的内容
                                ;
        ret
        END
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- 如果其它 C 编译器支持寄存器声明， C 源程序不需要修改。
- 要改变寄存器变量，在程序中增加寄存器变量声明。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果其它编译器支持寄存器声明， C 源程序不需要修改。
- 可以使用多少变量以及将其分配在哪个区域取决于其它 C 编译器的具体情况。

如何使用 saddr 区域 (sreg/__sreg)

声明为 sreg 或 __sreg 的外部变量和声明在函数内的静态变量分配到 saddr 区域。

[功能]

- 以关键字 sreg 或 __sreg 声明的外部变量和函数内静态变量 （称为 sreg 变量）自动分配到 saddr 区域 [FFE20H 至 FFEB3H] 并具有可再分配性。当这些变量超过上面所示的区域时，出现编译错误。
- sreg 变量处理方法与 C 源程序中的普通变量处理方法相同。
- char, short, int 和 long 型的 sreg 变量的每一个位都会自动变成 boolean 型变量。
- 被声明的 sreg 变量如没有赋初始值，则 0 作为初始值赋予。
- 声明在汇编源程序中的 sreg 变量，其地址可以引用 saddr 区域 [FFE20H 至 FFF1FH]。区域 [FFEB4H 至 FFEDFH] 被编译器使用，所以必须小心处理 (参阅 图 3-2.　存储器空间的使用)。

[效果]

- 对应 saddr 区域的指令代码的长度通常比对应存储器的指令代码长度要短。这有助于缩短目标代码和提高程序执行速度。

[使用方法]

- 在定义为变量的模块和函数中用关键字 sreg 和 __sreg 声明变量。只有指定静态存储的变量才能成为函数内部 sreg 变量。

```
sreg    类型名      变量名 / sreg    static  类型名      变量名
__sreg  类型名      变量名 / __sreg  static  类型名      变量名
```

- 在引用 sreg 外部变量的模块内声明下面的变量。它们还可以在函数内被描述。

```
extern  sreg    类型名    变量名 / extern  __sreg  类型名    变量名
```

[限制]

- 如果对函数指定 const 类型或 sreg/__sreg 类型，会输出警告信息，并且 sreg 声明被忽略。
- char 类型占用空间是其它类型占用空间的一半，long/float/double/long double/far pointer 类型占用空间是其它类型占用空间的两倍。
- 在 char 型变量之间有字节限制，但是在其它情况中存在字限制。
- 当指定了 -za 选项，只允许 __callt，禁止 callt。
- 在 int/short、 near pointer 和 pointer 情况中，每个装载的模块最多可使用 74 个变量 (当使用 saddr 区域 [FFE20H 至 FFEB3H] 时)。
  注意，当使用 bit 型和 boolean 型变量时，可用的变量数会减少。

[示例]

<C 源代码 >

```
extern sreg int hsmm0 ;
extern sreg int hsmm1 ;
extern sreg int *hsptr ;

void main ( ) {
        hsmm0 -= hsmm1 ;
}
```

以下的示例显示了由用户建立的 sreg 变量定义的代码。如果在 C 源程序中没有作出 extern 声明， 78K0RC 编译器输出以下代码。在此情况中，将不输出 ORG 伪指令。

```
        PUBLIC  _hsmm0  ; Declaration
        PUBLIC  _hsmm1  ;
        PUBLIC  _hsptr  ;

@@DATS  DSEG    SADDRP  ; Allocation to segment
        ORG     0FE20H  ;
_hsmm0 :        DS      ( 2 )   ;
_hsmm1 :        DS      ( 2 )   ;
_hsptr :        DS      ( 2 )   ;
```

在函数中输出以下代码。

```
movw    ax, _hsmm0
subw    ax, _hsmm1
movw    _hsmm0, ax
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- 如果其它编译器不使用关键字 sreg/__sreg，则不需要修改。
  要改变 sreg 变量，遵照上面的方法进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 通过 #define 作修改。对于详细信息参阅 "3.3.5　C 源代码修改 "。因此 , sreg 变量作为普通变量处理。

外部变量 / 内部静态变量的 saddr 自动分配选项 (-rd) 的使用方法

-rd 选项自动分配外部变量和内部静态变量进 saddr 区域。

[功能]

- 不管是否作了 sreg 声明，外部变量 / 内部静态变量 （除了 const 型外）自动分配到 saddr 区域。
- 取决于 *n* 值和指定的 m，外部静态变量和外部静态变量分配如下指定。

n 和搭	分配到 saddr 区域的变量
n	(1) *n* = 1 时 char 和 unsigned char 型变量 (2) *n* = 2 时 当 *n* = 1 时的变量，另外加上 short, unsigned short, int, unsigned int, enum 和 near pointer 类型的变量。 n = 4 时 当 n = 2 时的变量，另外加上 long, unsigned long, float, double 和 long double, far pointer 类型变量
m	结构体，联合体和数组
省略时	所有变量

- 不论以上规范如何，用关键字 sreg 声明的变量都分配到 saddr 区域时。
- 以上的规则也适用于由 extern 声明引用的变量，同这些变量分配到 saddr 区域一样完成处理。
- 由这此项分配到 sddr 区域的变量以 sreg 变量同样的方式进行处理。这些变量函数和限制描述如下 [如何使用 saddr 区域 (sreg/__sreg)]。

[规范方法]

- 指定 -rd[*n*][m] (*n* = 1, 2 或 4) 选项。

[限制]

- 在 -rd[*n*][m] 选项中，不同 n, m 值的模块之间不能相互链接。

内部静态变量的 saddr 自动分配选项的使用方法 (-rs)

-rs 选项将内部静态变量自动分配到 saddr 区域。

[功能]

- 将内部静态变量 （除 const 型）自动分配到 saddr 区 saddr 区域，而不管是否作出 sreg 声明。
- 取决于 *n* 的值和 m 的定义，内部静态变量的分配情况如下所示。

n 和搭	分配到 saddr 区域的变量
n	(1) 当 *n* = 1： char 和 unsigned char 型变量 (2) 当 *n* = 2： 当 *n* = 1 时的变量，另外加上 short, unsigned short, int, unsigned int, enum 和 near pointer 型变量 (3) 当 *n* = 4： 当 *n*= 2 时的变量，另外加上 long, unsigned long, float, double 和 long double, far pointer 型变量
m	结构体，联合体和数组
省略时	所有变量 （仅在这种情况下包括结构体，联合体和数组）

- 用关键字 sreg 声明的变量分配到 saddr 区域时，忽略上面的指定。
- 由这此项分配到 sddr 区域的变量以 sreg 变量同样的方式进行处理。这些变量函数和限制描述如 [如何使用 saddr 区域 (sreg/__sreg)] 中所述。

[规范方法]

- 指定 -rs[*n*][m] (*n* = 1, 2 或 4) 选项。

备注　　在 -rs[*n*][m] 选项中，指定不同 n, m 值的模块可以相互链接。

如何使用 sfr 区域 (sfr)

#pragma sfr 指令表示 sfr 名，能用来操作 C 源程序文件的特殊功能寄存器。

[功能]

- sfr 指的是一组特殊功能寄存器，诸如 78K0R 微控制器中各种外围设备的模式寄存器和控制寄存器。
- 通过声明使用 sfr 名称，对 sfr 区域的操作可以直接在 C 源代码级进行描述。
- sfr 变量为不具有初值的外部变量 （未定义）。
- 对只读 sfr 变量执行写入检查。
- 对只写 sfr 变量执行读取检查。
- 将非法数据分配到 sfr 变量将会导致编译错误。
- 可以使用的 sfr 名称都分配到由地址 [FFF00H 至 FFFFFH 和 F0000H 至 F07FFH 注] 组成的范围内。

注 取决于所使用器件不同而不同。

[效果]

- 对 sfr 区域的操作可以直接在 C 源代码级进行描述。
- 对应 sfr 区域的指令代码的长度比对应存储器的指令代码长度要短。这有助于缩短目标代码和提高程序执行速度。

[使用方法]

- 以 #pragma 预处理指令声明在 C 源代码中使用 sfr 名称，如下所示 （关键字 sfr 可以用大写或小写字母表示。）：

```
#pragma sfr
```

#pragma sfr 指令必须在 C 源代码行的开始处声明。
然而，如果指定了 #pragma PC （处理器型），则此后说明 #pragma sfr 指令应该紧随其后。
以下语句和指令可以放在 #pragma sfr 指令之前：

- 注释语句
- 没有定义或引用变量或函数的预处理指令

- 在 C 源程序中，对 sfr 名称的描述按照器件原有的 sfr 名称进行描述 （不改变）。。在此情况下，无需声明 sfr。

[限制]

- 所有 sfr 名称必须以大写字母表示。以小写字母表示的 sfr 会按普通变量进行处理。

[示例]

<C 源代码 >

```
#ifdef  __K0R__
#pragma sfr
#endif

void main ( void ) {
        PL0 -= ADCR ;
        /* ADCR = 10 ;  ==> error */
}
```

不输出有关声明的代码，且会有以下代码在函数中输出。

```
mov     a, PL0
sub     a, ADCR
mov     PL0, a
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- C 源程序中不依赖于器件或编译器的部分无需修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 ?pragma sfr

  示例如下所示。

```
#ifdef  __K0R__
#pragma sfr
#else
/*Declaration of variables*/
unsigned char   P0 ;
#endif

void main ( void ) {
        P0 = 0 ;
}
```

- 对于具有使用 sfr 或其替代功能函数的器件，必须创建专用库来以访问该区。

bit 型变量 (bit), boolean 型变量 (boolean/__boolean)

bit 和 boolean 型变量指定产生占有存储器区 1-bit 的空间。

[功能]

- 一个 bit 或 boolean 型变量按 1-bit 数据进行处理，并分配到 saddr 区域。
- 这个变量可以按与没有初值 （或有未知的值）的外部变量相同的处理方式处理。
- C 编译器对这个变量输出以下这些位操作指令：

```
MOV1 , AND1 , OR1 , XOR1 , SET1 , CLR1 , NOT1 , BT , BF
```

[效果]

- 可在 C 语言中执行基于汇编源程序级编程，并且能够以位为单位访问 saddr 和 sfr 区。

[使用方法]

- 在模块内声明 bit 或 boolean 型，其中要用到的 bit 或 boolean 型变量声明如下所示：
- __boolean 还可以替代 bit 来说明。

```
bit             变量名称
boolean         变量名称
__boolean       变量名称
```

- 在模块内声明 bit 或 boolean 型，其中要用到的 bit 或 boolean 型变量声明如下所示：

```
extern  bit             变量名称
extern  boolean         变量名称
extern  __boolean       变量名称
```

- char， int， short 和 long 型 sreg 变量 （除数组元素和结构体成员外）以及 8-bit 的 sfr 变量可以自动作为 bit 型变量使用。

```
变量名.n （其中 n =0 至 31）
```

[限制]

- 通过使用 CY （溢位）标志位完成 2 位或 boolean 型变量的操作。基于此原因，语句之间的溢出标志位的内容不能保证正确。
- 数组不能定义也不能引用。
- bit 或 boolean 型变量不能用于结构体或联合体成员。
- 此类型变量不能用作参数型函数。
- bit 型变量不能用作自动变量类型。
- 仅 bit 型变量而言，每个装载的模块最多可以使用 1184 个变量 (当 saddr 区域使用 [FFE20H 至 FFEB3H] 时) (正常模式)。
- 位变量声明时不能赋初值。
- 如果带有 const 关键字一起声明变量，则忽略 const 声明。

- 如下表所示，仅通过操作符和常数就可完成 0 和 1 的运算操作。

分类	运算符
Assignment	=
Bitwise AND	&, &=
Bitwise OR	|, |=
Bitwise XOR	^, ^=
Logical AND	&&
Logical OR	||
Equal	==
Not Equal	!=

- *, & (pointer 引用 ,address 引用)，不能完成 sizeof 操作。
- 当指定 -za 选项时，仅允许 _boolean。
- 在使用 sreg 变量情况中，如果指定 -rd， -rs(saddr 自动分配选项) 选项，将减少可用的 bit 类型的数量。

[示例]

<C 源代码 >

```
#define ON      1
#define OFF     0

extern bit data1 ;
extern bit data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while (data1 ) {
                data1 = data2 ;
                testb ( );
        }

        if ( data1 && data2 ) {
                chgb ( );
        }
}
```

这个示例是当用户为 bit 型变量生成定义代码时的情况。如果没有附加 extern 声明，则编译器输出以下的代码。此情况中，不输出 ORG 准 - 伪指令。

```
PUBLIC  _data1          ; Declaration
PUBLIC  _data2

@@BITS  BSEG            ; Allocation to segment
        ORG     0FE20H
_data1  DBIT
_data2  DBIT
```

在函数中输出以下的代码

```
set1    _data1                  ( 初始化 )
clr1    _data2                  ( 初始化 )
bf      data1, $?L0004          ( 判断 )
mov1    CY, _data2              ( 赋值 )
mov1    _data1, CY              ( 赋值 )
bf      _data1, $?L0005         ( 逻辑与表达式 )
bf      _data2, $?L0005         ( 逻辑与表达式 )
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 即使不使用关键字 bit, boolean 或 __boolean，也不需要修改 C 源程序。
- 要将变量改为 bit 或 boolean 型变量，请遵照上述使用方法中所描述的修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 必须使用 #define。详细信息参阅 "3.3.5　C 源代码修改 " (此情况的结果是， bit 或 boolean 型变量按 普通变量处理。)。

ASM 语句 (#asm - #endasm/__asm)

#asm 和 __asm 指令允许在 C 源程序代码中使用汇编语言语句。语句由 C 编译器产生嵌入汇编源代码中。

[功能]

(1) #asm - #endasm

- 通过 78K0RC 编译器预处理指令 #asm 和 #endasm，由用户描述的汇编源程序可以嵌入进汇编源文件中输出。
- #asm 和 #endasm 行将不输出。

(2) __asm

- 汇编指令将汇编代码描述输出为字符串文字，并嵌入汇编源程序文件。

[效果]

- 可在汇编源代码中操作 C 源代码的全局变量
- 在 C 源代码中执行不能描述的函数。
- 通过 C 编译器来手工优化汇编源程序输出，并且把它嵌入进 C 源代码 (为获得有效目标代码)

[使用方法]

(1) #asm #endasm

- 用 #asm 指令指示汇编源代码起始，用 #endasm 指令指示汇编源代码结束。在 #asm 与 #endasm 之间描述汇编源代码。

```
#asm
  :     /* Assembler source */
#endasm
```

(2) __asm

- 在 C 源程序中， ASM 语句以下面的格式描述：

```
__asm ( 字符串文字 );
```

- 字符串文字的说明方法符合 ANSI 规范，且通过使用换码字符串转义字符 （\n：换行， \t：制表键）或 \，可以使一行连续，也或可以连接字符串。

[限制]

- #asm 指令不允许嵌套。
  如果使用 ASM 语句，则不创建目标模块文件。而是，将创建汇编源代码文件。
  [Output assemble file] 由设置属性面板的 [Compile Options] 表设置为 "Yes"。 (有关设置的方法，参阅 "CubeSuite 78K0R build"。)
- 仅小写字母可以用于来说明 __asm。如果 __asm 以大小写字符混合的方式来说明，则编译器将其看作用户函数。
- 当指定 -za 选项时，仅允许 __asm。
- #asm-#endasm 和 __asm 块仅可以在 C 源代码的函数内说明。因此，汇编源代码输出到段名称为 @@CODE 或 @@CODELE 的 CSEG。

[示例]

(1) #asm - #endasm

<C 源代码 >

```
void main ( void ) {
#asm
        callt [init]
#endasm
}
```

< 编译器输出的目标代码 >

```
@@CODEL CSEG
_main :
        callt [init]
        ret
        END
```

在以上示例中，在 #asm 与 #endasm 之间的语句将作为汇编源代码程序输出到汇编源代码文件。

(2) __asm

<C 源代码 >

```
int     a, b ;

void main ( void ) {
        __asm ( "\tmovw ax, !_a \t ; ax <- a" );
        __asm ( "\tmovw !_b, ax \t ; b <- ax" );
}
```

< 编译器输出的目标代码 >

```
@@CODEL CSEG
_main :
        movw    ax, !_a         ; ax <- a
        movw    !_b, ax         ; b <- ax
        ret
        END
```

[兼容性]

- 如果 C 编译器支持 #asm 的，可以遵照 C 编译器指定的格式修改程序。
- 如果目标器件不同，则需要修改程序的汇编源代码部分。

汉字 (2 字节字符) (/* kanji */, // kanji)

C 源程序注释可以包含汉字 （多字节日文字符）。

[功能]

- 汉字代码可以在 C 源程序文件中描述。
- 在注释中的汉字代码作为注释的一部分处理，所以这种码不作为目标码来编译。
- 使用在注释中的汉字代码可以由选项或环境变量来指定。
  如果无选项指定，代码就以环境变量 LANG78K 设置为汉字代码。
- 如果汉字代码由选项和环境变量 LANG78K 两种指定，那么选项指定优先。
- 如果 "SJIS" 设置在环境变量 LANG78K 中，在注释中的汉字替代为移位 JIS 码。
- 如果 "EUC" 设置在环境变量 LANG78K 中，编译器把注释中的汉字类型理解为 EUC 码。
- 如果 "NONE" 设置在环境变量 LANG78K 中，编译器把注释理解为不含汉字码。
- 默认指定 SJIS 码。

[效果]

- 汉字码的使用，使日文编程者更容易理解注释，编写 C 源程序更轻松。

[使用方法]

- 通过编译器选项或环境变量设置汉字代码。 (如果使用默认设置，就不需要来设置。)

(1) 通过编译器选项设置

选项设置列出在下面的表中。

选项	说明
-zs	SJIS 移位 JIS 码)
-ze	EUC (EUC 码)
-zn	NONE (不使用汉字代码)

(2) 通过环境变量 LANG78K 设置

- 设置 "SJIS", "EUC" 或 "NONE"。（如果需要，也可以在如 autoexec.bat 文件中描述。）
- SJIS, EUC 或 NONE 的规范不区分大小写。
- 在 C 源程序文件中描述汉字字符要遵照 LANG78K 中的指定。

```
SET     LANG78K = SJIS   ( 移位 JIS 码 )
SET     LANG78K = EUC    (EUC 码 )
SET     LANG78K = NONE   ( 不使用汉字代码 )
```

[限制]

- 仅移位 JIS 码和 EUC 码能描述在注释中。除了注释外，只有 0x7F 字符或低于 0x7F 的 ASCII 码才能描述。不管全角字符还是半角片假名 （包含半角符号）都不能描述在除了注释外的任意地方。
  如果描述任何这些字符，所描述的字符码可能不会输出。

[示例]

<C 源代码 >

```
// main function
void main ( void ) {
        /* Comment */
}
```

汉字类型信息输出到汇编源程序。

< 编译器输出的目标代码 >

```
$KANJICODE SJIS
```

当 C 源程序内容输出到汇编源程序时，也输出注释中的汉字字符。

```
; line          1 : // main function
; line          2 : void main ( void ) {
@@CODEL CSEG
_main :
; line          3 :             /* Comment */
;  line          4 : }
```

[说明]

- 汉字代码仅可以在 C 源程序文件注释中描述。
- 当使用格式 "// comment", 时，指定编译器选项 -zp。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果在注释区域中汉字不能描述 (不是 "/* ...*/", or "// newline" 区域)，源文件必须修改。
- 如果汉字代码与指定在 CC78K0R 中不同，必须首先转换汉字代码。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 对于支持在注释中描述汉字字符的 C 编译器不需要修改 C 源程序。
- 如果 C 编译器不支持在注释中描述汉字字符，必须在 C 源程序中删除汉字字符。

中断函数 (#pragma vect/#pragma interrupt)

产生中断向量表，通过中断输出所需的目标代码。

[功能]

- 所描述的函数名称的地址被注册到对应于指定中断请求名称的中断向量表中。
- 中断函数输出代码，在函数开始时和结束时的堆栈对数据 （除使用在 ASM 语句外）进行保存或恢复 (如果寄存器区被指定，那么在此码之后)：
  - 寄存器
  - 寄存器变量所用的 saddr 区域
  - 用于工作的 saddr 区域
  - 运行时库的 saddr 区
  - 用于保存段信息的 saddr 区域
  - ES 和 CS 寄存器

  注，这取决于中断函数的规范或状态，保存 / 恢复分别进行，如下所示：
  - 如果指定不改变，则不管是否使用这些代码，改变寄存器区或保存 / 恢复寄存器内容的代码，以及保存 / 恢复 saddr 区域内容的代码不作输出。
  - 如果寄存器区指定，用来选择寄存器区的代码输出在中断函数的开始处，因此，不对寄存器保存或恢复。
  - 如果不指定不改变并且在中断函数中调用函数，那么不管是否指定使用寄存器，整个寄存器区被保存或恢复。

- 如果编译时不指定 -qr 选项，则不使用寄存器变量的 saddr 区域和工作的 saddr 区域，所以不输出保存 / 恢复的代码。

  如果保存代码的大小小于恢复代码，则输出恢复代码。下表总结了以上内容，并列出了保存 / 恢复区域。

保存 / 恢复区域	NO BANK	函数调用				不调用函数			
		未指定 -qr		指定 -qr		未指定 -qr		指定 -qr	
		堆栈	RBn	堆栈	RBn	堆栈	RBn	堆栈	RBn
使用寄存器	NG	NG	NG	NG	NG	OK	NG	OK	NG
全部寄存器	NG	OK	NG	OK	NG	NG	NG	NG	NG
使用运行时库的 saddr 区域， ES， CS 寄存器， 用于保存段信息的 saddr 区域	NG	NG	NG	NG	NG	OK	OK	OK	OK
全部运行时库的 saddr 区域， ES， CS 寄存器， 用于保存段信息的 saddr 区域	NG	OK	OK	OK	OK	NG	NG	NG	NG
使用寄存器变量的 saddr 区域	NG	NG	NG	OK	OK	NG	NG	OK	OK
用于工作的全部 saddr 区域	NG	NG	NG	OK	OK	NG	NG	NG	NG

Stack ： 指定使用的堆栈
RBn ： 指定寄存器组
OK ： 保存
NG ： 不保存

[效果]

- 中断函数可以在 C 源代码级描述。
- 由于寄存器区可以改变，保存寄存器的代码不输出，所以缩短了目标代码并且提高了执行速度。
- 无需知道向量表的地址，就可以识别中断请求名称。

[使用方法]

- 使用 #pragma 指令来指定中断请求名称、函数名称、堆栈开关、由编译器使用的寄存器以及是否保存 / 恢复 saddr 区域。在 C 源代码的开始处声明 #pragma 指令。 #pragma 指令描述在 C 源程序开始处 （如需关于中断请求的具体名称，参阅 所使用目标器件的用户手册）。关于软件中断 BRK，描述 BRK_I。
- 以下各项可以在此 #pragma 指令之前进行描述。
  - 注释
  - 预处理指令既不能定义也不能引用变量或函数

```
#pragmaΔvect(or interrupt)Δinterrupt-request-nameΔfunction-nameΔ

              [Stack-change-specification]┌Δ  Stack-usage-specification ┐
                                          │    No-change-specification  │
                                          └    Register-bank-specification┘
```

- *中断请求名*
  以大写字母描述。
  请参阅使用相关的目标设备的用户手册 （示例：NMI， INTP0 等）。
  关于软件中断 BRK，描述 BRK_I。
- *函数名*
  描述中断处理的函数名
- *堆栈改变规范*
  SP = 数组名称 [+ 偏移位置]（示例：SP = buff + 10）。
  通过 unsigned short 定义数组 （示例：unsigned short buff [5];）。
  为偏移位置指定偶数的 buff 大小或低于此大小。 (示例：在 unsigned short buff[5] 的情况中， buff 大小是 10 个字节，所以指定了偶数 10 或低于 10 的数。）
- *堆栈使用规范*
  STACK (默认)
- *不改变规范*
  NOBANK
- *寄存器区规范*
  RB0/RB1/RB2/RB3
- Δ
  空格

注意事项 由于 78K0RC 编译器的启动例程对寄存器区 0 初始化 所以一定要具体指定寄存器区 1 至 3。

[限制]

- 当不指定 -zf 选项时，不考虑存储器模式，把中断函数分配在 C0H 和 0FFFFH 之间的区域 。
  当指定 -zf 选项时，中断函数遵照存储器模式分配。另外，也可按 __near 或 __far 指定的分配区域来分配。
- near 区域以外区域中的数组不能指定为堆栈改变。如果指定将发生错误。
- 偶数以外的数值不能指定为偏移位置。如果指定将发生错误。
- 与其它微控制器不同， unsigned short 类型数组保留用于改变堆栈指针。
- 中断请求名必须以大写字母描述。
- 仅在一个模块内对中断请求名将做双重检验。
- 如果下面三个条件得到满足，寄存器的内容可以改变，但是编译器不对此作检验。
  如果指定要改变寄存器区，就要设置寄存器区，以使这些寄存器不会重叠。如果寄存器区重叠 , 就控制其中断，使之不发生重叠。
  当指定 NOBANK(不指定规范), 寄存器不保存。但是 , 要控制寄存器使其的内容不丢失。
  - 如果两个或两个以上中断发生
  - 如果在中断中使用相同的 BANK 发生两个或两个以上中断
  - 如果在描述 #pragma interrupt - 中指定 NOBANK 或寄存器组。
- callt/__callt/__rtos_interrupt/__flash/__flashf 函数不能指定为中断函数。
  只有当指定 -zf 选项时，才能指定 __far。
- 因为其中断函数不能具带有参数或返回值，所以指定具有 void 型的中断函数 （示例：void func （void） ;）。
- 即使 ASM 语句存在于中断函数中，保存所有寄存器和变量区域的代码也不输出。如果中断函数在汇编语句中使用了为编译器保留的区域或在 ASM 语句中调用函数，用户就必须保存寄存器和变量区域。
- 如果指定 leafwork1 至 16，输出警告并且忽略此规范 。
- 当指定改变堆栈，堆栈指针就改变为数据名称符号加偏移量的地址。数组名的区域不通过 #pragma 指令保留。需要分开定义作为全局 unsigned short 类型数组。
- 改变堆栈指针的代码产生在函数起始，设置堆栈指针返回的代码产生在函数的结束。
- 当关键字 sreg/__sreg 添加在数组上用于堆栈改变时，假定定义了两个或两个以上带有不同属性的相同名称的变量，并发生编译错误。可以由 -rd 选项在 sassd 区域中分配数组，但是代码大小和执行速度的效率不会提高，因为数组被使用为堆栈。推荐使用 saddr 区域而不是用堆栈。
- 用了不改变选项，就不能同时指定堆栈改变。如果指定将发生错误。
- 堆栈改变必须在堆栈使用 / 寄存器组规范前描述。如果堆栈改变描述在堆栈使用 / 寄存器组规范后，将发生错误。
- 如果在 #pragma vect/ #pragma interrupt 规范中，函数所指定的不改变，寄存器组或改变堆栈作为保存的目的地没有定义在相同的模块中， 输出警告信息，忽略堆栈改变。在此情况中，就使用默认堆栈。

[示例]

(1) 当指定寄存器组时

<C 源代码 >

```
#pragma interrupt INTP0 inter rb1

void inter ( void ) {
        /* Interrupt processing to INTP0 pin input*/
}
```

< 编译器输出的目标代码 >

```
@@VECT08        CSEG    AT      0008H ; INTP0
_@vect08:
        DW      _inter
@@BASE          CSEG    BASE
_inter :

        ; 切换寄存器组的代码
        ; 保存编译器使用的 saddr 区域代码
        ; 保存 ES 和 CS 寄存器
        ; INTP0 引脚输入的中断处理 （函数本体）
        ; 恢复 ES 和 CS 寄存器
        ; 恢复编译器使用的 saddr 区域代码
        reti
```

(2) 当堆栈改变和指定寄存器组时

<C 源代码 >

```
#pragma interrupt INTP0 inter sp = buff + 10 rb2

unsigned short  buff[5] ;
void func ( void );

void inter ( void ) {
        func ( );
}
```

< 编译器输出的目标代码 >

```
@@BASE      CSEG    BASE
_inter :
        sel     RB2             ; 改变寄存器组
        movw    ax, sp          ; 改变堆栈指针
        movw    sp, #_buff + 10 ;       :
        push    ax              ;       :
        movw    c, #0CH         ; 保存由编译器使用的 saddr
        dec     c               ;       :
        dec     c               ;       :
        movw    ax, _@SEGAX[c]  ;       :
        push    ax              ;       :
        bnz     $$ - 6          ;       :
        mov     a, ES           ; 保存 ES 和 CS 寄存器
        mov     x, a            ;       :
        mov     a, CS           ;       :
        push    ax              ;       :
        call    !!_func
        pop     ax              ; 恢复 ES 和 CS 寄存器
        mov     CS, a           ;       :
        mov     a, x            ;       :
        mov     CS, a           ;       :
        movw    de, #_@SEGAX    ; 恢复编译器使用的 saddr
        mov     c, #06H         ;       :
        pop     ax              ;       :
        movw    [de], ax        ;       :
        incw    de              ;       :
        incw    de              ;       :
        dec     c               ;       :
        bnz     $$ - 5          ;       :
        pop     ax              ; 恢复堆栈指针到原来的位置
        movw    sp, ax          ;       :
        reti

@@VECT08    CSEG    AT      0008H
_@vect08:
            DW      _inter
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用中断函数， C 源程序不需要修改。
- 要改变普通函数为中断函数，请遵照以上 [使用方法] 中所描述的过程修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma vect， #pragma interrupt 指令，中断函数可以作为普通函数使用。
- 当普通函数作为中断函数时，根据每个编译器的规范更改程序。

中断函数修饰符 (__interrupt, __interrupt_brk)

可用于描述向量表设置以及在其它文件中的中断函数定义。

[功能]

- 以 __interrupt 修饰符声明的函数被视为硬件中断函数，且通过不可屏蔽 / 可屏蔽中断函数的返回指令 RETI 执行返回。
- 通过用 __interrupt_brk 修饰符声明被认为是软件中断函数，并且由返回指令 RETB 对软件中断函数执行返回。
- 用此修饰符声明的函数被认为 (不可屏蔽 / 可屏蔽 / 软件) 中断函数，并保存或恢复寄存器和变量区域 (1) 和 (6) 以下 , 其作为编译器的工作区使用，保护或恢复到堆栈。
  如果在此函数中出现函数调用，则所有变量区保存到堆栈。

(1) 寄存器

(2) 寄存器变量所用的 saddr 区域

(3) 用于工作的 saddr 区域

(4) 运行时库的 saddr 区

(5) 用于保存段信息的 saddr 区域

(6) ES 和 CS 寄存器

备注 如果编译时未指定 -qr 选项 （默认），则不输出保存 / 恢复代码，因为不使用区 （2）和 （3）。

[效果]

- 通过此修饰符来声明函数，向量表的设置和中断函数定义可以在不同的文件中描述。

[使用方法]

- 将 __interrupt 或 __interrupt_brk 描述为中断函数的修饰符。

(1) 对于不可屏蔽 / 可屏蔽中断函数

```
__interrupt     void func ( ) {processing}
```

(2) 对于软件中断函数 >

```
__interrupt_brk void func ( ) {processing}
```

[限制]

- 当指定 -zf 选项时，不考虑存储器模式，把中断函数分配在 C0H 和 0FFFFH 之间的区域 。
  当指定 -zf 选项时，中断函数遵照存储器模式分配。另外，也可按 __near 或 __far 指定的分配区域来分配。
- 中断函数不能指定 callt/__callt/__rtos_interrupt/__flash/__flashf。

[注意事项]

- 仅声明此修饰符无法设定向量地址。向量地址必须通过使用 #pragma vect/interrupt 指令或汇编程序分别设置。
- Saddr 区域和寄存器都保存到堆栈。
- 即使设置了向量地址或通过 #pragma vect （或 interrupt 中断）改变了保存目的地 ...，如果在相同文件中不存在函数定义，忽略保存目的地的变化，且使用默认堆栈。
- 要在相同文件中定义中断函数为 #pragma vect （或 interrupt） ... 规范，即使未描述此修饰符，由 #pragma vect （或 interrupt ） ... 指定的函数名称也会被判定为中断函数。

  #pragma vect/interrupt 的详细信息 参阅 " 中断函数 (#pragma vect/#pragma interrupt)" 的 [使用方法]。

[示例]

- 按下面的格式中的声明或定义中断函数。通过 #pragma interrupt 设置向量地址的代码。

```
#pragma interrupt       INTP0   inter   RB1     /*The interrupt request name of*/
#pragma interrupt       BRK_I   inter_b RB2     /*The software interrupt is "BRK_I"*/

__interrupt     void inter ( );                 /*Prototype declaration*/
__interrupt_brk void inter_b ( );               /*Prototype declaration*/
__interrupt     void inter ( ) {processing};    /*Function body*/
__interrupt_brk void inter_b ( ) {processing};  /*Function body*/
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 除非支持中断函数，否则无需修改 C 源程序。
- 若需要，遵照以上 [使用方法] 中描述的过程修改中断函数。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 必须使用 #define 使中断修饰符按普通函数处理。
- 要使用中断修饰符用作中断函数，根据每个编译器的具体规范修改程序。

中断函数 (#pragma DI, #pragma EI)

在目标代码中嵌入指令 , 禁止 / 允许中断。

[功能]

- DI 和 EI 代码输出到目标，并建立一个目标文件。
- 如果省略 #pragma 指令，则 DI( ) 和 EI( ) 被视为普通函数。
- 如果 “DI() ；”在函数的开始时描述 （除自动变量、注释和预处理指令的声明外），则在函数预处理之前输出 DI 代码 （立即在函数名称标识符之后）。
- 为了在函数预处理后输出 DI 代码，在描述 "DI( );" 前开辟新块 （以 "{" 对此块划分界限）。
- 如果 “EI( );”在函数结束处描述 （除注释和预处理指令外），则在函数后处理后输出 EI 代码 （立即在代码 RET 之后）。
- 为了在函数后处理前输出 EI 代码，在描述 "EI( );" 后关闭新块 （以 "{" 对此块划分界限）。

[效果]

- 可以建立函数禁止中断。

[使用方法]

- 在 C 源代码的开始处描述 #pragma DI 和 #pragma EI 指令。
  但是，以下的语句和指令可以放在 #pragma DI 和 #pragma EI 指令之前 :
  - 注释语句
  - 其它 #pragma 指令
  - 既未定义又未引用变量或函数的预处理指令
- 在源程序中描述 DI( ) ；或 EI( ) 的方式和函数调用相同；。
- DI 和 EI 可以在 #pragma 之后用大写字母或小写字母描述。

[限制]

- 当使用这些中断函数时， DI 和 EI 不能用作函数名。
- DI 和 EI 必须用大写字母描述。如果用小写字母表示，则将其按普通函数进行处理。

[示例]

```
#ifdef  __K0R__
#pragma DI
#pragma EI
#endif
```

<C 源代码 >

```
#pragma DI
#pragma EI

void main ( void ) {
        DI ( );
        ;Function body
        EI ( );
}
```

< 编译器输出的目标代码 >

```
_main :
        di
        ; Preprocessing
        ;Function body
        ; Postprocessing
        ei
        ret
```

(1) 要在预处理之前 / 后处理之后输出 DI 和 EI

<C 源代码 >

```
#pragma DI
#pragma EI

void main ( void ) {
        {
                DI ( );
                ;Function body
                EI ( );
        }
}
```

< 编译器输出的目标代码 >

```
_main :
        ; Preprocessing
        di
        ;Function body
        ei
        ; Postprocessing
        ret
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用中断函数， C 源程序不需要修改。
- 要改变普通函数为中断函数，请遵照以上 [使用方法] 中所描述的过程修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma DI 和 #pragma EI 指令或通过 #ifdef 隔离， DI 和 EI 可以用作普通函数名 （例如：#ifdef__K0R__ ...#endif)。
- 当普通函数用作中断函数时，遵照每个编译器的规范修改程序。

CPU 控制指令 (#pragma HALT/STOP/BRK/NOP)

#pragma HALT/STOP/BRK/NOP 声明了嵌入 CPU 控制指令。

[功能]

- 下面的代码输出到目标代码来建立一个目标文件：
- 用于 HALT 操作 (HALT) 的指令
- 用于 STOP 操作 (STOP) 的指令
- BRK 指令
- NOP 指令

[效果]

- 用 C 程序可以使用微控制器的待机函数。
- 可以产生软件中断。
- 没有 CPU 运行的情况下时钟可以继续运行。

[使用方法]

- 在 C 源代码开始处说明 #pragma HALT， #pragma STOP#pragma NOP 和 #pragma BRK 指令。
- 以下各项可以在 #pragma 指令之前进行描述。
  - 注释语句
  - 其它 #pragma 指令
  - 既未定义又未引用变量或函数的预处理指令
- #pragma 后的关键字可以用大写或小写字母描述。
- 在 C 源代码中用大写字母以与函数调用相同格式的描述如下表示：

```
HALT ( );
STOP ( );
BRK ( );
NOP ( );
```

[限制]

- 当此函数使用时 , HALT, STOP, BRK 和 NOP 不能用作函数名。
- 以大写字母描述 HALT, STOP, BRK 和 NOP。如果用小写字母表示，则将其按普通函数进行处理。

[示例]

<C 源代码 >

```
#pragma HALT
#pragma STOP
#pragma BRK
#pragma NOP

void main ( void ) {
        HALT ( );
        STOP ( );
        BRK ( );
        NOP ( );
}
```

< 编译器输出的目标代码 >

```
@@CODEL CSEG
_main :
        halt
        stop
        brk
        nop
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用 CPU 控制指令，也不需要修改 C 源程序。
- 当使用 CPU 控制指令时，遵照以上 [使用方法] 所描述的过程修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 "#pragma HALT", "#pragma STOP", "#pragma BRK", 和 "#pragma NOP" 语句或用 #ifdef 隔离 , HALT, STOP, BRK, 和 NOP 可以用作函数名。
- 要使用这些指令作为 CPU 控制指令，遵照每个编译器的具体规范修改程序。

Bit 域声明 (类型说明符的扩展)

unsigned char, signed char, unsigned int, signed int, unsigned short 和 signed short 可以声明位域类型。

[功能]

- unsigned char 类型的位域不可跨越字节界限进行分配。
- unsigned int, signed int, unsigned short, signed short 类型的位域不能跨越字节界限进行分配，但是，当指定 -rc 选项时可以跨越字节界限进行分配。
- 同样大小的类型位域分配在字节单元同一字节单元 （或字单元）中。
  如果大小不同的类型，则位域分配在不同字节单元 （或字单元）中。
- unsigned short, signed short 类型按 unsigned int, signed int 类型作相应类似处理。

[效果]

- 可以保存存储器的内容，可以缩短目标代码且可以提高运行速度。

[使用方法]

- 作为位域类型说明符，unsigned char, signed char, signed int, unsigned short, signed short 类型可以指定附加到 unsigned int 类型中。
  声明如下。

```
struct  tag-name {
        unsigned char   field-name : bit-width ;
        unsigned char   field-name : bit-width ;
                :
        unsigned int    field-name : bit-width ;
};
```

[示例]

```
struct  tagname {
        unsigned char   A : 1 ;
        unsigned char   B : 1 ;
                :
        unsigned int    C : 2 ;
        unsigned int    D : 1 ;
                :
};
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 无需修改源程序。
- 改变类型说明符将 unsigned char， signed char, unsigned short, signed short 作为类型说明符。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果 unsigned char, signed char, signed int, unsigned short 和 signed short 不用作类型说明符，则无需修改源程序。
- 如果 unsigned char, signed char, signed int, unsigned short and signed short 用作类型说明符，则改为 unsigned int。

位域声明 （位域的分配方向）

-rb 选项改变位域分配次序。

[功能]

- 要改变位域分配的方向，当指定 -rb 选项时，位域从 MSB 端分配。
- 如果未指定 -rb 选项，将从 LSB 端分配位域。

[使用方法]

- 在编译时指定 -rb 选项可从 MSB 端分配位域。
- 当从 LSB 端分配位域时，不要指定该选项。

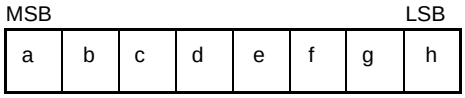
[示例]

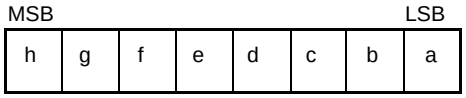
(1) 位域声明 1

```
struct  t {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
        unsigned char   g : 1 ;
        unsigned char   h : 1 ;
};
```

因为 a 至 h 为 8 位或更少的位，所以可分配在 1 字节的单元中。

从 MSB 进行位分配 通过指定 -rb 选项	从 LSB 进行位分配 不指定 -rb 选项

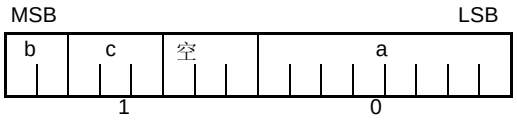




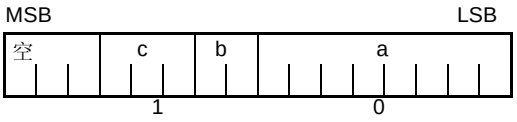
(2) 位域声明 2

```
struct  t {
        char a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned int    f : 5 ;
        unsigned int    g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 2 ;
};
```

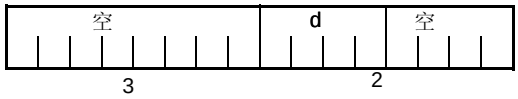
从 MSB 端进行位域分配
当指定 -rb 选项时

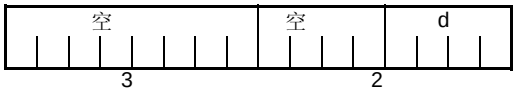


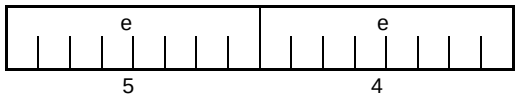
从 LSB 端进行位域分配
当不指定 -rb 选项时

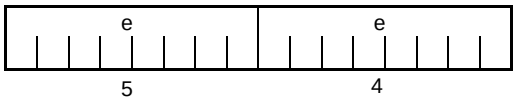


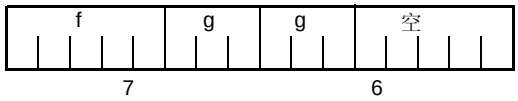
char 型成员 a 分配到第一字节单元。成员 b 和 c 分配到随后的第二个字节单元中。如果一个字节单元没有足够空间来容纳 char 型成员，则该成员将分配到后面的字节单元。在此情况下，如果在第二字节单元中仅有 3 位的空间且成员 d 有四位，则其将分配到第三个字节单元。

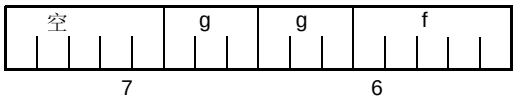


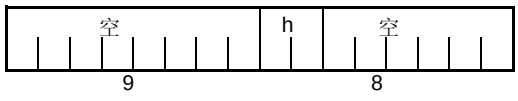


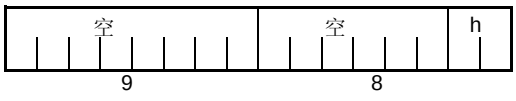




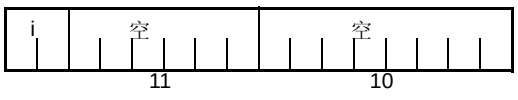


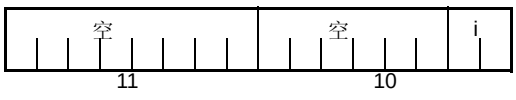






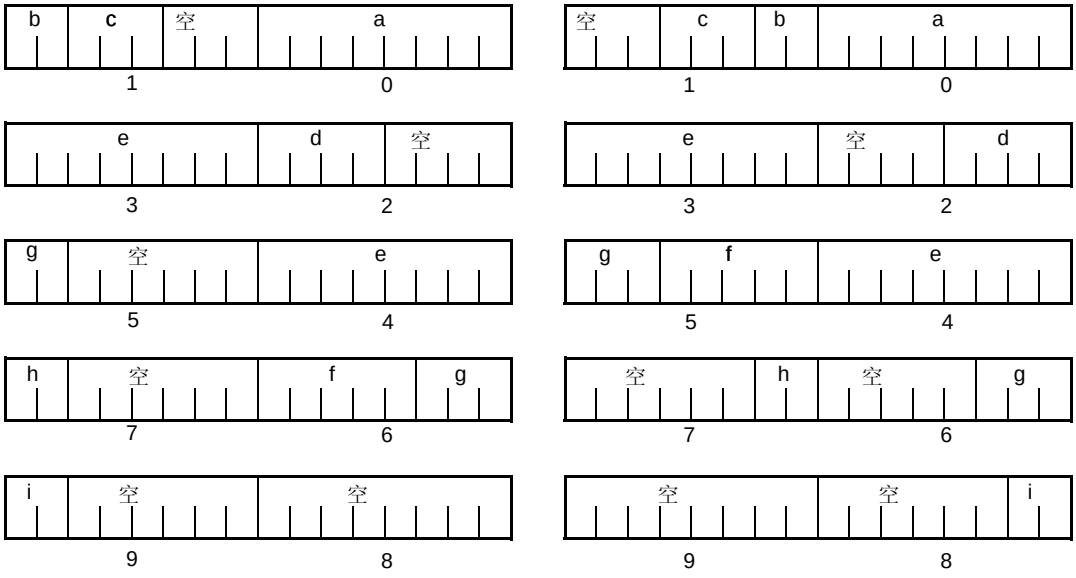
因为成员 g 为 unsigned int 型的位域，所以可跨字节界限进行分配。
因为 h 为 unsigned char 型位域，所以不能与 unsigned int 型的 g 位域分配在相同的字节单元中，而应分配在下一字节单元。





因为 i 为 unsigned int 型的位域，所以其分配在下一字单元。

当指定 -rc 选项时 （把结构体成员整合在一起），以上位域变为下述形式。

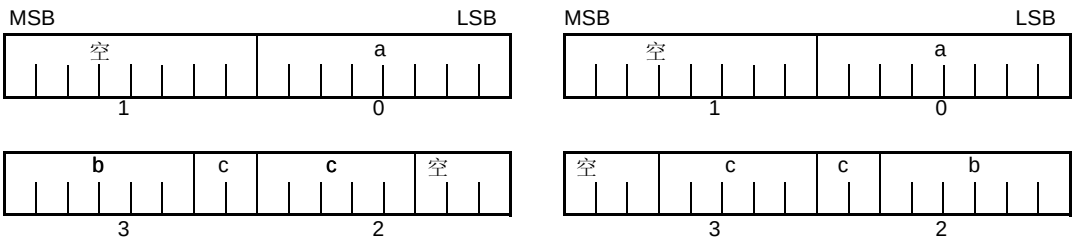


备注　　分配图下方的数字表示从该结构开始的字节偏移值。

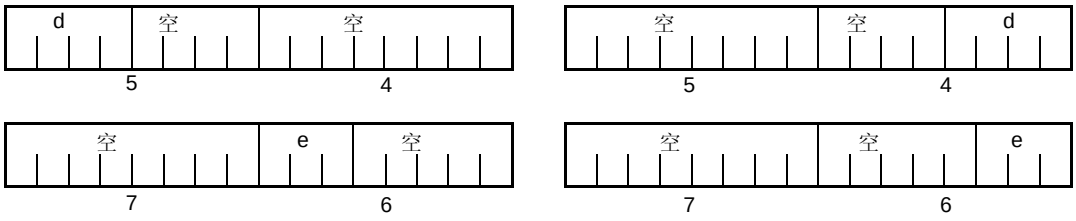
(3) 位域声明 3

```
struct  t {
        char a ;
        unsigned int    b : 6 ;
        unsigned int    c : 7 ;
        unsigned int    d : 4 ;
        unsigned char   e : 3 ;
        unsigned int    f : 10 ;
        unsigned int    g : 2 ;
        unsigned int    h : 5 ;
        unsigned int    i : 6 ;
};
```

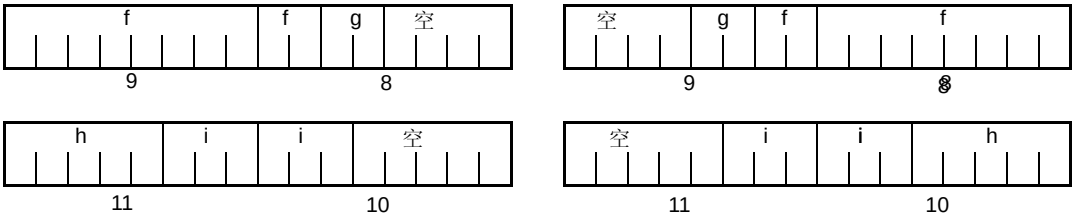
从 MSB 端进行位域分配 当指定 -rb 选项时	从 LSB 端进行位域分配 当不指定 -rb 选项时



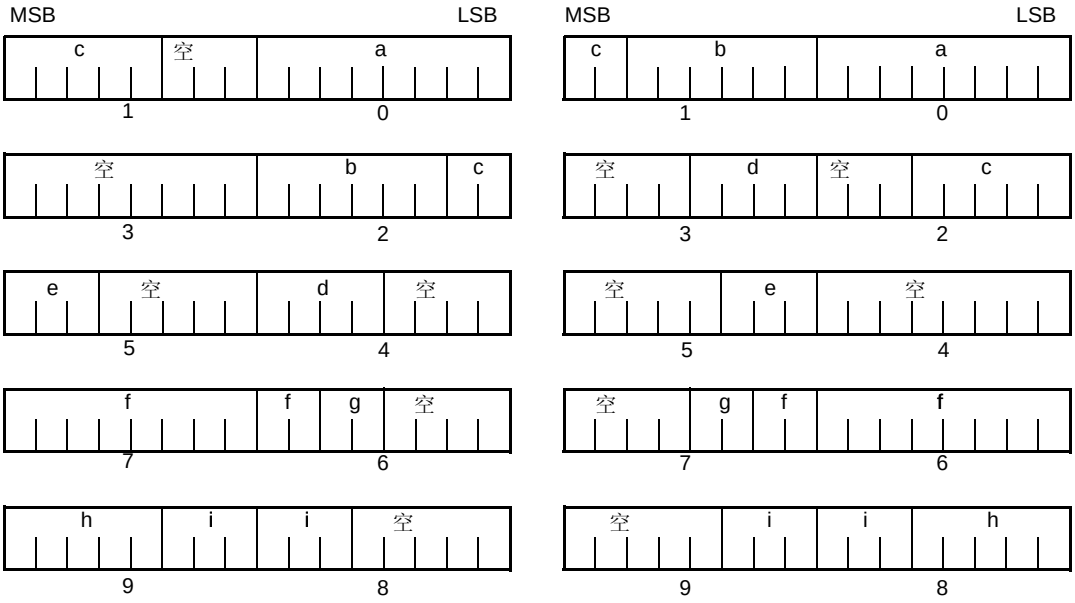
因为 b 和 c 为 unsigned int 型的位域，所以从下一字单元分配。
因为 d 也是 unsigned int 型的位域，所以分配在下一字单元。



因为 e 为 unsigned char 型的位域，所以其分配在下一字节单元。



f 、 g、 h 和 i 分配到单独的字单元。
当指定 -rc 选项时 （把结构体成员整合在一起），以上位域变为下述形式。



备注 分配图下方的数字表示从该结构开始的字节偏移值。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 无需修改源程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果使用了 -rb 选项，且代码和考虑到位域分配序列需配合编码，则必须修改源程序。

改变编译器输出区段名称 (#pragma section ...)

改变编译器输出区段 名称并指定起始地址。

[功能]

- 改变编译器输出区段名称并指定起始地址。
  如果省略起始地址，则按照默认办法分配。 有关编译器输出区段名称和默认位置的详细信息，请参阅 "3.6　段名称列表 "。
  此外，这些区段的位置可以通过省略起始地址，在链接时使用链接指令文件指定。 关于链接指令的详细信息，请参考 "5.1.1　链接指令 "。
- 要以指定的 AT 起始地址改变区段名称 @@CALT， callt 函数必须在源文件中的另一函数之前或之后说明。
- 如果在 #pragma 指令之后进行数据说明，则该数据位于数据改变后的区段中。
  可以再次改变指令，以便如果在重新改变指令之后进行数据说明，则该数据处于重新改变的区段中。
  如果在改变之前定义的数据又在改变之后被重定义，则其位于重新改变区段中。
  此外，该方法对于静态变量 （函数内）也同样有效。

[效果]

- 在一个文件中重复改变编译器输出区段，可以使每一块的位置相互独立，以使得数据可以分配至所希望的数据单元中。

[使用方法]

- 通过使用如下所示的 #pragma 指令去指定要改变的区段名称、新的区段名称和该区段的起始地址。
  在 C 源代码开始处说明此 #pragma 指令。
  以下各项可以在此 #pragma 指令之前进行描述。
  - 注释语句
  - 预处理指令既不能定义也不能引用变量或函数

  但是，在 BSEG 和 DSEG 中的所有区段，以及 CSEG 中的 @@CNST 和 @@CNSTL 区段均可在 C 源代码的任意位置说明，且可以重复执行重新改变指令。 要返回到初始区段名称，在改变的区段中说明编译器输出区段名称。
  在文件开始处如下声明：

```
#pragma section compiler-output-section-name    new-section-name    [ AT startaddress ]
```

- 在 #pragma 之后说明的关键字，确保以大写字母表示编译器输出的区段名称。
  section 和 AT 可以用大写或小写字母或其组合来表示。
- 其中要说明的新区段名称的格式必须符合汇编程序规范 （字段名称最多可以使用八个字母）。
- 仅可以使用 C 语言的十六进制数和汇编程序的十六进制数说明起始地址。

(1) C 语言的十六进制数

```
0xn/0xn ... n
0Xn/0Xn ... n
(n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

(2) 汇编程序的十六进制数

```
nH/n ... nH
nh/n ... nh
(n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

十六进制数必须以数字开始。
示例：以十六进制数表达值为 255 的数值，需在 F 前加零。 因此其为 0FFH。

- 对于 CSEG 中除 @@CNST 和 @@CNSTL 之外的区段，也就是函数所处的区段，此 #pragma 指令只能在 C 源代码开始处 （C 文本说明之后）说明。如果说明的话，将输出警告并忽略该说明。
- 如果此 #pragma 指令在 C 文本说明之后执行，则创建汇编源文件而不创建目标模块文件。
- 如果此 #pragma 指令在 C 文本说明之后，则包含此 #pragma 指令且没有 C 文本 （包括变量和函数的外部引用声明）的文件将不能被包含。 这将导致错误 （参考 " 编码错误示例 1"）。
- #include 语句不能在文件中出现，因为该文件在执行 C 文本说明之后执行此 #pragma 指令。 如果说明，将会导致错误 （参考 " 编码错误示例 2"）。
- 如果 #include 语句在 C 文本之后，则此 #pragma 指令不能在此语句之后说明。 如果说明，将会导致错误 （参考 " 编码错误示例 3"）。

  但是，如果头文件中包含 C 本体，那么将不会出错。

```
d1.h
        extern int      a;


d2.h
        #define VAR 1


d.c
        #include "d1.h";            // When there is a body of C and it's in #include,
        #include "d2.h"              // #pragma instruction of d.c isn't an error.
        #pragma section @@DATA ??DATA1
```

[限制]

- 用于表示向量表 （如 @@VECT02 等）的段的区段名称不能改变。
- 如果在 AT 起始地址中指定的两个或更多同名区段存在于另一个文件中时，将发生链接错误。
- 为编译器输出区段名 @@DATS、 @@BITS 和 @@INIS 指定地址范围从 FFE20H 至 FFEB3H， @@CALT 为 0x80 至 0xbf， @@CODE 和 @@BASE 为 0x0 至 0xffff， @@CNST 为镜像区域，其他区段为 0x0 至 0xffeff。

[示例]

块名 @@CODEL 变为 CC1，地址 2400H 指定为起始地址。
<C 源代码 >

```
#pragma section @@CODEL CC1 AT  2400H

void main ( void ) {
        ;Function body
}
```

< 编译器输出的目标代码 >

```
CC1     CSEG    AT      2400H
_main :
        ; Preprocessing
        ;Function body
        ; Postprocessing
        ret
```

在下列代码示例中， C 主代码之后接 #pragma 指令。
内容分配在 ?/

(1) 示例 1

```
#pragma section @@DATA          ??DATA
int             a1 ;                    // ??DATA
sreg int        b1 ;                    // @@DATS
int             c1 = 1 ;                // @@INIT and @@R_INIT
const int       d1 = 1 ;                // @@CNST
#pragma section @@DATS          ??DATS
int             a2 ;                    // ??DATA
sreg int        b2 ;                    // ??DATS
int             c2 = 1 ;                // @@INIT and @@R_INIT
const int       d2 = 1 ;                // @@CNST
#pragma section @@DATA          ??DATA2
// ??DATA is automatically closed and ??DATA2 becomes valid
int             a3 ;                    // ??DATA2
sreg int        b3 ;                    // ??DATS
int             c3 = 3 ;                // @@INIT and @@R_INIT
const int       d3 = 3 ;                // @@CNST
#pragma section @@DATA          @@DATA
// ??DATA2 is closed and processing returns to the default @@DATA
#pragma section @@INIT          ??INIT
#pragma section @@R_INIT        ??R_INIT

int             a4 ;                    // @@DATA
sreg int        b4 ;                    // ??DATS
// ROMization is invalidated unless both names (@@INIT and @@R_INIT) are changed.  This
// is the user's responsibility.
```

```
int             c4 = 1 ;                // ??INIT and ??R_INIT
const int       d4 = 1 ;                // @@CNST
#pragma section @@INIT          @@INIT
#pragma section @@R_INIT        @@R_INIT
// ??INIT and ??R_INIT are closed and processing returns to the default setting
#pragma section @@BITS          ??BITS
__boolean e4 ;                          // ??BITS
#pragma section @@CNST          ??CNST
char            *const p = "Hello" ;    // p and "Hello" are both ??CNSTT
```

(2) 示例 2

```
#pragma section     @@DATA      ??DATA1
int             a1 ;                    // ??DATA
sreg int        b1 ;                    // @@DATS
int             c1 = 1 ;                // @@INIT and @@R_INIT
const int       d1 = 1 ;                // @@CNST
#pragma section @@DATS          ??DATS
int             a2 ;                    // ??DATA
sreg int        b2 ;                    // ??DATS
int             c2 = 1 ;                // @@INIT and @@R_INIT
const int       d2 = 1 ;                // @@CNST
#pragma section @@DATA          ??DATA2
// ??DATA is automatically closed and ??DATA2 becomes valid
int             a3 ;                    // ??DATA2
sreg int        b3 ;                    // ??DATS
int             c3 = 3 ;                // @@INIT and @@R_INIT
const int       d3 = 3 ;                // @@CNST
#pragma section     @@DATA      @DATA
// ??DATA2 is closed and processing returns to the default @@DATA
#pragma section @@INIT          ??INIT
#pragma section @@R_INIT        ??R_INIT
// ROMization is invalidated unless both names (@@INIT and @@R_INIT) are changed.  This
//  is the user's responsibility.
int             a4 ;                    // @@DATA
sreg int        b4 ;                    // ??DATS
int             c4 = 1 ;                // ??INIT and ??R_INIT
const int       d4 = 1 ;                // @@CNST
#pragma section @@INIT          @@INIT
#pragma section @@R_INIT        @@R_INIT
// ??INIT and ??R_INIT are closed and processing returns to the default setting
#pragma section @@BITS          ??BITS
__boolean e4 ;                          // ??BITS
#pragma section @@CNST          ??CNST
char * const p = "Hello" ;              // p and "Hello" are both ??CNSTT
--
#pragma section     @@INIT      ??INIT1
#pragma section     @@R_INIT    ??R_INIT1
```

```
#pragma section     @@DATA      ??DATA1
char        c1 ;
int         i2 ;
#pragma section     @@INIT      ??INIT2
#pragma section     @@R_INIT    ??R_INIT2
#pragma section @@DATA          ??DATA2
char        c1 ;
int         i2 = 1 ;
#pragma section     @@DATA      ??DATA3
#pragma section     @@INIT      ??INIT3
#pragma section     @@R_INIT    ??R_INIT3
extern char c1 ;                        // ??DATA3
int         i2 ;                        // ??INIT3 and ??R_INIT3
#pragma section     @@DATA      ??DATA4
#pragma section     @@INIT      ??INIT4
#pragma section     @@R_INIT    ??R_INIT4
```

在 C 主代码之后指定 #pragma 指令 , 所受限制在下列编码错误示例中说明。

(3) 编码错误示例 1

```
a1.h
        #pragma section @@DATA  ??DATA1         // File containing only the #pragma
                                                //  section

a2.h
        extern int      func1( void );s
        #pragma section @@DATA  ??DATA2         // File containing the main C code
                                                // followed by the #pragma directive.

a3.h
        #pragma section @@DATA  ??DATA3         // File containing only the #pragma
                                                //  section

a4.h
        #pragma section     @@DATA      ??DATA3
        extern int      func2 ( void );         // File that includes the main C code.

a.c
        #include "a1.h"
        #include "a2.h"
        #include "a3.h"         // <- Error
                                // Because the a2.h file contains the main C code
                                // followed by this #pragma directive, file a3.h, which
                                // includes only this #pragma directive, cannot be
                                // included.
        #include "a4.h"
```

(4) 编码错误示例 2

```
b1.h
        const int i ;

b2.h
        const int j ;

        #include "b1.h"          // This does not result in an error since it is not
                                 // file (b.c) in which the main C code is followed by
                                 // this #pragma directive.

b.c
        const int       k ;
        #pragma section     @@DATA      ??DATA1
        #include "b2.h"          // <- Error
                                // Since an #include statement cannot be coded afterward
                                // in file (b.c) in which the main C code is followed by
                                 // this #pragma directive.
```

(5) 编码错误示例 3

```
c1.h
        extern int       j ;
        #pragma section @@DATA  ??DATA1 // This does not result in an error since the
                                        //  #pragma directive is included and
                                        // processed before the processing of c3.h.

c2.h
        extern int       k ;
        #pragma section @@DATA  ??DATA2 // <- Error
                                        // This #include statement is specified after
                                        // the main C code in c3.h, and the #pragma
                                        // directive cannot be specified afterward.

c3.h
        #include "c1.h"
        extern int       i ;
        #include "c2.h"
        #pragma section @@DATA  ??DATA3 // <- Error
                                        // This #include statement is specified after
                                        // the main C code, and the #pragma directive
                                        //  cannot be specified afterward.
```

```
c.c
        #include "c3.h"
        #pragma section @@DATA  ??DATA4 // <- Error
                                        // This #include statement is specified after
                                        // the main C code in c3.h, and the #pragma
                                        // directive cannot be specified afterward.
        int     i ;
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不支持区段名称改变函数，则无需修改源程序。
- 要改变区段名称，请根据上述使用方法中描述的程序修改源程序。

(1) 从 78K0R C 编译器导入到其他 C 编译器

- 以 #ifdef 删除或隔离 #pragma section...。
- 要改变区段名称，请根据每个编译器的具体规范修改程序。

[注意事项]

- 区段 （section）等同于汇编程序中的段 （assembler）。
- 编译器不会检查新区段名是否与其他符号重名。 因此，用户必须通过汇编输出汇编列表自行检查区段名是否重复。
- 当指定了 -zf 选项，每个区段名都将改变，因此第二个掮
- 如果通过 #pragma 区段改变了与 ROM 化相关的区段名[注]，必须由用户自行更改启动例程。

注 ROM 化相关的区段名

```
@@R_INIT, @@R_INIS, @@RLINIT, @@INITL, @@INIT, @@INIS
```

这里是一些改变启动例程 （cstart.asm 或 cstartn.asm）和终止例程 （rom.asm）的示例，与改变 ROM 化相关区段名相联系。

<C 源代码 >

```
#pragma section @@R_INIT        RTT1
#pragma section @@INIT          TT1
```

如果通过说明以上所示的 #pragma section 已经改变存储具有初值的外部变量的区段名称，则用户必须在启动例程中添加要存储到新区段的外部变量的初始化处理程序。

因此，如下所示，将以下两项添加到启动例程：新区段开始的第一个标号声明和复制初值的部分，并将结束标号的声明添加到终止例程。

RTT1_S 和 RTT1_E 为区段 RTT1 的开始和结束标号名称， TT1_S 和 TT1_E 为区段 TT1 的开始和结束标号名称。

(1) 改变启动例程 cstartx.asm

(a) 添加标号的声明，表示已改变名称区段的结束

```
  :
#pragma section @@R_INIT        RTT1
#pragma section @@INIT          TT1
EXTRN   RTT1_E, TT1_E          ; Adds EXTRN declaration of RTT1_E and TT1_E
  :
```

(b) ĂĚş”Š€÷µĄ”“—Ⅱ Đ°µ▯ RTT1«ŻąŚłĄ÷ýµľTT1«ŻąŚµ▯ ŠĂ–ÚĄŽ¬Î°Ł

```
  :
LDATS1 :
        MOVW    AX, HL
        CMPW    AX, #LOW _?DATS
        BZ      $LDATS2
        MOV     [HL + 0], #0
        INCW    HL
        BR      $LDATS1
LDATS2 :
        MOV     ES, #HIGH RTT1_S
        MOV     HL, #LOWW RTT1_S
        MOV     DE, #LOWW TT1_S
LTT1 :
        MOVW    AX, HL
        CMPW    AX, #LOWW TT1_E
        BZ      $LTT2
        MOV     A, ES : [HL]
        MOV     [DE], A
        INCW    HL
        INCW    DE
        BR      $LTT1
LTT2 :
;
        CALL    !!_main                 ; main ( );
        CLRW    AX
        CALL    !!_exit                 ; exit ( 0 );
        BR      $$
;
```

添加区段，将初值从 RTT1 区段复制到 TT1 区段

(c) 设置已改变名称区段的起始标号。

```
  :
@@R_INIT    CSEG    UNIT64KP
_@R_INIT :
@@R_INIS        CSEG    UNIT64KP
_@R_INIS :
@@INIT          DSEG
_@INIT :
@@DATA          DSEG
_@DATA :
@@INIS          DSEG    SADDRP
_@INIS :
@@DATS          DSEG    SADDRP
_@DATS :

RTT1            CSEG    UNIT64KP        ; 指示 TT1 区段的开始
RTT1_S :                                ; 添加标签设置
TT1             DSEG    BASEP           ; 指示 TT1 区段的开始
TT1_S :                                 ; 添加标签设置

@@CODEL CSEG
@@CALT      CSEG    CALLT0
@@CNST      CSEG    MIRRORP
@@BITS      BSEG
;
                END
```

(2) 更改终止模块 rom.asm

注意事项　不要改变目标模块名 "@rom" 和 "@rome"。

(a) 添加标号的声明，表示已改变名称区段的结束

```
NAME    @rom
;
PUBLIC          _?R_INIT, _?R_INIS
PUBLIC          _?INIT, _?DATA, _?INIS, _?DATS

PUBLIC          RTT1_E, TT1_E          ; Adds RTT1_E and TT1_E

;
@@R_INIT    CSEG    UNIT64KP
_?R_INIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG
_?INIT :
@@DATA          DSEG
_?DATA :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS
        :
```

(b) 设置表示结束的标号

```
  :
RTT1    CSEG    UNIT64KP        ; Adds the label setting indicating the end of the
                                ; RTT1 section.
RTT1_E :                        ; Adds the label setting

RTT1    CSEG    UNIT64KP        ; Adds the label setting indicating the end of the
                                ; RTT1 section.
TT1_E :                         ; Adds the label setting

;
        END
```

二进制常量 (0bxxx)

编译器支持 0bxxx 表示法，用于表达 C 源代码中的二进制常量。

[功能]

- 在可以描述整数常量的位置描述二进制常量

[效果]

- 常量可以用二进制位字符串说明，而不用八进制或十六进制数替换。同时还提高了可读性。

[使用方法]

- 说明 C 源代码中的二进制常量。
  以下展示二进制常量的描述方法。

```
0b      binary-number
0B      binary-number
```

备注 二进制数 :"0" 或 "1".

- 二进制常量以 0b 或 0B 开始，且随后是数 0 或 1 的列表。
- 二进制常量的值以 2 为底进行计算。
- 二进制常量的类型为下表中表示的第一个值。

下标二进制数：	int, unsigned int, long int, unsigned long int
下标 u 或 U：	unsigned int, unsigned long int
下标 l 或 L：	long int, unsigned long int
下标 u 或 U 和下标 l 或 L：	unsigned long int

[示例]

<C 源代码 >

```
unsigned        i ;

i = 0b11100101 ;
```

编译器的输出对象与下列语句的效果相同。

```
unsigned        i ;

i = 0xE5 ;
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 无需修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果编译器支持二进制常量，则需要修改以满足各编译器的具体规范。
- 如果编译器不支持二进制常量，则需要修改为其他整数格式，诸如八进制、十进制和或十六进制。

模块名称改变函数 (#pragma name)

目标代码模块名在 C 源代码中的名称可以任意改变。

[功能]

- 将指定的模块名称的前 254 个字母输出到目标模块文件的符号信息表。
- 当指定了 -g2 选项，将指定的模块名称的前 254 个字母作为符号信息 （MOD_NAM）输出到汇编表文件，且当指定 -ng 选项时，作为 NAME 准伪指令将指定的模块名称的前 254 个字母输出到汇编表文件。
- 如果模块名称指定了 255 个或更多字母，则输出警告消息。
- 如果描述中出现未经认可的字母，则出现错误且处理程序出现异常。
- 如果存在一个以上的 #pragma 指令，则之后描述的任何指令都会被启用，且输出警告消息。

[效果]

- 对象的模块名称可以改为任何名称。

[使用方法]

- 以下显示说明方法。

```
#pragma name    module-name
```

模块名称必须由 OS 授权为文件名称的字符组成，除？（?
区分大写和小写字母。

[示例]

```
#pragma name    module1
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不支持模块名称改变函数，则无需修改。
- 要改变模块名称，请根据以上使用方法中描述的过程对源程序进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma name ... 或由由 #ifdef 将其屏蔽。
- 要改变模块名称，请根据每个编译器的具体规范修改程序。

循环移位函数 (#pragma rot)

输出对表达值移位的代码到目标代码中，通过直接内联展开。

[功能]

- 把表达式值进行移位的代码输出到目标中，通过直接内联展开而不是函数调用，并生成对象文件。
- 如果没有对应的 #pragma 指令，则循环移位函数被视为普通函数。

[效果]

- 即使不进行移位过程，循环移位函数还可以通过 C 源代码或 ASM 说明语句来实现。

[使用方法]

- 在源文件中描述的方法和函数调用的格式相同。
  具有以下四种函数名称。

```
rorb, rolb, rorw, rolw
```

(1) unsigned char rorb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
将 *x* 向右移位 *y* 次。

(2) unsigned char rorb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
将 *x* 向左移位 *y* 次。

(3) unsigned int rorw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
将 *x* 向右移位 *y* 次。

(4) unsigned int rolw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
将 *x* 向左移位 *y* 次。

- 通过模块的 #pragma rot 指令声明循环移位函数的使用方法。
  然而，以下各项可以在 #pragma rot 之前说明。
  - 注释
  - 其它 #pragma 指令
  - 不产生定义 / 引用的变量以及不产生定义 / 引用的函数的预处理指令
- #pragma 后的关键字可以用大写或小写字母描述。

[示例]

<C 源代码 >

```
#pragma rot
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned char   c ;

void main ( void ) {
        c = rorb ( a, b );
}
```

< 输出汇编源程序 >

```
        mov     x, !_b
        mov     a, !_a
L0003 :
        ror     a, 1
        dec     x
        bnz     $L0003
```

[限制]

- 循环移位函数名称不能用作函数名称。
- 循环移位函数名称必须用小写字母表示。如果循环移位函数用大写字母表示，则其作为普通函数进行处理。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不使用循环移位函数，则无需修改。
- 要改为循环移位函数，请根据以上使用方法描述的过程对源程序进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma rot 语句或由 #ifdef 屏蔽。
- 要使用循环移位函数，请根据每个编译器的具体规范 （#asm， #endasm 或 asm(); 等）修改程序。

乘法函数 (#pragma mul)

输出表达值与以直接内联目标相乘的代码。

[功能]

- 输出表达式值相乘对象的代码输出到目标中，通过直接内联展开而不是函数调用，并生成对象文件。
- 如果没有对应的 #pragma 指令，则乘法函数被视为普通函数。

[效果]

- 生成代码会自动对应乘法指令输入输出的数据类型的大小。因此，生成的代码比普通乘法表达式的具有更快的执行速度和更小的代码量。

[使用方法]

- 描述格式与源程序中的函数调用相同。
  以下显示了乘法函数列表。

```
mulu
```

(1) unsigned int mulu (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
进行 *x* 和 *y* 的无符号乘法。

- 通过模块的 #pragma mul 指令声明乘法函数的使用方法。
  然而，以下各项可以在 #pragma mul 之前说明。
  - 注释
  - 其它 #pragma 指令
  - 既未定义又未引用变量或函数的预处理指令
- #pragma 后的关键字可以用大写或小写字母描述。

[限制]

- 用于乘法的函数不可作为函数名 （当声明 #pragma 时）。
- 用于乘法的函数必须用小写字母描述。如果用大写字母表示，则将其按普通函数进行处理。

[示例]

<C 源代码 >

```
#pragma mul

unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned int    i ;

void main ( void ) {
        i = mulu ( a, b );
}
```

< 编译器输出的目标代码 >

```
mov     x, !_b
mov     a, !_a
mulu    x
movw    !_i, ax
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不使用乘法函数，则无需修改。
- 要改为乘法函数，请根据以上使用方法描述的过程对源程序进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma mul 语句或由 #ifdef 屏蔽。乘法函数名称不能用作函数名称。
- 要使用乘法函数，请根据每个编译器的具体规范 （#asm， #endasm 或 asm(); 等）修改程序。

除法函数 (#pragma div)

输出从对象除以表达式值的代码。

[功能]

- 输出从对象除以表达式值的代码。
- 如果没有对应的 #pragma 指令，则除法函数被视为普通函数。

[效果]

- 生成的代码与 78K0 C 编译器兼容，且会自动对应除法指令 I/O 的数据类型大小。因此，生成的代码比普通除法表达式具有更快的执行速度和更小的尺寸。

[使用方法]

- 描述格式与源程序中的函数调用相同。
  具有以下两种除法函数名称。

```
divuw, moduw
```

(1) unsigned int divuw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
进行 *x* 和 *y* 的无符号除法并返回商。

(2) unsigned char moduw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
进行 *x* 和 *y* 的无符号除法并返回余数。

- 通过模块化的 #pragma div 指令声明除法函数的使用方法。
  然而，以下各项可以在 #pragma div 之前说明。
  - 注释
  - 其它 #pragma 指令
  - 不产生定义 / 引用的变量以及不产生定义 / 引用的函数的预处理指令
- #pragma 后的关键字可以用大写或小写字母描述。

[限制]

- 不内联展开除法函数，而通过库调用。
- 除法函数名称不能用作函数名称。
- 除法函数名称必须用小写字母表示。如果用大写字母表示，则将其按普通函数进行处理。

[示例]

<C 源代码 >

```
#pragma div

unsigned int    a = 0x1234 ;
unsigned char   b = 0x12 ;
unsigned char   c ;
unsigned int    i ;

void main ( void ) {
        i = divuw ( a, b );
        c = moduw ( a, b );
}
```

< 编译器输出的目标代码 >

```
mov     c, !_b
movw    ax, !_a
call    !@@divuw
movw    !_i, ax
mov     c, !_b
movw    ax, !_a
call    !@@divuw
mov     a，c
mov     !_c，a
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- 如果编译器不使用除法函数，则无需修改。
- 要改为除法函数，请根据以上使用方法描述的过程对源程序进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma div 语句或由 #ifdef 屏蔽。除法函数名称可用作函数名称。
- 要使用除法函数，请根据每个编译器的具体规范 （#asm， #endasm 或 asm(); 等）修改程序。

BCD 运算函数 (#pragma bcd)

出对象中的表达式值执行 BCD 操作的代码，通过直接内联展开。

[功能]

- 输出对象中的表达式值执行 BCD 操作的代码，通过直接内联展开而不是函数调用，并生成对象文件。
  但是，bcdtob, btobcde, bcdtow, wtobcd 和 btobcd 函数不通过内联。
- 如果没有对应的 #pragma 指令，则 BCD 运算函数被视为普通函数。

[效果]

- 即使不描述 BCD 运算过程， BCD 运算函数也可通过 C 源程序或 ASM 语句实现。

[使用方法]

- 与源程序中的函数调用代码的格式相同。
  如下所示，有 13 种类型的 BCD 运算函数名称。

(1) unsigned char moduw (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
由 BCD 调整指令实现十进制加法。

(2) unsigned char sbbcdb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
由 BCD 调整指令实现十进制减法。

(3) unsigned int adbcdbe (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
由 BCD 调整指令实现十进制加法 （带结果扩展）。

(4) unsigned int sbbcdbe (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
由 BCD 调整指令实现十进制减法 （带结果扩展）。
如发生借位，则更高的位设为 0x99。

(5) unsigned int adbcdw (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
由 BCD 调整指令实现十进制加法。

(6) unsigned int sbbcdw (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
由 BCD 调整指令实现十进制减法。

(7) unsigned long adbcdwe (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
由 BCD 调整指令实现十进制加法 （带结果扩展）。

(8) unsigned long sbbcdwe (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
由 BCD 调整指令实现十进制减法 （带结果扩展）。
如发生借位，则更高的位设为 0x9999。

(9) unsigned char bcdtob (*x*) ;
unsigned char *x* ;
十进制数值转换为二进制数值。

(10) unsigned int btobcde (*x*) ;
unsigned char *x* ;
二进制数值转换为十进制数值。

(11) unsigned int bcdtow (*x*) ;
unsigned int *x* ;
十进制数值转换为二进制数值。

(12) unsigned int wtobcd (*x*) ;
unsigned int *x* ;
十进制数值转换为二进制数值。
但是，如果 x 的值超出 10000 的话，将返回 0xffff。

(13) unsigned char btobcd (*x*) ;
unsigned char *x* ;
十进制数值转换为二进制数值。
但是，忽略溢出。

- 使用由模块中的 #pragma bcd 指令声明的 BCD 运算函数。但是，下列项目可在 #pragma bcd 之前编码。
  - 注释
  - 其它 #pragma 指令
  - 既未定义又未引用变量或函数的预处理指令
- #pragma 后可用大小写字母对关键字进行描述。

[限制]

- BCD 运算函数名称不可作为函数名称。
- BCD 运算函数以小写字母进行编码。如果使用大写字母，这些函数将作为普通函数处理。

[示例]

<C 源代码 >

```
#pragma  bcd

unsigned char   a = 0x12 ;
unsigned char   b = 0x34 ;
unsigned char   c ;

void main ( void ) {
        c = adbcdb ( a, b );
        c = sbbcdb ( b, a );
}
```

< 编译器输出的目标代码 >

```
mov     a, !_a
add     a, !_b
add     a, !BCDADJ
mov     !_c, a
mov     a, !_b
sub     a, !_a
sub     a, !BCDADJ
mov     !_c, a
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用 BCD 运算函数，则无需修改。
- 要将另一函数改为 BCD 运算函数，请根据以上使用方法对源程序进行修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma bcd 声明或以 #ifdef 将其隔离。 BCD 运算函数名称可以用作函数名称。
- 要将 #pragma bcd 用作 BCD 运算函数，请根据每个编译器的具体规范修改程序 （#asm， #endasm 或 asm(); 等）。

数据插入函数 (#pragma opc)

插入常数数据到当前地址中。

[功能]

- 插入常数数据到当前地址中。
- 当没有对应的 #pragma 指令时，数据插入函数被视为普通函数。

[效果]

- 专用数据和指令可以不使用 ASM 语句嵌入代码区。
  当 ASM 使用时，没有汇编器就不能获得目标代码。但是，如果使用数据插入函数，则目标代码可以在没有汇编器的情况下获得。

[使用方法]

- 在源文件中使用大写字母描述，其描述方法与函数调用的格式相同。
- 数据插入的函数名称为 __OPC。

(1) void __OPC (unsigned char *x* , ...) ;

将参数中描述的常量值插入到当前地址。
参数只可以使用常量。

- 数据插入的函数使用 #pragma opc 指令声明。
  然而，以下各项可以在 #pragma opc 之前说明。
  - 注释
  - 其它 #pragma 指令
  - 不产生定义 / 引用的变量以及不产生定义 / 引用的函数的预处理指令
- #pragma 后的关键字可以用大写或小写字母描述。

[限制]

- 数据插入函数名称不能用作函数名称 （当指定 #opc 时）。
- __OPC 必须以大写字母表示。如果用小写字母表示，则将其按普通函数进行处理。

[示例]

<C 源代码 >

```
#pragma opc

void main ( void ) {
        __OPC ( 0xA7 ) ;
        __OPC ( 0x51, 0x12 ) ;
        __OPC ( 0x30, 0x34, 0x12 ) ;
}
```

< 编译器输出的目标代码 >

```
_main :
; line  4 : __OPC ( 0xA7 ) ;
        DB      0AFH
; line  5 : __OPC ( 0x51, 0x12 ) ;
        DB      051H
        DB      012H
; line  6 : __OPC ( 0x30, 0x34, 0x12 ) ;
        DB      030H
        DB      034H
        DB      012H
; line  7 : }
        ret
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不使用数据插入函数，则无需修改。
- 要改为数据插入函数，使用上面的 [使用方法] 进行程序修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 通过 #ifdef 屏蔽或删除 #pragma opc 语句。数据插入函数名称可以用作函数名称。
- 要用作数据插入函数，必须根据 C 编译器的规范来改变源程序 (#asm, #endasm 或 asm( ) ; , 等)。

RTOS 中断处理 (#pragma rtos_interrupt ...)

可以描述 RX78K0R 中断处理。

[功能]

- 以 #pragma rtos_interrupt 指令指定的函数名作为 78K0RRTOS RX78K0R 的中断处理函数。
- 注册描述函数名的地址到中断向量表作为中断请求名。
- RTOS 中断处理产生的代码按如下的次序。

(1) 用 call !!addr20 指令来调用内核符号 __kernel_int_entry。

(2) 保存由编译器使用的地址区域

(3) 保证局部变量区域（当存在局部变量时）

(4) 函数体

(5) 释放局部变量区域（当存在局部变量时）

(6) 恢复由编译器使用的地址区域

(7) 使用 br !!addr20 指令无条件跳转到 label _ret_int

[效果]

- RTOS 中断处理可以描述在 C 源代码级中。
- 由于中断请求名已经定义，所以向量表的地址不需要再定义。

[使用方法]

- 中断请求名 函数名由 #pragma 指令指定。

```
#pragmaΔrtos_interrupt[Δinterrupt-request-nameΔfunction-name]
```

- 此 #pragma 指令描述在 C 源程序的开始处。
- 以下各项可以在 #pragma 指令之前进行描述。
  - 注释
  - 不产生定义 / 引用的变量以及不产生定义 / 引用的函数的预处理指令
- 在 #pragma 之后描述的关键字，中断请求名必须以大写字母描述。其它的关键字可以用大写或小写字母描述。

[限制]

- 当不指定 -zf 选项时，不考虑存储器模式，把 RTOS 中断处理分配在 C0H 和 0FFFFH 之间的区域 。
  当指定 -zf 选项时，中断函数遵照存储器模式分配。另外，也可按 __near 或 __far 指定的分配区域来分配。
- 中断请求名以大写字母描述。
- 软件中断和不可屏蔽中断不可以为其指定中断请求名，如果指定将发生错误。
- 在一个模块单元中，中断请求仅做双倍检查。
- RTOS 中断处理不能指定 callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf。
  只有当指定 -zf 选项时，才能指定 __far。
- ret_int/_kernel_int_entry 不能用作为函数名。

[示例]

<C 源代码 >

```
#pragma rtos_interrupt  INTP0   intp


int     i ;


void intp ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        func ();
}
```

< 编译器输出的目标代码 >

```
@@BASE          CSEG    BASE
_intp :
        call    !!__kernel_int_entry
        movw    ax, _@RTARG0    ;  保存由编译器使用的 saddr 区域
        push    ax              ;
        movw    ax, _@RTARG2    ;
        push    ax              ;
        movw    ax, _@RTARG4    ;
        push    ax              ;
        movw    ax, _@RTARG6    ;
        push    ax              ;
        movw    ax, _@SEGAX     ;
        push    ax              ;
        movw    ax, _@SEGDE     ;
        push    ax              ;
        subw    sp, #06H        ; 保护局部变量区域
        movw    hl, sp
;  line 5 :     int     a[3] ;
; line 6 :      a[0] = 1;
        onew    ax
        movw    [hl], ax        ;  a
```

```
; line 7 :       func ();
         call    !!_func
; line 8 :      }
        addw    sp, #06H        ; 释放局部变量区域
        pop     ax              ; 恢复由编译器使用的 saddr 区域
        movw    _@SEGDE, ax     ;
        pop     ax              ,
        movw    _@SEGAX, ax     ;
        pop     ax              ,
        movw    _@RTARG6, ax    ;
        pop     ax              ,
        movw    _@RTARG4, ax    ;
        pop     ax              ;
        movw    _@RTARG2, ax    ;
        pop     ax              ;
        movw    _@RTARG0, ax    ;
        br      !!_ret_int

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- 如果编译器不支持 RTOS 中断处理 则不需要修改源程序。
- 要改为 RTOS 中断处理，使用上面的 【使用方法】进行修改程序。

(2) 从 78K0RC 编译器导入到其他 C 编译器 >

- 删除 #pragma rtos_interrupt 声明，可用作普通函数。
- 要用作 ROTS 中断处理，必须遵照 C 编译器的规范来修改源程序。

RTOS (__rtos_interrupt) 中断处理修饰符

对于 RX78K0R 的矢量设置和中断处理的描述在分开的文件中说明。

[功能]

- 用 with the __rtos_interrupt 修饰符描述的函数视为 RTOS 中断处理。 RTOS 中断处理使用的寄存器详细情况和 saddr 的保存及恢复参考 "RTOS 中断处理 (#pragma rtos_interrupt ...)"。

[效果]

- 对于 RTOS 的向量表设置和中断处理函数的定义能在不同的文件中描述。

[使用方法]

- __rtos_interrupt 修饰符添加到 RTOS 中断处理函数上。

```
__rtos_interrupt        void func ( )        {processing}
```

[限制]

- 当不指定 -zf 选项时，不考虑存储器模式，把 RTOS 中断处理分配在 C0H 和 0FFFFH 之间的区域。当指定 -zf 选项时，中断函数遵照存储器模式分配。另外，也可按 __near 或 __far 指定的分配区域来分配。
- RTOS 中断处理不能指定 callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf。
  只有当指定 -zf 选项时，才能指定 __far。
- ret_int/__kernel_int_entry 不能用作为函数名。

[注意事项]

- 仅用此修饰符不能设置向量地址。
  向量地址的设置必须用另外的 #pragma 指令或相应的汇编语句来进行。
- 当 RTOS 中断处理定义在与 #pragma rtos_interrupt 文件相同文件中时，即使此修饰符没有描述 ，以 #pragma rtos_interrupt 指定的函数名也判断为 RTOS 中断处理函数。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不支持 RTOS 中断处理 则不需要修改源程序。
- 要改为 RTOS 中断处理函数，使用上面的 【使用方法】进行修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 通过 #define 可以改变 (详细情况，参考 "3.3.5　C 源代码修改 ")。
  通过这些改变， RTOS 中断处理修饰符能处理为普通函数。
- 要使用 RTOS 中断处理，需要根据每种编译器的规范修改程序。

RTOS 任务函数 (#pragma rtos_task)

对于 RX78K0R 以 #pragma rtos_task 指定的函数视为 RTOS 任务函数。

[功能]

- 以 #pragma rtos_task 指定的函数名视为 RTOS 任务函数。
- 如果指定了函数名称，但是在同一文件中没有发现实体定义，则将发生错误。
- RTOS 任务函数的预处理不为帧指针 / 寄存器变量保存寄存器。后处理不输出。
- RTOS 系统调用 ext_tsk 是始终调用在 #pragma rtos_task 的最后。
- 下列 RTOS 系统调用函数可用。

  void ext_tsk (void) ;

  Calls RTOS system call ext_tsk。

  然而， ext_tsk 在 ext_tsk 实体定义，中断函数， RTOS 中断处理中调用时，将发生错误。
- RTOS 调用 ext_tsk 是使用 br ！！ addr20 指令调用。如果 ext_tsk 在函数的最后出现，后处理不输出。
- 任务函数可以不带参数指定或仅用不超过 4 字节的参数指定来编码，但是不能指定返回数值。
  如果指定两个或两个以上参数，指定 5 个字节或 5 个字节以及指定返回数值，将发生错误。

[效果]

- RTOS 中断任务可以描述在 C 源代码级中。
- 帧指针 / 寄存器变量相关寄存器保存和后处理不输出，所以代码效率得到提升。

[使用方法]

- 对下面的 #pragma 指令指定函数名。

```
#pragmaΔrtos_task[Δtask-function-name]
```

- #pragma 指令描述在 C 源程序的开始处。
  然而，以下各项可以在 #pragma 指令之前进行描述。
  - 注释
  - 不产生定义 / 引用的变量以及不产生定义 / 引用的函数的预处理指令
- #pragma 之后的关键字可以用大写或小写字母描述。

[限制]

- RTOS 的任务函数不能指定 callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf。
  只有当指定 -zf 选项时，才能指定 __far。
- 不能使用调用普通函数相同的方法调用 RTOS 的任务函数。
- RTOS 系统调用的函数名 ext_tsk 不能用作函数名。
- #pragma rtos_task 没有写到 C 源程序， ext_tsk 不解释为系统的 RTOS 调用。因此，即使 ,ext_tsk 从 RTOS 中断处理中调用 ext_tsk，也不输出下面的错误。

  E0778：在中断函数中不能调用 ext_tsk

  解决方法：
  - 通过 #pragma rtos_task 清除任务函数的使用。
  - 不从 RTOS 中断处理中调用 ext_tsk。

[示例]

<C 源代码 >

```
#pragma rtos_task       func
#pragma rtos_task       func2

void func ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( ) ;
}

void func2 ( int x ) {
        int     a[3] ;
        a[0] = 1 ;
}

void func3 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( ) ;
}

void func4 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        if ( a[0] ) {
                ext_tsk ( ) ;
        }
}
```

< 编译器输出的目标代码 >

```
@@CODEL CSEG
_func :
        subw    sp, #06H        ; 保存帧指针
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ;  a
        br      !!_ext_tsk      ; 通过写 ext_tsk 调用 ext_tsk
        br      !!_ext_tsk      ; 通过任务函数 ext_tsk 的调用始终输出
                                ; 不输出结尾
_func2 :
        push    ax              ; 不保存帧指针
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ;  a
        br      !!_ext_tsk      ; 通过任务函数 ext_tsk 的调用始终输出
                                ; 不输出结尾
_func3 :
        push    hl              ;  不保存帧指针
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ;  a
        br      !!_ext_tsk      ; 如果 ext_tsk 在中间被调用，输出结尾
                                ; 函数
_func4 :
        push    hl              ;  不保存帧指针
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ;
        clrw    bc
        cmpw    ax, bc
        skz
        br      !!_ext_tsk      ;  如果 ext_tsk 在中间被调用，输出结尾
        addw    sp, #06H        ;
        pop     hl
        ret
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果编译器不支持 RTOS 任务函数 则不需要修改源程序。
- 要改为 RTOS 任务函数，使用上面的 【使用方法】修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果删除 #pragma rtos_task 声明， RTOS 任务函数就用作为普通函数。
- 要用作 ROTS 任务函数，必须遵照 C 编译器的规范来修改源程序。

闪存区域分配方式 (-zf)

用 -zf 选项允许在闪存区域定位程序。没有指定 -zf 选项，允许使用函数链接引导区的目标。

注意事项　此函数允许闪存重写器件函数。

[功能]

- 产生一个目标文件定位在闪存中。
- 不能从引导区域引用在闪存区域中的外部变量。
- 能从闪存区域中引用引导区域中的外部变量。
- 在引导区域程序中和闪存区域程序中不能定义相同的外部变量和相同的全局函数

[效果]

- 可以在闪存中定位程序。
- 没有指定 -zf 选项，允许使用函数链接引导区的目标。

[使用方法]

- 在编译期间指定 -zf 选项。

[限制]

- 对闪存区域使用启动例程或库。

闪存区域跳转表和闪存区域分配 (#pragma ext_table)

#pragma 指令指定闪存跳转表的起始地址。启动例程和中断函数可以定位在闪存区域中，并且可以从引导区域调用函数到闪存区域。

注意事项　此函数允许闪存重写器件函数。

[功能]

- -zf 选项决定了启动例程，中断函数或从引导区调用函数到闪存区域的跳转表首地址。
- 跳转表的首地址开始的 64 个地址专门用于中断函数（包含启动例程），它们占据 4 字节区域。
- 普通函数的跳转表一般分配在 " 跳转表首地址 +4 * 64" 之后。跳转表的每一项占据 4 字节区域。关于 ext_func ID 数值的更多信息参阅 " 从引导区到闪存区的函数调用 #pragma ext_func)"。
- -zz 选项决定了跳转表的起始地址。
- 当仅指定 -zt 选项时， -zz 选项被认为具有相同值。
- 当仅指定 -zz 选项时， -zt 选项被认为具有相同值。

[效果]

- 启动例程和中断函数可以定位在闪存区域中。
- 函数可以从引导区域调用到闪存区域。

[使用方法]

- 使用 -zt 选项，按如下方式来指定闪存跳转区域的起始地址。

```
-ztxxxxxh : xxxxx = 0c0h to 0edfffh注
```

- 使用 -zz 选项，按如下方式来指定闪存跳转区域的起始地址。

```
-zzxxxxxh : xxxxx = 0c0h to 0edfffh注
```

注　不同器件的地址不同。

[限制]

- 可以指定为闪存跳转表起始地址的地址范围是 0C0H 至 0EDFFFH。 (然而，根据目标器件 0EDFFFH 可能变化。)
- 当源程序包含 #pragma ext_func 时，以及当指定 -zf 选项并且程序包含 #pragma vect, #pragma interrupt 或 #pragma rtos_interrupt 指令时，这两者之一必须指定 -zf 选项。如果未指定 -zz 或 -zt 选项，将发生错误。
- 2000H 是默认的中断服务程序库向量表的起始地址 (_@vect00 至 _@vect7e)。默认的在中断向量库中跳转表的起始地址是 2000H。
- 链接器 -zb 选项也指定了闪存跳转表的起始地址。链接器 -zb 选项和闪存区域的起始地址必须指定为相同的地址 。如果地址不一致，则会产生错误。
- 如果分配闪存跳转表的地址小于闪存跳转表的起始地址，就会发生错误。
- 如果要建立的程序定位在引导区域或闪存区域中， -zt 或 -zz 选项必须用来指定分配闪存区域的地址和闪存分支表。
- 当模块用不同的 -zt 或 -zz 选项编译链接时，就发生错误。

- 当引导区域或闪存区域中的 ROM 数据不能定位在 near 区域时， ROM 数据指针强制处理为 far 指针。 (参见下面的 [注]。) 因此，在 small 和 medium 模式中，当调用取 (const *) 参数的标准库函数时， suffix "_f" 必须加在库函数名之后 （警告 W0072 始终输出）。

  下面的标准库函数取 (const *) 参数。

  sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/stltoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

[兼容性]

(1) 从其它 C 编译器导入到 78K0RC 编译器

- 要指定闪存区域跳转表的首地址，遵照上面 [使用方法] 改变此地址。

(2) 从 78K0R C 编译器导入到其他 C 编译

- 要指定闪存区域跳转表的首地址，需要进行下述更改。

[注意事项]

- 闪存跳转表的起始地址和镜像区域的起始地址影响 near/far 规范的处理。如果 near/far 区域规范不同于实际存储器分布，在命令行分析时警告 W0070 和 W007 仅出现一次。
- 当闪存跳转表的起始地址在镜像区域，在 64KB 内：near/far 区域规范如下没有更改。（参阅 " 图 3-3.　存储器映射示例 1"。）
- 当闪存跳转表的起始地址在镜像区域，不在 64KB 内：闪存区域函数定位在 far 区域。（参阅 " 图 3-4.　存储器映射示例 2"。）
- 当闪存跳转表的起始地址在镜像区域的结束地址之上，在 64KB 内：闪存区域的 ROM 数据定位在 far 区域。（参阅 " 图 3-5.　存储器映射示例 3"。）
- 当闪存跳转表的起始地址在镜像区域的结束地址之上，不在 64KB 内：闪存区域函数定位在 far 区域，闪存 ROM 数据区域也定位在 far 区域。（参阅 " 图 3-6.　存储器映射示例 4"。）
- 当闪存跳转表的起始地址在镜像区域的起始地址之下，在 64KB 内：引导区域的 ROM 数据定位在 far 区域。（参阅 " 图 3-7.　存储器映射示例 5"。）
- 当闪存跳转表的起始地址在镜像区域的起始地址之下，不在 64KB 内：引导区域 ROM 数据定位在 far 区域，闪存区域 ROM 数据也定位在 far 区域。（参阅 " 图 3-8.　存储器映射示例 6"。）
- 当引导区域或闪存区域 ROM 数据不能定位在 near 区域：

- 当引导区域或闪存区域 ROM 数据不能置于 near 区域时或当闪存区域函数不能置于 near 区域时：施加以下的约束。

表 3-13.　当没有引导镜像区域时， ROM 数据的处理

区域	定义	外部声明	由指针指向目标
引导区	始终 far	始终 far	始终 far
闪存区	near 或 far	始终 far	始终 far

表 3-14. 当没有闪存镜像区域时， ROM 数据的处理

区域	定义	外部声明	由指针指向目标
引导区	near 或 far	near 或 far	始终 far
闪存区	始终 far	始终 far	始终 far

表 3-15. 当闪存区域的开始区不在 64KB 内时，函数处理

区域	定义	外部声明	由指针指向目标
引导区	near 或 far	near 或 far 注	始终 far
闪存区	始终 far	始终 far	始终 far

注 函数由 #pragma ext_func 指定驻在闪存中，所以它们始终是 far。

图 3-3. 存储器映射示例 1

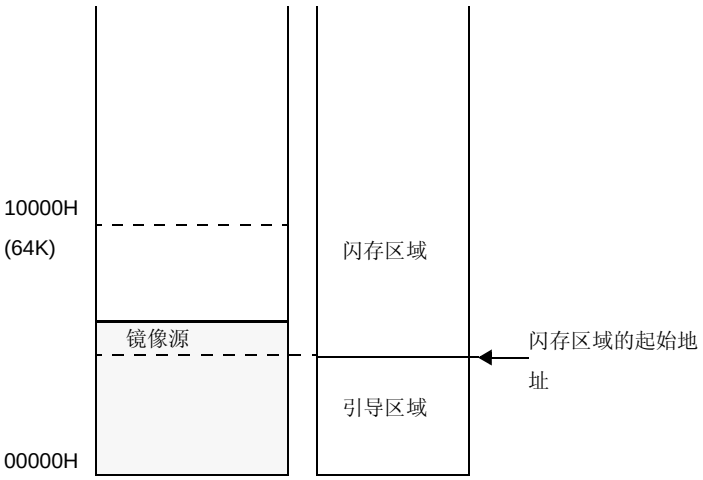


图 3-4. 存储器映射示例 2

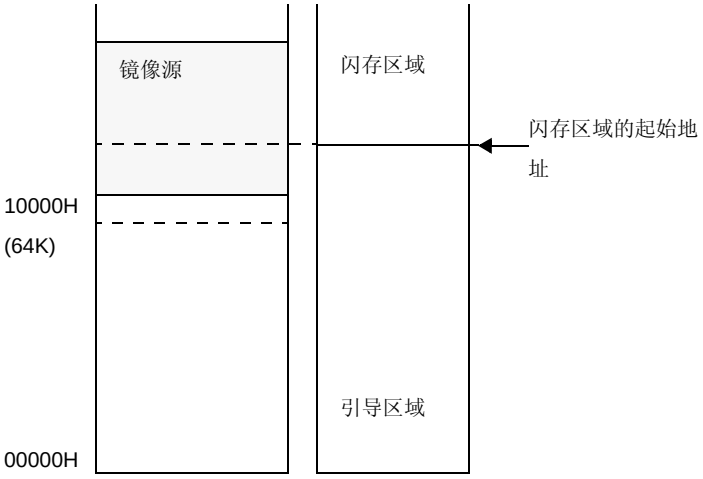


图 3-5. 存储器映射示例 3

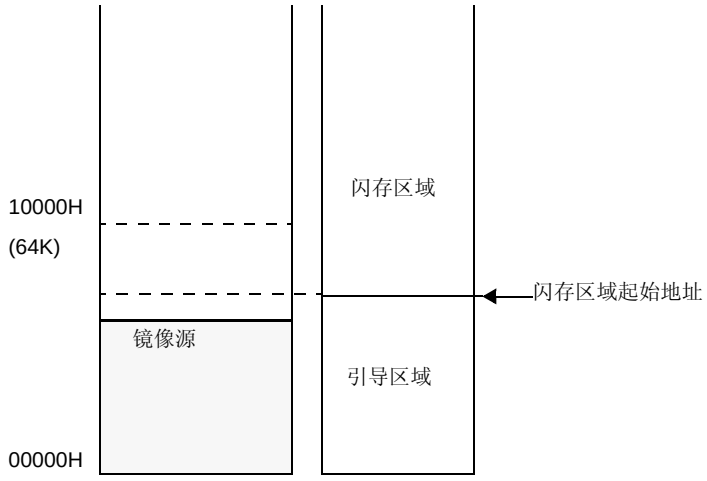


图 3-6. 存储器映射示例 4

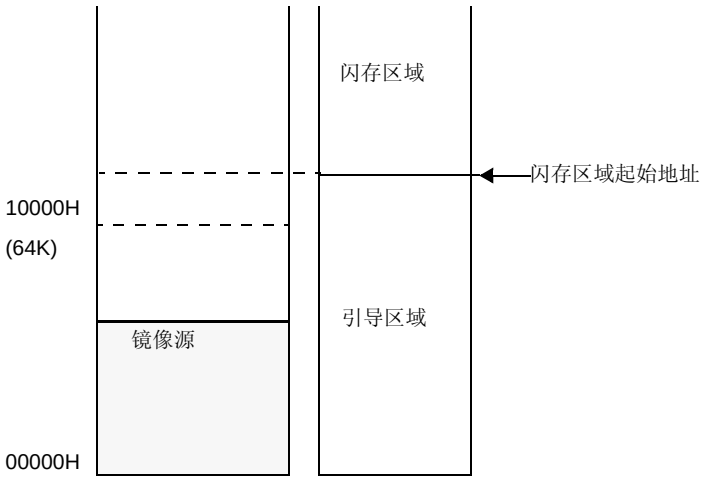


图 3-7. 存储器映射示例 5

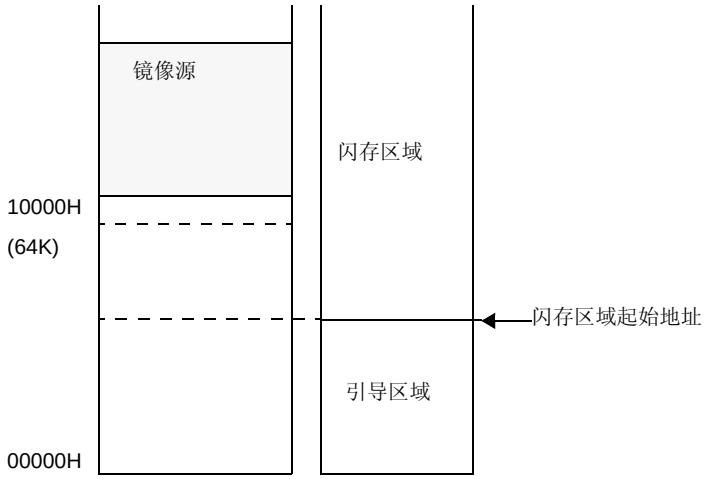
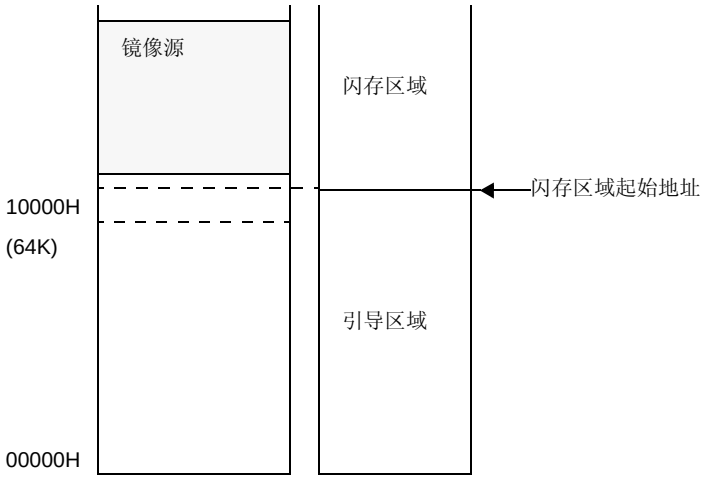


图 3-8. 存储器映射示例 6



[示例]

要在地址 2000H 以后产生跳转表和放置中断函数：

<C 源代码 >

```
#pragma interrupt       INTP0   intp

void intp ( void ) {
}
```

(1) 放置中断函数到引导区域 (无 -zf 指定 , 指定 -zt2000h)

< 编译器输出的目标代码 >

```
                PUBLIC  _intp
                PUBLIC  _@vect06
@@BASE          CSEG    BASE
_intp :
                reti

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

在中断向量表中设置中断函数的首地址。

(2) 放置中断函数到闪存区域 (无 -zf 指定，指定 -zt2000h)

< 编译器输出的目标代码 >

```
                PUBLIC  _intp
@ECODE          CSEG    BASE
_intp :
                reti

@EVECT06        CSEG    AT      0200CH
                br      !!_intp
```

在跳转表中设置中断函数的首地址。
跳转表的地址值是 2000H + 4 * (0006H / 2) 因为跳转表的首地址是 200CH 和中断向量地址 (2 字节) 是 0006H。
中断向量库设置地址 200CH 在中断向量表中。

< 中断向量 06 的库 >

```
                PUBLIC  _@vect06


@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      200CH
```

从引导区到闪存区的函数调用 #pragma ext_func)

#pragma 指令指定函数名和从引导区域调用到闪存区域中的 ID 数值。可以从引导区域调用函数到闪存区域中。

注意事项　此函数允许闪存重写器件函数。

[功能]

- 从引导区域调用函数到闪存区域是经过闪存跳转表来执行的。
- 在引导区域中的函数从闪存区域可以直接调用。

[效果]

- 可以从引导区域调用函数到闪存区域中。

[使用方法]

- #pragma 指令指定函数名和从引导区域调用到闪存区域中的 ID 数值。

```
#pragma ext_func        函数名  ID 数值
```

- 此 #pragma 指令描述在 C 源程序的开始处。
- 以下各项可以在此 #pragma 指令之前进行描述。
  - 注释
  - 既未定义又未引用变量或函数的的预处理指令。

[限制]

- ID 数值设置在 0 至 255 (0xFF)。
- 如果没有指定 -zt 选项或 -zz 选项，文件包含 #pragma ext_func 编译时，就发生错误。
- 相同的函数带有不同 ID 值，或不同的函数有相同的 ID 值，编译时就发生错误。
  下面 (1) 和 (2) 是错误的。

(1)
```
#pragma ext_func f1 3
#pragma ext_func f1 4
```

(2)
```
#pragma ext_func f1 3
#pragma ext_func f2 3
```

- 如果从引导区域调用函数到闪存区域，在闪存区域中没有对应的函数定义，链接器无法执行检查。这属于用户的责任。
- callt 函数只能定位在引导区域。如果调用函数定义在闪存区域 (当指定 -zf 选项时)，产生一个错误。

- 当为 small 或 medium 模式指定 -rf 选项时以及为 large 模式指定 -rn 选项时，当调用取 (const *) 参数的标准库函数时 suffix "_f" 必须加到库函数名中 （警告 W0072 始终输出）。
  下面的标准库参数取 (const *) 参数。
  sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/stltoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

[注意事项]

- 当在 small 或 medium 模式中指定 -r 选项时以及对 large 模式指定 -rn 选项时，程序终止以符合 ANSI 标准。如果指定严格 ANSI 选项 -za 的话，将产生警告 W0073。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果 #pragma ext_func 不使用，就不需要修改。
- 要完成从引导区域调用函数到闪存区域，按照上面的 [使用方法] 修改程序。

(2) 从 78K0R C 编译器导入到其他 C 编译

- 删除 #pragma ext_func 指令或用 #ifdef 来隔离。
- 要完成从引导区域调用函数到闪存区域，程序需要下面的修改。

[示例]

- 地址在 2000H 之后生成跳转表，从引导区域调用闪存区域的函数 f1 和 f2。

<C 源代码 >

```
(1) 引导区域端

#pragma interrupt INTP0 intf0
#pragma ext_func f1 3
#pragma ext_func f2 4

void f1 ( ), f2 ( );

void func ( ) {
        f1 ( );
        f2 ( );
}
```

```
(2) 闪存区域端

#pragma interrupt INTP1 intf1
#pragma ext_func f1 3
#pragma ext_func f2 4

void f1 ( ) {
}

void f2 ( ) {
}

void intf1 ( ) {
}
```

备注 **1.** #pragma ext_func f1 3 表示函数 f1 跳转目的地定位在跳转表的开始地址是 + 4 * 64 + 4 * 3。
2. #pragma ext_func f2 4 表示函数 f2 的跳转目的地定位在跳转表的开始地址是 + 4 * 64 + 4 * 4。
3. 4 * 跳转表开始的 64 字节专用于描述中断函数 （包含启动例程）。

< 编译器输出的目标代码 >

(1) 当分配跳转表地址在 64KB 之内时

```
(1) 引导区域端（没有指定 -zf 选项和指定 -zt2000h 选项）

@@CODEL         CSEG
_func :
        call    !0210CH
        call    !02110H
        ret
@@VECT08        CSEG    AT    0008H
_@vect08:
                DW      _intf0
```

```
(2) 闪存区域端 (specified -zf)

@ECODEL         CSEG
_f1:
        ret
_f2:
        ret
_intf1:
        reti

@EVECT0A        CSEG    AT    02014H
        br      !!_intf1

@EXT03          CSEG    AT    0210CH
        br      !!_f1
        br      !!_f2
```

```
(3)0A 的中断向量库

@@VECT0A        CSEG    AT    000AH
_@vect0a:
                DW      2014H
```

(2) 当分配跳转表地址不在 64KB 之内时
当跳转表起始地址是 1300H 时

```
(1) 引导区域端 ( 没有指定 -zf 选项和指定 -zt2000h 选项 )

@@CODEL         CSEG
_func :
        call    !!01310CH
        call    !!013110H
        ret

@@VECT08        CSEG    AT    0008H
_@vect08:
         DW      _intf0
```

(2) 闪存区域端（指定 -zf 选项）

```
@ECODEL          CSEG
_f1:
        ret
_f2:
        ret
_intf1:
        reti

@EVECT0A         CSEG    AT    013014H
        br       !!_intf1

@EXT03           CSEG    AT    01310CH
        br       !!_f1
        br       !!_f2
```

(3)0A 的中断向量库

```
@@BASE      CSEG    BASE
?@vect0a:       br      !!013014H

@@VECT0A         CSEG    AT    000AH
_@vect0a:
                 DW      ?@vect0a
```

镜像源程序区说明

-mi0/-mi 选项命令编译器为指定的镜像源程序区域产生代码。

[功能]

- 当指定 -mi0 选项时， MAA 中的 0 代码产生。
- 当指定 -mi1 选项时， MAA 中的 1 代码产生。
- 当模块已经用不同的 -mi0/-mi1 选项指定时，发生链接错误。
- 当指定 -mi 选项时， MAA 中的 0 代码产生。
- 通过默认链接器的 -mi 选项来设置编译 -mi 选项的数值。
- 除非指定，否则 -mi 选项设置为 0。
- 如果连接器 -mi 选项的数值与编译器 -mi 选项的数值不同时，会发生链接错误。
- 对于镜像区域和 MAA 位的更多信息，参阅目标器件的用户手册。

[效果]

- 编译器产生指定镜的像源程序区域代码。

[使用方法]

- 在编译时，指定 -mi0 或 -mi1 选项。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- -mi 选项可以指定选择镜像源程序区域。源程序不需要修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 源程序无需修改就可以在其它 C 编译器上编译。

参数 / 返回值的 int 展开限制方法 (-zb)

在编译期间指定 -zb 选项，可以减少目标代码和提高执行速度。

[功能]

- 当函数返回值的类型定义为 char/unsigned char 时，不生成返回值的 int 展开代码。
- 当定义了函数参数的原型且原型的参数定义为 char/unsigned char 时，不生成参数的 int 展开代码。

[效果]

- 因为不生成 int 展开代码，可以减少目标代码并提高执行速度。

[使用方法]

- 在编译期间指定 -zb 选项 。

[示例]

<C 源代码 >

```
unsigned char   func1 ( unsigned char x, unsigned char y );
unsigned char   c, d, e;

void main ( void ) {
        c = func1 ( d, e );
        c = func2 ( d, e );
}

unsigned char   func1 ( unsigned char x, unsigned char y ) {
        return x + y ;
}
```

(1) 当指定 -rb 选项时

< 编译器输出的目标代码 >

```
_main :
; line  5 :              c = func1 ( d, e );
        mov     x, !_e
        push    ax
        mov     x,!_d                   ; 不进行 int 扩展
        call    !_func1
        pop     ax
        mov     a, c
        mov     !_c, a
; line  6 :             c = func2 ( d, e );
        mov     x, !_e
        clrb    a                       ; 因为没有原形声明，执行 int 扩展
                                        ; 原型声明
        push    ax
        mov     x, !_d
        mov     x, #00H
        xch     a, x                        ; 因为没有，所以执行 int 扩展
                                        ; 原型声明
        call    !_func2
        pop     ax
        mov     a, c
        mov     !_c, a

;  line          7 : }
        ret
; line 8 :
; line 9 :      unsigned char func1 ( unsigned char x, unsigned char y ) {
_func1 :
        push    hl
        push    ax
        movw    ax, sp
        movw    hl, ax
        mov     a, [hl]
        mov     x, a
        mov     a, [hl + 6]
        movw    hl, ax
; line  10 :            return x + y ;
        mov     a, l
        add     a, h
        mov     c, a                    ; 不执行 int 扩展
;  line          11 : }
        pop     ax
        pop     hl
        ret
```

[限制]

- 对此函数的函数体定义与原型声明不同，则程序可能错误运行操作。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果对于所有函数体定义的原型声明无法正确运行，则要进行校正原型声明。或者，不指定 -zb 选项。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 无需修改。

存储器运算函数 (#pragma inline)

目标文件由标准库输出函数以直接内联的内存复制和内存设置产生。

[功能]

- 目标文件由标准库存储器运算函数 memcpy 和 memset 输出产生，方法是直接内联而不是函数调用。
- 当没有 #pragma 指令时，生成调用标准库函数的代码。

[效果]

- 与调用标准库函数相比，提高了执行速度。
- 如果指定字符数数量为常量，则可以缩短目标代码。

[使用方法]

- 在源程序中描述的方法与函数调用的格式相同。
- 以下各项可以在此 #pragma 指令之前进行描述。
  - 注释
  - 其它 #pragma 指令
  - 既未定义又未引用变量或函数的预处理指令

[示例]

<C 源代码 >

```
#pragma inline


char    ary1[100], ary2[100] ;


void main ( void ) {
        memset ( ary1, 'A', 50 );
        memcpy ( ary1, ary2, 50 );
}
```

< 编译器输出的目标代码 >

```
_main :
        push    hl
; line  5 :    memset ( ary1, 'A', 50 );
        movw    de, #loww ( _ary1 )
        mov     a, #041H        ; 65
        mov     c, #032H        ; 50
L0003 :
        mov     [de], a
        incw    de
        dec     c
        bnz     $L0003
;  line  6 :     memcpy ( ary1, ary2, 50 );
        movw    de, #loww ( _ary1 )
        movw    hl, #loww ( _ary2 )
        mov     c, #032H        ; 50
L0005 :
        mov     a, [hl]
        mov     [de], a
        incw    de
        incw    hl
        dec     c
        bnz     $L0005
;  line          7 : }
        pop     hl
        ret
```

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用存储器运算函数则不需要修改源程序。
- 当改变存储器运算函数时，请使用上面的方法。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 删除 #pragma inline 指令或使用 #ifdef 来隔离。

绝对地址分配规范 (__directmap)

在模块中 __directmap 声明变量是以绝对地址定义分配的。变量可分配到任意地址。

[功能]

- 由 __directmap 和 static 变量声明的外部变量初值在函数中被认为是分配地址的说明，该变量分配到指定的地址。
  使用整型变量指定分配地址。
- C 源程序代码中的 __directmap 变量按静态变量处理。
- 因为初始值被认为是分配地址规范，所以不能定义初始值，而保留未定义值。
- 可指定的地址规范范围，为保留区域和指定的地址通过模块链接保留区域范围，以及变量双重检查范围显示在下表中。

条目	范围	
	当指定小型模式或 中型模式时	当指定大型模式 时
地址规范范围	0xf0000 - 0xfffff	0x00000 - 0xfffff
保留区域范围	0xffd00 - 0xffeff	0xffd00 - 0xffeff
双重检查范围	器件内部 RAM 的 起始地址 - 结束地址	器件内部 RAM 的 起始地址 - 结束地址

- 如果地址规范在地址规范之外，则发生错误。
- 用 __directmap 声明的变量不能分配到下面区域之外的扩展区域。如果分配，将输出错误。
  - saddr 区域 (0xffe20 至 0xffeff)
  - sfr 区域或 saddr 区域重叠 (0xfff00 至 0xfff1f) 的区域
  - sfr 区域 (0xfff20 至 0xfffff)
  - 第二个 sfr 区域 （取决于所使用器件而不同。）
- 如果通过 __directmap 声明的变量分配地址重复并且在双重检查范围内，则警告信息 （W0762）输出并显示重复变量的地址。
- 如果地址规范范围在 saddr 区域之内，则自动进行 __sreg 声明和产生 saddr 指令。
- 如果 char/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long 类型变量由 __directmap 声明为位引用， sreg/__sregt 必须与 __directmap 一起被指定。如果没有一起指定，将发生错误。
- 如果指定的地址范围在 near 区域，变量被认为在 near 区域用于访问。
- 如果指定的地址范围既不是 saddr 区域也不是 near 区域，变量被认为在 far 区域用于访问。
- 如果没有指定 __near 或 __far 类型修饰符，变量按照存储器模式规范被访问。

- 如果指定了类型修饰符，变量按照规范被访问。如果指定的地址范围和类型修饰符相矛盾，将输出错误。下表列出了地址规范范围，存储器模式和类型修饰符之间的关系。

地址规范范围		类型修饰符					
		__near __sreg	__far __sreg	__sreg	__near	__far	不指定
在 saddr 区域内	访问方式	sreg	sreg	sreg	sreg	sreg	sreg
	指针长度	2 个字节	4 个字节	小型 ：2 个字节 中型 ：2 个字节 大型 ：4 个字节	2 个字节	4 个字节	小型 ：2 个字节 中型 ：2 个字节 大型 ：4 个字节
在 near 区域内	访问方式	错误	错误	错误	near	far	小型 ：near 中型 ：near 大型 ：far
	指针长度				2 个字节	4 个字节	小型 ：2 个字节 中型 ：2 个字节 大型 ：4 个字节
在 far 区域内	访问方式	错误	错误	错误	错误	far	小型 ：错误 中型 ：错误 大型 ：far
	指针长度					4 个字节	小型 ：错误 中型 ：错误 大型 ：4 个字节

[效果]

- 一个或多个变量可以分配于相同的任意地址。

[使用方法]

- 在模块中声明 __directmap，将变量分配在已被定义的绝对地址中。

```
__directmap                  类型名   变量名 = 分配地址规范 ;
__directmap static           类型名   变量名 = 分配地址规范;
__directmap __sreg           类型名   变量名 = 分配地址规范;
__directmap __sreg static    类型名   变量名 = 分配地址规范;
```

- 如果为结构体 / 联合体 / 数组声明 __directmap，在括号 {} 指定地址。

[示例]

<C 源代码 >

```
__directmap     char    c = 0xffe00 ;
__directmap     __sreg  char    d = 0xffe20 ;
__directmap     __sreg  char    e = 0xffe21 ;
__directmap  struct x {
        char a ;
        char b ;
} xx = { 0xffe30 };

void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

< 编译器输出的目标代码 >

```
        PUBLIC  _main
_c      EQU     0FFE00H                 ; 通过 __directmap 为变量声明地址
_d      EQU     0FFE20H                 ; 由 EQU 定义
_e      EQU     0FFE21H                 ;
_xx     EQU     0FFE30H                 ;
        EXTRN   __mmfe00                ; 用于链接保留区域模块
        EXTRN   __mmfe20                ; EXTRN 输出
        EXTRN   __mmfe21                ;
        EXTRN   __mmfe30                ;
        EXTRN   __mmfe31                ;
@@CODEL CSEG
_main :
; line 10 :
        oneb    !loww ( _c )
; line 11 :
        mov     _d, #012H               ; saddr 指令输出因为地址
; line 12 :                             ; specified in saddr area
        set1    _e.5                    ; 因为已使用了 __sreg，所以可做 BIT 操作
; line 13 :
        mov     _xx, #05H               ; saddr 指令输出因为地址
                                        ; 指定在 saddr 区域
; line 14 :
        mov     _xx + 1, #0AH           ; saddr 指令输出因为地址
                                        ; 指定在 saddr 区域
; line 15 :
        ret
        END
```

[限制]

- 不能为函数参数，返回值或自动变量指定 __directmap。如果指定了这些，将发生错误。
- 如果地址指定在保留区域范围之外，变量区域将不作保留，有必要建立描述指令文件或建立用于保留此区域的独立模块。
- __directmap 变量不能用 extern 声明，因为它与静态变量处理方式相同。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 如果不使用关键字 __directmap，不需要修改。
- 要改变为 __directmap 变量，按照上面描述的方法作修改。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 使用 #define 可以实现兼容性 (详细信息参阅 "3.3.5　C 源代码修改)。
- 当 __directmap 被使用为绝对地址分配规范时，按照每种编译器的规格来修改。

near/far 区域规范

当函数或变量声明时，分配函数或变量的地方通过加到 __near 或 __far 类型描述来特别定义。

[功能]

- 函数或变量的位置通过指定 __near 或 __far 类型修饰符特别指定。

修饰符	位置
__near 类型修饰符	near 区域 (数据：0F0000H 至 0FFFFFH，代码：000000H 至 00FFFFH)
__far 类型修饰符	far 区域 (000000H 至 0FFFFFH)

- near 区域的指针是 2 字节长度， far 区域的指针是 4 字节长度。
- 如果在声明相同变量和函数时使用了 __near 和 __far 类型修饰符，将发生错误。
- 在语法上， _near 和 __far 类型修饰符作为类型修饰符处理。
- 如果用 __callt， __interrupt， __rtos_interrupt， __interrupt_brk， __sreg 或 __boolean 一起指定， __near 或 __far 类型修饰符被忽略。
- 如果一起指定 __near 和 __far 类型修饰符，将发生错误。
- 如果指定自动变量，参数或寄存器变量， __near 或 __far 类型修饰符被忽略。
- 在 near 区域中的变量不使用 ES 寄存器就可存取。
  指针长度是 2 字节。
- 在 far 区域中的变量要通过设置 ES 寄存器来存取。
  指针长度是 4 字节。
- 在 near 区域的函数用 !addr16 调用，在 far 区域的函数用 !!addr20 调用。
- 由于没有不引用 CS 寄存器就可调用指针的指令，所以一定要设置 CS 寄存器来调用函数指针。
- 在 near 区域中的函数指针输出代码设置 CS 寄存器为 0.
- far 指针的最高字节总是未定义。
- 从 near 指针或 int 转换到 far 指针以及从 near 指针转换到 long 产生如下操作。
  - "0xf" 加到变量指针的高位字节 (0 是特殊的，产生零扩展)。
  - 函数指针是零扩展。
- far 指针用低位 2 字节加和减，高位字节不改变。
- ptrdiff_t 总是 int 类型。
- far 指针使用低位 3 字节做等式运算。
- far 指针使用低位 2 字节做关系操作。要比较不指向相同目标的指针，指针必须转换到 unsigned long 类型。如果指定 -za 选项，低位 3 字节用于作比较。
- 按照指定的存储器模式，字符串常量分配于 far 或 near 区域。

存储器模式	位置
小型模式	near 区域
中型模式	near 区域
大型模式	far 区域

- 当使用大型模式时，自动变量，参数和 sreg 变量的指针是 4 字节长度。
- 如果同一变量或函数在定义模块中声明为 _near，在另一模块中声明为 _far，或相反情况时，将进行以下错误检查。（参阅 " 编码示例 2"。）
  - 当变量或函数被引用时，如果 1）其已经在定义模块中声明为 __near， 2）在模块中声明为 __far，在被引用的地方发生链接错误。
  - 完成多至 8 个指针，数组或函数声明任意组合的错误检查。
  - 只有指定 -g 选项时才完成检查。

[效果]

- __far 类型修饰符的规范允许函数和变量分配到 far 区域并且允许引用。
- __near 类型修饰符的规范允许函数和变量分配到 near 区域并且允许引用。
  函数和变量分配到 near 区域可以用短指令被调用或引用。

[使用方法]

- 声明的函数或变量时添加 __near 或 __far 类型修饰符。

[示例]

(1) 编码示例 1

```
__near int i1;
__far  int i2;
__far  int * __near p1;
__far  int * __near * __far p2;
__far  int func1();
__far  int * __near func2();
__near int ( * __far fp1 )();
__far  int * __near ( * __near fp2 )();
__near int * __far ( * __near fp3 )();
__near int * __near ( * __far fp4 )();
```

- i1 是 int 类型，分配到 near 区域。
- i2 是 int 类型，分配到 far 区域。
- p1 是 4- 字节类型的变量，该变量指向 " 在 far 区域中的 int 类型 "。变量本身被分配到 near 区域。
- p2 是 2- 字节变量，该变量指向 near 区域中的 4- 字节类型，指向 "far 区域中的 int 类型 "。变量本身被分配到 far 区域。
- func1 是函数，其返回 "int 类型 "。函数本身被分配到 far 区域。
- func2 是返回 4 字节类型的函数，该函数指向 " 在 far 区域中的 int 类型 "。函数本身被分配到 near 区域。
- fp1 是 2 字节类型的变量，该变量指向 " 在 near 区域中的函数，返回 int 类型 "。变量本身被分配到 far 区域。
- fp2 是 2 字节类型的变量，该变量指向在 near 区域中的函数，返回 4 字节类型，指向
- "far 区域的 int 类型 "。变量本身被分配到 near 区域。
- fp3 是 4 字节变量，该变量指向 far 区域中的函数，返回 2 字节类型，指向 "near 区域中的 int 类型 "。变量本身被分配到 near 区域。
- fp4 是 2 字节变量，该变量指向 near 区域中的函数，返回 2 字节类型，指向 "near 区域中的 int 类型 "。变量本身被分配到 far 区域。

(2) 编码示例 2

- 下面的示例解释了，如果同一变量或函数在定义模块中声明为 near，在另一模块中声明为 far，或相反情况时，将进行以下错误检查。

```
(1)a.c


/* Definitions */
int     __near i1;
int     __far  i2;
int     __near * __near nnp1;
int     __near * __near nnp2;
int     __near * __far  fnp1;
int     __near * __near nnp3;
int     __far  * __near nfp1;
int     __far  * __near nffunc1(){}
int     __far  * __near nffunc2(){}
int     __far  * __far  fffunc1(){}
int     __near * __far  fnfunc1(){}
int     __far  * __far  fffunc2(){}
```

```
(2)b.c

/* extern  declarations */
extern int     __far  i1;
extern int     __near i2;
extern int     __near * __near nnp1;
extern int     __near * __far  nnp2;
extern int     __near * __near fnp1;
extern int     __far  * __near nnp3;
extern int     __near * __near nfp1;
extern int     __far  * __near nffunc1();
extern int     __far  * __far  nffunc2();
extern int     __far  * __near fffunc1();
extern int     __far  * __far  fnfunc1();
extern int     __near * __far  fffunc2();

void main ( void ) {
        i1 = 1;           /* OK */
        i2 = 1;           /* Error */
        *nnp1 = 1;        /* OK */
        *nnp2 = 1;        /* OK */
        *fnp1 = 1;        /* Error */
        *nnp3 = 1;        /* Error */
        *nfp1 = 1;        /* Error */
        nffunc1 ( );      /* OK */
        nffunc2 ( );      /* OK */
        fffunc1 ( );      /* Error */
        fnfunc1 ( );      /* Error */
        fffunc2 ( );      /* Error */
}
```

[限制]

- 即使指定了 __far 类型修饰符，数据也不能分配到超出 64KB 界限的扩展区域。
  函数可以分配到超过 64KB 界限的扩展区域。

[兼容性]

(1) 从其它的 C 编译器导入到 78K0R C 编译器

- 如果保留字 __near 或 __far 没有使用，不需要修改此代码。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 如果 __near 或 __far 不使用，不需要修改此代码。
- 如果使用 __near 或 __far 类型修饰符， #define 可以使用于 near/far 区域的规范。

[注意事项]

- 如果使用低位 2 字节作为关系运算，数据不能分配于 64KB 界限的最后字节。如果分配，由链接器或编译器将输出错误。
  因为关系运算是由 ANSI 编译操作完成，[注] 该运算使用了指向数组以外范围的指针。

注　关系运算上的 ANSI 规定限制
如果表达式 P 指向数组目标的元素，表达式 Q 指向该数组目标的最后的元素，指针表达式 Q+1 是大于表达式 P。

- far 区域的指针大小是 4 字节，但是计算对象是低位 3 字节，所以最高位字节字节总是无定义。

< 示例 >

```
union   tag {
        __far unsigned short    *ptr;
        unsigned long           ldata;
} un;
```

数值写入 un.ptr，然后 un.ldata 被引用；最高字节成为无定义。为了保证 un.ldata 的最高字节是 0，联合体必须首先用 0 对 un.ldata 初始化。

- 链接器用下面块类型和重定位属性的组合方式检查块的数据位置。
  DSEG UNIT64KP
  DSEG PAGE64KP
  CSEG PAGE64KP
- 如果使用 #pragma section 或链接指令文件改变上面再定位属性，链接器不检查块的数据位置。

存储器模式规范

编译时，分配函数或变量的地点可以通过 -ms、 -mm 或 -ml 选项按存储器模式来指定。

[功能]

- 指定函数或变量的位置。

存储器模式	数据	功能
小型模式	near 区域	near 区域
中型模式	near 区域	far 区域
大型模式	far 区域	far 区域

- 如果指定 __near 或 __far 类型修饰符，则指定的 __near 或 __far 类型修饰符取得优先。
- 小型模式
  包含 64KB 数据部分和 64KB 代码部分；总共 128KB。
  数据 ROM 分配在 0000H 至 0FFFFH 或 10000H 至 1FFFFH，镜像映射在 FxxxxH。
  代码分配在 00000H 至 0FFFFH。
  由于通过指定 __far 类型修饰符， CS 寄存器值可以改变，当调用函数指针时，需设置 CS 寄存器。
- 中型模式
  变量分配到 near 区域，函数分配到 area 区域。包含 64KB 数据部分和 1MB 代码部分。
  ROM 数据分配在 0000H 至 00FFFFH 或 010000H 至 01FFFFH，镜像映射在 FxxxxH。分配代码没有限制。
- 大型模式
  变量和函数定分配到 far 区域。包含 1MB 数据部分和 1MB 代码部分。分配数据和代码没有限制。

[使用方法]

- 编译时指定 -ms， -mm 或 -ml 选项。

选项	说明
-ms	小型模式
-mm	中型模式
-ml	大型模式

[示例]

<C 源代码 >

```
int     i ;
int     *p ;
void func( void ) { }
void ( *fp )( void );

void main ( void ) {
        int     r;

        r = i ;          /* 数据访问 */
        func();          /* 函数调用 */
        r = *p ;         /* 数据指针 */
        fp();            /* 函数指针 */
}
```

< 编译器输出的目标代码 >

(1) 当使用小型模式时

```
movw    hl, !_i
call    !_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
movw    ax, !_fp
mov     CS, #00H        ; 0
call    ax
```

(2) 当使用中型模式时

```
movw    hl, !_i
call    !!_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
mov     a, !_fp + 2
mov     CS, a
movw    ax, !_fp
call    ax
```

(3)　当使用大型模式时 >

```
mov     ES, #highw ( _i )
movw    hl, ES:!_i
call    !!_func
mov     ES, #highw ( _p )
mov     a, ES:!_p + 2
movw    de, ES:!_p
mov     ES, a
movw    ax, ES:[de]
movw    hl, ax
mov     ES, #highw ( _fp )
mov     a, ES:!_fp + 2
mov     CS, a
movw    ax, ES:!_fp
call    ax
```

[限制]

- 即使指定大型模式，数据也分配到超出 64KB 界限的扩展区域中。
- 指定了不同存储模式的模块不能相互链接。
- 每个装载模块文件分配到 far 区域，有 / 没有初始值的变量的容量各是 (64K - 1) 字节 （注：如果指定 -za 选项是 64KB)。
  此容量可以通过改变块名来增加，使用 " 改变编译器输出区段名称 (#pragma section ...)" 的函数，可以用特定文件中有 / 没有初始值的变量包含到另外的输出块名中。
  在此情况中，启动例程和终止例程必须修改 (参考 " 改变编译器输出区段名称 (#pragma section ...)"[与改变 ROM 化的块名有关的修改启动例程的示例])。
- 每个输出块名的最大容量不改变。
- 如果不指定 -za 选项，数据就不能分配在 64KB 界限的最后字节 (参考 "near/far 区域规范 " 中的注意事项)。

分配 ROM 数据说明

ROM 数据的分配位置可以特别指定在 near 或 far 区域。

[功能]

- -rf 选项把 ROM 数据放置在 far 区域。
- -rn 选项把 ROM 数据放置在 near 区域。
- 当 -rf 和 -rn 选项都不指定， ROM 数据的放置取决于存储器模式。
- ROM 数据的放置按下面从高到低的优先次序的规范决定。

(1) near 或 far 通过闪存区域的起始地址规范和镜像源程序区域的地址来规定。 (参阅 " 闪存区域跳转表和闪存区域分配 (#pragma ext_table)")。

(2) __near 或 __far 关键字

(3) -rn 或 -rf 选项规范

(4) 存储器模式

- ROM 数据涉及下面数据类型。
  - 变量声明为 const
  - 字符串文字
  - auto aggregate 类型变量的初始值 （数组和结构体）
  - 开关语句分支表

[效果]

- 允许分配 ROM 数据进任意的 far 或 near 区域。

[使用方法]

- 在编译时指定 -rf 或 rn.

[限制]

- 当由不同模块指定 const 变量时，按照以下 ROM 数据规范的优先级放置并完成错误检查。关于错误检查，请参阅 “near/far 区域规范 ?。

[兼容性]

(1) 从其它 C 编译器导入到 78K0R C 编译器

- 用指定 -rf 或 -rn 选项通过重编译来指定 ROM 数据的放置。不需要修改源程序。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 在其它 C 编译器上编译不需要修改源程序。

3.3.5 C 源代码修改

当使用扩展函数时，编译器可生成有效的目标代码。但是，这些函数是基于 78K0R 单片机设计的。如果程序使用了这些扩展函数，那么当导入其他器件时必须进行修改。

这部分就说明了把程序从其他 C 编译器导入 78K0R C 编译器的技巧，以及从 78K0R C 编译器导入其他 C 编译器的技巧。

(1) 从其它 C 编译器导入到 78K0R C 编译器

- #pragma 注

  如果其他 C 编译器支持 #pragma 命令，那么必须修改 C 源代码。 C 源代码的修改方式和程度取决于其他 C 编译器的规范。

- 扩展规范

  如果其他 C 编译器使用诸如附加关键字之类的扩展规范，那么必须修改 C 源代码。 C 源代码的修改方式和程度取决于其他 C 编译器的规范。

注 #pragma 为 ANSI 支持的预处理命令之一。编译器会把 #pragma 之后的字符串识别为命令。如果 C 编译器不支持该命令，那么将忽略命令 #pragma，编译器将继续执行直到正常结束。

(2) 从 78K0R C 编译器导入到其他 C 编译器

- 由于 78K0R C 编译器添加了关键字作为扩展功能，必须删除 C 源代码中的关键字或用 #ifdef 命令使其无效才能导入其他 C 编译器。

下列为一些示例。

(a) 使关键字无效 （同样可应用于 callf、 sreg 和 norec）

```
#ifndef __K0R__
#define callt           /* Makes callt as ordinary function */
#endif
```

(b) 从一种类型转换为另一种类型

```
#ifndef __K0R__
#define bit     char    /* Changes bit type to char type variable*/
#endif
```

3.4　函数调用接口

该部分说明了函数调用接口的特性。

- 返回值（所有函数通用）
- 普通函数调用接口

3.4.1　返回值

- 函数的返回值存放于寄存器或进位标志中。
- 返回值的存放位置如下所示。

表 3-16.　返回值的存放位置

类型	存放位置
1 位	CY
1 或 2 个字节的整数	BC
near 指针	BC
4 个字节的整数	BC（较低），DE（较高）
far 指针	BC（较低），DE（较高）
浮点数	BC（较低），DE（较高）
结构	返回的结构复制到函数的存放位置，其地址备份将存放于 BC 和 DE 寄存器内。

3.4.2　普通函数调用接口

(1) 参数传递

- 第二个和之后的参数传递至堆栈上函数。
- 第一个参数通过寄存器或堆栈传递到函数的定义端。

下表显示了第一个参数传递的位置。

表 3-17.　第一个参数的传递位置（函数调用端）

类型	存放位置
1 个字节的数据[注] 2 个字节的数据[注]	AX
near 数据的指针	AX
3 个字节的数据[注] 4 个字节的数据[注]	AX, BC
指向函数的指针， far 数据的指针	AX, BC
浮点数	AX, BC
其他	位于堆栈

注　1 至 4 个字节的数据，包括结构体、联合体和指针。

(2) 参数和自动变量的存放位置

- 参数或自动变量在函数顶部被分配至寄存器，通过寄存器声明参数或自动变量或指定 -qv 选项。其他的参数和自动变量存放于堆栈中。
- 如果通过堆栈传递到函数定义端的参数未分配到寄存器，那么传递位置就是分配位置。
- 除非没有堆栈框架，否则参数和自动变量将分配至寄存器 HL。
  若指定 -qr 选项，参数和自动变量还可分配至 _@KREGxx。关于 _@KREGxx，请参考 ?3.5　saddr 区域标签列表 ?。
- 参数和自动变量以引用频率的顺序分配至寄存器。
  不经常引用的参数和自动变量可能不会被分配至寄存器，即使参数和自动变量是由寄存器声明或指定了 -qv 选项。
- 寄存器通过函数定义端对参数或自动变量进行分配、存储和恢复。

(3) 示例

(a) 示例 1

<C 源代码 >

```
void func0 ( register int, int );

void main ( void ) {
        func0 ( 0x1234, 0x5678 );
}

void func0 ( register int p1, int p2 ) {
        register int    r ;
        int     a;
        r = p2 ;
        a = p1 ;
}
```

<1> - 当指定 -qr 选项时

< 汇编由编译器产生的源代码 >

```
_main :
; line  4 :     func0 ( 0x1234, 0x5678 );
        movw    ax,  #05678H    ; 22136
        push    ax                      ; 2nd and following arguments passed on
                                        ; stack
        movw    ax, #01234H     ; 4660  ; 1st argument passed in register
        call    !!_func0                ; Function call
        pop     ax                      ; Release stack used for function call
; line  5 : }
        ret
; line 6 :
; line  7 : void func0 ( register int p1, int p2 ) {
_func0 :
        push    hl
        movw    de, _@KREG14
        push    de                      ; Save saddr area for register variable
        movw    de, _@KREG12
        push    de                      ; Save saddr area for register variable
        movw    _@KREG14, ax            ; Assign 1st argument p1 to saddr
        push    ax                      ; Reserve storage for auto variable a
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, [hl+12]     ; p2    ; Argument p2
        movw    _@KREG12, ax    ; r     ; Auto variable r
; line 11 :     a = p1 ;
        movw    ax, _@KREG14    ; p1    ; Argument p1
        movw    [hl], ax        ; a     ; Auto variable a
; line 12 : }
        push    ax                      ; Reserve storage for auto variable a
        pop     ax
        movw    _@KREG12, ax            ; Restores saddr area for register
                                        ; argument
        pop     ax
        movw    _@KREG14, ax            ; Restores saddr area for register
                                        ; argument
        pop     hl
        ret
```

(b) Example 2

<C 源代码 >

```
void func1 ( int, register int );

void main ( void ) {
        func1 ( 0x1234, 0x5678 );
}

void func1 ( int p1, register int p2 ) {
        register int    r ;
        int     a;
        r = p2 ;
        a = p1 ;
}
```

<1> 当指定 -qr 选项时

< 汇编由编译器产生的源代码 >

```
_main :
; line  4 :     func1 ( 0x1234, 0x5678 );
        movw    ax,  #05678H    ; 22136
        push    ax                      ; 2nd and following arguments passed on
                                        ; stack
        movw    ax, #01234H     ; 4660  ; 1st argument passed in register
        call    !!_func1                ; Function call
        pop     ax                      ; Release stack used for function call
; line  5 : }
        ret
; line 6 :
; line  7 : void func1 ( int p1, register int p2 ) {

_func0 :
        push    hl
        push    ax                      ; Place 1st argument p1 on stack
        movw    de, _@KREG14
        push    de                      ; Save saddr area for register variable
        movw    de, _@KREG12
        push    de                      ; Save saddr area for register variable
        movw    ax, [sp+12]
        movw    _@KREG12, ax            ; Assign argument p2 to saddr
        push    ax                      ; Reserve storage for auto variable a
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, _@KREG12    ; p2    ; Argument p2
        movw    _@KREG14, ax    ; r     ; Auto variable r
; line 11 :     a = p1 ;
```

```
        movw    ax, [hl+6]      ; p1    ; Argument p1
        movw    [hl], ax        ; a     ; Auto variable a
; line 12 : }
        push    ax                      ; Reserve storage for auto variable a
        pop     ax
        movw    _@KREG12, ax            ; Restores saddr area for register
                                        ; variable
        pop     ax
        movw    _@KREG14, ax            ; Restores saddr area for register
                                        ; variable
        pop     ax                      ; Release storage for 1st argument p1
        pop     hl
        ret
```

3.5 saddr 区域标签列表

78K0R C 编译器使用下列标签对 saddr 区域中的地址进行引用。因此，下表中的名称不能用于 C 和汇编源程序。

(1) 寄存器变量

标签名称	地址
_@KREG00	0FFEB4H
_@KREG01	0FFEB5H
_@KREG02	0FFEB6H
_@KREG03	0FFEB7H
_@KREG04	0FFEB8H
_@KREG05	0FFEB9H
_@KREG06	0FFEBAH
_@KREG07	0FFEBBH
_@KREG08	0FFEBCH
_@KREG09	0FFEBDH
_@KREG10	0FFEBEH
_@KREG11	0FFEBFH
_@KREG12	0FFEC0H 注
_@KREG13	0FFEC1H 注
_@KREG14	0FFEC2H 注
_@KREG15	0FFEC3H 注

注 当寄存器对函数参数进行声明或定义 -qv 和 -qr 选项时，参数将位于 saddr 区域。

(2) 关于工作

标签名称	地址
_@NRARG0	0FFEC4H
_@NRARG1	0FFEC6H
_@NRARG2	0FFEC8H
_@NRARG3	0FFECAH
_@NRAT00	0FFECCH
_@NRAT01	0FFECDH
_@NRAT02	0FFECEH
_@NRAT03	0FFECFH
_@NRAT04	0FFED0H
_@NRAT05	0FFED1H
_@NRAT06	0FFED2H
_@NRAT07	0FFED3H

(3) 关于段信息

标签名称	地址
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

(4) 实时运行库的参数

标签名称	地址
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

3.6 段名称列表

这部分说明了编译器输出的全部段以及其位置。

下表显示了在该部分表格中出现的重定位属性。

- CSEG 重定位属性

CALLT0	对于指定段定位，使其起始地址为 2 的倍数，范围是 80H 至 BFH。
AT 绝对表达式	定位指定段至绝对地址 （在 00000H 至 FFEFFH 范围内）。
UNITP	对于指定段定位，使其起始地址为 2 的倍数，为 C0H 至 EFFFEH 范围内的任意位置。

- DSEG 重定位属性

SADDRP	对于指定段定位，使其起始地址为 2 的倍数，在 FFE20H 至 FFEFFH 范围内的 saddr 区域。
UNITP	对于指定段定位，使其起始地址为 2 的倍数，在 RAM 中默认范围内的任意位置。

3.6.1 段名称列表

(1) 程序区域和数据区域

区段名	段类型	重定位 属性	说明
@@CODE	CSEG	BASE	代码部分段 （位于 near 区域）
@@CODEL	CSEG		代码部分段 （位于 far 区域）
@@LCODE	CSEG	BASE	库代码段 （位于 near 区域）
@@LCODEL	CSEG		库代码段 （位于 far 区域）
@@CNST	CSEG	MIRRORP	ROM 数据[注 1]
@@CNSTL	CSEG	PAGE64KP	ROM 数据[注 1]
@@R_INIT	CSEG	UNIT64KP	near 初始化数据段 （带初始值）
@@RLINIT	CSEG	UNIT64KP	far 初始化数据段 （带初始值）
@@R_INIS	CSEG	UNIT64KP	初始化数据段 （带初始值的 sreg 变量）
@@CALT	CSEG	CALLT0	callt 函数表段
@@VECT*nn*	CSEG	AT 00*mm*H	向量表段[注 2]
@@BASE	CSEG	BASE	callt 函数和中断函数段
@@LBASE	CSEG	BASE	库和 callt 函数段
@@INIT	DSEG	BASEP	数据区域段 （带有初始值，位于 near 区域）
@@INITL	DSEG	UNIT64KP	数据区域段 （带有初始值，位于 far 区域）
@@DATA	DSEG	BASEP	数据区域段 （不带初始值，位于 near 区域）
@@DATAL	DSEG	UNIT64KP	数据区域段 （不带初始值，位于 far 区域）
@@INIS	DSEG	SADDRP	数据区域段 （带初始值的 sreg 变量）
@@DATS	DSEG	SADDRP	数据区域段 （不带初始值的 sreg 变量）

区段名	段类型	重定位 属性	说明
@@BITS	BSEG		布尔运算类型和比特类型变量段

注 1. ROM 数据涉及下面数据类型。
- const 变量段
- switch-case 语句的表引用
- 未知字符串常量
- 自动变量的初始值数据

2. *nn* 和 *mm* 的值根据中断类型而变化。

(2) 闪存存储器区域

区段名	段类型	重定位 属性	说明
@ECODE	CSEG	BASE	代码部分段 （位于 near 区域）
@ECODEL	CSEG		代码部分段 （位于 far 区域）
@LECODE	CSEG	BASE	库代码段 （位于 near 区域）
@LECODEL	CSEG		库代码段 （位于 far 区域）
@ECNST	CSEG	MIRRORP	ROM 数据注 1
@ECNSTL	CSEG	PAGE64KP	ROM 数据注 1
@ER_INIT	CSEG	UNIT64KP	near 初始化数据段 （带初始值）
@ERLINIT	CSEG	UNIT64KP	far 初始化数据段 （带初始值）
@ER_INIS	CSEG	UNIT64KP	初始化数据段 （带初始值的 sreg 变量）
@EVECT*nn*	CSEG	AT *mmmm*H	向量表段注 2
@EXT*xx*	CSEG	AT *yyyy*H	闪存区域跳转表段注 3
@EINIT	DSEG	BASEP	数据区域段 （带有初始值，位于 near 区域）
@EINITL	DSEG	UNIT64KP	数据区域段 （带有初始值，位于 far 区域）
@EDATA	DSEG	BASEP	数据区域段 （不带初始值，位于 near 区域）
@EDATAL	DSEG	UNIT64KP	数据区域段 （不带初始值，位于 far 区域）
@EINIS	DSEG	SADDRP	数据区域段 （带初始值的 sreg 变量）
@EDATS	DSEG	SADDRP	数据区域段 （不带初始值的 sreg 变量）
@EBITS	BSEG		布尔运算类型和比特类型变量段
@ECALT	CSEG		虚拟段

注 1. ROM 数据涉及下面数据类型。
- const 变量段
- switch-case 语句的表引用
- 未知字符串常量
- 自动变量的初始值数据

2. *nn* 和 *mmmm* 的值根据中断类型而变化。
3. *xx* 和 *yyyy* 根据闪存区域函数的 ID 而变化。

3.6.2　段的定位

段类型	位置（默认）
CSEG	ROM
BSEG	RAM 的 saddr 区域
DSEG	RAM

3.6.3　C 源代码示例

```
#pragma INTERRUPT       INTP0   inter   rb1     /* Interrupt vector*/

void main ( void );                             /* Function prototype declaration */
const   int     i_cnst = 1 ;                    /* const variable*/
callt   void f_clt ( void );                    /* callt function prototype declaration*/
boolean b_bit ;                                 /* boolean type variable*/
long    l_init = 2 ;                            /* Initialized external variable*/
int     i_data ;                                /* Uninitialized external variable*/
s
__sreg  int     sr_inis = 3 ;                   /* Iinitialized sreg variables*/
__sreg  int     sr_dats ;                       /* Uninitialized sreg variable*/

void main ( void ) {                            /* Function definition*/
        int     i ;
        i = 100 ;
}

void inter ( void ) {                           /* Interrupt function definition*/
}

callt   void f_clt ( void ) {                   /* callt function definition*/
}
```

3.6.4　输出汇编器模块的指令

汇编语言源程序中的命令和指令集的输出根据目标器件而不同。
详情，请参阅 “第 4 章　汇编语言规范”。

(1) Small 模型

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -ms -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
```

```
$KANJICODE SJIS
$TOL_INF    03FH, 0100H, 00H, 00H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; Segment for const variable
_i_cnst :   DW      01H             ; 1

@@R_INIT    CSEG    UNIT64KP            ; Segment for initialization data (Initialized
                                        ; external variable)
            DW      00002H, 00000H  ; 2

@@INIT      DSEG    BASEP               ; Segment for tentative data (with initial value)
_l_init :   DS      ( 4 )

@@DATA      DSEG    BASEP               ; Segment for tentative data (without initial
                                        ; value)
_i_data :   DS      ( 2 )

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3

@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP              ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function */
; line 2 :
```

```
; line  3 : void main ( void );              /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;     /* const variable */
; line  5 : callt   void f_clt ( void );     /* calltFunction prototype declaration */
; line  6 : boolean b_bit ;                  /* boolean type variable */
; line  7 : long    l_init = 2 ;             /* Initialized external variable */
; line  8 : int     i_data ;                 /* Uninitialized external variable */
; line  9 : sreg    int     sr_inis = 3 ;    /* Iinitialized sreg variable */
; line 10 : sreg    int     sr_dats ;        /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                  /* Function definition */

@@CODE       CSEG  BASE                 ; Segment for code portion
_main :
        push    hl
; line 13 :         int     i ;
; line 14 :         i = 100 ;
        movw    hl, #064H          ; 100
; line 15 : }
        pop     hl
        ret
; line 16 :
; line 17 : void inter ( ) {                 /* Interrupt function definition */

@@BASE          CSEG    BASE            ; Segment for callt and interrupt function
_inter :
; line 18 : }
         reti
; line 19 :
; line 20 : callt   void f_clt ( ) {        /* callt function definition */
_f_clt:
; line 21 : }
         ret
@@VECT06        CSEG    AT      0006H   ; Segment for vector table
_@vect06 :
                DW      _inter
                END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

(2) 中型模式

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -mm -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE SJIS
$TOL_INF    03FH, 0100H, 00H, 04000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; Segment for const variable
_i_cnst :   DW      01H                 ; 1

@@R_INIT    CSEG    UNIT64KP            ; Segment for initialization data (Initialized
                                        ; external variable)
            DW      00002H, 00000H  ; 2

@@INIT      DSEG    BASEP              ; Segment for tentative data (with initial value)
_l_init :   DS      ( 4 )

@@DATA      DSEG    BASEP               ; Segment for tentative data (without initial
value)
_i_data :   DS      ( 2 )

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3
```

```
@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP              ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function */
; line 2 :
; line  3 : void main ( void );             /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;    /* const variable */
; line  5 : callt   void f_clt ( void );     /* calltFunction prototype declaration */
; line  6 : boolean b_bit ;                  /* boolean type variable */
; line  7 : long    l_init = 2 ;             /* Initialized external variable */
; line  8 : int     i_data ;                 /* Uninitialized external variable */
; line  9 : __sreg  int     sr_inis = 3 ;    /* Iinitialized sreg variable */
; line 10 : __sreg  int     sr_dats ;        /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                  /* Function definition */

@@CODEL     CSEG                        ; Segment for code portion
_main :
         push    hl
; line 13 :          int     i ;
; line 14 :          i = 100 ;
         movw    hl, #064H           ; 100
; line 15 : }
         pop     hl
         ret
; line 16 :
; line 17 : void inter ( ) {                 /* Interrupt function definition */

@@BASE   CSEG    BASE                   ; Segment for callt and interrupt function
_inter :
; line 18 : }
         reti
; line 19 :
; line 20 : callt   void f_clt ( ) {         /* callt function definition*/
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG    AT      0006H          ; Segment for vector table
_@vect06 :
         DW      _inter
         END
```

```
; Target chip : uPDxxxx
; Device file : xx.xxx
```

(3) 大型模式

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -ml -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE SJIS
$TOL_INF    03FH, 0100H, 00H, 08000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3

@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP             ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )
```

```
@@CNSTL     CSEG    PAGE64KP            ; Segment for const variable
_i_cnst :   DW      01H                 ; 1

@@RLINIT    CSEG    UNIT64KP            ; Segment for initialization data (with initial
                                        ; value)
            DW      00002H, 00000H  ; 2

@@INITL     DSEG  UNIT64KP          ; Segment for tentative data (with initial value)
_l_init :   DS    ( 4 )

@@DATAL     DSEG  UNIT64KP              ; Segment for tentative data (without initial
                                        ; value)
_i_data :   DS    ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function */
; line 2 :
; line  3 : void main ( void );             /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;    /* const variable */
; line  5 : callt   void f_clt ( void );    /* calltFunction prototype declaration */
; line  6 : __boolean       b_bit ;         /* boolean type variable */
; line  7 : long    l_init = 2 ;            /* Initialized external variable */
; line  8 : int     i_data ;                /* Uninitialized external variable */
; line  9 : __sreg  int     sr_inis = 3 ;    /* Iinitialized sreg variable */
; line 10 : __sreg  int     sr_dats ;        /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                 /* Function definition */

@@CODEL     CSEG                        ; Segment for code portion
_main :
        push    hl
; line 13 :         int     i ;
; line 14 :         i = 100 ;
        movw    hl, #064H         ; 100
; line 15 : }
        pop     hl
        ret
; line 16 :
; line 17 : void inter ( ) {                /* Interrupt function definition */

@@BASE          CSEG    BASE            ; Segment for callt and interrupt function
_inter :
; line 18 : }
        reti
; line 19 :
; line 22 : callt   void f_clt ( ) {        /* callt function definition */
_f_clt:
```

```
; line 23 : }
        ret
@@VECT06        CSEG    AT      0006H   ; Segment for vector table
_@vect06 :
                  DW      _inter
                    END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

第 4 章 汇编语言规范

本章解释了 78K0R 汇编器支持的汇编语言规范。

4.1 源程序的描述方法

该部分解释了源程序的描述方法、表达和运算符。

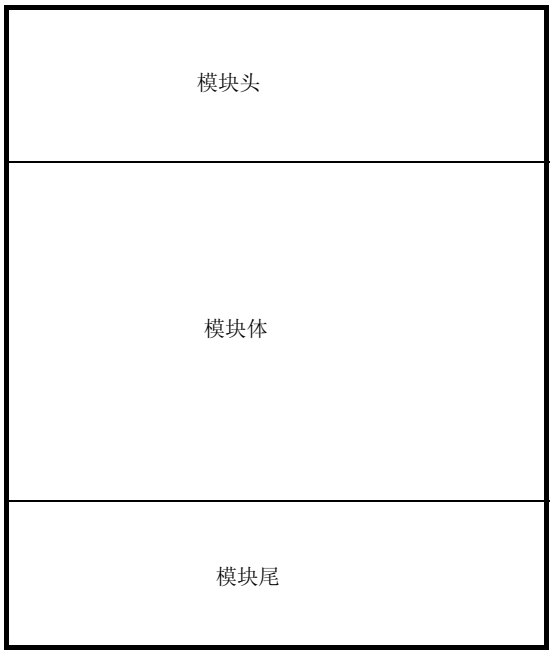
4.1.1 基本构造

当一个源程序被分割成多个模块进行描述时，每一个模块成为一个输入单元，在汇编器中称为源模块（如果一个源程序由单个模块组成的话，“源程序”就等同于“源模块”）。

成为汇编器中的一个输入单元的源模块主要由下列三部分组成：

- 模块头（模块头）
- 模块体（模块体）
- 模块尾（模块尾）

图 4-1. 源模块的结构



(1) 模块头

在模块头部分，可对下列显示的控制指令进行描述。注意这些控制指令只能在模块头中进行描述。

此外，可省略模块头。

表 4-1.　可在模块头中描述的指令

指令类型	说明
与汇编器选项具有相同功能的控制指令	- PROCESSOR - DEBUG/NODEBUG/DEBUGA/NODEBUGA - XREF/NOXREF - SYMLIST/NOSYMLIST - TITLE - FORMFEED/NOFORMFEED - WIDTH - LENGTH - TAB - KANJICODE
由 C 编译器或其他高级程序输出的特殊控制指令	- TOL_INF - DGS - DGL

(2) 模块体

以下指令不能在模块体中描述。

- 与汇编器选项具有相同功能的控制指令

其他所有的命令、控制指令和指令均可在模块体中进行描述。
模块体必须被分割成一个个单元进行描述，这些单元称为 “ 段”。
段由如下相关命令进行定义。

- 代码段
  由 CSEG 命令定义
- 数据段
  由 DSEG 命令定义
- 位段
  由 BSEG 命令定义
- 绝对段
  由 CSEG、 DSEG 或 BSEG 命令加上一个绝对地址 （AT 位置地址）作为重定位属性。该段也可由 ORG 命令进行定义。

模块体可被配置为任何段的整合。
但是，数据段和位段必须先于代码段定义。

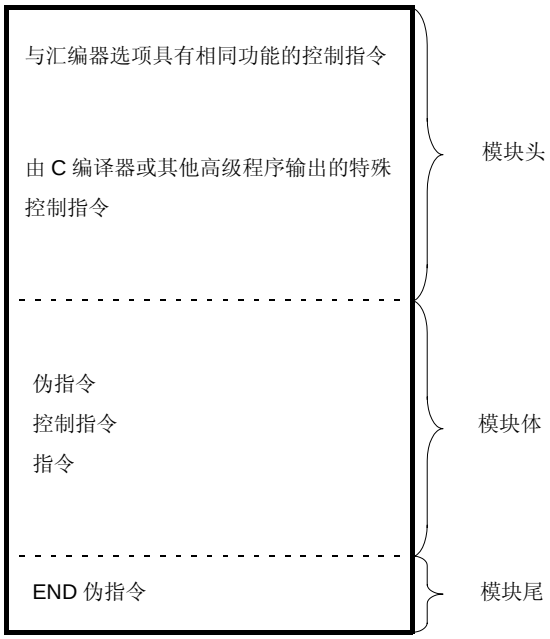
(3) 模块尾

模块尾意味着源模块的结束。必须在该部分描述 END 命令。
如果在 END 命令之后出现任何注释、空格、 tab 键或换行符以外的内容的话，汇编器将输出一条警告信息并忽略 END 命令之后出现的所有字符。

(4) 源程序的整体配置

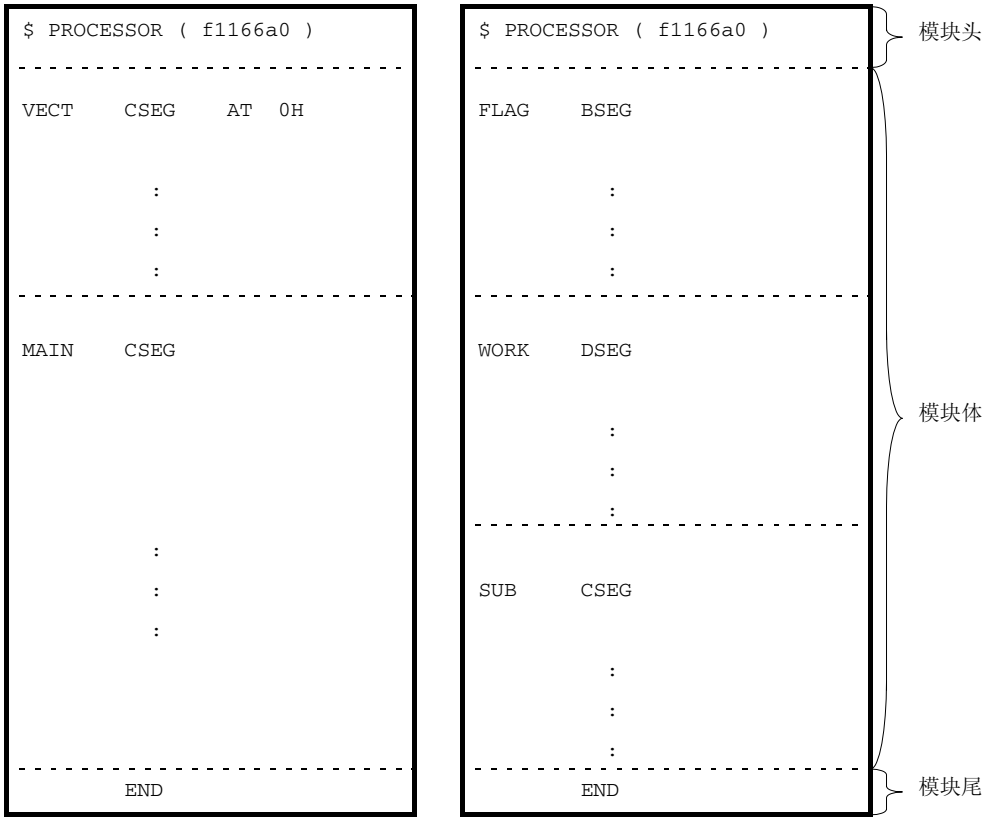
下列显示了源模块（源程序）的整体配置。

图 4-2. 源程序的整体配置



下列显示了简单源模块结构的示例。

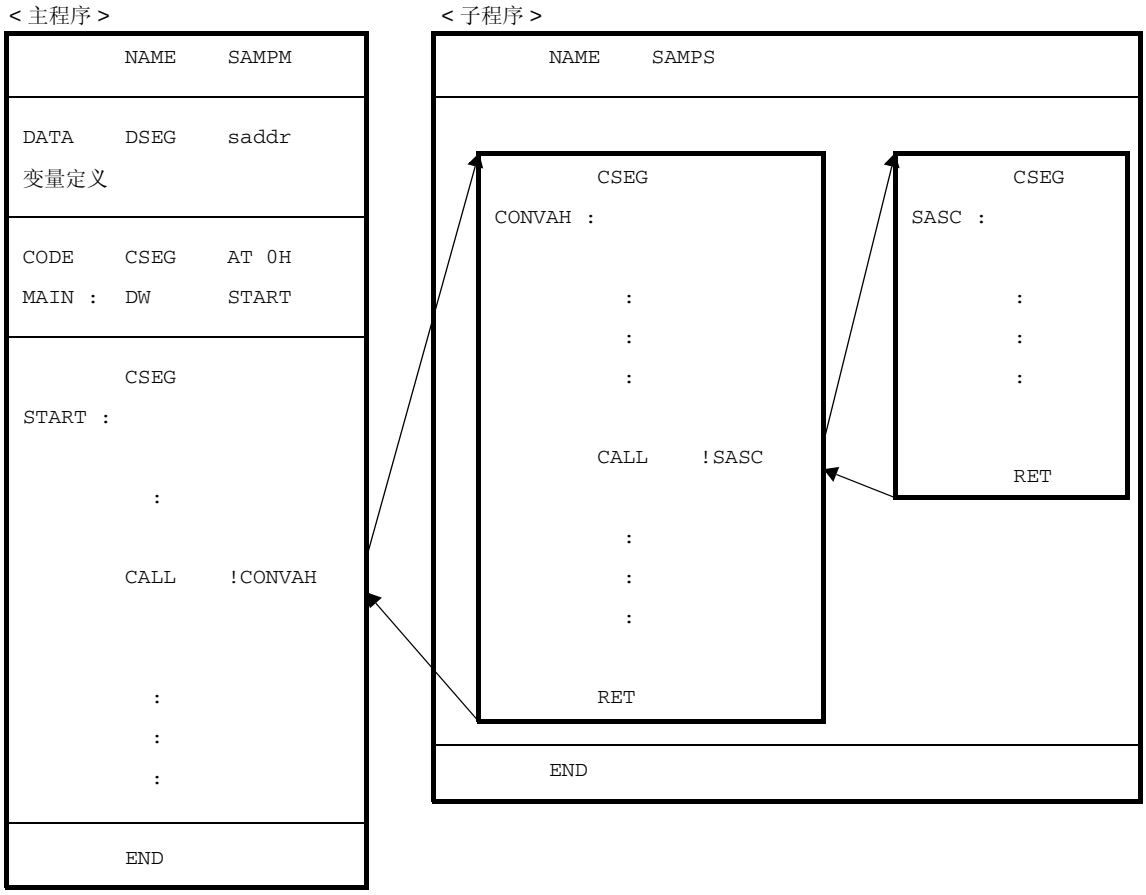
图 4-3. 源模块结构示例



(5) 代码示例

在该部分，源模块（源程序）的描述示例显示为样例程序。
样例程序的结构可简单描述如下。

图 4-4. 样例源程序配置



- 主程序

```
        NAME    SAMPM                   ; (1)
; *******************************************
;       HEX -> ASCII Conversion Program
;               main-routine
; *******************************************

PUBLIC  MAIN, START                     ; (2)
EXTRN   CONVAH                          ; (3)
EXTRN   _@STBEG                         ; (4)   <- 错误

DATA    DSEG    AT      0FFE20H         ; (5)
HDTSA : DS      1
STASC : DS      2

CODE    CSEG    AT      0H              ; (6)
```

```
MAIN :  DW      START

        CSEG                            ; (7)
START :
        ; chip initialize
        MOVW    SP, #_@STBEG

        MOV     HDTSA , #1AHex 2-code data in HL reqistor
        MOVW    HL, #LOWW ( HDTSA )     ; set h

        CALL    !CONVAH         ; convert ASCII <- HEX
                                ; output BC-register <- ASCII code
        MOVW    DE, #LOWW ( STASC )     ; set DE <- store ASCII code table
        MOV     A , B
        MOV     [ DE ] , A
        INCW    DE
        MOV     A , C
        MOV     [ DE ] , A
        BR      $$
        END                             ; (8)
```

(1) 模块名称声明

(2) 从另一个模块中引用为一个外部引用符号的符号声明

(3) 从另一个模块中定义为一个外部引用符号的符号声明

(4) 堆栈协议符的声明。当连接程序带有 -s 选项时，该声明将由连接程序产生。（如果连接程序未指定 -s 选项，将发生错误。）

(5) 数据段开始的声明 （将位于 saddr 内）

(6) 代码段开始的声明 （将位于绝对段，起始地址为 0H）

(7) 另一个代码段开始的声明 （绝对程序代码段结束）

(8) 模块结束的声明

- 子程序

```
        NAME    SAMPS           ; (9)
; ************************************************************
;       HEX -> ASCII Conversion Program
;               sub-routine
;
;   input condition     : ( HL )        <- hex 2 code
;   output condition    : BC-register   <- ASCII 2 code
; ************************************************************
```

```
PUBLIC  CONVAH                  ; (10)

        CSEG                            ; (11)
CONVAH :
        XOR     A , A
        ROL4    [HL]            ; hex upper code load (12)
        CALL    !SASC
        MOV     B, A            ; store result

        XOR     A , A
        ROL4    [HL]            ; hex lower code load
        CALL    !SASC
        MOV     C, A            ; store result
        RET

; ************************************************************
;       subroutine      convert ASCII code
;
;   input       Acc ( lower 4bits )     <- hex code
;   output      Acc                     <- ASCII code
; ************************************************************

SASC :
        CMP     A, #0AH         ; check hex code > 9
        BC      $SASC1
        ADD     A, #07H         ; bias ( +7H )
SASC1 :
        ADD     A, #30H         ; bias ( +30H )
        RET
        END                             ; (13)
```

(9) 模块名称声明

(10) 从另一个模块中引用为一个外部定义符号的符号声明

(11) 代码段开始声明

(12) ROL4 指令是属于 78K0 系列的指令， 78K0R 系列不支持该指令。必须指定汇编－兼容选项汇编该模块。
关于汇编选项 （兼容）的信息，请参考 78K0R 构建用户手册。

(13) 模块结束的声明

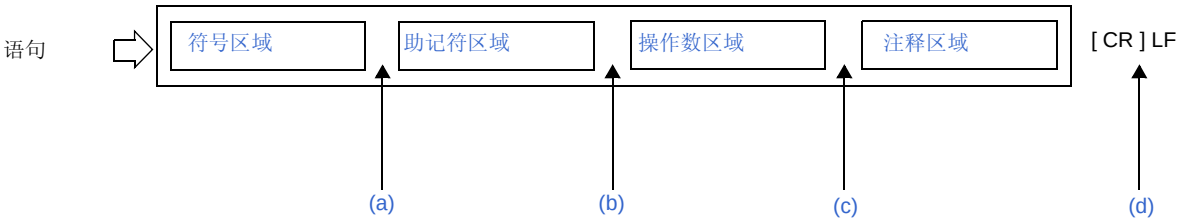
4.1.2　描述方法

(1) 配置

源程序由语句组成。

一个语句由 4 个区域组成，显示如下。

图 4-5. 语句区域



(a) 符号区域和助记符区域必须由冒号 （:）或一个或多个空格或 tab 键隔开。（是选择冒号还是空格取决于助记符区域中的指令。）

(b) 助记符区域和操作数区域必须由冒号 （:）或一个或多个空格或 tab 键隔开。根据助记符区域中的指令，可能不需要操作符字段。

(c) 如果使用注释区域，必需用分号 （;）隔开。

(d) 源程序中的每一行均由一个 LF 代码结束。（LF 代码前可能出现一个 CR 代码。）

- 一条语句必须在一行内进行描述。每行最多可包含 2048 个字符长度 （包括 CR/LF）。 TAB 和 CR （如果出现）每个都计作一个字符。如果一行内的字符超出 2048，将会产生一个警告并且超出第 2048 字符之后的字符将被忽略，不进行汇编。然后，超出的字符仍然会显示在汇编器的列表文件中。
- 只有由 CR 组成的行不会输出到汇编器列表文件。
- 以下描述也有效。
  - 虚拟行 （没有语句描述的行）
  - 只有符号字段组成的行
  - 只包含注释字段组成的行

(2) 字符集

源文件可包含以下三种字符。

- 语言字符
- 字符数据
- 注释字符

(a) 语言字符

语言字符是指在源程序中用于描述指令的字符。

语言字符集包括字母、数字和特殊字符。

表 4-2. 字母字符

名称		字符
数字字符		0 1 2 3 4 5 6 7 8 9
字符 字符	大写	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
	小写	a b c d e f g h i j k l m n o p q r s t u v w x y z

表 4-3. 特殊字符

字符	名称	主要用途	
?	问号	相当于字母字符	
@	Circa	相当于字母字符	
_	下划线	相当于字母字符	
	Blank	区域分隔符	分隔符 符号
HT (09H)	Tab 码	相当于空格	
,	逗号	操作数分隔符	
:	冒号	标签分隔符	
;	分号	符号说明注释区域的开始	
CR (0DH)	返回码	符号表示一行的结束（在汇编器中被省略）	
LF (0AH)	换行码	符号表示一行的结束	
+	加号	加法运算符或正号	汇编器运算符
-	减号	减法运算符或负号	
*	星号	乘法运算符	
/	斜线	除法运算符	
.	句号	比特位置说明符	
( )	左括号和右括号	符号规定进行算术运算的顺序	
< >	大于和小于号	比较运算符	
=	等号	比较运算符	
í	单引号	- 符号表示字符内容的开始或结束 - 符号表示一个完整的宏参数	
$	美元符号	- 符号表示位置计数器 - 符号表示控制指令的开始等同于汇编器选项 - 符号指定相对寻址	
&	‘和’号	连接符（用于宏程序）	
#	升号	符号指定立即选址	
!	感叹号	符号指定绝对寻址	
[ ]	方括号	符号指定间接寻址	

(b) 字符数据

符号数据是指用于写文字字符串、字符串和一些控制指令引号内运算的字符 （TITLE, SUBTITLE, INCLUDE）。

注意事项 1.　可使用除了 00H 外的全部字符 （包括 kanji，根据不同操作系统代码可能会不同）。如果出现 00H，将产生错误并且从 00H 至结尾单引号 （'）内的所有字符将被忽略。

2.　当出现非法字符时，汇编器会在汇编列表中以感叹号 （!）取代它。（CR （0DH）字符不会输出到汇编列表中。）

3.　Windows OS 会把代码 1AH 视为文件结束代码 （EOF）。输入数据不能包含该代码。

(c) 注释字符

注释字符用于书写注释。

注意事项　注释字符和字符数据拥有相同的字符集。然后，注释中出现 00H 不会报错。在汇编列表中 00H 以 “!” 出现。

(3) 符号区域

符号区域用于给地址和数据对象命名的符号。符号使得程序更易于理解。

(a) 符号类型

根据用途和定义方法，符号可进行如下分类。

符号类型	作用	定义方法
名称	用于源程序中地址和数据对象的名称。	在符号区域中写入 EQU、 SET 或 DBIT 命令。
标签	用于源程序中地址和数据对象的标签。	在冒号 （:）后写入名称。
外部参考名称	用于由其他源模块定义的参考符号。	在操作数区域中写入 EXTRN 或 EXTBIT 命令。
段名称	在连接时使用。	在符号区域写入 CSEG、 DSEG、 BSEG 或 ORG 命令。
模块名称	在符号调试时使用。	在操作数区域中写入 NAME 命令。
宏名称	用于在源程序中命名宏。	在符号区域中写入 MACRO 命令。

注意事项　可在符号区域中写入的四种符号为名称、标签、段名称和宏名称。

(b) 符号描述的规定

在写符号时需遵循以下规定。

- 符号中可使用的字符是字母字符和特殊字符 （?、 @、 _）。

  符号中的首字符不可为数字 （0 至 9）。

- 最大符号长度为 256 字符。

  超出最大长度的字符将被忽略。

- 保留字不能用作符号。

  查看 “4.5 保留字”，获取保留字列表。

- 同样的符号只能定义一次。

  然而，采用 SET 命令定义的名称可通过 SET 命令被再次定义。

- 汇编器区分大写和小写字符。

- 当符号区域中写入标签时，必须在标签名称后直接加冒号 （:）。

< 正确的符号示例 >

```
CODE01   CSEG             ;  “CODE01”是段名称。
VAR01    EQU     10H      ;  “VAR01”是名称。
LAB01 :  DW      0        ;  “LAB01”是标签。
         NAME    SAMPLE   ;  “SAMPLE”是模块名称。
MAC1     MACRO            ;  “MAC1”是宏名称。
```

< 不正确的符号示例 >

```
1ABC     EQU     3        ; 首字符为数字。
LAB      MOV     A, R0    ;  “LAB”是标签，必须与助记符区域隔开
                          ; 可采用冒号 （:）进行分隔。
FLAG :   EQU     10H      ; 名称中不需要冒号 （:）。
```

< 符号过长的示例 >

```
A123456789B12...Y123456789Z123456        EQU      70H
            257 个字符                    ; 最后一个字符 （6）被省略
                                         ; 因为其长度超出最大符号长度。
                                         ; 该符号被定义为
                                         ; A123456789B12...Y123456789Z12345
```

< 仅由符号构成的语句的示例 >

```
ABCD :                   ; ABCD 被定义为标签。
```

(c) 符号中需注意的几点

??Rannnn （其中 nnnn = 0000 至 FFFF）是汇编器每次会自动取代的符号，它在宏本体中产生一个本地符号。一不小心，它就会导致重复定义，从而出现错误。

当段定义命令未给定名称时，汇编器会自动产生一个名称。这些段名称显示如下。

重复的段名称定义是错误的。

段名称	命令	重新配置属性
?A0nnnnn (nnnnn = 00000 - FFFFF)	ORG 命令	无
?CSEG	CSEG 命令	UNIT
?CSEGUP		UNITP
?CSEGT0		CALLT0
?CSEGFX		FIXED
?CSEGSI		SECUR_ID
?CSEGB		BASE
?CSEGP64		PAGE64KP
?CSEGU64		UNIT64KP
?CSEGMIP		MIRRORP
?CSEGOB0		OPT_BYTE
?DSEG	DSEG 命令	UNIT
?DSEGUP		UNITP
?DSEGS		SADDR
?DSEGSP		SADDRP
?DSEGBP		BASEP
?DSEGP64		PAGE64KP
?DSEGU64		UNIT64KP
?BSEG	BSEG 命令	UNIT

(d) 符号属性

所有名称和标签都具有一个值和一个属性。
值是指定义数据对象的值，如数值或地址本身的值。
段名称、模块名称和宏名称不具有值。
下面表格列出了符号属性。

属性类型	分类	值
NUMBER	- 为分配的数字常量命名 - 由 EXTRN 命令定义的符号 - 数字常量	十进制表示法 ： 0 至 1048575 十六进制表示法 ： 00000H 至 FFFFFH （无符号）
ADDRESS	- 定义为标签的符号 - 由 EQU 和 SET 命令定义的标签名称	十进制表示法 ： 0 至 1048575 十六进制表示法 ： 0H 至 FFFFFH
BIT	- 作为位值定义的名称 - 在 BSEG 中的名称 - 由 EXTBIT 命令定义的符号	0H 至 FFFFFH
SFR	通过 EQU 命令，名称定义为 SFR	SFR 区域
SFRP	通过 EQU 命令，名称定义为 SFR	

属性类型	分类	值
CSEG	由 CSEG 命令定义的段名称	这些属性类型均不包含值。
DSEG	由 DSEG 命令定义的段名称	
BSEG	由 BSEG 命令定义的段名称	
MODULE	由 NAME 命令定义的模块名称。（若无指定，模块名称来自输入的源文件名的初始名称。）	
MACRO	由 MACRO 命令定义的宏名称	

< 示例 >

```
TEN      EQU     10H          ; 名称 TEN 具有 NUMBER 属性且值为 10H。
         ORG     80H
START :  MOV     A, #10H      ; 标签 START 具有 ADDRESS 属性且值为 80H。
BIT1     EQU     0FFE20H.0    ; 名称 BIT1 具有 BIT 属性且值为 0FFE20H.0。
```

(4) 助记符区域

在助记符区域写入指令助记符、命令和宏参考指令。
如果指令或命令需要一个或多个操作数时，助记符区域必须通过一个或多个空格或 tab 键与操作数区域分开。
然而，如果首个操作数以 “#”、“$”、“!” 或 “[” 开始的话，即使助记符区域和首个操作数区域之间无任何分隔，语句的汇编仍然可正常进行。

< 正确助记符的示例 >

```
MOV      A, #0H
CALL     !CONVAH
RET
```

< 不正确助记符的示例 >

```
MOVA     #0H              ; 助记符区域和操作数区域之间无空格。
C ALL     !CONVAH         ; 助记符区域内包含一个空格。
ZZZ                       ; 78K0R 系列中不包含 ZZZ 指令。
```

(5) 操作数区域

在操作数区域，对于需要操作数的指令、命令或宏参考命令进行写操作数 （数据）。
一些指令和命令不需要操作数，而另一些则需要两个或更多。
当您提供两个或更多操作数时，用逗号 （,）隔开。
下列数据类型可显示于操作数区域：

- 常量 （数字或字符串）
- 字符串
- 寄存器名称
- 特殊字符 ($ # ! [ ])
- 段定义命令的重定义属性
- 符号
- 表达式
- 位项

所需操作数的大小和属性由指令或命令决定。参阅 "4.1.16　操作数特性 " 获取详细信息。
参阅目标器件用户手册，以获取指令集操作数的格式和惯用表示法。

以下部分说明了可显示于操作数区域的数据类型。

(a) 常量

常量是一个固定值或数据条目，也可作为立即数据。

常量分为数字常量和字符串常量。

- 数字常量

数字常量可表示为二进制、八进制、十进制或十六进制。

下列表格列出了数字常量的表示法。

数字常量为无符号的 32 位数据。

可用数值范围是 0 <= n <= 0FFFFFFFFH。

使用符号运算符来表示负值。

类型	符号	示例
二进制	在数字之后添加 “B” 或 “Y”。	1101B 1101Y
八进制	在数字之后添加 “O” 或 “Q”。	74O 74Q
十进制	只需写出数字，或在数字之后添加 “D” 或 “T”。	128 128D 128T
十六进制	- 在数字之后添加 “H”。 - 如果数字以 “A”、“B”、“C”、“D”、“E” 或 “F” 开头时，在其前面添加 “0”。	8CH 0A6H

- 字符串常量

字符串常量由显示在 "(2)　字符集 " 中的一串字符表示，在一组单引号 （'）内。

汇编器把串常量转换为 7 位 ASCII 码，奇偶位为 0。

串常量的长度为 0 至 2 个字符。

如果要把单引号字符归入字符串，中需写两遍。

< 字符串常量的示例 >

```
'ab'            ; 6162H
'A'             ; 0041H
'A'''           ; 4127H
' '             ; 0020H  （1 个空格的字符）
```

(b) 字符串

字符串常量由显示在 "(2)　字符集 " 中的一串字符表示，在一组单引号 （'）内。

字符串的主要用途是作为操作数用于 DB 和 CALL 命令以及 TITLE 和 SUBTITLE 控制指令。

< 特殊字符的示例 >

```
        CSEG
MAS1 :  DB      'YES'           ; 以字符串 “YES” 初始化。
MAS2 :  DB      'NO'            ; 以字符串 "NO" 初始化。
```

(c) 寄存器名称

下列寄存器可在操作数区域内命名：

- 通用寄存器
- 通用寄存器组
- 特殊功能寄存器

通用寄存器和通用寄存器组可采用其绝对名称 （R0 至 R7 和 RP0 至 RP3）进行描述，以及其功能名称（X、 A、 B、 C、 D、 E、 H、 L、 AX、 BC、 DE 和 HL）。

根据指令类型的不同，在操作数区域中描述的寄存器名称可能有所不同。关于每个寄存器名称的具体描述方法，请查阅每个器件的用户手册以获取器件开发时所使用的软件。

(d) 特殊字符

下表列出了可出现在操作数区域的特殊字符。

特殊字符	功能
$	- 表示具有操作数的指令的位置地址 （或多字节指令时的地址首字节）。 - 表示分支指令的相对地址。
!	- 表示分支指令的绝对地址。 - 表示 addr16 规范，允许采用 MOV 指令访问整个存储器空间。
#	- 表示立即数据。
[]	- 表示间接寻址。

< 特殊字符的示例 >

```
Address     Source program
100             ADD     A, #10H
102     LOOP :  INC     A
103             BR      $$ - 1      ; 操作数中的第二个 $ 表示地址
                                    ; 103H。说明 “BR $ - 1” 发生在
                                    ; 同一个操作中。
105             BR      !$ + 100H   ; 操作数中的第二个 $ 表示地址
                                    ; 105H。说明 “BR $ + 100H” 发生在
                                    ; 同一个操作中。
```

(e) 段定义命令的重定义属性

重定义属性可出现在操作数区域内。

查阅 "4.2.2　段定义命令 " 以获取更多关于重定义属性的信息。

(f) 符号

当符号出现在操作数区域中时，分配给该符号的地址 （或值）会编程操作数的值。

< 符号值的示例 >

```
VALUE   EQU     1234H
        MOV     AX, #VALUE      ; 也可写为 MOV      AX, #1234H
```

(g) 表达式

表达式是常量、位置地址 （由 $ 表示）、名称、标签和运算符的组合，用于求值。

表达式可指定为指令操作数，其中可指定数值。

查阅 "4.1.3　表达式和运算符 " 以获取更多关于表达式的信息。

< 表达式的示例 >

```
TEN     EQU     10H
        MOV     A, #TEN - 5H
```

在该示例中，“TEN - 5H”是一个表达式。

在该表达式中，名称和数值用 ?。表达式的值为 0BH，因此该表达式可重新写作 “MOV A , #0BH”。

(h) 位项

位项由位的位置符获得。

查阅 "4.1.14　比特位置说明符 " 以获取更多关于位项的信息。

< 位项的示例 >

```
CLR1    A.5
SET1    1 + 0FFE30H.3   ; 该操作数的值为 0FFE31H.3
CLR1    0FFE40H.4 + 2   ; 该操作数的值为 0FFE40H.6
```

(6) 注释区域

在分号（;）之后的注释区域对注释进行描述。
注释区域的续行从冒号开始到新的一行代码结束或文件的 EOF 码。
注释有助于理解和维护程序。
汇编器不会对注释进行处理，但是会输出到汇编列表中。
可在注释区域中出现的字符为那些显示在 "(2)　字符集 " 中的字符。

< 注释的示例 >

```
        NAME    SAMPM
; ***************************************
;       HEX -> ASCII Conversion Program
;               main-routine
; ***************************************

        PUBLIC  MAIN , START
        EXTRN   CONVAH
        EXTRN   @STBEG

DATA    DSEG    saddr
HDTSA : DS      1
STASC : DS      2

CODE    CSEG    AT 0H
MAIN :  DW      START

        CSEG
START :
                                ; chip initialize
        MOVW    SP, #_@STBEG

        MOV     HDTSA , #1AH
        MOVW    HL, #HDTSA      ; set hex 2-code data in HL register

        CALL    !CONVAH         ; convert ASCII <- HEX
                                ; output BC-register <- ASCII code

        MOVW    DE, #STASC      ; set DE <- store ASCII code table
        MOV     A , B
        MOV     [ DE ] , A
        INCW    DE
        MOV     A , C
        MOV     [ DE ] , A
        BR      $$
        END
```

只有注释区域的行

只有注释区域的行

注释
位于注释
注释区域

4.1.3　表达式和运算符

表达式可以是一个符号、常量、位置地址 （由 $ 表示）或位项，运算符则与上述中的一项相结合或者是多个运算符的结合。

表达式中异于运算符的元素称为项，根据其在表达式中的顺序，从左到右可称为第一项、第二项以及第四项。

汇编器支持显示在？表 4-4.　运算符类型”中的运算符。运算符拥有优先级，这决定了它们在计算中的顺序。优先顺序显示在 “表 4-5.　运算符的优先级？中。

可通过把项和运算符置于括号 “( )”中的方法改变计算顺序。

< 示例 >

```
MOV     A, #5 * ( SYM + 1 )
```

在上述示例中， "5 * (SYM+1)。“5”为表达式的第一项、“SYM”为第二项，以及 “1”为第三项。运算符为 "*", "+" 和 "()"。

表 4-4.　运算符类型

运算符类型	运算符
算术运算符	+, -, *, /, MOD, +sign, +sign
逻辑运算符	NOT, AND, OR, XOR
关系运算符	EQ (=), NE (< >), GT (>), GE (>=), LT (<), LE (<=)
移位运算符	SHR, SHL
字节分离运算符	HIGH, LOW
字分离运算符	HIGHW, LOWW
特殊运算符	DATAPOS, BITPOS, MASK
其它运算符	( )

上述运算符可分为一元运算符、特殊一元运算符、二元运算符、 N 元运算符和其他运算符。

一元运算符	+sign, -sign, NOT, HIGH, LOW, HIGHW, LOWW
特殊一元运算符	DATAPOS, BITPOS
二元运算符	+, -, *, /, MOD, AND, OR, XOR, EQ (=), NE (<>), GT (>), GE (>=), LT (<), LE (<=), SHR, SHL
N 元运算符	MASK
其他运算符	( )

表 4-5. 运算符的优先级

优先	级别	运算符
高	1	+ sign, - sign, NOT, HIGH, LOW, HIGHW, LOWW, DATAPOS, BITPOS, MASK
	2	*, /, MOD, SHR, SHL
	3	+, -
	4	AND
	5	OR, XOR
低	6	EQ (=), NE (<>), GT (>), GE (>=), LT (<), LE (<=)

表达式根据下列规则进行运算。

- 运算顺序由运算符的优先级决定。
  当两个运算符拥有相同优先级时，运算顺序为从左至右；一元运算符除外，其运算顺序为从右至左。
- 括号 “( )”内的子表达式比括号外的子表达式先运算。
- 运算允许包含连续的一元运算符。
  示例：
  1 = - - 1 == 1
  -1 = -+1 = -1
- 表达式采用无符号的 32 位数值进行运算。
  如果立即数超出 32 位，则超出的数值将被省略。
- 如果常量的值超出 32 位，则会产生错误，并且其值计算为 0。
- 在除法中，小数被省略。
  如果除数为 0，则会产生错误，并且其值计算为 0。
- 负数表示为二的补集。
- 当源文件汇编时，外部参考符号的求值为 0 （该求值在链接时间内决定）。
- 在操作数区域中，表达式的运算结果必须满足指令的要求，成为有效的操作数。
  当表达式包含可重定位或外部参考项并且指令需求 8 位操作数时，目标代码为最不重要的 8 位操作数的值以及 16 位的需求信息将输出到重定位信息。然后，连接器会检查前一数值是否超出了 8 位的范围。如果溢出，则链接发生错误。
  在绝对表达式的情况下，该数值在汇编阶段决定并在该阶段进行检查，用于测试结果是否符合所要求的范围。例如， MOV 指令需要 8 位操作数，因此操作数范围必须在 0H 至 0FFH。

(1) 正确的示例

```
MOV     A, #'2*' AND 0FH
MOV     A, #4 * 8 * 8 - 1
```

(2) 不正确的示例

```
MOV     A, #'2*.
MOV     A, #4 * 8 * 8
```

(3) 求值的示例

表达式	求值
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0FFFFFFFFH
-1 > 1	00H (False)
EXT[Note] + 1	1

注 EXT ： 外部参考符号

4.1.4 算术运算符

以下算术运算符可用。

运算符	概要
+	第一个项值与第二项值相加
-	第一个项值与第二项值相减
*	第一个项值与第二项值相乘。
/	表达式的第 1 项值除以第 2 项值，并返回结果的整数部分。
MOD	获取表达式第 1 项值除以第 2 项值结果中的余数。
+sign	照常返回该项值。
+sign	寻找该项值 2 的余数。

+

第一个项值与第二项值相加

[功能]

返回表达式中第 1 项和第 2 项相加值的总数。

[应用示例]

```
        ORG     100H
START : BR      !$ + 6          ; (1)
```

(1) BR 指令产生到 " 当前位置地址加 6" 的跳转，也就是到地址 "100H + 6H = 106H"。
因此，在以上例子中的 （1）也可描述为：START : BR !106H

-

第一个项值与第二项值相减

[功能]

返回第 1 项值减第 2 项值的结果。

[应用示例]

```
        ORG     100H
BACK :  BR      BACK - 6H       ;  (1)
```

(1) BR 指令产生到 " 分配给 BACK 的地址减 6" 的跳转，也就是到地址 "100H - 6H = 0FAH"。
因此，在以上例子中的 （1）也可描述为：BACK : BR !0FAH

*

第一个项值与第二项值相乘

[功能]

返回表达式中第 1 项和第 2 项相乘 (乘积) 的结果。

[应用示例]

```
TEN     EQU     10H
        MOV     A, #TEN * 3     ;  (1)
```

(1) 使用 EQU 指令，在名为 "TEN" 中定义值为 "10H"。
"#" 表示中间数据。表达式 "TEN * 3" 与 "10H * 3" 相同，并返回数值 "30H"。
因此，在以上表达式中的 （1）也可描述为：MOV A , #30H

/

表达式的第 1 项值除以第 2 项值，并返回结果的整数部分。

[功能]

表达式的第 1 项值除以第 2 项值，并返回结果的整数部分。
将截断结果中十进制数的小数部分。如果除法运算的除数 （第 2 项）为 0，则产生错误

[应用示例]

```
MOV     A, #256 / 50          ;  (1)
```

(1) "256 / 50" 相除的结果为 5 并余 6。
运算符返回数值 "5", 它是除法结果的整数部分。
因此，在以上表达式中的 （1）也可描述为：MOV A , #5

MOD

获取表达式第 1 项值除以第 2 项值结果中的余数。

[功能]

获取表达式第 1 项值除以第 2 项值结果中的余数。
如果除数 （第 2 项）为 0，则产生错误。
之前需要空格且后面是 MOD 运算符。

[应用示例]

```
MOV     A, #256  MOD  50       ;  (1)
```

(1) "256 / 50" 相除的结果为 5 并余 6。
MOD 运算符返回余数 6。
因此，在以上表达式中的 （1）也可描述为：MOV A , #6

+sign

照常返回该项值。

[功能]

不改变返回表达式中该项数值。

[应用示例]

```
FIVE    EQU     +5      ;  (1)
```

(1) 不改变返回该项数值 "5"。
使用 EQU 指令定义名为 "FIVE" 的值为 "5"。

+sign

寻找该项值 2 的余数。

[功能]

通过两个余数返回表达式该项数值。

[应用示例]

```
NO      EQU     -1      ;  (1)
```

(1) -1 成为 1 除以 2 的余数。

二进制数 2 的余数 0000 0000 0000 0000 0000 0000 0000 0001 成为 :

1111 1111 1111 1111 1111 1111 1111 1111

因此 , 使用 EQU 指令 , 在名为 "TEN" 中定义值为 "0FFFFFFFFH"。

4.1.5 逻辑运算符

以下逻辑运算符可用。

运算符	概要
NOT	每一位取逻辑非 （NOT)。
AND	对第 1 项值和第 2 项值的每一位取 AND(逻辑与）运算。
OR	对第 1 项值和第 2 项值的每一位取逻辑 OR 运算。
XOR	对第 1 项值和第 2 项值的每一位取 OR （逻辑或）运算。

NOT

每一位取逻辑非 （NOT)。

[功能]

对表达式中某项值逐位取反，并返回结果。
在 NOT 运算符和某项之间需要用空格分隔。

[应用示例]

```
MOVW    AX, #LOWW ( NOT 3H )   ; (1)
```

(1) 对 "3H" 进行逻辑非如下所示 :

NOT)	0000	0000	0000	0000	0000	0000	0000	0011
	1111	1111	1111	1111	1111	1111	1111	1100

返回 0FFFFFFFCH。
因此 , 对 (1) 也做如下描述 : MOVW AX , #LOWW #0FFFFFFFCH

AND

对第 1 项值和第 2 项值的每一位取 AND(逻辑与）运算。

[功能]

在表达式的第 1 项值和第 2 项值之间逐位进行 AND(逻辑积），并返回结果。
之前需要空格且后面是 AND 运算符。

[应用示例]

```
MOV     A, #6FAH  AND  0FH     ;  (1)
```

(1) 在 "6FAH" 和 "0FH" 之间进行 AND 运算如下所示：

	0000	0000	0000	0000	0000	0110	1111	1010
AND)	0000	0000	0000	0000	0000	0000	0000	1111
	0000	0000	0000	0000	0000	0000	0000	1010

返回结果 "0AH"。因此，在以上表达式中的 （1）也可做如下描述：
MOV A , #0AH

OR

对第 1 项值和第 2 项值的每一位取逻辑 OR 运算。

[功能]

在表达式的第 1 项值和第 2 项值之间逐位进行 OR (逻辑和)，并返回结果。
之前需要空格且后面是 OR 运算符。

[应用示例]

```
MOV     A, #0AH  OR  1101B     ;  (1)
```

(1) 在 "0AH" 和 "1101B" 之间进行 OR 运算如下所示：

	0000	0000	0000	0000	0000	0000	0000	1010
OR)	0000	0000	0000	0000	0000	0000	0000	1101
	0000	0000	0000	0000	0000	0000	0000	1111

返回结果 "0FH"。
因此，在以上表达式中的 （1）也可描述为：MOV A , #0FH

XOR

对第 1 项值和第 2 项值的每一位取 OR （逻辑或）运算。

[功能]

在表达式的第 1 项值和第 2 项值之间逐位进行 OR （逻辑或）运算，并返回结果。之前需要空格且后面是 XOR （逻辑异或）运算符。

[应用示例]

```
MOV     A, #9AH  XOR  9DH      ;  (1)
```

(1) 在 "9AH" 和 "9DH" 之间进行 XOR （逻辑异或）运算如下所示：

	0000	0000	0000	0000	0000	0000	1001	1010
XOR)	0000	0000	0000	0000	0000	0000	1001	1101
	0000	0000	0000	0000	0000	0000	0000	0111

返回结果 "7H"。
因此，在以上表达式中的 （1）也可描述为：MOV A , #7H

4.1.6　关系运算符

以下关系运算符可用。

运算符	概要
EQ (=)	比较第 1 项的值是否与第 2 项的值相等。
NE (< >)	比较第 1 项的值是否与第 2 项的值不相等。
GT (>)	比较第 1 项值是否大于第 2 项值。
GE (>=)	比较第 1 项值是否大于或等于第 2 项值。
LT (<)	比较第 1 项值是否小于第 2 项值。
LE (<=)	比较第 1 项值是否小于或等于第 2 项值。

EQ (=)

比较第 1 项的值是否与第 2 项的值相等。

[功能]

如果表达式的第 1 项值与第 2 项值相等，则返回 0FFH(真），如果两个值不相等，则返回 00H(假）。
之前需要空格且后面是 EQ 运算符。

[应用示例]

```
A1      EQU     12C4H
A2      EQU     12C0H

        MOV     A, #A1  EQ  ( A2 + 4H )         ;  (1)
        MOV     X, #A1  EQ   A2                 ;  (2)
```

(1) 在 （1）上面，表达式 "A1 EQ (A2 + 4H)" 变成 "12C4H EQ (12C0H + 4H)"。
因为第 1 项值与第 2 项值相等，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "A1 EQ A2" 变成 "12C4H EQ 12C0H"。
因为第 1 项值与第 2 项值不相等，所以运算符返回 00H。

NE (< >)

比较第 1 项的值是否与第 2 项的值不相等。

[功能]

如果表达式的第 1 项值与第 2 项值不相等，则返回 0FFH(真），如果两个值相等，则返回 00H(假）。
之前需要空格且后面是 NE 运算符。

[应用示例]

```
A1      EQU     5678H
A2      EQU     5670H

        MOV     A, #A1  NE   A2                ;  (1)
        MOV     A, #A1  NE  ( A2 + 8H )        ;  (2)
```

(1) 在 （1）上面，表达式 "A1 NE A2" 变成 "5678H NE 5670H"。
因为第 1 项值与第 2 项值不相等，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "A1 NE (A2 + 8H)" 变成 "5678H NE (5670H + 8H)"。
因为第 1 项值与第 2 项值相等，所以运算符返回 00H。

GT (>)

比较第 1 项值是否大于第 2 项值。

[功能]

如果表达式的第 1 项值大于第 2 项值，则返回 0FFH (真)，如果第 1 项值小于或等于第 2 项值，则返回 00H (假)。之前需要空格且后面是 GT 运算符。

[应用示例]

```
A1      EQU     1023H
A2      EQU     1013H

        MOV     A, #A1  GT   A2                ;  (1)
        MOV     X, #A1  GT  ( A2 + 10H )       ;  (2)
```

(1) 在 （1）上面，表达式 "A1 GT A2" 变成 "1023H GT 1013H"。
因为第 1 项值大于第 2 项值，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "A1 GT (A2 + 10H)" 变成 "1023H GT (1013H + 10H)"。
因为第 1 项值与第 2 项值相等，所以运算符返回 00H。

GE (>=)

比较第 1 项值是否大于或等于第 2 项值。

[功能]

如果表达式的第 1 项值大于或等于第 2 项值，则返回 0FFH (真)，如果第 1 项值小于第 2 项值，则返回 00H (假)。之前需要空格且后面是 GE 运算符。

[应用示例]

```
A1      EQU     2037H
A2      EQU     2015H

        MOV     A, #A1  GE   A2                ;  (1)
        MOV     X, #A1  GE  ( A2 + 23H )       ;  (2)
```

(1) 在 （1）上面，表达式 "A1 GE A2" 变成 "2037H GE 2015H"。
因为第 1 项值大于第 2 项值，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "A1 GE (A2 + 23H)" 变成 "2037H GE (2015H + 23H)"。
因为第 1 项值小于第 2 项值，所以运算符返回 00H。

LT (<)

比较第 1 项值是否小于第 2 项值。

[功能]

如果表达式的第 1 项值小于第 2 项值，则返回 0FFH (真)，如果第 1 项值大于或等于第 2 项值，则返回 00H (假)。之前需要空格且后面是 LT 运算符。

[应用示例]

```
A1      EQU     1000H
A2      EQU     1020H

        MOV     A, #A1  LT  A2                ;  (1)
        MOV     X, # ( A1 + 20H )  LT  A2     ;  (2)
```

(1) 在 （1）上面，表达式 "A1 LT A2" 变成 "1000H LT 1020H"。
因为第 1 项值小于第 2 项值，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "(A1 + 20H) LT A2" 变成 "(1000H + 20H) LT 1020H"。
因为第 1 项值与第 2 项值相等，所以运算符返回 00H。

LE (<=)

比较第 1 项值是否小于或等于第 2 项值。

[功能]

如果表达式的第 1 项值小于或等于第 2 项值，则返回 0FFH (真)，如果第 1 项值大于第 2 项值，则返回 00H (假)。之前需要空格且后面是 LE 运算符。

[应用示例]

```
A1      EQU     103AH
A2      EQU     1040H

        MOV     A, #A1  LE  A2                ;  (1)
        MOV     X, # ( A1 + 7H )  LE  A2      ;  (2)
```

(1) 在 （1）上面，表达式 "A1 LE A2" 变成 "103AH LE 1040H"。
因为第 1 项值小于第 2 项值，所以运算符返回 0FFH。

(2) 在 （2）上面，表达式 "(A1 + 7H) LE A2" 变成 "(103AH + 7H) LE 1040H"。
因为第 1 项值大于第 2 项值，所以运算符返回 00H。

4.1.7移位运算符

以下移位运算符可用。

运算符	概要
SHR	只获取首项右移值并在第二项中出现。
SHL	只获取首项左移值并在第二项中出现。

SHR

只获取首项右移值并在第二项中出现。

[功能]

获取返回值通过表达式的第 1 项值右移位到由第 2 项值指定的二进制位数。全零等于移动指定移位的二进制位数到高字节位。

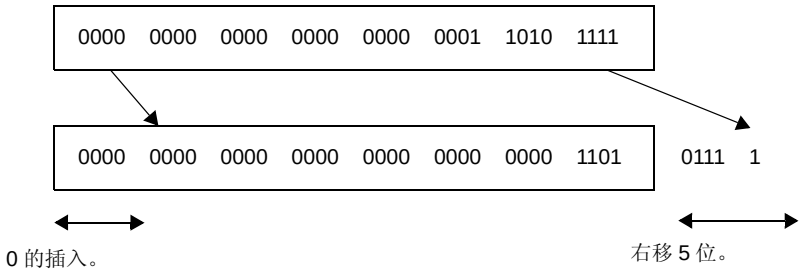
之前需要空格且后面是 SHR 运算符。

如果二进制转移位数为 0，照现在的样子返回第 1 项值。如果二进制转移位数超过 32, 则自动用零填充空的地方。

[应用示例]

```
MOV     A, #01AFH  SHR  5      ;  (1)
```

(1) 这个运算符对数值 "01AFH" 右移 5 位。



返回数值 "000DH"。

因此，在以上例子中的 （1）也可描述为：MOV A , #0DH

SHL

只获取首项左移值并在第二项中出现。

[功能]

获取返回值通过表达式的第 1 项值左移位到由第 2 项值指定的二进制位数。全零等于移动指定移位的二进制位数到低字节位。

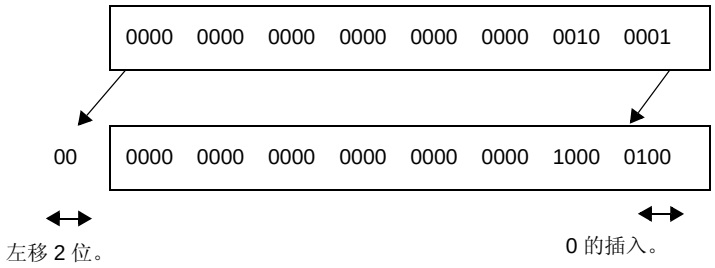
之前需要空格且后面是 SHL 运算符。

如果二进制转移位数为 0，照现在的样子返回第 1 项值。如果二进制转移位数超过 32, 则自动用零填充空的地方。

[应用示例]

```
MOV     A, #21H  SHL  2        ;  (1)
```

(1) 这个运算符对数值 "21H" 左移 2 位。



返回数值 "84H"。

因此，在以上例子中的 （1）也可描述为：MOV A , #84H

4.1.8　字节分离运算符

以下字节分离运算符可用。

运算符	概要
HIGH	返回某项的高 8 位值。
LOW	返回某项的低 8 位值。

HIGH

返回某项的高 8 位值。

[功能]

返回某项的高 8 位值。
在 HIGH 运算符和某项之间需要用空格分隔。

[应用示例]

```
MOV     A, #HIGH  1234H        ;  (1)
```

(1) 通过执行 MOV 指令，这个运算符返回表达式 "1234H" 的高 8 位值 "12H"。
因此，在以上例子中的 （1）也可描述为：MOV A , #12H

[备注]

进行 SFR 名称的 HIGH 操作，使用以下任一个描述的方法。

```
HIGHΔSFR-name
```

或 ,

```
HIGH[Δ]([Δ]SFR-name[Δ])
```

从对 NUMBER 绝对属性的操作数的运算获取结果。
对 SFR 名称不能进行其他操作。
< 示例 >

```
符号字段      记忆单元字段  操作数字段
                MOV             R0, #HIGH PM0
                MOV             R1, #HIGH PM1 + 1H      ; 等于 ( HIGH PM1 ) + 1
                MOV             R1, #HIGH ( PM1 + 1H )  ; 返回错误因为
                                                        ; 除 HIGH, LOW 以外的操作数 ,
                                                        ; 设定 HIGHW, 和  LOWW
                                                        ; 作为 SFR 名称
```

LOW

返回某项的低 8 位值。

[功能]

返回某项的低 8 位值。
在 LOW 运算符和某项之间需要用空格分隔。

[应用示例]

```
MOV     A, #LOW  1234H          ;  (1)
```

(1) 通过执行 MOV 指令，这个运算符返回表达式 "1234H" 的低 8 位值 "34H"。
因此，在以上例子中的 （1）也可描述为：MOV A , #34H

[备注]

进行 SFR 名称的 LOW 操作，使用以下任一个描述的方法。

```
LOWΔSFR-name
```

或 ,

```
LOW[Δ]([Δ]SFR-name[Δ])
```

从对 NUMBER 绝对属性的操作数的运算获取结果。
对 SFR 名称不能进行其他操作。
< 示例 >

```
符号字段      记忆单元字段   操作数字段
                MOV             R0, #LOW PM0
                MOV             R1, #LOW PM1 + 1H       ; 等于 #( LOW PM1 ) + 1
                MOV             R1, #LOW ( PM1 + 1H )   ; 返回错误因为
                                                        ; 除 HIGH, LOW 以外的操作数 ,
                                                        ; 设定 HIGHW, 和  LOWW
                                                       ; 作为 SFR 名称
```

4.1.9 字分离运算符

以下字分离运算符可用。

运算符	概要
HIGHW	返回某项的高 16 位值。
LOWW	返回某项的低 16 位值。

HIGHW

返回某项的高 16 位值。

[功能]

返回某项的高 16 位值。
在 HIGHW 运算符和某项之间需要用空格分隔。

[应用示例]

```
MOVW    AX, #HIGHW     12345678H       ; (1)

MOV     ES, #HIGHW     LAB             ; (2)
MOVW    AX, ES:!LAB
```

(1) 通过执行 MOVW 指令，这个运算符返回表达式 "12345678H" 的高 16 位值 "1234H"。
因此，在以上例子中的 （1）也可描述为：MOVW AX , #1234H

(2) 通过在行 （2) 上执行 MOV 指令，标签 LAB 的高端地址设置为 ES 寄存器。

[备注]

进行 SFR 名称的 HIGHW 操作，使用以下任一个描述的方法。

```
HIGHWΔSFR-name
```

或 ,

```
HIGHW[Δ]([Δ]SFR-name[Δ])
```

从对 NUMBER 绝对属性的操作数的运算获取结果。
对 SFR 名称不能进行其他操作。
< 示例 >

```
符号字段    记忆单元字段  操作数字段
            MOVW          RP0, #HIGHW PM0
            MOVW          RP1, #HIGHW PM1 + 1H    ; 等于 #( HIGHW PM1 ) + 1
            MOVW          RP1, #HIGHW ( PM1 + 1H ); 返回错误因为
                                                  ; 除 HIGH, LOW 以外的操作数 ,
                                                  ; 设定 HIGHW, 和  LOWW
                                                  ; 作为 SFR 名称
```

LOWW

返回某项的低 16 位值。

[功能]

返回某项的低 16 位值。
在 LOW 运算符和某项之间需要用空格分隔。

[应用示例]

```
MOVW    AX, #LOWW      12345678H       ; (1)
```

(1) 通过执行 MOVW 指令，这个运算符返回表达式 "12345678H" 的低 16 位值 "5678H"。
因此，在以上例子中的 （1）也可描述为：MOVW AX , #5678H

[备注]

进行 SFR 名称的 LOWW 操作，使用以下任一个描述的方法。

```
LOWWΔSFR-name
```

或 ,

```
LOWW[Δ]([Δ]SFR-name[Δ])
```

从对 NUMBER 绝对属性的操作数的运算获取结果。
对 SFR 名称不能进行其他操作。
< 示例 >

```
符号字段      记忆单元字段  操作数字段
              MOVW          RP0, #LOWW PM0
              MOVW          RP1, #LOWW PM1 + 1H     ; 等于 #( LOWW PM1 ) + 1
              MOVW          RP1, #LOWW ( PM1 + 1H ) ; 返回错误因为
                                                    ; 除 HIGH, LOW 以外的操作数 ,
                                                    ; 设定 HIGHW, 和  LOWW
                                                    ; 作为 SFR 名称
```

4.1.10　特殊运算符

以下特殊运算符可用。

运算符	概要
DATAPOS	获取位符号的地址部分。
BITPOS	获取位符号的位部分。
MASK	获取 16 位数值，该数值指定位的位置为 1 且所有其他为 0。

DATAPOS

获取位符号的地址部分。

[功能]

返回位符号的地址部分 （字节地址）。

[应用示例]

```
SYM     EQU     0FE68H.6                ;  (1)

        MOV     A, !DATAPOS  SYM        ;  (2)
```

(1) EQU 指令定义名为 "SYM" 的数值为 0FE68H.6。

(2) "DATAPOS SYM" 表示为 "DATAPOS 0FE68H.6", 并返回 "0FE68H" 。
因此，在以上例子中的 （2）也可描述为：MOV A , !0FE68H

BITPOS

获取位符号的位部分。

[功能]

返回位符号的位部分 （位位置）。

[应用示例]

```
SYM     EQU     0FE68H.6                ;  (1)

        CLR1    [HL].BITPOS  SYM        ;  (2)
```

(1) EQU 指令定义名为 "SYM" 的数值为 0FE68H.6。

(2) "BITPOS.SYM" 表示为 "BITPOS 0FE68H.6", 并返回 "6" 。
CLR1 指令清除 [HL].6 为 0。

MASK

获取 16 位数值 , 该数值指定位的位置为 1 且所有其他为 0。

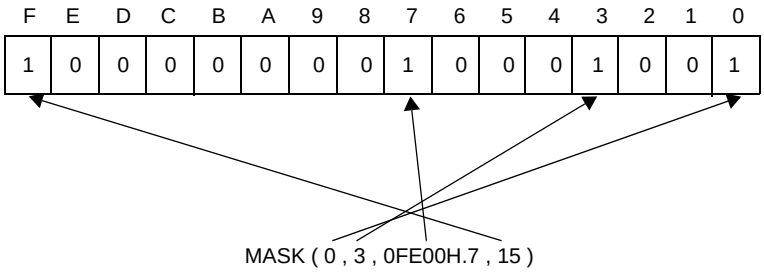
[功能]

返回 16 位数值，该数值指定位位置为 1 且设置所有其他位为 0。

[应用示例]

```
MOVW    AX, #MASK ( 0, 3, 0FE00H.7, 15 )    ;  (1)
```

(1) MOVW 指令返回数值 "8089H"。



4.1.11 其它运算符

以下运算符也可用。

运算符	概要
( )	在 ( ) 中把计算区分优先次序

( )

在 ( ) 中把计算区分优先次序

[功能]

括号中的运算要比括号外的运算优先。
这个运算符用于改变其他运算的优先顺序。
如果在多层中都嵌入了括号，则优先计算最内部括号中的表达式。

[应用示例]

```
MOV     A, # ( 4 + 3 ) * 2
```

按照表达式 (1), (2) 顺序进行计算，并作为结果返回值 "14" 。
如果没有使用括号。

按照以上显示的顺序 (1), (2) 进行计算，并作为结果返回值 "10" 。
有关运算符的优先顺序，参见 表 4-5. 运算符的优先级。

4.1.12 运算中的限制

表达式的运算是通过运算符把各项连接。可作为项进行描述的元素有常量、 $、名称和标签。每项具有重定位属性和符号属性。

根据每项中的重定位属性和符号属性的类型，可在每项上使用的运算符是受到限制的。因此，当描述表达式时，必须注意表达式中每一项的重定位属性和符号属性。

(1) 运算符和重定位属性

组成表达式的每一项均具有重定位属性和符号属性。

如果以重定位属性对项进行分类，可分为 3 类：绝对项、重定位项和外部参考项。

下表显示重定位属性和其特性，以及相对应的项。

表 4-6. 重定位属性的类型

类型	特性	对应的元素
绝对项	在汇编时间定义的值或常量的项	- 常量 - 绝对程序段中的标签 - $ 表示绝对程序段中的位置地址 - 由绝对值定义的名称，如常量或标签和上述列出的 $。
重定位项	不在汇编时间内定义的具有值的项	- 在可重定位段中定义的标签 - $ 表示可重定义段中的可重定义地址 - 由可重定位符号定义的名称
外部参考项[注]	其他模块中的符号外部参考项	- 由 EXTRN 命令定义的标签 - 由 EXTBIT 命令定义的名称

注 共有 6 种运算符可采用外部参考项作为运算目标；它们是 "+"、 "-"、 "HIGH"、 "LOW"、 "HIGHW" 和 "LOWW"。然而，一个表达式中只允许出现一个外部参考项，并且必须与 "+" 相连。

下表列出了按重定位属性分类的用于表达式的各运算符和项的组合。

表 4-7. 按重定位属性组合的运算符和项 （可重定位项）

运算符类型	项的可重定位属性			
	X:ABS Y:ABS	X:ABS Y:REL	X:REL Y:ABS	X:REL Y:REL
X + Y	A	R	R	-
X - Y	A	-	R	注 1
X * Y	A	-	-	-
X / Y	A	-	-	-
X MOD Y	A	-	-	-
X SHL Y	A	-	-	-
X SHR Y	A	-	-	-

运算符类型	项的可重定位属性			
	X:ABS Y:ABS	X:ABS Y:REL	X:REL Y:ABS	X:REL Y:REL
X EQ Y	A	-	-	注 1
X LT Y	A	-	-	注 1
X LE Y	A	-	-	注 1
X GT Y	A	-	-	注 1
X GE Y	A	-	-	注 1
X NE Y	A	-	-	注 1
X AND Y	A	-	-	-
X OR Y	A	-	-	-
X XOR Y	A	-	-	-
NOT X	A	A	-	-
+ X	A	A	R	R
- X	A	A	-	-
HIGH X	A	A	注 2	注 2
LOW X	A	A	注 2	注 2
HIGHW X	A	A	注 2	注 2
LOWW X	A	A	注 2	注 2
MASK (X)	A	A	-	-
DATAPOS X.Y	A	-	-	-
BITPOS X.Y	A	-	-	-
MASK (X.Y)	A	-	-	-
DATAPOS X	A	A	R	R
BITPOS X	A	A	A	A

ABS ： 绝对项
REL ： 可重定位项
A ： 结果为绝对项
R ： 结果为可重定位项
- ： 不可运算

注 **1.** 只有当 X 和 Y 在同一段中定义时才能运算，并且当 X 和 Y 不是可重定位项时，只能通过 HIGH、LOW、 HIGHW、 LOWW 或 DATAPOS 进行运算。

2. 当 X 和 Y 不是可重定位项时，只能通过 HIGH、 LOW、 HIGHW、 LOWW 或 DATAPOS 进行运算。

共有 6 种运算符可采用外部参考项作为运算目标；它们是 "+"、 "-"、 "HIGH"、 "LOW"、 "HIGHW" 和 "LOWW"。然而，一个表达式中只允许出现一个外部参考项，并且必须与 "+" 相连。
下列显示了可进行组合的运算符和项，按照重定位属性进行分类。

表 4-8. 按重定位属性组合的运算符和项 （外部参考项）

运算符类型	项的可重定位属性				
	X:ABS Y:EXT	X:EXT Y:ABS	X:REL Y:EXT	X:EXT Y:REL	X:EXT Y:EXT
X + Y	E	E	-	-	-
X - Y	-	E	-	-	-
+ X	A	E	R	E	E
HIGH X	A	注 1	注 2	注 1	注 1
LOW X	A	注 1	注 2	注 1	注 1
HIGHW X	A	注 1	注 2	注 1	注 1
LOWW X	A	注 1	注 2	注 1	注 1
MASK (X)	A	-	-	-	-
DATAPOS X.Y	-	-	-	-	-
BITPOS X.Y	-	-	-	-	-
MASK (X.Y)	-	-	-	-	-
DATAPOS X	A	E	R	E	E
BITPOS X	A	E	A	E	E

ABS ： 绝对项
EXT ： 外部参考项
REL ： 可重定位项
A ： 结果为绝对项
E ： 结果为外部参考项
R ： 结果为可重定位项
- ： 不可运算

注 **1.** 如果 X 和 Y 不是外部参考项时，只能通过 HIGH、LOW、HIGHW、LOWW、DATAPOS 或 BITPOS 进行运算。

2. 如果 X 和 Y 不是可重定位项时，只能通过 HIGH、 LOW、 HIGHW、 LOWW 或 DATAPOS 进行运算。

(2) 运算符和符号属性

组成表达式的每一项除了重定位属性之外还具有符号属性。
如果把项按符号属性进行分类，可分为两大类：NUMBER 项和 ADDRESS 项。
下表显示了表达式中使用的符号属性的类型和相关项。

表 4-9. 运算中的符号属性

符号属性类型	相关项
NUMBER 项	- 带 NUMBER 属性的符号 - 常量
ADDRESS 项	- 带 ADDRESS 属性的符号 - "$" 表示位置计数器

下列显示了可进行组合的运算符和项，按照符号属性进行分类。

表 4-10. 按符号属性组合的运算符和项

运算符类型	项的符号属性			
	X:ADDRESS Y:ADDRESS	X:ADDRESS Y:NUMBER	X:NUMBER Y:ADDRESS	X:NUMBER Y:NUMBER
X + Y	A	A	A	N
X - Y	N	A	N	N
X * Y	N	N	N	N
X / Y	N	N	N	N
X MOD Y	N	N	N	N
X SHL Y	N	N	N	N
X SHR Y	N	N	N	N
X EQ Y	N	N	N	N
X LT Y	N	N	N	N
X LE Y	N	N	N	N
X GT Y	N	N	N	N
X GE Y	N	N	N	N
X NE Y	N	N	N	N
X AND Y	N	N	N	N
X OR Y	N	N	N	N
X XOR Y	N	N	N	N
NOT X	A	A	N	N
+ X	A	A	N	N
- X	A	A	N	N
HIGH X	A	A	N	N
LOW X	A	A	N	N
HIGHW X	A	A	N	N
LOWW X	A	A	N	N
DATAPOS X	A	A	N	N
MASK X	N	N	N	N

ADDRES ： ADDRESS 项
NUMBER ： NUMBER 项
A ： 结果为 ADDRESS 项
N ： 结果为 NUMBER 项
- ： 不可运算

(3) 如何检查运算限制

下列示例为如何中断重定位属性和符号属性的运算。

```
BR      $TABLE + 5H
```

这里的 "TABLE" 假设为可重定位代码段中的已定义标签。

(a) 运算和重定位属性

"TABLE + 5H" 是一个可重定位项 + 一个绝对项，因此可使用 " 表 4-7.　按重定位属性组合的运算符和项（可重定位项） " 规则。

运算符类型　　...　X + Y
项的重定位属性　　...　X:REL, Y:ABS

因此，可以知道结果是 "R"， , 或更具体点就是可重定位项。

(b) [运算符和符号属性]

"TABLE + 5H" 是一个 ADDRESS 项 + 一个 NUMBER 项，因此可使用 " 表 4-10.　按符号属性组合的运算符和项 " 规则。

运算符类型　　...　X + Y
项的重定位属性　　...　X:ADDRESS, Y:NUMBER

因此，可以知道结果是 "A"， , 或更具体点就是 ADDRESS 项。

4.1.13　绝对表达式的定义

绝对表达式是指带有值的表达式，该值在汇编时间内通过计算得到。

下列属于绝对表达式的类别：

- 常量
- 只由常量组成的表达式 （常量表达式）
- 常量、由常量表达式定义的 EQU 符号和 SET 符号
- 上述运算的表达式

备注　　只有符号可进行反向参照。

4.1.14　比特位置说明符

通过使用 (.) 比特位置说明符使得比特访问得以实现。

(1) 描述格式

```
X[Δ].[Δ]Y
 └──────┘
位项
```

X （第一项）		Y （第二项）
通用寄存器	A	表达式 （0 - 7）
控制寄存器	PSW	表达式 （0 - 7）
特殊功能寄存器	sfr 注	表达式 （0 - 7）
存储器	[HL] 注	表达式 （0 - 7）

注　　关于特定描述的详细信息，请查阅每个器件的用户手册。

(2) 功能

- 第一项是字节地址，第二项是比特位置。这使得访问指定位成为可能。

(3) 说明

- 使用比特位置说明符的表达式称为位项。
- 比特位置说明符不受运算符优先顺序的影响。左边定义为项 1，右边定义为项 2。

- 下列限制用于第一项：
  - 可指定 NUMBER 或 ADDRESS 属性的表达式、 SFR 名称支持的 8 位读取或寄存器名称 (A)。
  - 如果第一项为绝对表达式，那么其区间必须为 0H 至 0FFFFFH。
  - 可指定外部参考符号。
- 下列限制用于第二项：
  - 表达式的值必须在 0 至 7 范围内。当超出该范围时，会产生错误。
  - 可仅指定绝对 NUMBER 属性的表达式。
  - 不可指定外部参考符号。

(4) 运算和重定位属性

- 下表显示了按重定位属性组合的项 1 和项 2。

组合项 X：	ABS	ABS	REL	REL	ABS	EXT	REL	EXT	EXT
组合项 Y：	ABS	REL	ABS	REL	EXT	ABS	EXT	REL	EXT
X.Y	A	-	R	-	-	E	-	-	-

ABS ： 绝对项
REL ： 可重定位项
EXT ： 外部参考项
A ： 结果为绝对项
E ： 结果为外部参考项
R ： 结果为可重定位项
- ： 不可运算

(5) 位符号的值

- 当采用位于操作数区域 EQU 命令的比特位置说明符对位符号进行定义时，位符号的值显示如下：

操作数类型	符号值
A.bit 注 2	1.bit
PSW.bit 注 2	FFFFAH.bit
sfr 注 1.bit 注 2	FFFXXH.bit 注 3
expression.bit 注 2	XXXXXH.bit 注 4

注 **1.** 关于详细的描述，请参阅各器件的用户手册。
2. bit = 0 - 7
3. FFFXXH 是 sfr 地址
4. XXXXXH 是表达式的值

(6) 示例

```
SET1    0FFE20H.3
SET1    A.5
CLR1    P1.2
SET1    1 + 0FFE30H.3     ; 等于 0FFE31H.3
SET1    0FFE40H.4 + 2     ; 等于 0FFE40H.6
```

4.1.15　标识符

标识符是一种名称，用于符号、标签和宏等。

标识符的描述遵循下列基本规则。

- 标识符由字母数字字符和作为字符使用的符号 （?、 @ 和 _）组成。
  但是，第一个字符不能为数字 （0 至 9）。
- 保留字不能用作标识符。
  对于保留字的定义，请参阅 "4.5　保留字 "。
- 汇编器区分大小写。

4.1.16　操作数特性

指令和命令需要一个或多个在大小和地址方面不同的操作数，这些操作数在所需的操作数值和操作数符号属性内。

例如，指令 "MOV r , #byte" 的功能是把 "byte" 所指的送到寄存器 "r"。由于寄存器是 8 位的，因此？字节？的数据大小必须等于或小于 8 位。

由于第二个操作数 （100H）用 8 位不足以表达，因此语句搪 OV R0 , #100H

因此，在描述操作数时，必须把下列几点铭记于心。

- 指令 （数值、名称和标签）中的操作数是否符合大小和地址范围。
- 指令 （名称和标签）中的操作数的符号属性是否合适。

(1) 操作数值的大小和地址范围

作为指令操作数的数值、名称和标签在大小和地址范围上存在条件限制。

对于指令，操作数的大小和地址范围受操作数表达的限制。对于命令，则受到命令类型的限制。

限制条件如下。

表 4-11.　指令操作数值范围

操作数表达	取值范围	
byte	8 位值　　：0H 至 0FFH	
字	word [B] word [C] word [BC]	(1)　数字常量和 NUMBER 属性的符号 0H 至 FFFFH (2)　ADDRESS 属性的符号 在下列区域中的任意一个 - F0000H 至 FFFFFH - 当 MAA=0 时，区域 （01000H 至 0xxxxH）被镜像到 RAM 空间；当 MAA=1 时，区域 （11000H 至 1xxxxH）[注 1] 被镜像到 RAM 区间
	ES : word [B] ES : word [C] ES : word [BC]	(1)　数字常量和 NUMBER 属性的符号 0H 至 FFFFH (2)　ADDRESS 属性的符号 0H 至 FFFFFH
	除上面之外	16 位值　　：0H 至 FFFFH
saddr	FFE20H 至 FFF1FH [注 4]	
saddrp	FFE20H 至 FFF1FH 偶数[注 4]	
sfr	FFF20H 至 FFFFFH：特殊功能寄存器符号 （SFR 符号）、数字常量和 NUMBER 属性符号[注 5]	
sfrp	FFF20H 至 FFFFFH：特殊功能寄存器符号 （支持 16 位运算的 SFR 符号，仅偶数）、数字常量和 NUMBER 属性符号[注 5]	

操作数表达	取值范围	
addr20	!!addr20	0H 至 FFFFFH
	$addr20	0H 至 FFFFFH, 且分支目标从分支指令或调用指令的下一地址起的（-80H）至 （+7FH）范围内
	$!addr20	0H 至 FFFFFH, 且分支目标从分支指令或调用指令的下一地址起的（-8000H）至 （+7FFFH）范围内
addr16	!addr16 （BR 和 CALL 指令）	0H 至 FFFFH （数字常量和符号可指定的范围相同）
	!addr16 注 2 （除 BR 和 CALL 指令以外）	(1) 数字常量和 NUMBER 属性的符号注 3 0H 至 FFFFH (2) ADDRESS 属性的符号注 3 下列中的一个： - F0000H 至 FFFFFH - 当 MAA=0 时，该区域被镜像到 RAM 空间 （例如：01000H 至 0xxxxH ；或当 MAA=1 时，该区域被镜像到 RAM 空间 （例如：11000H 至 1xxxxH）注 1
	ES:!addr16	(1) 数字常量或 NUMBER 属性的符号注 3 0H 至 FFFFH (2) ADDRESS 属性的符号注 3 0H 至 FFFFFH
	!addr16.bit	(1) DBIT 符号、 SFBIT 或 SABIT 属性的位符号，位符号由 EQU 命令定义 （但是只有当操作数包括含有 ADDRESS 属性的符号时） 下列中的一个： - F0000H 至 FFFFFH - 当 MAA=0 时，该区域被镜像到 RAM 空间 （例如：01000H 至 0xxxxH ；或当 MAA=1 时，该区域被镜像到 RAM 空间 （例如：11000H 至 1xxxxH）注 1 (2) 除去上述情况的位符号 0H 至 FFFFH
	ES:!addr16.bit	(1) DBIT 符号、 SFBIT 或 SABIT 属性的位符号，位符号由 EQU 命令定义 （但是只有当操作数包括含有 ADDRESS 属性的符号时） 0H 至 FFFFFH (2) 除去上述情况的位符号 0H 至 FFFFH
addr5	0080H 至 00BFH （CALLT 表格区域，仅限偶数值）	
bit	3 位值 ： 0 至 7	
n	2-bit value ： 0 to 3	

注 **1.** 被镜像到 RAM 空间的区域的地址范围根据器件不同而不同。关于详细信息，请参考目标器件的用户手册。

2. 为了把 sfr 或 2ndsfr 作为操作数进行描述，可把它们指定为 !sfr 或 !2ndsfr。这些是 !addr16 作为操作数输出的代码。
对于 2ndsfr 进行描述时可以不加 “!”。将输出相同的 !addr16 操作数代码。

3. 在 16 位数据中只能指定偶数位的地址。

4. 为了和 78K0 兼容， FE20H 至 FF1FH 的范围只能用于指定数字常量和 NUMBER 属性的符号。

5. 针对数字常量和 NUMBER 属性的符号，在指定地址不执行 SFR 读写访问的检查。

下列示例解释了为什么 addr16 或字操作数的符号属性会对指定操作数范围有影响。

更多关于符号属性的信息，请查阅 "(d) 符号属性 "。

(a) !addr16 （除 BR 和 CALL 以外的指令）

该部分解释了为什么用于指定 !addr16 操作数 （除 BR 和 CALL 以外的指令）的值的范围在 (1) 数字常量和 NUMBER 属性的符号以及 (2)ADDRESS 属性的符号之间存在差异。
下列为一个示例。

```
NUMBER0          EQU     0F100H                  ; (a)
NUMBER1          EQU     0F102H
NUMBER2          EQU     0F103H

D0               DSEG    AT  0FF100H
ADDRESS0:        DS      1
ADDRESS1:        DS      1
ADDRESS2:        DS      1

                 CSEG
                 MOV     !NUMBER0, A             ; (b)
                 MOV     !0F100H, A              ; (c)
                 MOV     !ADDRESS0, A            ; (d)
```

(a) 行包含一个 NUMBER 属性的符号。下面解释了当该 NUMBER 属性的字符被指定为 !addr16 操作数时的情况。
指令集中的插 OV !addr16, A。在示例中的 (b) 行，寄存器 A 中的值发送到地址 0FF100H。 (a) 行中的 NUMBER 属性的符号可由 (c) 行中的值取代。即， NUMBER0 符号 （指定为 !addr16 操作数的 NUMBER 属性的字符）和数值 0F100H 均表示相同地址，称为 “0F100H”。至于范围，作为 !addr16 操作数的 NUMBER 属性符号 （除 BR 和 CALL 以外的指令）具有 0H 至 FFFFH 的值。这些值指定的地址为 F0000H 至 FFFFFH。

接下来解释当相同的过程在 ADDRESS0 标签和 ADDRESS 属性的符号上执行时的情况。
addr16 的范围是 0000H 至 FFFFH，此时 (d) 行中 ADDRESS 符号的值在 RAM 存储器空间内的地址为 FxxxxH 至 FFFFFH。通常，这将产生一个错误。因此在程序开发时，应在操作数允许范围内 （F0000H 至 FFFFFH）提供操作数标签，如 ADDRESS0 （ADDRESS 属性的符号）。
总之， !addr16 操作数 （除 BR 和 CALL 以外的指令）即 ADDRESS 属性的符号可具有 F0000H 至 FFFFF 的值。这使得可以像指定 !addr16 操作数一样指定它们。

此外，当 ROM 区域镜像到 RAM 区域时，可支持 !addr16。
如下示例所示。

```
M0              CSEG    MIRRORP
ADDRESS0:       DB      12H
ADDRESS1:       DB      34H
ADDRESS2:       DB      56H
                CSEG
                MOV     A, !ADDRESS0            ; (e)
```

MO 段位于 ROM 空间中镜像到 RAM 空间的那部分。当 MAA=0 时， MO 段位于 01000H 至 0xxxxH ；当 MAA=1 时，位于 11000H 至 0xxxxH。因为这样，在 (e) 行中 ADDRESS0 符号的值的范围从 01000H 至 0xxxxH 或从 11000H 至 1xxxxH。在进行程序开发时，像 (e) 行中的符号那样可作为镜像段中符号的参考。其中 !addr16 的范围是 01000H 至 0xxxxH 或 11000H 至 1xxxxH。

总之，带有 ADDRESS 属性的 !addr16 符号 （除 BR 和 CALL 以外的指令）可具有 01000H 至 0xxxxH 或 11000H 至 0xxxxH 的值。这使得可以像指定 !addr16 操作数一样指定它们。

(b) ES:!addr16

该部分解释了为什么用于指定 ES:!addr16 操作数的值的范围在 (1) 数字常量和 NUMBER 属性的符号以及 (2)ADDRESS 属性的符号之间存在差异。

下列为一个示例。

```
DATA            CSEG    AT      12345H
ADDRESS0:       DB      12H
ADDRESS1:       DB      34H
ADDRESS2:       DB      56H


                CSEG
                MOV     ES, #HIGHW ADDRESS0     ; (f)
                MOV     A, ES:!ADDRESS0         ; (g)
```

(f) 和 (g) 行的语句把数据从 ADDRESS0 发送至寄存器 A。

addr16 的范围是 0000H 至 FFFFH。但 ADDRESS0 符号在 (g) 行的值为 12345H。通常，这将产生一个错误。

因此，在程序开发中，为了可写成像 (g) 行这样的语句，需允许 ADDRESS0 的范围是 0H 至 FFFFFH。

总之， ES:!addr16 操作数即 ADDRESS 属性的符号，可如其本身进行指定。从 0H 至 FFFFFH 的值也可如其本身进行指定。

(c) !addr16.bit, ES:!addr16.bit

该部分解释了为什么 !addr16.bit 和 ES:!addr16.bit 值的范围在 (1) DBIT 符号、 SFBIT 和 SABIT 属性的位符号、由 EQU 命令定义的位符号 （但仅限操作数中包含 ADDRESS 属性的符号时）和 (2) 其他符号之间有所差异。

如下示例所示。

```
                BSEG
DBITSYM0        DBIT                            ; (h)
DBITSYM1        DBIT
DBITSYM2        DBIT

BIT1_PM0        EQU     PM0.1                   ; (i)
BIT2_P0         EQU     P0.2                    ; (j)

                DSEG
ADDRESS0:       DS      1
ADDRESS1:       DS      1
ADDRESS2:       DS      1

ADR_BIT0        EQU     ADDRESS0.0              ; (k)
ADR_BIT1        EQU     ADDRESS0.1
ADR_BIT2        EQU     ADDRESS0.2

                CSEG
                SET1    !DBITSYM0               ; (l)
                SET1    !BIT1_PM0               ; (m)
                SET1    !BIT2_P0                ; (n)
                SET1    !ADR_BIT0               ; (o)
```

正如 (l) 至 (o) 行所述， (h) 行描述的 DBIT 符号、 (i) 和 (j) 行描述的 SFBIT 属性和 SABIT 属性的位符号以及 (k) 行中的由 EQU 命令定义的位符号 （仅限操作数中包含 ADDRESS 属性的符号时）作为操作数用于 !addr16.bit 是可能的，因此根据描述的符号属性，各值的范围是不同的。

同样， ES:!addr16.bit 操作数值的范围也根据符号属性而定。

(d) 字

该部分解释了为什么字操作数值的范围在 (1) 数字常量和 NUMBER 属性的符号和 (2)ADDRESS 属性的符号之间存在差异。

如下示例所示。

```
                DSEG
ADDRESS0:       DS      1
ADDRESS1:       DS      1
ADDRESS2:       DS      1
                CSEG
                MOV     B, #0
                MOV     ADDRESS0[B], A          ; (p)
                MOV     C, #1
                MOV     ADDRESS0[C], A          ; (q)
                MOVW    BC, #2
                MOV     ADDRESS0[BC], AX        ; (r)
```

由于在一个字需要操作数时，通常会为其指定标签 （ADDRESS 属性的字符），如 (p) 至 (r) 行中的 word[B]、 word[C] 和 word[BC] 指令，因此可以像 !addr16 一样，通过编写代码即可方便地指定标签。同样，编写 ES:word[B]、 ES:word[C] 和 ES:word[BC] 指令代码也很简单。

表 4-12. 命令操作数值的范围

指令 类型	命令	取值范围
段的定义	CSEG AT	0H 至 0FFFFFH （不包括 SFR 和 2ndSFR）
	DSEG AT	0H 至 0FFFFFH （不包括 SFR 和 2ndSFR）
	BSEG AT	0H 至 0FFFFFH （不包括 SFR 和 2ndSFR）
	ORG	0H 至 0FFFFFH （不包括 SFR 和 2ndSFR）
符号定义	EQU	20 位值 0H 至 FFFFFH
	SET	20 位值 0H 至 FFFFFH
存储器初始化和区域保留	DB	8 位值 0H 至 FFH
	DW	16 位值 0H 至 FFFFH
	DG	20 位值 0H 至 FFFFFH
	DS	16 位值 0H 至 FFFFH
自动分支指令选择	BR/CALL	0H 至 FFFFFH

(2) 指令所需操作数的大小

指令可分为机器指令和命令。当需要立即数据或符号时，所需操作数的大小根据指令或命令而不同。当源代码指定的数据大于所需的操作数时，将发生错误。

表达式采用无符号的 32 位数值进行运算。当运算结果超出 0FFFFFFFFH （32 位）时，将会产生警告信息。然而，当指定可重定位或外部符号为操作数时，该值不能由汇编器决定。在这种情况下，链接器将决定值并进行范围检查。

(3) 操作数的符号属性和重定位属性

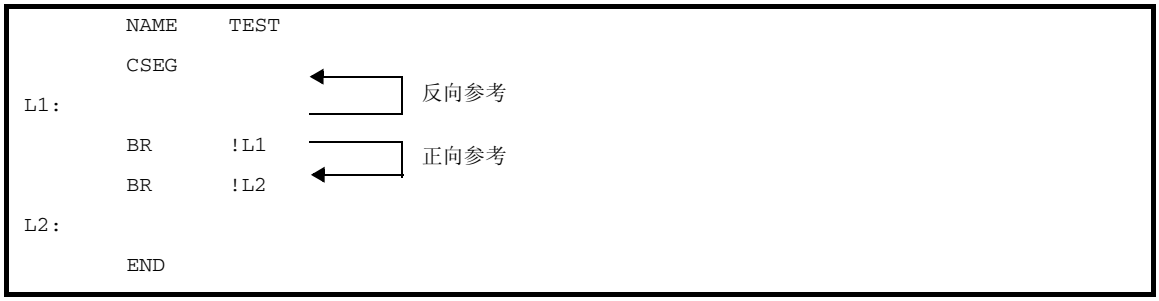
当名称、标签和 $（表示位置计数器）作为指令操作数进行描述时，它们是否可以作为操作数进行描述。这取决于符号属性和重定位属性（请查阅 “4.1.12 运算中的限制 ”）。

当名称和标签作为指令操作数进行描述时，它们是否可以作为操作数进行描述。这取决于参考的方向。

以名称和标签作为参考方向的话，可以是反向参考或正向参考。

- 反向参考：一个名称或标签作为操作数进行参考，在名称或标签之前的一行定义
- 正向参考：一个名称或标签作为操作数进行参考，在名称或标签之后的一行定义

< 示例 >



下列显示了符号属性、重定位属性以及名称和标签的参考方向。

表 4-13. 作为操作数描述的符号性质

	符号属性	NUMBER		ADDRESS				NUMBER ADDRESS		sfr 保留字[注 1]	
	重定位属性	属性项		属性项		可重定位项		外部参考项			
	参考模式	反向	正向	反向	正向	反向	正向	反向	正向	sfr	2ndsfr
说明格式	byte	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	字	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	saddr	OK	OK	OK	OK	OK	OK	OK	OK	OK[注 3]	NG
	saddrp	OK	OK	OK	OK	OK	OK	OK	OK	OK[注 2.4]	NG
	sfr	OK	OK	NG	NG	NG	NG	NG	NG	OK[注 2 和 5]	NG
	sfrp	OK	OK	NG	NG	NG	NG	NG	NG	OK[注 2 和 6]	NG
	addr20	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	addr16	OK	OK	OK	OK	OK	OK	OK	OK	OK[注 7]	OK[注 7]
	addr5	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	位	OK	OK	NG	NG	NG	NG	NG	NG	NG	NG
	n	OK	OK	OK	OK	NG	NG	NG	NG	NG	NG

正向 : 正向参考
反向 : 反向参考
OK : 可进行描述
NG : 错误
- : 不可描述

注 **1.** 作为 EQU 命令操作数的符号指定 sfr 或 sfrp （saddr 和 sfr 不重叠的 sfr 区域）时只能反向参考。禁止正向参考。

2. 如果位于 saddr 区域的 sfr 保留字用于指令 （in which a combination of sfr/sfrp changed from saddr/saddrp exists in the operand combination）描述的话，代码将作为 saddr/saddrp 输出。

3. saddr 区域的 sfr 保留字

4. saddr 区域的 sfrp 保留字

5. 只有 sfr 保留字允许 8 位读取

6. 只有 sfr 保留字允许 16 位读取

7. !sfr 和 !2ndsfr 只能指定为 !addr16 操作数 （除 BR 和 CALL 以外的指令）。

表 4-14. 作为命令操作数描述的符号性质

	符号属性		NUMBER		ADDRESS, SADDR						BIT					
	重定位 属性		属性项		属性项		可重定位项		外部参考项		属性项		可重定位项		外部参考项	
	参考模式		反向	正向	反向	正向	反向	正向	反向	正向	反向	正向	反向	正向	反向	正向
命令	ORG		OK 注 1	-	-	-	-	-	-	-	-	-	-	-	-	-
	EQU 注 2		OK	-	OK	-	OK 注 3	-	-	-	OK	-	OK 注 3	-	-	-
	SET		OK 注 1	-	-	-	-	-	-	-	-	-	-	-	-	-
	DB	容量	OK 注 1	-	-	-	-	-	-	-	-	-	-	-	-	-
		初始值	OK	OK	OK	OK	OK	OK	OK	OK	-	-	-	-	-	-
	DW	容量	OK 注 1	-	-	-	-	-	-	-	-	-	-	-	-	-
		初始值	OK	OK	OK	OK	OK	OK	OK	OK	-	-	-	-	-	-
	DG	容量	OK 注 1	-	-	-	-	-	-	-	-	-	-	-	-	-
		初始值	OK	OK	OK	OK	OK	OK	OK	OK	-	-	-	-	-	-
	DS		OK 注 4	-	-	-	-	-	-	-	-	-	-	-	-	-
	BR/CALL		OK	-	-	-	-	-	-	-	-	-	-	-	-	-

正向 : 正向参考
反向 : 反向参考
OK : 可进行描述
- : 不可描述

注 **1.** 只能对绝对表达式进行描述。

2. 如果对包括下列模式之一的表达式进行描述，将出现错误。

- ADDRESS 属性 - ADDRESS 属性
- 与 ADDRESS 属性相关操作数的 ADDRESS 属性

- HIGH，绝对 ADDRESS 属性
- LOW，绝对 ADDRESS 属性
- HIGHW，绝对 ADDRESS 属性
- LOWW，绝对 ADDRESS 属性
- DATAPOS，绝对 ADDRESS 属性
- MASK，绝对 ADDRESS 属性
- 通过上述 8 种模式的优化，会对运算结果产生影响。

3. 由 HIGH/LOW/HIGHW/LOWW/DATAPOS/MASK 运算符组成的项是不允许具有可重定位项的。
4. 参见 “4.2.4 存储器初始化、区域保留命令 "。

4.2 指令

本章说明指令。

命令是 78K0R 汇编器执行一系列处理时所必须的所有类型的指令。

4.2.1 概要

作为汇编结果，指令被翻译成目标代码 （机器语言），而基本上命令不会转换为目标代码。

命令主要包含下述功能：

- 对源程序进行描述
- 初始化存储器和保留存储空间
- 为汇编器和链接器执行处理提供所需信息

下表显示了命令的类型。

表 4-15. 命令列表

类型	伪指令
段定义命令	CSEG, DSEG, BSEG, ORG
符号定义命令	EQU, SET
存储器初始化、区域保留命令	DB, DW, DG, DS, DBIT
链接指令	EXTRN, EXTBIT, PUBLIC
目标模块名的声明指令	NAME
分支指令自动选择指令	BR, CALL
宏指令	MACRO, LOCAL, REPT, IRP, EXITM, ENDM
汇编终止指令	END

以下部分详细解释每条命令。

在每条指令格式的描述中，“[ ]”表示方括号中的参数可从规范中省略，“...”表示相同格式的重复描述。

4.2.2 段定义命令

源模块通过分割每个段单元进行描述。

段命令用于定义那些？段？。

有四种类型的段。

- 代码段
- 数据段
- 位段
- 绝对程序段

段类型决定了其在存储器区域中的映射。

下面显示了每个段的定义方法和在存储器地址内的映射。

表 4-16. 段的定义方法和存储器地址位置

段类型	定义方法	存储器地址位置
代码段	CSEG 命令	内部或外部 ROM 地址区域
数据段	DSEG 命令	内部或外部 RAM 地址区域
位段	BSEG 命令	内部 RAM saddr 区域
绝对程序段	通过 CSEG、DSEG 和 BSEG 命令指定位置地址 （AT 位置地址）至重定位属性	已定义的地址

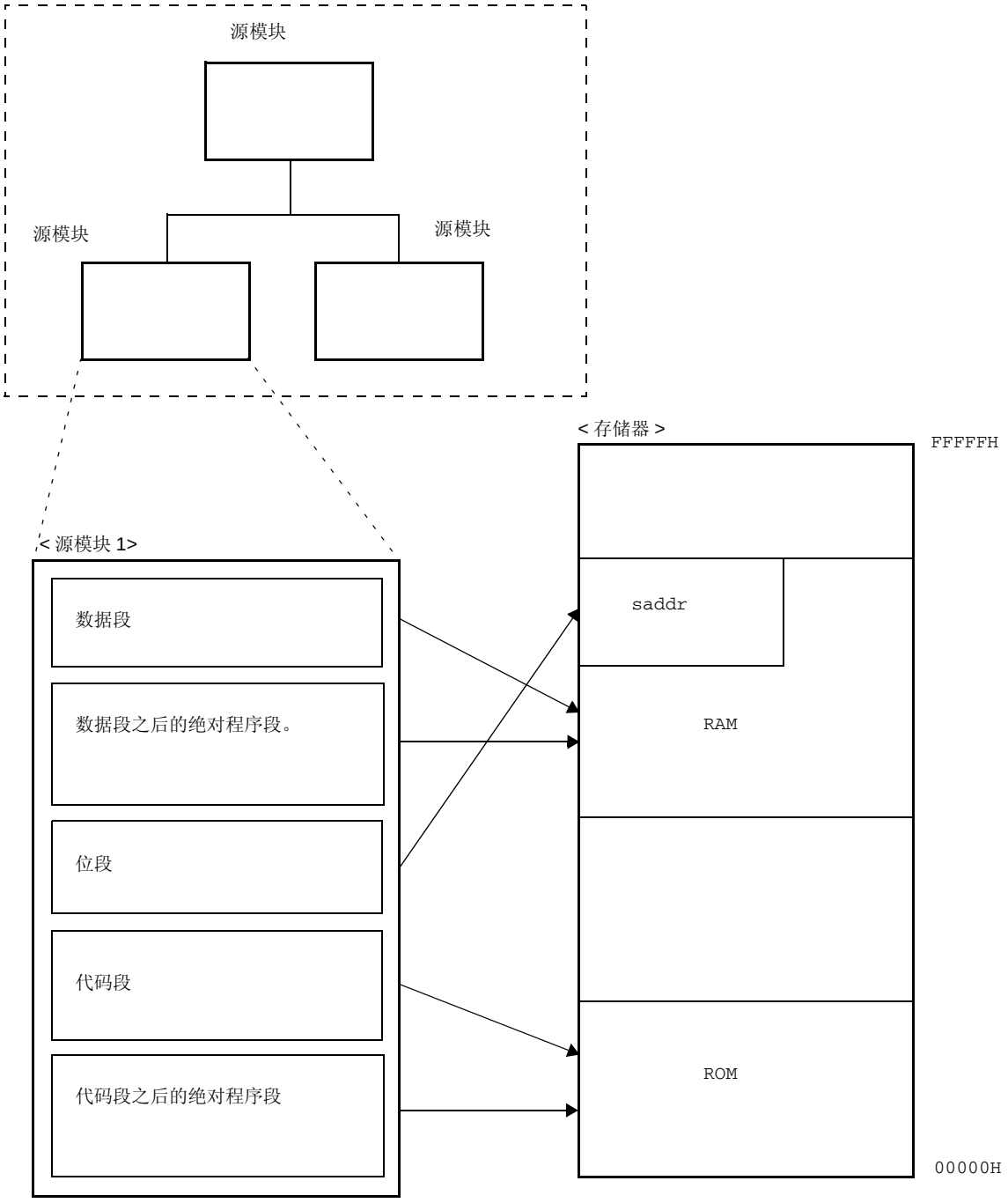
当用户希望在存储器中设置地址映射时需定义绝对程序段。针对堆栈区域，用户必须设置堆栈指针并在数据段中保留一定区域。

此外，段不能位于下列区域。

选项字节区域	C0 至 C2H （用户选项字节） C3H （片上调试选项字节）
当指定安全 ID 时	C4H 至 CDH
当使用片上调试函数时	02H 至 03H， CE 至 D7H （用于片上调试） 用户可通过 -go 选项指定程序区域的起始地址

段映射的示例如下所示。

图 4-6. 段存储器映射



可获得下列段中定义的命令。

控制指令	概要
CSEG	说明汇编器中代码段的起始位置
DSEG	说明汇编器中数据段的起始位置
BSEG	说明汇编器中位段的起始位置
ORG	由定位计数器的操作数定义的表达式值。

CSEG

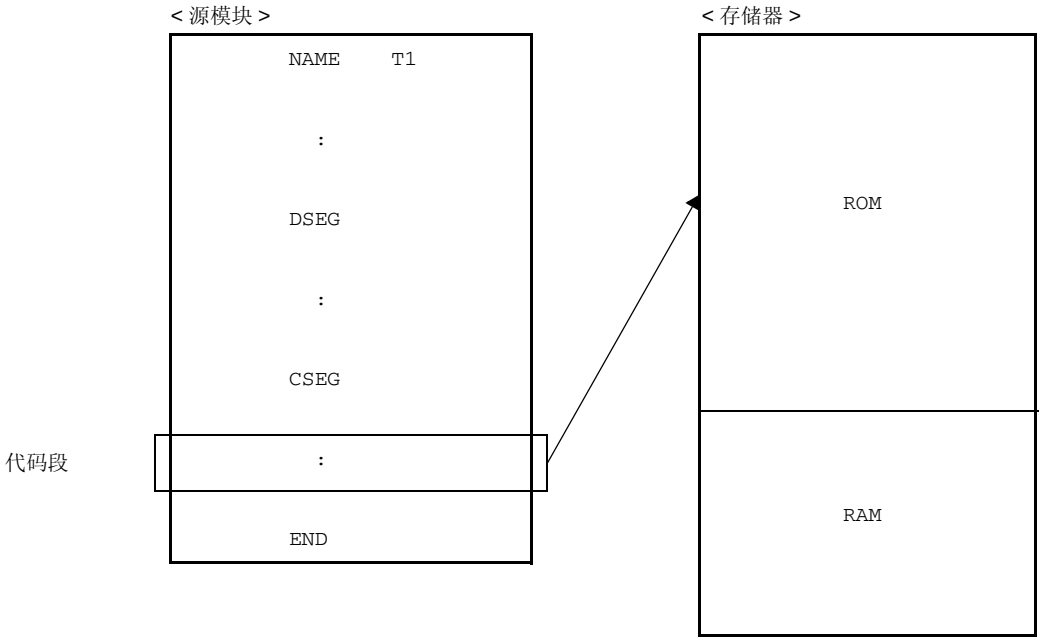
说明汇编器中代码段的起始位置。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*段名称*]	CSEG	[*重定位属性*]	[; *注释*]

[功能]

- CSEG 命令说明汇编器中代码段的起始位置。
- CSEG 命令后描述的全部指令属于代码段直到其遇到段定义命令 （CSEG、 DSEG、 BSEG 或 ORG）或 END 命令为止，最后这些指令转换为机器语言后存放于 ROM 地址中。



[使用]

- CSEG 命令用于描述指令，并在代码段定义 DB、 DW 等命令。
  但是，为了从固定地址重新定位代码段，必须在操作数区域内把 “*绝对表达式*？描述为重定位属性。
- 描述诸如子程序的单一功能单元，应定义为单个代码段。
  如果单元相对较大或子程序运用较广 （如，可用于开发其他程序）的话，子程序应定义为单个模块。

[说明]

- 代码段的起始地址可由 ORG 命令定义。
  也可通过描述重定位属性 “*绝对表达式*？来定义。
- 重定位属性定义了代码段的位置地址范围。

表 4-17. CSEG 的重定位属性

重定位属性	格式描述	说明
CALLT0	CALLT0	让汇编器对已定义段进行定位，这样段的起始地址变为 2 的倍数，地址范围是 00080H 至 000BFH。
FIXED	FIXED	让汇编器把已指定段的起始位置指定在 000C0H 至 0FFFFH 的地址范围
BASE	BASE	让汇编器把已指定段的起始位置指定在 000C0H 至 0FFFFH 的地址范围
AT	AT *绝对表达式*	让汇编器把已定义段定位至绝对地址 （SFR 和 2ndSFR 除外）。
UNIT	UNIT	让汇编器把已定义段定位至任何地址 （存储器区域 “ROM” 内的 000C0H 至 EFFFFH 地址）。
UNITP	UNITP	让汇编器把已定义段定位至任何地址，这样起始地址可能为偶数 （存储器区域 “ROM” 内的 000C0H 至 EFFFFH 地址）。
IXRAM	IXRAM	让汇编器把已定义段定位至任何地址 （存储器区域揜 OM
SECUR_ID	SECUR_ID	这是安全 ID 指定的属性。除安全 ID 外未指定。 让汇编器把已指定段的起始位置指定在 000C4H 至 000CDH 的地址范围。
PAGE64KP	PAGE64KP	让汇编器把已定义段定位至揜 OM 同名段但位于不同文件中的不可合并。
UNIT64KP	UNIT64KP	让汇编器把已定义段定位至揜 OM 同名段可合并。
MIRRORP	MIRRORP	让汇编器把已定义段定义至 RAM 空间的镜像区域，当 MAA = 0 时，范围为 01000H 至 0xxxxH ；当 MAA = 1 时，范围为 11000H 至 1xxxxH。[注]
OPT_BYTE	OPT_BYTE	其使用用户选项字节和片上调试指定属性。除用户选项字节和片上调试外未定义。 让汇编器把已指定段的起始位置指定在 000C0H 至 000C3H 的地址范围。

注 在 RAM 空间镜像的地址范围根据所选器件而不同。

- 若未指定重定位属性至代码段，汇编器将假设揖 NIT
- 若定义了在 “表 4-17. CSEG 的重定位属性” 中以外的重定位属性，汇编器将输出错误并假设 “UNIT” 已指定。若每个代码段的大小超出重定位属性指定的区域，将产生错误。
- 如果由重定位属性揖 T

- 通过 CSEG 命令在符号区域描述段名称，可进行代码段命名。如果未设定段名至代码段，汇编器将自动给代码段一个缺省名。
  下列为代码段的缺省段名。

重定位 属性	缺省段名
CALLT0	?CSEGT0
FIXED	?CSEGFX
UNIT (或省略)	?CSEG
UNITP	?CSEGUP
IXRAM	?CSEGIX
BASE	?CSEGB
SECUR_ID	?CSEGSI
PAGE64KP	?CSEGP64
UNIT64KP	?CSEGU64
MIRRORP	?CSEGMIP
OPT_BYTE	?CSEGOB0
AT	段名不可省略。

- 当 C 编译器输出的缺省段中，下列段的大小为 0 时，链接器将更改重定位属性。

片断名	重定位 属性	大小为 0 时的重定位属性
@@CALT	CSEG CALLT0	CSEG UNIT
@@CNST	CSEG MIRRORP	CSEG UNIT

- 当重定位属性为 AT 时，若省略段名称的话将产生错误。
- 如果两个或多个代码段具有相同重定位属性 （除 AT）的话，这些代码段可能具有相同的段名。
  这些同名的代码段在汇编器处理时作为一个代码段处理。
  如果同名代码段的重定位属性不同的话，将产生错误。因此，每个重定位属性的代码段数量为 1。
- 可把代码段分为单元进行描述。相同重定位属性和同名的代码段可作为一系列段由汇编器在一个模块中描述。

注意事项 1. 对于重定位属性为 AT 的代码段的描述不能分为单元进行。

2. 有必要的话，可插入一个字节的间隔，这样由重定位属性 CALLT0 指定的地址可能为偶数。

- 在两个或更多不同模块中的同名数据段，只有当其重定位属性为 UNIT、 CALLT0、 FIXED、 UNITP、 BASE、 PAGE64KP、 UNIT64KP、 MIRRORP 或 SECUR_ID 时才能定义，并在链接时合并为一个数据段。
- 段名称不能引用为符号。
- 汇编器能够输出的总段数为 256 个不同的名称，包括 ORG 命令中定义的名称。同名段算作一个。
- 可作为段名的最大字符数为 8。
- 段名中的大小写字符是有区别的。

- 通过使用 OPT_BYTE 指定用户选项字节和片上调试。
  如果设定芯片的选项字节不具有选项字节特性时，将产生错误。
  当不是指定给芯片的用户选项字节具有用户选项字节特性时，把缺省段 “?CSEGOB0”定义到每个地址并读取器件文件作为初始值。

[示例]

```
        NAME    SAMP1
C1      CSEG            ; (1)

C2      CSEG    CALLT0  ; (2)

        CSEG    FIXED   ; (3)

C1      CSEG    CALLT0  ; (4)   <- Error

        CSEG                            ; (5)
        END
```

(1) 汇编器把段名译为 “C1”，重定位属性译为 “UNIT”。

(2) 汇编器把段名译为 “C2”，重定位属性译为 “CALLT0”。

(3) 汇编器把段名译为 “?CSEGFX”，重定位属性译为 “FIXED”。

(4) 由于在 (1) 当中，段名 “C1”被定义为重定位属性 “UNIT”，因此产生错误。

(5) 汇编器把段名译为 “C1”，重定位属性译为 “UNIT”。

DSEG

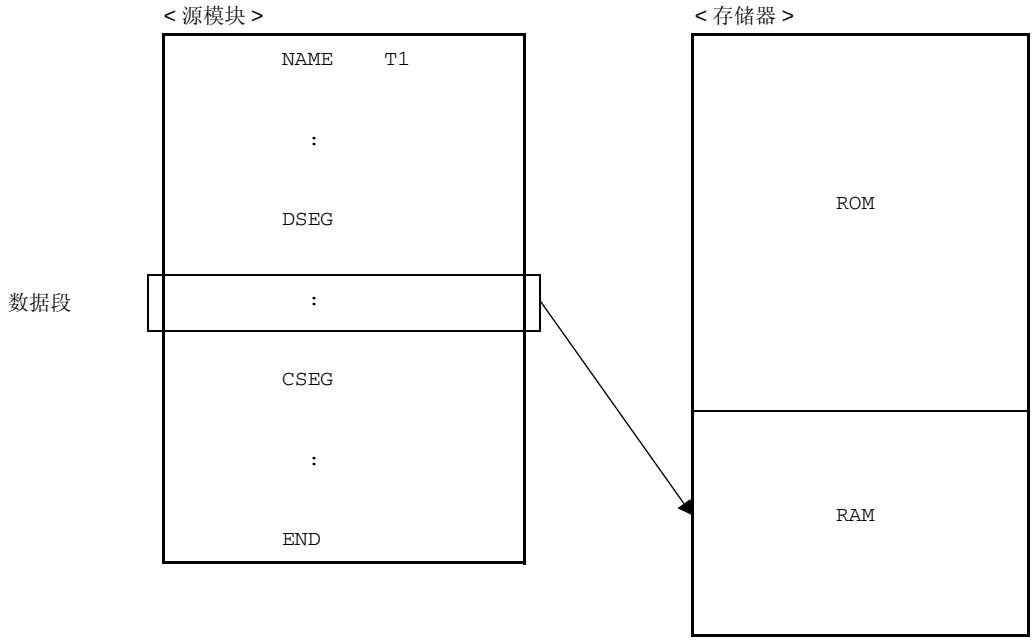
说明汇编器中数据段的起始位置。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*段名称*]	DSEG	[*重定位属性*]	[; *注释*]

[功能]

- DSEG 命令说明汇编器中数据段的起始位置。
- 在 DSEG 命令之后， DS 命令定义属于数据段的存储器，直到其遇到段定义命令 （CSEG、 DSEG、 BSEG 或 ORG）或 END 命令为止，且最终保留在 RAM 地址中。



[使用]

- DS 命令主要用于描述由 DSEG 命令定义的数据段。
  数据段位于 RAM 区域。因此，数据段中不能描述指令。
- 在数据段中，用于程序的 RAM 工作区由 DS 命令保留并且每个工作区都附有一个标签。在描述源程序时使用该标签。
  作为数据段保留的每个区域由链接器进行定位，因此不会与 RAM 上的其他工作区重叠 （堆栈区域和其他模块定义的工作区）。
  如果数据段与通用寄存器重叠，链接器将输出错误信息。可通过警告信息说明选项 （-w）改变警告信息的输出等级。

-w 定义的值	目标检查
0	无区域
1	RB0
2	RB0 至 RB3

[说明]

- 数据段的起始地址可由 ORG 命令定义。
  也可在 DSEG 命令的操作数区域中，在绝对程序表达式之后描述重定位属性揸 T
- 重定位属性定义了数据段的位置地址范围。
  下列为重定位属性适用的数据段。

表 4-18. DSEG 的重定位属性

重定位 属性	格式描述	说明
SADDR	SADDR	让汇编器把指定段定义在 saddr 区域 (saddr 区域 : FFE20H 至 FFEFFH)。
SADDRP	SADDRP	让汇编器把指定段定义在 saddr 区域的偶数地址 （saddr 区域 : FFE20H g 至 FFEFFH)。
AT	AT *绝对表达式*	让汇编器把已定义段定位至绝对地址 （SFR 和 2ndSFR 除外）。
UNIT	UNIT 或无定义	让汇编器把指定段定义在 内部或任何外部位置 （在名称为揸 AM
UNITP	UNITP	让汇编器把指定段定义在 偶数地址的内部或任何外部位置 （在名称为揸 AM
BASEP	BASEP	让汇编器把指定段定位在内部 RAM 区域，这样可能从偶数地址开始（不包括 saddr 区域：FxxxxH 至 FFEFFH). 注 在没有 ES 引用的组织数据访问时使用。
PAGE64KP	PAGE64KP	让汇编器把已定义段定位至揸 AM 同名段但位于不同文件中的不可合并。
UNIT64KP	UNIT64KP	让汇编器把已定义段定位至揸 AM 同名段可合并。

注 xxxx 表示的地址根据所使用的器件而不同。

- 由 78K0 汇编器提供的重定位属性也可进行描述，其功能与揸 NIT
  下表列出了 78K0 提供的 DSEG 重定位属性。

重定位属性	格式描述
IHRAM	IHRAM
LRAM	LRAM
DSPRAM	DSPRAM
IXRAM	IXRAM

- 若未指定重定位属性至数据段，汇编器将假设揸 NIT

- 若定义了在 “表 4-18. DSEG 的重定位属性”中以外的重定位属性，汇编器将输出错误并假设 “UNIT”已指定。若每个数据段的大小超出重定位属性指定的区域，将产生错误。
- 如果由重定位属性揸 T
- 机器语言指令 （包括 BR 命令）不能用于描述数据段。若使用，将输出错误且忽略该行。
- 通过 DSEG 命令在符号区域描述段名称，可进行数据段命名。如果未给数据段设定段名，汇编器将自动给出缺省段名。

  下列为数据段的缺省段名。

重定位 属性	缺省段名
SADDR	?DSEGS
SADDRP	?DSEGSP
UNIT （或无定义）	?DSEG
UNITP	?DSEGUP
IHRAM	?DSEGIH
LRAM	?DSEGL
DSPRAM	?DSEGDSP
IXRAM	?DSEGIX
BASEP	?DSEGBP
PAGE64KP	?DSEGP64
UNIT64KP	?DSEGU64
AT	段名不可省略。

- 当 C 编译器输出的缺省段中，下列段的大小为 0 时，链接器将更改重定位属性。

片断名	重定位属性	大小为 0 时的重定位属性
@@INIS	DSEG SADDRP	DSEG UNITP
@@DATS	DSEG SADDRP	DSEG UNITP
@EINIS	DSEG SADDRP	DSEG UNITP
@EDATS	DSEG SADDRP	DSEG UNITP

- 如果两个或多个数据段具有相同重定位属性 （除 AT）的话，这些数据段可能具有相同的段名。
  这些段在汇编器处理时作为一个数据段处理。
- 可把数据段分为单元进行描述。相同重定位属性和同名的代码段可作为一系列段由汇编器在一个模块中描述。

注意事项 1. 对于重定位属性为 AT 的代码段的描述不能分为单元进行。

2. 当重定位属性为 SADDR 时，有必要的话，插入一个字节的间隔，这样 DESG 命令之后描述的地址可能为偶数。

- 如果重定位属性为 SADDRP，定位指定段的话，在 DSEG 命令之后描述的地址是 2 的倍数。

- 如果同名代码段的重定位属性不同的话，将产生错误。因此，每个重定位属性的代码段数量为 1。
- 在两个或更多不同模块中的同名数据段，只有当其重定位属性为 UNIT、UNITP、SADDR、SADDRP、LRAM、IHRAM、 DSPRAM、 IXRAM、 BASEP、 PAGE64KP 或 UNIT64KP 时才能定义，并在链接时合并为一个数据段。
- 段名称不能引用为符号。
- 汇编器能够输出的总段数为 255 个不同的段，包括 ORG 命令中定义的名称。同名段算作一个。
- 可作为段名的最大字符数为 8。
- 段名中的大小写字符是有区别的。

[示例]

```
        NAME    SAMP1
        DSEG                    ; (1)
WORK1 : DS      2
WORK2 : DS      1
        CSEG
        MOV     A, !WORK2       ; (2)
        MOV     A, WORK2        ; (3)   <- Error
        MOVW    DE, #WORK1      ; (4)
        MOVW    AX, WORK1       ; (5)   <- Error
        END
```

(1) 数据段的开始由 DSEG 命令定义。
由于省略了重定位属性，因此假设为 “UNIT”。缺省段名为 "?DSEG"。

(2) 该描述对应于 "MOV A, !addr16"。

(3) 该描述对应于 "MOV A, saddr"。
可重定位标签 “WORK2” 不能描述为 “saddr”。因此，进行该描述时会产生错误。

(4) 该描述对应于 "MOVW rp, #word"。

(5) 该描述对应于 "MOVW AX, saddrp"。
可重定位标签 “WORK1” 不能描述为 “saddrp”。因此，进行该描述时会产生错误。

BSEG

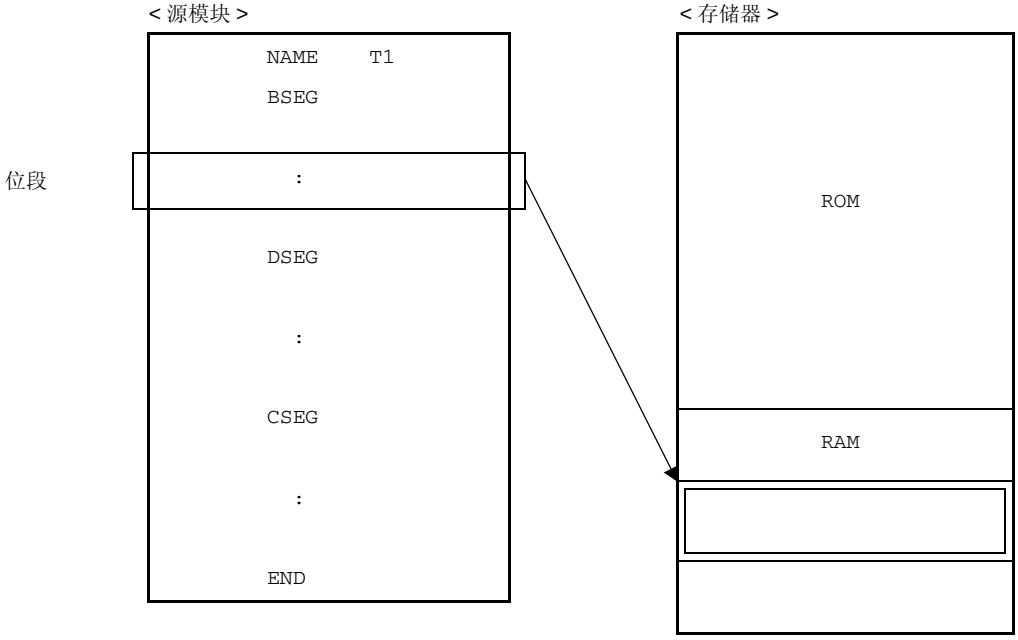
说明汇编器中位段的起始位置。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*段名称*]	BSEG	[*重定位属性*]	[; 注释]

[功能]

- BSEG 命令说明汇编器中位段的起始位置。
- 位段用于定义源模块中使用的 RAM 地址。
- BSEG 命令后由 DBIT 命令定义的存储器区域属于位段直到其遇到段定义命令 （CSEG、 DSEG 或 BSEG）或 END 命令为止。



[使用]

- 在位段中描述的 DBIT 命令由 BSEG 命令定义。
- 位段中不能描述指令。

[说明]

- 位段的起始地址可通过描述重定位属性区域的括 T *绝对程序表达式*？定义。
- 重定位属性定义了位段的位置地址范围。
  下列为重定位属性适用的位段。

表 4-19. BSEG 的重定位属性

重定位属性	格式描述	说明
AT	AT *绝对表达式*	让汇编器把指定段的起始地址定位于绝对地址的位 0。禁止在位单元中定义 （00000H 至 FFFFFH）（SFR 和 2ndSFR 除外）。
UNIT	UNIT 或无定义	让汇编器把指定段定位于任何位置 （FFE20H 至 FFEFFH）。

- 若未指定重定位属性至位段，汇编器将假设挃 NIT
- 如果定义了除表 3-5 中列出的重定位属性的话，汇编器将输出错误且假设已设定挃 NIT。若每个位段的大小超出重定位属性指定的区域，将产生错误。
- 在汇编器和链接器中，位段中的位置计数器以 “0xxxxx.b”的形式出现 （字节地址为 5 位十六进制数，位的位置为 1 位十六进制数 （0 至 7）。

(1) 绝对

字节地址	位的位置							
	0	1	2	3	4	5	6	7
0FFE20H	0FFE20H.0	.1FFE20H.0	.2FFE20H.0	.3FFE20H.0	.4FFE20H.0	.5FFE20H.0	.6FFE20H.0	.7FFE20H.0
0FFE21H	0FFE21H.0	.1FFE21H.0	.2FFE21H.0	.3FFE21H.0	.4FFE21H.0	.5FFE21H.0	.6FFE21H.0	.7FFE21H.0

(2) 可重定位

字节地址	位的位置							
	0	1	2	3	4	5	6	7
0H	0H.0	0H.1	0H.2	0H.3	0H.4	0H.5	0H.6	0H.7
1H	1H.0	1H.1	1H.2	1H.3	1H.4	1H.5	1H.6	1H.7

备注 拥有可重定位的位段，字节地址可在段的开始指定字节单元中的偏移量。
在目标转换器输出的符号表中，显示和输出了已定义位的区域开始部分的位偏移。

符号值	位偏移
0FFE20H.0	0000
.1FFE20H.0	0001
.2FFE20H.0	0002
:	:
.7FFE20H.0	0007
0FFE21H.0	0008
.1FFE21H.0	0009
:	:
0FFE80H.0	0300

符号值	位偏移
:	:

- 如果由重定位属性揇 T
- 通过 BSEG 命令在符号区域描述段名称，可进行位段命名。如果未给位段设定段名，汇编器将自动给出缺省段名。

  下表显示了缺省的段名。

重定位属性	缺省段名
UNIT （或无定义）	?BSEG
AT	段名不可省略。

- 当 C 编译器输出的缺省段中，下列段的大小为 0 时，链接器将更改重定位属性。

片断名	重定位属性	大小为 0 时的重定位属性
@@BITS	BSEG UNIT (位于 SADDR 区域)	BSEG UNIT (位于 RAM 区域)

- 如果重定位属性是揇 NIT。这些段在汇编器处理时作为一个单一段处理。

  因此，每个重定位属性的重名段数量为 1。
- 重名的位段名必须具有相同的重定位属性 UNIT （当重定位属性为 AT 时，禁止设定多个段为同名）。
- 如果重名段的重定位属性不是 UNIT 模式，那么将输出错误且该行被忽略。
- 处于两个或更多不同模式的重名位段将在链接时间合并为一个单一位段。
- 段名称不能引用为符号。
- 段位将通过链接器放置在 0H 至 FFFFFH 的地址。
- 标签不能在位段中进行描述。
- 只有 DBIT, EQU, SET, PUBLIC, EXTBIT, EXTRN, MACRO, REPT, IRP, ENDM 命令、宏定义和宏引用可在位段中进行描述。若对上述指令以外的指令进行描述，将产生错误。
- 汇编器输出的总段数最多为 256 个不同段，由 ORG 命令定义段。同名段算作一个。
- 可作为段名的最大字符数为 8。
- 段名中的大小写字符是有区别的。

[示例]

```
        NAME    SAMP1

FLAG    EQU     0FFE20H
FLAG0   EQU     FLAG.0          ; (1)
FLAG1   EQU     FLAG.1          ; (2)

        BSEG                    ; (3)
FLAG2   DBIT

        CSEG
        SET1    FLAG0           ; (4)
        SET1    FLAG2           ; (5)
        END
```

(1) 位地址（**0FFE20H** 中的位 **0**）根据字节地址范围进行定义。

(2) 位地址（**0FFE20H** 中的位 **1**）根据字节地址范围进行定义。

(3) 位段由 **BSEG** 命令进行定义。由于省略了重定位属性，因此默认重定位属性为挎 **NIT**
在每个段位中， **DBIT** 命令为每个位定义位工作区域。位段应在模块本体的头部分进行描述。
位段中定义的位地址 **FLAG2** 的位置可无需考虑位地址范围进行放置。

(4) 该描述可由 “**SET1 FLAG.0**”替换。该 **FLAG** 表示字节地址。

(5) 在该描述中，无需考虑给定的字节地址范围。

ORG

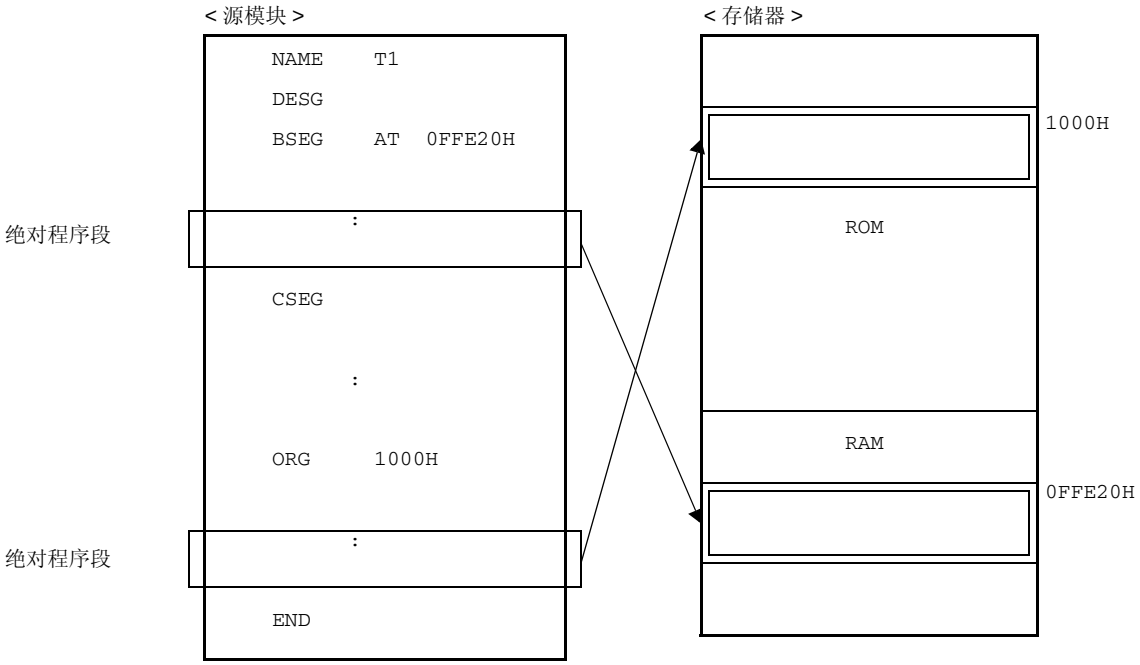
由定位计数器的操作数定义的表达式值。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*段名称*]	ORG	[*重定位属性*]	[; *注释*]

[功能]

- ORG 命令设置由定位计数器的操作数定义的表达式值。
- 在 ORG 命令之后，直到遇到段定义命令（CSEG、DSEG、BSEG 或 ORG）或 END 命令之前，已描述的指令或保留的存储器空间均属于绝对段，并且由操作数指定其存放地址。



[使用]

- 指定 ORG 命令把代码段或数据段定位至指定地址。

[说明]

- 由 ORG 命令定义的绝对程序段在该 ORG 命令执行之前属于由 CSEG 或 DSEG 命令定义的代码段或数据段。在属于数据段的绝对程序段中，不能进行指令描述。属于位段的绝对程序段不能由 ORG 命令进行描述。
- 由 ORG 命令定义的段位或数据位翻译为重定位属性 “AT” 的代码段或数据段。
- 通过 ORG 命令在符号区域描述段名称，可进行绝对程序段的命名。
  可识别为段名的最大字符数为 8。
- 在模块中，由 ORG 命令定义的同名段与由 CSEG 或 DESG 命令定义的 AT 属性段以相同的方式处理。
- 在不同模块中，由 ORG 命令定义的同名段与由 CSEG 或 DESG 命令定义的 AT 属性段以相同的方式处理。

- 如果未对绝对程序段指定段名，汇编器将自动分配缺省段名 ??A0nnnnn”，其中 “nnnnn”表示指定段的 5 个数字的十六进制起始地址 （ 00000 至 FFFFF）。
- 如果 ORG 命令之前未出现 CSEG 或 DSEG 命令，那么由 ORG 命令定义的绝对程序段将翻译为代码段中的绝对程序段。
- 如果名称或标签作为 ORG 命令的操作数进行描述时，那么该名字或标签必须为已在源模块中定义了的绝对项。
- 如果采用非法对象对绝对表达式进行描述，或绝对表达式的评估值超过 00000H 至 FFEFFH 范围的话，汇编器将输出错误并继续处理程序，假设绝对表达式的值为 00000H。
- 用于操作数的绝对表达式在无符号的 32 位单元中进行评估。
- 段名称不能引用为符号。
- 汇编器输出的总段数最多为 256 个不同段，由段定义命令定义段。同名段算作一个。
- 可作为段名的最大字符数为 8。
- 段名中的大小写字符是有区别的。

[示例]

```
        NAME    SAMP1


        DSEG
        ORG     0FFE20H         ; (1)
SADR1 : DS      1
SADR2 : DS      1
SADR3 : DS      2

MAIN0   ORG     100H
        MOV     A, SADR1        ; (2)   <- Error

        CSEG                            ; (3)
MAIN1   ORG     1000H           ; (4)
        MOV     A, SADR2
        MOVW    AX, SADR3
        END
```

(1) 定义属于数据段的绝对程序段。
该绝对程序段将定位于短直接寻址区域，从地址 “FFE20H”开始。由于省略了段名的定义，汇编器将自动分配名称 “?A0FFE20?。

(2) 产生一个错误，因为在属于数据段的绝对程序段中不能进行指令描述。

(3) 该命令声明了代码段开始。

(4) 该绝对程序段位于从 "1000H"

4.2.3　符号定义命令

符号定义命令指定那些用于写入源模块的数据名称。有了这些名称，数据值的规范就更清晰，并且更便于理解源模块的详细信息。

符号定义命令指定了汇编器中源模块所使用的值的名称。

可获得下列符号定义命令。

控制指令	概要
EQU	由操作数定义的表达式值以及带属性的数值数据并定义为名称。
SET	由操作数定义的表达式值以及带属性的变量并定义为名称。

EQU

由操作数定义的表达式值以及带属性的数值数据并定义为名称。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
name	EQU	*expression*	[; *comment*]

[功能]

- EQU 命令定义的名称具有在操作数区域内指定的表达式值和属性 （符号属性和重定位属性）。

[使用]

- 由 EQU 命令对源模块中使用的数值数据进行命名，并在命令的操作数中对该名称进行描述，从而以该名称取代数值数据。
  在源模块中经常使用的数值数据，推荐为其定义一个名称。如果必须在源模块中修改数据值，那么只需修改名称的操作数值。

[说明]

- EQU 命令可在源程序中的任何位置进行描述。
- 由 EQU 命令定义的符号不能由 SET 命令重新定义，标签也是如此。此外，由 SET 命令定义的符号或标签不能由 EQU 命令重新定义。
- 当名称或标签在 EQU 命令的操作数中进行描述时，所使用的名称或标签已在源模块中进行定义。
  该命令的操作数不能描述为外部引用项。
  不能描述 SFR 和 SFR 位符。
- 不能描述包含由 HIGH/LOW/HIGHW/LOWW/DATAPOS/BITPOS 运算符建立的项的表达式，该表达式在其操作数中具有一个可重定位项。
- 如果表达式包含下列描述的任一操作数模式的话，将出现错误。

(1) 带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2

下列条件 (1) 和 (2) 中的任意一个均符合上述表达式 (a) 或 (b)：

(a) (a) 如果带有 ADDRESS 属性的表达式 1 中的标签 1 和带有 ADDRESS 属性的表达式 2 中的标签 2 属于相同段且如果在两个标签之间描述 BR 命令，其中目标码的字节数不能确定

(b) (b) 如果标签 1 和标签 2 位于不同段，且如果在段和标签起始部分之间描述 BR 命令，其中目标码的字节数不能确定

(2) 带有 ADDRESS 属性和属性相关运算符的表达式 1 - 带有 ADDRESS 属性的表达式 2

(3) 带有 ADDRESS 属性的 HIGH 绝对表达式

(4) 带有 ADDRESS 属性的 LOW 绝对表达式

(5) 带有 ADDRESS 属性的 HIGHW 绝对表达式

(6) 带有 ADDRESS 属性的 LOWW 绝对表达式

(7) 带有 ADDRESS 属性的 DATAPOS 绝对表达式

(8) 带有 ADDRESS 属性的 BITPOS 绝对表达式

(9) 下列 (a) 满足 (3) 至 (8) 的表达式：

(a) 如果在带有 ADDRESS 属性的表达式的标签和标签所属段的开始部分之间描述 BR 命令，其中目标码的字节数不能立即确定

- 如果在操作数的描述格式内存在错误，汇编器将输出错误信息，但是将假设存放的操作数值为其可分析的符号区域中描述的名称值。
- 由 EQU 命令定义的名称，在同一源模块中不能进行重新定义。
- 由 EQU 命令定义的名称具有位值的话，将具有地址和作为数值的位的位置。
- 下表显示了可描述为 EQU 命令的操作数的位值以及这些位值可被引用的范围。

操作数类型	符号值	引用范围
A.bit 注 1	1.bit	只能在同一模块内引用。
PSW.bitye	0FFFFAH.bit	
sfr 注 2.bit 注 1	0FFFXXH 注 3.bit	
2ndsfr 注 2.bit 注 1	0FXXXXH 注 4.bit	
saddr.bit 注 1	0FFXXXH 注 5.bit	可从另一模块引用。
expression.bit 注 1	0XXXXXH 注 6.bit	

注 **1.** bit = 0 - 7

2. 关于详细的描述，请参阅各器件的用户手册。

3. 0FFFXXH： sfr 的地址

4. 0FXXXXH： 2ndsfr 区域

5. 0FXXXXH： saddr 区域 （0FFE20H 至 0FFF1FH）

6. 0XXXXXH： 0H 至 0FFFFFH

[示例]

```
        NAME    SAMP1

WORK1   EQU     0FFE20H         ; (1)
WORK10  EQU     WORK1.0         ; (2)
P02     EQU     P0.2            ; (3)
A4      EQU     A.4             ; (4)
PSW5    EQU     PSW.5           ; (5)

        SET1    WORK10          ; (6)
        SET1    P02             ; (7)
        SET1    A4              ; (8)
        SET1    PSW5            ; (9)
        END
```

(1) 名称 “WORK1” 具有值 “0FFE20H”、符号属性揑 UMBER

(2) 名称揥 ORK10
在上述 (1) 中，操作数中描述的 “WORK1” 已被定义值 “0FFE20H”。

(3) 名称揚 02

(4) 名称揑 4

(5) 名称揚 SW5

(6) 该描述相当于 "SET1 saddr.bit"。

(7) 该描述相当于 "SET1 sfr.bit"。

(8) 该描述相当于 "SET1 A.bit"。

(9) 该描述相当于 "SET1 PSW.bit"。

在 (4) 至 (5) 中被定义为 "A.bit" 和 "PSW.bit" 的名称只能在相同模块中进行引用。
被定义为 "sfr.bit"、"saddr.bit" 和 "*expression*.bit" 的名称还可作为外部定义符号从另一模块中进行引用（参考 "4.2.5 链接指令 "）。

作为应用示例中的源模块汇编结果，将产生下列汇编列表。

```
        Assemble list

ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT

  1       1                                NAME    SAMP
  2       2
  3       3           ( FFE20 )       WORK1   EQU     0FFE20H         ; (1)
  4       4           ( FFE20.0 )     WORK10  EQU     WORK1.0         ; (2)
  5       5           ( FFF00.2 )     P02     EQU     P0.2            ; (3)
  6       6           ( 00001.4 )     A4      EQU     A.4             ; (4)
  7       7           ( FFFFA.5 )     PSW5    EQU     PSW.5           ; (5)
  8       8
  9       9   00000   710220                  SET1    WORK10          ; (6)
 10      10   00003   712200                  SET1    P02             ; (7)
 11      11   00006   71CA                    SET1    A4              ; (8)
 12      12   00008   715AFA                  SET1    PSW5            ; (9)
 13      13                                    END
```

在汇编列表的 (2) 至 (5) 行，位值中的位地址值定义为在目标代码区域内显示的名称。

SET

由操作数定义的表达式值以及带属性的变量并定义为名称。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
name	SET	*absolute-expression*	[; *comment*]

[功能]

- SET 命令定义的名称具有在操作数区域内指定的表达式值和属性 （符号属性和重定位属性）。
- 由 SET 命令定义的名称的值和属性可在相同模块中重新定义。
  这些值和属性始终有效，直到相同名称重新定义为止。

[使用]

- 对在源模块中作为名称使用的数值数据 （变量）进行命名，并在命令的操作数中对其进行描述，从而取代数值数据 （变量）。
  若欲改变源模块中名称的值，可再次使用 SET 命令对相同的名称赋予不同的值。

[说明]

- 在 SET 命令的操作数区域中必须描述绝对表达式。
- SET 命令可在源程序中的任何位置进行描述。
  但是，由 SET 命令定义的名称不能向前引用。
- 如果在 SET 命令定义的语句中检测到错误的话，汇编器会输出错误信息，但将假设存放的操作数值是其能够分析的符号区域中描述的名称的值。
- 由 EQU 命令定义的符号不能由 SET 命令重新定义。
  由 SET 命令定义的符号不能由 EQU 命令重新定义。
- 不能定义位符号。

[示例]

```
        NAME    SAMP1
COUNT   SET     10H             ; (1)

        CSEG
        MOV     B, #COUNT       ; (2)
LOOP :
        DEC     B
        BNZ     $LOOP

COUNT   SET     20H             ; (3)

        MOV     B, #COUNT       ; (4)
        END
```

(1) 名称 “COUNT”具有值 “10H”、符号属性 “NUMBER”和重定位属性 “ABSOLUTE”。这些值和属性将有效，直到在下列 (3) 中由 SET 命令重新定义。

(2) 名称 "COUNT" 的值 "10H" 发送至寄存器 B。

(3) 名称 "COUNT" 的值改变为 "20H"。

(4) 名称 "COUNT" 的值 "20H" 发送至寄存器 B。

4.2.4　存储器初始化、区域保留命令

存储器初始化命令定义了程序所使用的常量数据。
定义的数据值作为目标代码产生。
区域保留命令确保了程序所使用的存储器区域。
可获得下列存储器初始化和分区命令。

控制指令	概要
DB	字节区域的初始化
DW	字区域的初始化
DG	32 位（4 个字节）中 20 位区域的初始化
DS	确保由操作数指定的字节数存储区域。
DBIT	在位段中确保 1 位的存储区域。

DB

字节区域的初始化

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	DB	(*size*)	[; *comment*]
		or	
[*label*:]	DB	*initial-value*[, ...]	[; comment]

[功能]

- DB 命令使汇编器初始化字节区域。
  初始化的字节数量可定义为捞 ize
- DB 命令还可让汇编器以字节单位初始化存储区域，并在操作数区域中指定初始值。

[使用]

- 当定义在程序中使用的表达式或字符串时使用 DB 命令。

[说明]

- 若在操作数区域中插入一个数值，汇编器就会假设指定了一个 size。否则，会假设一个初始值。

(1) 带 size 的规范：

(a) 如果在操作数区域指定 size，汇编器初始化的区域等同于指定的带有值 "00H" 的字节数。

(b) 绝对表达式必须作为 size 进行描述。如果进行非法的 size 描述，汇编器将输出错误信息并停止执行初始化。

(2) 带初始值的规范：

(a) 表达式

表达式的值必须为 8 位数据。因此，操作数值必须在 0H 至 0FFH 范围内。如果超出 8 位，汇编器将只使用低八位的值作为有效数据并输出一个错误。

(b) 字符串

如果描述字符串作为操作数，将为每个字符串中的字符保留为 8 位 ASCII 码。

- DB 命令不能在位段中进行描述。
- DB 命令的语句中可指定两个或更多初始值。
- 作为初始值，可描述包含可重定位符号或外部引用符号的表达式。

[示例]

```
                NAME    SAMP1
                CSEG
WORK1 :         DB      ( 1 )                   ; (1)
WORK2 :         DB      ( 2 )                   ; (1)
                CSEG
MASSAG :        DB      'ABCDEF'                ; (2)
DATA1 :         DB      0AH, 0BH, 0CH           ; (3)
DATA2 :         DB      ( 3 + 1 )               ; (4)
DATA3 :         DB      'AB' + 1                ; (5)   <- Error
                END
```

(1) 由于指定了 **size**，汇编器将以值 **"00H"** 对每个字节区域进行初始化。

(2) 一个 **6** 字节的区域以字符串 **'ABCDEF'** 进行初始化。

(3) 一个 **3** 字节的区域以 **"0AH, 0BH, 0CH"** 进行初始化。

(4) 一个 **4** 字节的区域以 **"00H"** 进行初始化。

(5) 由于表达式 **"AB" + 1** 的值为 **4143H (4142H + 1)**，超出了 **0** 至 **0FFH** 范围，
因此该描述发生错误。

DW

字区域的初始化

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	DW	(*size*)	[; *comment*]
		or	
[*label*:]	DW	*initial-value*[, ...]	[; comment]

[功能]

- DW 命令使汇编器初始化字区域。
  初始化的字的数量可定义为捞 ize
- DW 命令还可让汇编器以字单位 （2 个字节）初始化存储区域，并在操作数区域中指定初始值。

[使用]

- 当定义 16 位数字常量如程序使用的地址或数据时，使用 DW 指令。

[说明]

- 若在操作数区域中插入一个数值，汇编器就会假设指定了一个 size ； 否则就假定初始值。

(1) 带 size 的规范：

(a) 如果在操作数区域指定 size，汇编器初始化的区域等同于指定的带有值 "00H" 的字的数量。

(b) 绝对表达式必须作为 size 进行描述。如果进行非法的 size 描述，汇编器将输出错误并停止执行初始化。

(2) 带初始值的规范：

(a) 常量
16 位或更少。

(b) 表达式
表达式的值必须以 16 位数据存放。
字符串不可描述为初始值。

- DW 命令不能在位段中进行描述。
- 指定初始值的高 2 位数字存放于 HIGH 地址，低 2 位数字存放于 LOW 地址。
- DW 命令的语句中可指定两个或更多初始值。
- 作为初始值，可描述包含可重定位符号或外部引用符号的表达式。

[示例]

```
                NAME    SAMP1
                CSEG
WORK1 :         DW      ( 10 )          ; (1)
WORK2 :         DW      ( 128 )         ; (1)
                CSEG
                ORG     10H
                DW      MAIN            ; (2)
                DW      SUB1            ; (2)

                CSEG
MAIN :
                CSEG
SUB1 :

DATA :          DW      1234H, 5678H    ; (3)
                END
```

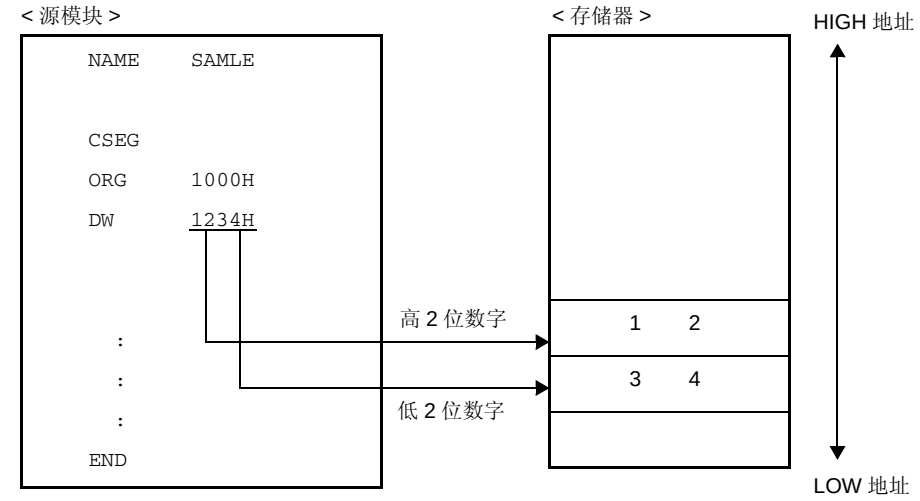
(1) 由于指定了 size，汇编器将以值 "00H" 对每个字进行初始化。

(2) DW 指令定义向量入口地址。

(3) 一个 2 个字的区域以 "34127856" 值进行初始化。

注意事项　存储器的 HIGH 地址用字的高 2 位数字作为初始值。存储器的 LOW 地址用字的低 2 位数字作为初始值。

< 示例 >



DG

32 位 （4 个字节）中 20 位区域的初始化

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	DG	(*size*)	[; *comment*]
		or	
[*label*:]	DG	*initial-value*[, ...]	[; comment]

[功能]

- DG 指令让汇编器初始化 32 位 （4 个字节）单元中的 20 位区域。初始值或 size 可定义为操作数。
- DG 命令还可让汇编器以 4 个字节单位初始化存储区域，并在操作数区域中指定初始值。

[使用]

- 当定义 20 位数字常量如程序使用的地址或数据时，使用 DG 指令。

[说明]

- 若在操作数区域中插入一个数值，汇编器就会假设指定了一个 size ；否则就假定初始值。

(1) 带 size 的规范：

(a) 如果在操作数区域指定 size，汇编器初始化的区域等同于指定的带有值 "00H" 的数字 x4 个字节。

(b) 绝对表达式必须作为 size 进行描述。如果进行非法的 size 描述，汇编器将输出错误并停止执行初始化。

(2) 带初始值的规范：

(a) 常量

20 位或更少。

(b) 表达式

表达式的值必须以 16 位数据存放。
字符串不可描述为初始值。

- DG 命令不能在位段中进行描述。
- 指定初始值的最高位字节存放于 HIGH WORD 地址，最低位字节存放于 LOW 地址，且低 2 位字节中较高的那个字节存放于存储器中的 HIGH 地址。
- DW 命令的语句中可指定两个或更多初始值。
- 作为初始值，可描述包含可重定位符号或外部引用符号的表达式。

[示例]

```
                NAME    SAMP1

DATA1 :         DG      12345H, 56789H          ; (1)
DATA2 :         DG      ( 10 )                  ; (2)
                END
```

(1)　一个 4 字节的区域以 "4523010089670500" 值进行初始化。

(2)　40 字节 (10 x 4 字节) 区域以 "00H" 进行初始化。

注意事项　对于 20 位的值，存储器的 HIGH WORD 地址由最高位字节初始化，存储器的 LOW 地址由最低位字节初始化， HIGH 地址由低 2 位中的较高位字节初始化。

< 示例 >

< 源模块 >

```
NAME    SAMP1
        CSEG
DATA1:  DG      12345H, 56789H
          :
        END
```

< 存储器 >

	00
HW	05
H	67
L	89
	00
HW	01
H	23
L	45

HIGH 地址

LOW 地址

HW :　HIGH WORD

H　:　HIGH

L　:　LOW

DS

确保由操作数指定的字节数存储区域。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*label*:]	DS	*absolute-expression*	[; *comment*]

[功能]

- DS 指令让汇编器为操作数区域中指定的字节数保留存储器区域。

[使用]

- DS 指令主要用来保留程序中所使用的存储器 （RAM）区域。
  若指定标签，那么保留的存储器区域中的首地址的值将分配给该标签。在源模块中，该标签用于描述对存储器的操作。

[说明]

- 由 DS 指令保留的区域内容为未知 （不确定）。
- 定义的绝对表达式由无符号的 16 位操作数进行评估。
- 当操作数值为 ?
- DS 命令不能在位段中进行描述。
- 由 DS 指令定义的符号 （标签）只能逆向引用。
- 只有下述从绝对表达式中扩展的参数可在操作数区域进行描述：
  - 常量
  - 带进行运算的常量的表达式 （常量表达式）
  - 定义带有常量或常量表达式的 EQU 符号或 SET 符号 ADDRESS
  - 带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2
    如果 " 带有 ADDRESS 属性 " 的表达式 1
    然而，下列两种情况下会发生错误：
    - 如果标签 1 和标签 2 属于相同段且如果在这两个标签之间描述 BR 指令，其中目标码的字节数不能立即确定
    - 如果标签 1 和标签 2 位于不同段，且如果在任意段和标签所属段的起始部分之间描述 BR 命令，其中目标码的字节数不能确定
  - 在上述 (1) 至 (4) 的任意表达式中包含运算。
- 下列参数不可在操作数区域内描述：
  - 外部引用符号
  - 由 EQU 指令定义的 ? 带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2
  - 位置计数器 ($) 可在以 ? 带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2
  - 由 EQU 指令定义的带有 ADDRESS 属性的表达式的符号，该表达式允许进行带有 HIGH/LOW/DATAPOS/BITPOS 运算符的运算。

[示例]

```
                NAME    SAMPLE
                DSEG
TABLE1 :        DS      10              ; (1)
WORK1 :         DS      2               ; (2)
WORK2 :         DS      1               ; (3)
                CSEG
                MOVW    HL, #TABLE1
                MOV     A, !WORK2
                MOVW    BC, #WORK1
                END
```

(1) 保留了 10 字节的工作区，但是工作区的内容为未知 （不确定）。标签 "TABLE1" 位于起始地址。

(2) 保留 1 个字节的工作区。

(3) 保留 2 个字节的工作区。

DBIT

在位段中确保 1 位的存储区域。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*name*]	DBIT	*None*	[; *comment*]

[功能]

- DBIT 指令让汇编器在位段中保留 1 位的存储区域。

[使用]

- 使用 DBIT 指令在位段中保留位区域。

[说明]

- DBIT 指令只能在位段中进行描述。
- 由 DBIT 指令保留的 1 位区域内容为未知 （不确定）。
- 如果在符号区域内指定名称，该名称具有作为其值的地址和位的位置。
- 定义的名称可在需要 saddr.bit、 addr16.bit 和 ES:addr16.bit 的地方进行描述。

[示例]

```
        NAME    SAMPLE
        BSEG
BIT1    DBIT            ; (1)
BIT2    DBIT            ; (1)
BIT3    DBIT            ; (1)

        CSEG
        SET1    BIT1    ; (2)

        CLR1    BIT2    ; (3)
        END
```

(1) 通过这三个 DBIT 指令，汇编器将保留 3 个 1 位区域并为其定义名称 (BIT1、 BIT2 和 BIT3)，每个均具有作为值的地址和位的位置。

(2) 该描述对应于 "SET1 saddr.bit" 且描述在上述 (1) 中保留的作为操作数 "saddr.bit" 的位区域名称 "BIT1"。

(3) 该描述对应于 "CLR1 saddr.bit" 且描述作为 "saddr.bit" 的名称 "BIT2"。

4.2.5　链接指令

链接指令用来阐明引用其他模块定义的符号时的关系。可以理解为当一个程序分成模块 1 和模块 2 进行编写的情况。

当需要在模块 1 中引用由模块 2 定义的符号时，由于在两个模块中都没有进行任何声明，因此不能使用该符号。所以，需要在模块中分别显示 " 我想使用 " 或 " 我不想使用 "。

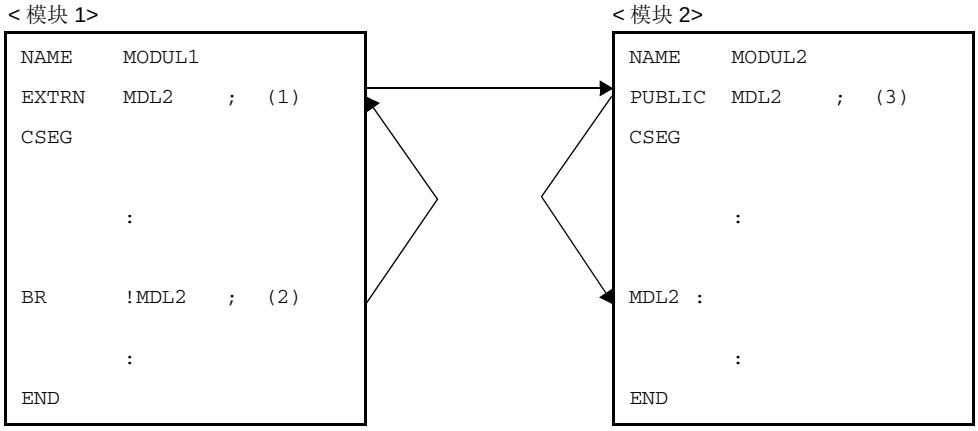
在模块 1 中，需作出 " 我想引用由其他模块定义的符号 " 这样的外部引用声明。与此同时，在模块 2 中需作出 " 该符号可由其他模块进行引用 " 这样的外部定义声明。

只有当外部引用和外部定义声明同时生效后，才能引用该符号。

链接指令就是用来建立这之间的关系的，并可使用下列指令。

- 符号外部引用声明：EXTRN 以及 EXTBIT 指令。
- 符号外部定义声明：PUBLIC 指令。

图 4-7. 两个模块的符号之间的关系



在上述模块中，为了使模块 2 中定义的 "MDL2" 符号在 (2) 中引用，在 (1) 中通过 EXTRN 指令声明外部引用。

在模块 2 的 (3) 中，通过 PUBLIC 指令对由模块 1 引用的 "MDL2" 符号进行外部定义声明。

该外部引用和外部定义符号是否正常响应由链接器检查。

可获得下列链接指令。

控制指令	概要
EXTRN	向链接器声明，把另一个模块中的符号 （位符除外）引用到本模块中。
EXTBIT	指令向链接器声明，把另一个模块中的位符 引用到本模块中。
PUBLIC	向链接器声明，在操作数区域中描述的符号由另一模块引用。

EXTRN

向链接器声明，把另一个模块中的符号 （位符除外）引用到本模块中。

[格式描述]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	EXTRN	*symbol-name*[, ...]	[; comment]
		or	
[*label*:]	EXTRN	BASE(*symbol-name*[, ...])	[; comment]

[功能]

- EXTRN 指令向链接器声明，把另一个模块中的符号 （位符除外）引用到本模块中。

[使用]

- 当引用由另一模块定义的符号时，必须使用 EXTRN 指令把该符号声明为外部引用。
- 操作结果根据操作数描述格式的不同而不同。

BASE(symbol-name[, ...])	可引用在 64KB 区域 （0H 至 0FFFF）内的符号作为指定符号。
不指定重定位属性	在链接器定位之后，根据 PUBLIC 声明的区域进行引用。

[说明]

- EXTRN 指令可在源程序中的任何位置进行描述 （参考 "4.1.1 基本构造 "）。
- 在操作数区域最多可指定 20 个符号，每个符号名由逗号 （,）隔开。
- 当引用的符号具有位值时，该符号必须采用 EXTBIT 指令，将其声明为外部引用。
- 由 EXTRN 指令声明的符号必须由 PUBLIC 指令在另一模块中声明。
- 即使由 EXTRN 指令声明的符号未在该模块中引用，也不会输出错误。
- EXTRN 指令中的操作数不能描述宏 （参考 "4.4 宏 "，获取宏名的信息）。
- 在整个模块中，一个符号只能由 EXTRN 指令声明一次。若第二次或更多次得对该符号进行 EXTRN 声明，链接器将输出警告。
- 已经声明过的符号不能作为 EXTRN 指令的操作数进行描述。同样的，已经声明为 EXTRN 的符号不能由其他指令进行重定义或声明。
- 由 EXTRN 指令定义的符号可在 64KB 区域 （0H 至 0FFFFH）内进行引用。以揃 ASE(symbol name)

[示例]

- 模块 1

```
        NAME    SAMP1
        EXTRN   SYM1, SYM2, BASE (SYM3) ; (1)
        CSEG
S1 :    DW      SYM1                    ; (2)
        MOV     A, SYM2                 ; (3)
        BR      !SYM3                   ; (4)
        END
```

- 模块 2

```
        NAME    SAMP2
        PUBLIC  SYM1, SYM2, SYM3        ; (5)

        CSEG
SYM1    EQU     0FFH                    ; (6)
DATA1   DSEG    SADDR
SYM2 :  DB      012H                    ; (7)
C1      CSEG    BASE
SYM3 :  MOV     A, #20H                 ; (8)
        END
```

(1) EXTRN 指令声明的 "SYM1"、 "SYM2" 和 "SYM3" 符号，在 (2)、 (3) 和 (4) 中作为外部引用使用。在操作数区域可能描述两个或更多符号。

(2) DW 指令引用符号 "SYM1"。

(3) MOV 指令引用符号 "SYM2"，且输出的代码引用了 saddr 区域。

(4) BR 指令引用符号 "SYM3"，且输出代码处于 64KB 区域（0H 至 0FFFFH）。

(5) 符号 "SYM1"、 "SYM2" 和 "SYM3" 声明为外部定义。

(6) 定义符号 "SYM1"。

(7) 定义符号 "SYM2"。

(8) 定义符号 "SYM3"。

EXTBIT

指令向链接器声明，把另一个模块中的位符引用到本模块中。

[格式描述]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	EXTBIT	*bit-symbol-name*[, ...]	[; comment]

[功能]

- EXTBIT 指令向链接器声明，把另一个模块中的位符引用到本模块中。

[使用]

- 当引用的符号具有位值并由另一模块定义时，必须使用 EXTBIT 指令把该符号声明为外部引用。

[说明]

- EXTBIT 指令可在源程序中的任何位置进行描述。
- 在操作数区域最多可指定 20 个符号，每个符号由逗号 （,）隔开。
- 由 EXTBIT 指令声明的符号必须由 PUBLIC 指令在另一模块中声明。
- 在整个模块中，一个符号只能由 EXTBIT 指令声明一次。若第二次或更多次得对该符号进行 EXTBIT 声明，链接器将输出警告。
- 即使由 EXTRN 指令声明的符号未在该模块中引用，也不会输出错误。

[示例]

- 模块 1

```
NAME    SAMP1
EXTBIT  FLAG1, FLAG2    ; (1)
CSEG
SET1    FLAG1           ; (2)
CLR1    FLAG2           ; (3)
        END
```

- 模块 2

```
        NAME    SAMP2
        PUBLIC  FLAG1, FLAG2    ; (4)
        BSEG
FLAG1   DBIT                    ; (5)
FLAG2   DBIT                    ; (6)
        CSEG
        NOP
        END
```

(1) EXTBIT 指令声明 "FLAG1" 和 "FLAG2" 符号作为外部引用使用。
在操作数区域可能描述两个或更多符号。

(2) SET1 指令引用符号 "FLAG1"。
该描述相当于 "SET1 saddr.bit"。

(3) CLR1 指令引用符号 "FLAG2"。
该描述相当于 "CLR1 saddr.bit"。

(4) PUBLIC 指令定义符号 "FLAG1" 和 "FLAG2"。

(5) DBIT 指令定义符号 "FLAG1" 作为 SADDR 区域的位符。

(6) DBIT 指令定义符号 "FLAG2" 作为 SADDR 区域的位符。

PUBLIC

向链接器声明，在操作数区域中描述的符号由另一模块引用。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*label*:]	PUBLIC	*symbol-name*[, ...]	[; comment]

[功能]

- PUBLIC 指令向链接器声明，在操作数区域中描述的符号由另一模块引用。

[使用]

- 当定义由另一个模块引用的符号 （包括位符）时，必须使用 PUBLIC 指令把该符号声明为外部定义。

[说明]

- PUBLIC 指令可在源程序中的任何位置进行描述。
- 在操作数区域最多可指定 20 个符号，每个符号名由逗号 （,）隔开。
- 操作数区域描述的符号必须在相同模块中定义。
- 在整个模块中，一个符号只能由 PUBLIC 指令声明一次。第二次或更多次得对该符号进行 PUBLIC 的声明将被链接器忽略。
- 每个位区域中的位符均可由 PUBLIC 进行声明。
- 下列符号不能作为 PUBLIC 指令的操作数：

(1) 由 SET 指令定义的名称

(2) 在同一模块中由 EXTRN 或 EXTBIT 指令定义过的符号

(3) 段名称

(4) 模块名称

(5) 宏名称

(6) 不在该模块中定义的符号

(7) 符号定义的带 SFBIT 属性、 EQU 指令的操作数符号

(8) 符号定义的带 EQU 指令的 sfr 和 2ndSFR （但是，不包括溢出的 sfr 和 saddr 的区域）

[示例]

- 模块 1

```
        NAME    SAMP1
        PUBLIC  A1, A2          ; (1)
        EXTRN   B1
        EXTBIT  C1

A1      EQU     10H
A2      EQU     0FFE20H.1

        CSEG
        BR      B1
        SET1    C1
        END
```

- 模块 2

```
        NAME    SAMP2
        PUBLIC  B1              ; (2)
        EXTRN   A1
        CSEG
B1:
        MOV     C, #LOW ( A1 )
        END
```

- 模块 3

```
        NAME    SAMP3
        PUBLIC  C1              ; (3)
        EXTBIT  A2
C1      EQU     0FFE21H.0
        CSEG
        CLR1    A2
        END
```

(1) PUBLIC 指令声明符号 "A1" 和 "A2" 由其他模块引用。

(2) PUBLIC 指令声明符号 "B1" 由其他模块引用。

(3) PUBLIC 指令声明符号 "C1" 由其他模块引用。

4.2.6 目标模块名的声明指令

目标模块名指令为汇编器生成的目标模块命名。
可获得下列目标模块名的声明指令。

控制指令	概要
NAME	把操作数区域中描述的目标模块名分配到由汇编器输出的目标模块中。

NAME

把操作数区域中描述的目标模块名分配到由汇编器输出的目标模块中。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*label*:]	NAME	*object-module-name*	[; *comment*]

[功能]

- NAME 指令把操作数区域中描述的目标模块名分配到由汇编器输出的目标模块中。

[使用]

- 调试器的符号调试中，每个目标模块所需的模块名。

[说明]

- NAME 指令可在源程序中的任何位置进行描述。
- 关于模块名描述的惯例，请参考 "(3)　符号区域 " 中的符号描述惯例。
- 可指定为模块名的字符是由汇编器软件的操作系统允许的，除 "(" (28H) 或 ")"(29H) 或双字节字符以外的字符。
- 只有通过 NAME 指令，才能把模块名作为操作数进行描述。
- 如果省略 NAME 操作符，汇编器将假设输入源模块文件的主名 （前 256 字符）作为模块名。主名将转换成大写字母用于检索。
  如果指定两个或更多模块名，汇编器将输出警告并忽略第二个及之后的模块名声明。
- 操作数区域描述的模块名不能超出 256 个字符。
- 符号名中的大小写字符是有区别的。

[示例]

```
        NAME    SAMPLE  ; (1)
        DSEG
BIT1 :  DBIT

        CSEG
        MOV     A , B
        END
```

(1)　NAME 指令声明 "SAMPLE" 为模块名。

4.2.7 分支指令自动选择指令

有两种可直接从分支地址写入操作数的无条件分支指令："BR !addr20" 和 "BR $addr20"。

至于这些指令，由于指令的字节数不同，用户有必要根据分支目标范围选择合适的操作数之后才使用这些指令，为的是使程序具有较好的存储器效率。

由于这个原因， 78K0R 汇编器具有一个指令，可根据分支目标的范围自动选择 2、3 或 4 个字节的分支指令。称之为分支目标指令自动选择指令。

可获得下列分支指令自动选择指令。

控制指令	概要
BR	根据操作数指定的表达式值的范围，汇编器会自动选择 2、 3 或 4 字节的分支指令并生成相应的目标码。
CALL	根据操作数指定的表达式值的范围，汇编器会自动选择 3 至 4 字节的调用分支指令并生成相应的目标码。

BR

让汇编器根据操作数区域内指定的表达式值的范围自动选择 2、 3 或 4 字节的 BR 分支指令并生成相应的目标码。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	BR	*expression*	[; *comment*]

[功能]

- BR 指令让汇编器根据操作数区域内指定的表达式值的范围自动选择 2、 3 或 4 字节的 BR 分支指令并生成相应的目标码。

[使用]

- 在下列显示的分支指令中，汇编器决定了分支目标的地址范围并自动选择和输出指令，尽可能使用最少的字节数。如果不清楚是否可用 2 字节分支指令描述的话，就使用 BR 指令。

分支指令	说明
"BR $addr20"(2 字节)	可用于分支目标在 -80H 至 +7FH 内的地址范围，从 BR 指令之后的地址开始。
"BR !addr20"(3 字节)	可用于分支目标在 64KB 内的地址范围。
"BR $!addr20"(3 字节)	计算分支目标的位移，并可在位移处于 -8000H 至 +7FFFH 范围内使用。
"BR !!addr20"(4 字节)	用于上述以外的情况

如果操作数 （分支目标）位于与指令不同的可重定位段中且在 BASE 区域之外，指令将由 4 字节指令代替并输出。
如果指令和操作数 （分支目标）位于不同段且在 BASE 区域之外，其类型将不同，即使操作数位于绝对程序段，指令将由 4 字节指令代替。
如果指令和操作数 （分支目标）位于不同段且位于 BASE 区域内，指令将由 3 字节指令 （BR !addr20）代替。

备注 不同类型是指：BR 指令位于绝对程序段的不同可重定位段，或 BR 指令位于可重定位段的绝对程序段。

- 如果明确知道应采用 2、 3 或 4 字节分支指令进行描述的话，则直接采用合适的指令进行描述。与描述 BR 指令相比，这可以缩短汇编时间。

[说明]

- BR 指令只能用于代码段。
- 直接跳转目的作为 BR 指令的操作数进行描述。 "$" 表示不能描述表达式开始时的当前位置计数器。
- 为了优化，必须满足下列条件。
  - 表达式中不能出现超过一个的标号或向前引用符号。

- 不要由 ADDRESS 属性描述 EQU 符号。
- 不要把 EQU 定义的符号用于描述？带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2
- 当使用 HIGH/LOW/HIGHW/LOWW/ DATAPOS/BITPOS 运算符进行运算时，不要描述带有 ADDRESS 属性的表达式。

如果不满足上述条件，将选择 4 字节的 BR 指令。

[示例]

```
ADDRESS                 NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  BR      L1                      ; (1)
00052H                  BR      L2                      ; (2)
00055H                  BR      L3                      ; (3)

0007DH          L1 :
0FFFFH          L2 :
10000H          L3 :

                C2      CSEG    AT      20050H
20050H                  BR      L4                      ; (4)
27FFFH          L4 :
                                END
```

(1) BR 指令产生 2 字节分支指令 （BR $addr20），因为该行和分支目标之间的位移在 -80H 和 +7FH 范围内。

(2) BR 指令的分支目标在 64KB 内，因此 BR 指令由 3 字节的分支指令 （BR !addr20）代替。

(3) BR 指令将由 4 字节的分支指令 （BR !!addr20）代替。

(4) BR 指令将由 3 字节分支指令 （BR !addr20）代替，因为该行和分支目标之间的位移在 -8000H 和 +7FFFH 范围内。

CALL

让汇编器根据操作数区域内指定的表达式值的范围自动选择 3 或 4 字节的 CALL 分支指令并生成相应的目标码。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	CALL	*expression*	[; *comment*]

[功能]

- CALL 指令让汇编器根据操作数区域内指定的表达式值的范围自动选择 3 或 4 字节的 CALL 分支指令并生成相应的目标码。

[使用]

- 在下列显示的分支指令中，汇编器决定了分支目标的地址范围并自动选择和输出指令，尽可能使用最少的字节数。如果不清楚是否可用 3 字节分支指令描述的话，就使用 CALL 指令。

分支指令	说明
"CALL !addr20"(3 字节)	可用于分支目标在 64KB 内的地址范围。
"CALL $!addr20"(3 字节)	计算分支目标的位移，并可在位移处于 -8000H 至 +7FFFH 范围内使用。
"CALL !!addr20"(4 字节)	用于上述以外的情况

如果操作数 （分支目标）位于与指令不同的可重定位段中且在 BASE 区域之外，指令将由 4 字节指令代替并输出。
如果指令和操作数 （分支目标）位于不同段且在 BASE 区域之外，其类型将不同[注]，即使操作数位于绝对程序段，指令将由 4 字节指令代替。
如果指令和操作数 （分支目标）位于不同段且位于 BASE 区域内，指令将由 3 字节指令 （BR !addr20）代替。

注 不同类型是指：CALL 指令位于绝对程序段的不同可重定位段，或 CALL 指令位于可重定位段的绝对程序段。

- 如果明确知道应采用 3 或 4 字节分支指令进行描述的话，则直接采用合适的指令进行描述。与描述 CALL 指令相比，这可以缩短汇编时间。

[说明]

- CALL 指令只能用于代码段。
- 直接跳转目的作为 CALL 指令的操作数进行描述。
- 为了优化，必须满足下列条件。
  - 表达式中不能出现超过一个的标号或向前引用符号。
  - 不要由 ADDRESS 属性描述 EQU 符号。
  - 不要把 EQU 定义的符号用于描述 ? 带有 ADDRESS 属性的表达式 1 - 带有 ADDRESS 属性的表达式 2

- 不要描述带有 ADDRESS 属性的表达式

如果不满足上述条件，将选择 4 字节的 CALL 指令。

[示例]

```
ADDRESS                 NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  CALL    L1                      ; (1)
00053H                  CALL    L2                      ; (2)
08052H          L1 :
0FFFFH          L2 :


                C2      CSEG    AT      20050H
20050H                  CALL    L3                      ; (3)
27FFFH          L3 :
                        END
```

(1) CALL 指令的分支目标在 64KB 内，因此 CALL 指令由 3 字节的分支指令 （CALL !addr20）代替。

(2) CALL 指令将由 4 字节的分支指令 （CALL !!addr20）代替。

(3) CALL 指令将由 3 字节分支指令 （CALL !addr20）代替，因为该行和分支目标之间的位移在 -8000H 和 +7FFFH 范围内。

4.2.8 宏指令

描述一个源程序时，对每个高使用频率的指令组系列逐一描述是低效率的。这也增加了出错的可能性。
通过宏指令使用宏功能，这样就不需要对相同种类的指令组系列多次描述，并且能提高编制代码的效率。
宏的基本功能是取代一系列的语句。
可获得下列宏指令。

控制指令	概要
MACRO	对在 MACRO 指令和 ENDM 指令之间描述的一系列语句执行宏，该宏被赋予在符号区域中指定的宏名。
LOCAL	说明了指定在操作栏中的符号名是局部符号，仅在宏体内有效。
REPT	仅写在 REPT 指令和 ENDM 指令之间的系列语句所指定的表达数值能重复发挥作用。
IRP	当虚数由运算在 IRP 指令和 ENDM 指令之间系列语句指定的实数替代时，仅实数的数量能重复发挥作用。
EXITM	发展了以 MACRO 指令定义的宏体，并且通过 REPT-ENDM, IRP-ENDM 重复被强制完成。
ENDM	完成了定义为宏函数的语句集。

MACRO

对在 MACRO 指令和 ENDM 指令之间描述的一系列语句执行宏，该宏被赋予在符号区域中指定的宏名。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
macro-name	MACRO	[*formal-parameter*[, ...]]	[; comment]
	:		
	宏体		
	:		
	ENDM		[; *comment*]

[功能]

- 对在 MACRO 指令和 ENDM 指令之间描述的一系列语句 (调用宏体)MACRO 指令执行宏，该宏是指定在符号域中赋予的宏名。

[使用]

- 在源程序中用宏名定义一系列频繁使用的语句。在定义之后只能描述已定义的宏名 (参阅 "(2)　参考宏 "), 并且宏体对应的宏名得到扩展。

[说明]

- MACRO 指令必须与 ENDM 指令成对出现。
- 对于在符号域中要描述的宏名，参阅描述在 "(3)　符号区域 " 中的符号约定 。
- 为了引用宏，要在助记符区域中描述宏名。
- 对于在操作数区域中描述形式参数，应使用与符号约定那样描述的规则。
- 最多 16 个形式参数可以描述每一个宏体。
- 形式参数只有在宏体中有效。
- 如果保留字作为形式参数描述的话，将发生错误。但是 , 如果描述用户定义的符号，将优先识别其作为形式参数。
- 形式参数的数目必须与实参数的数目相同。
- 定义在宏体中的名字和标号 , 如果以 LOCAL 指令声明，则对于一次性宏扩展有效。
- 宏嵌套 （即在宏体内引用其它的宏），允许包含 REPT 和 IRP 指令在内的最多八级嵌套。
- 在单个源文件模块内可定义的宏的数目没有规定限制。换句话说，存储器有效的空间有多大，宏就可定义多大。
- 交叉参考列表不输出形式参数行、参考行和符号名。
- 在宏体中，不能定义两个或更多个段。如果定义，将输出错误。

[示例]

```
        NAME    SAMPLE

ADMAC   MACRO   PARA1, PARA2    ; (1)
        MOV     A, #PARA1
        ADD     A, #PARA2
        ENDM                        ; (2)

        ADMAC   10H, 20H        ; (3)
        END
```

(1) 宏由指定宏名 "ADMAC" 和两个形式参数 "PARA1" 和 "PARA2" 定义。

(2) 这个指令指示了宏定义的结束。

(3) 宏 "ADMAC" 被引用。

LOCAL

说明了指定在操作数区域中的符号名是局部符号，该符号仅在宏体内有效

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
None	LOCAL	*symbol-name*[, ...]	[; comment]

[功能]

- LOCAL 指令说明了在操作数区域中指定的符号名是局部符号，该局部符号仅在宏体内有效。

[使用]

- 如果在宏体内宏定义的符号不止一次被引用，汇编器将针对该符号输出双重定义的错误。通过使用 LOCAL 指令，可以引用 (或调用) 宏，在宏体内定义的符号宏可以多次引用。

[说明]

- 对于在操作数区域中要描述的宏名，参阅描述在 "(3) 符号区域 " 中的符号约定 。
- 在每个宏扩展时，声明为 LOCAL 的符号将由 "??RAnnnn" 符号替代 (其中 n = 0000 至 FFFF)。在宏替换后的符号 "??RAnnnn" 将按全局变量方式处理，保存在符号表中，并且以后可以以符号名 "??RAnnnn" 引用。
- 如果符号在宏体内声明并且不止一次引用，这意味着符号在源程序模块中将定义不止一次。因此，需要声明该符号是局部符号，仅在宏体内有效。
- LOCAL 指令只能在宏定义内使用。
- 必须在操作数区域内定义符号前描述 LOCAL 指令 (换句话说， LOCAL 指令必须在宏体开始时描述)。
- 在源程序模块内， LOCAL 指令定义的所有符号名必须不同 (换句话说，对于在每个宏内使用的局部符号是不能重名)。
- 在操作数区域中指定局部符号的数量没有限制，只要在一个行内即可。然而 , 在宏体内的符号数量限制在 64 个。如果声明 65 个或更多的局部变量 , 汇编器将输出一个错误并作为空宏体保存这宏定义。即使调用该宏，也不会发生扩展。
- 以 LOCAL 指令定义的宏不能嵌套。
- 以 LOCAL 指令定义的符号不能从宏外部被调用 (引用)。
- 在操作数区域中保留字不能描述为符号名。但是 , 如果描述与用户定义相同的符号，将优先识别其作为形式参数。
- 声明为 LOCAL 指令操作数的符号不在交叉参考列表和符号列表中输出。
- LOCAL 指令的语句行不与宏扩展同时输出。
- 如果在宏定义内作出 LOCAL 声明，当其符号与宏定义的形式参数具有相同的名字时，将输出一个错误。

[示例]

```
        NAME    SAMPLE

        ; Macro definition
MAC1    MACRO
        LOCAL   LLAB            ; (1)
LLAB :                          ;
        BR      $LLAB           ; (2)
        ENDM                    ;

        ; Source text
REF1 :  MAC1                    ; (3)

??RA0000 :
        BR      $??RA0000       ; (2)

        BR      !LLAB           ; (4)     <- Error

REF2 :  MAC1                    ; (5)

??RA0001 :
        BR      $??RA0001       ; (2)
        END
```

(1) LOCAL 指令定义符号名 "LLAB" 为局部符号。

(2) 在宏 MAC1 内， BR 指令引用了局部符号 "LLAB"。

(3) 宏引用调用宏 MAC1。

(4) 由于局部符号 "LLAB" 在宏 MAC1 定义之外被引用 , 因此该描述将产生一个错误。

(5) 宏引用调用宏 MAC1。

上述汇编列表的应用示例显示如下。

```
        Assemble list

ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT

  1       1                                  NAME    SAMPLE
  2       2                    M       MAC1  MACRO
  3       3                    M             LOCAL   LLAB            ; (1)
  4       4                    M       LLAB :
  5       5                    M             BR      $LLAB           ; (2)
  6       6                    M             ENDM
  7       7
  8       8     000000                 REF1 :  MAC1                    ; (3)
          9                    #1      ;
 10             000000         #1      ??RA0000:
 11             000000  14FE   #1             BR      $??RA0000       ; (2)
  9      12
 10      13     000002  2C0000                 BR      !LLAB           ; (4)
***  ERROR E2407 , STNO  13 ( 0 ) Undefined symbol reference 'LLAB'
***  ERROR E2303 , STNO  13 ( 13 ) Illegal expression
 11      14
 12      15     000005                 REF2 :  MAC1                    ; (5)
 16                            #1      ;
 17             000005         #1      ??RA0001 :
 18             000005  14FE   #1             BR      $??RA0001       ; (2)
 13      19
 14      20                                 END
```

REPT

让汇编器重复展开在该指令和 ENDM 指令之间描述的一系列语句，展开的次数等于在操作数区域中指定的表达式数值。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	REPT	*absolute-expression*	[; *comment*]
	:		
	ENDM		[; *comment*]

[功能]

- REPT 指令让汇编器重复展开在该指令和 ENDM 指令之间描述的一系列语句 (调用 REPT-ENDM 块)，展开的次数等于在操作数区域中指定的表达式数值。

[使用]

- 使用 REPT 和 ENDM 指令在源程序中重复描述一系列语句。

[说明]

- 如果 REPT 指令不是与 ENDM 指令成对出现就会发生错误。
- 在 REPT-ENDM 块、宏引用、 REPT 指令和 IRP 指令中，可最多八级嵌套。
- 如果 EXITM 指令出现在 REPT-ENDM 块中 , 汇编器将终止 REPT-ENDM 块的子扩展。
- 汇编控制指令可在 REPT-ENDM 块中描述。
- 宏定义不能在 REPT-ENDM 块中描述。
- 在操作数区域中描述的绝对表达式以无符号 16 位数运作。
  如果表达式的值是 0，就不展开。

[示例]

```
NAME    SAMP1
CSEG
    ; REPT-ENDM block
REPT    3               ; (1)
        INC     B
        DEC     C
            ; Source text
ENDM                        ; (2)
END
```

(1) REPT 指令让汇编器连续三次展开 REPT-ENDM 块。

(2) 该指令指示了 REPT-ENDM 块的结束。

当上面源程序汇编时， REPT-ENDM 的扩展如下列汇编列表所示：

```
NAME    SAMP1
CSEG
REPT    3
        INC     B
        DEC     C
ENDM
        INC     B
        DEC     C
        INC     B
        DEC     C
        INC     B
        DEC     C
END
```

由语句 (1) 和 (2) 定义的 REPT-ENDM 块被展开了三次。
在汇编列表中，源程序模块中由 REPT 指令定义的语句 (1) 和 (2) 没有显示。

IRP

让汇编器重复展开在 IRP 指令和 ENDM 指令之间描述的一系列语句，当以指定在操作数区域的实参数（次序从左开始）替代形式参数时，展开的次数等于实参数的数目。

[格式描述]

符号区域	助记符区域	操作数区域	注释区域
[*label*:]	IRP	*formal-parameter*,<[*actual-parameter*[, ...]]>	[; *comment*]
	:		
	ENDM		[; *comment*]

[功能]

- IRP 指令让汇编器重复展开在 IRP 指令和 ENDM 指令之间描述的一系列语句 (调用 IRP-ENDM 块)，当以指定在操作数区域的实参数（次序从左开始）替代形式参数时，展开的次数等于实参数的数目。

[使用]

- 使用 IRP 和 ENDM 指令来描述语句时，只有一些语句成为变量并重复出现在源程序中。

[说明]

- IRP 指令必须与 ENDM 指令成对。
- 操作数区域中最多可描述 16 个实参数。
- 在 IRP-ENDM 块、宏引用、 REPT 指令和 IRP 指令中，可最多八级嵌套。
- 如果 EXITM 指令出现在 IRP-ENDM 块中 , 将终止由汇编器进行的之后 IRP-ENDM 块的扩展。
- 宏定义不能在 IRP-ENDM 中描述。
- 汇编控制指令可以在 IRP-ENDM 中描述。

[示例]

```
NAME    SAMP1
CSEG

IRP     PARA, <0AH, 0BH, 0CH>           ; (1)
    ; IRP-ENDM block
ADD     A, #PARA
MOV     [ DE ] , A
ENDM                        ; (2)
    ; Source text
END
```

(1) 形式参数是 "PARA"，实参数是下面三个："0AH"， "0BH" 和 "0CH"。
当以实参数 "0AH"、 "0BH" 和 "0CH" 替代形式参数 "PARA" 时， IRP 指令使汇编器三次展开 IRP-ENDM 块 (即实参数的数量)。

(2) 这个指令指示了 IRP-ENDM 块的结束。

当上面源程序汇编时， IRP-ENDM 块如下面汇编列表所示展开：

```
NAME    SAMP1
CSEG
    ; IRP-ENDM block
ADD     A, #0AH         ; (3)
MOV     [ DE ] , A
ADD     A, #0BH         ; (4)
MOV     [ DE ] , A
ADD     A, #0CH         ; (5)
MOV     [ DE ] , A
    ; Source text
END
```

由语句 (1) 和 (2) 定义的 IRP-ENDM 块被展开三次 (等于实参数的数目)。

(3) 在此 ADD 指令中， PARA 用 0AH 替代。

(4) 在此 ADD 指令中， PARA 用 0BH 替代。

(5) 在此 ADD 指令中， PARA 用 0CH 替代。

EXITM

强制终止由 MACRO 指令定义的宏体扩展和 REPT-ENDM 或 IRP-ENDM 块的重复运行。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
[*label*:]	EXITM	*None*	[; *comment*]

[功能]

- EXITM 指令强制终止由 MACRO 指令定义的宏体扩展和 REPT-ENDM 或 IRP-ENDM 块的重复运行。

[使用]

- 该功能主要用于条件汇编功能 (参阅 "4.3.7　条件汇编控制指令 ") 在 MACRO 指令定义的宏体中使用的时候。
- 如果条件汇编功能与其它指令共同在宏体中使用时，源程序中不能汇编的部分，只有在强制用 EXITM 指令值从宏返回控制时才可汇编。。在这种情况中，一定要使用 EXITM 指令。

[说明]

- 如果 EXITM 指令在宏体中表述， ENDM 指令以上的指令作为宏体保存。
- EXITM 指令仅在宏扩展期间指示宏的结束。
- 如果在 EXITM 指令的操作数区域中描述事项，汇编器将输出一个错误，但仍然执行 EXITM 进程。
- 如果 EXITM 指令出现在宏体中，汇编器将 IF/_IF/ELSE/ELSEIF/_ELSEIF/ENDIF 块的嵌套级强制返回到汇编器进入到宏体的那级。
- 如果在 INCLUDE 文件中出现 EXITM 指令，在宏体中产生展开 INCLUDE 控制指令，汇编器接受 EXITM 有效并且在这级上终止宏扩展。

[示例]

```
        NAME    SAMP1
MAC1    MACRO                           ; (1)
          ; macro body
        NOT1    CY
$       IF ( SW1 )                      ; (2)    <- IF block
                BT      A.1, $L1
                EXITM                   ; (3)
$       ELSE                            ; (4)    <- ELSE block
                MOV1    CY, A.1
                MOV     A, #0
$       ENDIF                           ; (5)
$       IF ( SW2 )                      ; (6)    <- IF block
                BR      [HL]
$       ELSE                            ; (7)    <- ELSE block
                BR      [DE]
$       ENDIF                           ; (8)
          ; Source text
        ENDM                        ; (9)

        CSEG
$       SET ( SW1 )                     ; (10)
        MAC1                            ; (11)   <- Macro reference
L1 :    NOP
        END
```

(1) "MAC1" 在宏体内使用条件汇编功能 (2) 和 (4) 至 (8)。

(2) 这里为条件汇编定义了一个 IF 块。
如果 "SW1" 为真 (不是 "0"),ELSE 块就不汇编。

(3) 这个指令强制终止 (4) 中的宏体扩展及其后的语句。
如果省略 EXITM，当宏展开时，汇编器就进行 (6) 及其后的汇编处理。

(4) 这里为条件汇编定义了一个 ELSE 块。
如果 "SW1" 为假 ("0"), 就汇编 ELSE 块。

(5) ENDIF 控制指令指示了条件汇编的结束。

(6) 这里为条件汇编定义了另一个 IF 块。
如果 "SW2" 是真 (不是 "0"), 就汇编 IF 块。

(7) 这里为条件汇编定义了另一个 ELSE 块。
如果 "SW2" 为假 ("0")，就汇编 ELSE 块。

(8) ENDIF 指令指示了 (6) 和 (7) 中的条件汇编处理结束。

(9) 这个指令指示了宏体的结束。

(10) SET 控制指令对开关 "SW1" 取真值 (不是 "0") 并且设置条件汇编的条件。

(11) 这个宏引用调用宏 "MAC1"。

备注　　这里的示例中 , 使用了条件汇编控制指令。参阅 "4.3.7　条件汇编控制指令 ". 参见 "4.4　宏 " 的宏体和宏扩展。

上述汇编列表的应用示例显示如下。

```
        NAME    SAMP1
MAC1    MACRO                           ; (1)
          :
        ENDM                        ; (9)
        CSEG
$       SET ( SW1 )                     ; (10)
        MAC1                    ; (11)
          ; Macro-expanded part
        NOT1    CY
$       IF ( SW1 )
                BT      A.1, $L1
          ; Source text
L1 :    NOP
        END
```

宏 "MAC1" 的宏体通过引用 (11) 中的宏进行扩展。

由于在 (10) 中的开关 "SW1" 设置为真，因此在宏体中的第一个 IF 块被汇编。由于 EXITM 是描述在 IF 块的末尾 , 因此不执行随后的宏扩展。

ENDM

命令汇编器终止执行定义为宏功能的一系列语句。

[格式描述]

符号区域	助记符区域	操作数区域	注释区
None	ENDM	*None*	[; *comment*]

[功能]

- ENDM 指令命令汇编器终止执行定义为宏功能的一系列语句。

[使用]

- ENDM 指令必须始终在 MACRO、 REPT 和 / 或 IRP 指令后的一系列语句的末尾描述。

[说明]

- 在 MACRO 指令和 ENDM 指令之间描述的语句是宏体。
- 在 REPT 指令和 ENDM 指令之间描述的语句是 REPT-ENDM 块。
- 在 IRP 指令和 ENDM 指令之间描述的语句是 IRP-ENDM 块。

[示例]

(1) MACRO-ENDM

```
        NAME    SAMP1
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
          :
        END
```

(2) REPT-ENDM

```
NAME    SAMP2
CSEG
  :
REPT    3
        INC     B
        DEC     C
ENDM
  :
END
```

(3) IRP-ENDM

```
NAME    SAMP3
CSEG
  :
IRP     PARA, <1, 2, 3>
        ADD     A, #PARA
        MOV     [ DE ] , A
ENDM
  :
END
```

4.2.9 汇编终止指令

汇编终止指令指定了汇编器中的源模块完成。. 该汇编终止指令必须描述在每个源模块的末尾。

汇编器处理源模块直到出现汇编完成指令。因此 , 在带有 REPT 块和 IRP 块的源模块中，如果汇编指令出现在 ENDM 之前， REPT 块和 IRP 块将不起作用。

下列为可获得的汇编终止指令。

控制指令	概要
END	源模块终止声明

END

源模块终止声明

[格式描述]

符号区域	助记符区域	操作数区域	注释区
None	END	*None*	[; *comment*]

[功能]

- END 指令指定了汇编器中的源模块的结束。

[使用]

- END 指令必须描述在每个源模块的末尾。

[说明]

- 汇编器继续汇编源模块直到 END 指令出现在源模块中。因此， END 指令需要在每个源文件模块的末尾。
- 在 END 指令之后总是输入换行码 （LF）。
- 如果不是空白， tab， LF 或注释的任意语句出现在 END 指令之后，汇编器就输出一个警告信息。

[示例]

```
NAME    SAMPLE
DSEG
  :
CSEG
  :
END                             ; (1)
```

(1) 始终在每个源文件模块的末尾描述 END 指令。

4.3 控制指令

本章描述控制指令。

控制指令为汇编器的运行提供详细的指令。

4.3.1 概要

控制指令为汇编器的运行提供详细的指令，因此也写入源程序中。

控制指令不会成为目标代码生成的对象。

控制指令的分类显示如下。

表 4-20. 控制指令列表

控制指令类型	控制指令
汇编目标类型规范控制指令	PROCESSOR
调试信息输出控制指令	DEBUG, NODEBUG, DEBUGA, NODEBUGA
交叉引用列表输出规范控制指令	XREF, NOXREF, SYMLIST, NOSYMLIST
Include 控制指令	INCLUDE
汇编列表控制指令	EJECT, LIST, NOLIST, GEN, NOGEN, COND, NOCOND, TITLE, SUBTITLE, FORMFEED, NOFORMFEED, WIDTH, LENGTH, TAB
条件汇编控制指令	IF, _IF, ELSEIF, _ELSEIF, ELSE, ENDIF, SET, RESET
Kanji 代码控制指令	KANJICODE
其他控制指令	TOL_INF, DGS, DGL

控制指令与指令一样在源程序中指定。

此外，在 " 表 4-20. 控制指令列表 " 所示的控制指令中，下列指令可作为汇编器选项写入，甚至在汇编器运行时可写入命令行。

表 4-21. 控制指令和汇编器选项

控制指令	汇编选项
PROCESSOR	-c
DEBUG/NODEBUG	-g/-ng
DEBUGA/NODEBUGA	-ga/-nga
XREF/NOXREF	-kx/-nkx
SYMLIST/NOSYMLIST	-ks/-nks
TITLE	-lh
FORMFEED/NOFORMFEED	-lf/-nlf
WIDTH	-lw
LENGTH	-ll
TAB	-lt
KANJICODE	-zs/-ze/-zn

4.3.2　汇编目标类型规范控制指令

汇编目标类型规范控制指令规定了在源模块文件中的汇编目标类型。
有下列汇编目标类型规范控制指令。

控制指令	概要
PROCESSOR	在源模块文件中指定汇编目标类型。

PROCESSOR

在源模块文件中指定汇编目标类型。

[格式描述]

```
[Δ]$[Δ]PROCESSOR[Δ]([Δ]processor-type[Δ])
[Δ]$[Δ]PROCESSOR[Δ]([Δ]processor-type[Δ])；缩写形式
```

[功能]

- 在源模块文件中， PROCESSOR 控制指令指定了用于汇编的目标器件的处理器类型。

[使用]

- 用于汇编的目标器件的处理器类型必须在源模块文件或汇编器的启动命令行中指定。
- 如果在每个源模块文件中省略对用于汇编的目标器件的处理器类型的指定，那么在每次汇编操作时必须指定处理器类型。因此，在每个源模块文件中通过指定用于汇编的目标器件，可在启动汇编器时为您节省时间和减少麻烦。

[说明]

- PROCESSOR 控制指令只能在源模块文件的头文件部分进行定义。如果控制指令在其他地方被定义，那么汇编程序将终止。
- 关于可定义处理器的名称 , 见 用户手册中的设备使用或者？设备文件操作的注意事项 "。
- 如果定义的处理器类型不同于实际经由汇编处理的目标设备，则该汇编程序将会中止。
- 只有一个 PROCESSOR 控制指令可以在模块的头部进行定义。
- 该用于汇编的目标器件的处理器类型也可在汇编程序的起始命令行中，通过汇编选项 （-c）进行定义。如果指定的处理器类型在源模块文件和启动命令行之间存在差异，该汇编程序将输出一个警告信息，并优先考虑处理启动命令行中的定义的处理器类型。
- 即使汇编程序的选项 （-c）已经在启动命令行中设定，汇编程序会在 PROCESSOR 控制指令中进行语法检查。
- 如果处理器类型没有在任何源模块文件或启动命令行中设定，汇编程序将中止。

[应用示例]

```
$ PROCESSOR ( f1166a0 )
$DEBUG
$        XREF

         NAME    TEST
           :
         CSEG
```

4.3.3　调试信息输出控制指令

调试信息输出控制指令可以在源模块文件中针对目标模块文件进行调试信息输出的设定。
有下列调试信息输出控制指令。

控制指令	概要
DEBUG	在目标模块文件中添加局部符号信息。
NODEBUG	不能在目标模块文件中添加局部符号信息。
DEBUGA	在目标模块文件中添加汇编源文件调试信息。
NODEBUGA	不能在目标模块文件中添加汇编源文件调试信息。

DEBUG

在目标模块文件中添加局部符号信息。

[格式描述]

```
[Δ]$[Δ]DEBUG            ; 默认假设
[Δ]$[Δ]DG               ; 缩写形式
```

[功能]

- DEBUG 控制指令通知汇编器，将局部符号信息添加到目标模块文件中。
- NODEBUG 控制指令通知汇编器，将局部符号信息不添加到目标模块文件中。然而，在这种情况下，段名称是输出到一个目标模块文件中。

[使用]

- 当包含局部符号的调试运行时，可以使用 DEBUG 控制指令。

[说明]

- DEBUG 或 NODEBUG 控制指令只能在源模块文件的头部进行定义。
- 如果 BEBUG 或者 NODEBUG 控制指令省略，那么汇编器会默认为 DEBUG 控制指令已经进行了定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 在启动命令行中使用汇编程序选项 （-g/-ng) 可以定义另外的局部符号信息。
- 如果在源模块文件中的控制指令规格不同于启动命令行中的规格，则在启动命令行中的规格的优先级较高。
- 即使定义了汇编程序中的选项 （-ng），汇编程序仍可以通过 DEBUG 或者 NODEBUG 控制指令来执行语法检查。

NODEBUG

不能在目标模块文件中添加局部符号信息。

[格式描述]

```
[Δ]$[Δ]NODEBUG
[Δ]$[Δ]NODG                    ; 缩写形式
```

[功能]

- NODEBUG 控制指令通知汇编器，将局部符号信息不添加到目标模块文件中。然而，在这种情况下，段名称是输出到一个目标模块文件中。
- " 局部符号信息 " 指的是除了模块名称以及 PUBLIC, EXTRN 和 EXTBIT 之外的符号。

[使用]

- 下列为使用 NODEBUG 控制指令的情况 :
- 符号调试的执行仅仅是用于全局符号
- 无符号的调试执行
- 只是目标是需要的 （同样也是为 PROM 的赋值）

[说明]

- DEBUG 或 NODEBUG 控制指令只能在源模块文件的头部进行定义。
- 如果 BEBUG 或者 NODEBUG 控制指令省略，那么汇编器会默认为 DEBUG 控制指令已经进行了定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 在启动命令行中使用汇编程序选项 （-g/-ng) 可以定义另外的局部符号信息。
- 如果在源模块文件中的控制指令规格不同于启动命令行中的规格，则在启动命令行中的规格的优先级较高。
- 即使定义了汇编程序中的选项 （-ng），汇编程序仍可以通过 DEBUG 或者 NODEBUG 控制指令来执行语法检查。

DEBUGA

在目标模块文件中添加汇编源文件调试信息。

[格式描述]

```
[Δ]$[Δ]DEBUGA              ; 默认假设
```

[功能]

- DEBUGA 控制指令通知汇编器，将汇编程序源调试信息添加到目标模块文件中。

[使用]

- 在进行汇编程序级的调试时，使用 DEBUGA 控制指令。在进行汇编程序级的调试时，需要一个集成调试器。

[说明]

- DEBUGA 或 NODEBUGA 控制指令只能在选择的源模块文件的头部进行定义。
- 如果 BEBUGA 或者 NODEBUGA 控制指令省略，那么汇编器会默认为 DEBUGA 控制指令已经进行了定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 在启动命令行中，使用汇编程序选项 （-ga/-nga) 可以定义另外汇编程序源调试信息。
- 如果在源模块文件中的控制指令规格不同于启动命令行中的规格，则在启动命令行中的规格的优先级较高。
- 即使定义了汇编程序中的 （-nga）选项，汇编程序仍可以通过 DEBUGA 或者 NODEBUGA 控制指令来执行语法检查。
- 如果汇编调试信息输出的 C 编译器 , do not describe the debug information output control instructions when assembling the output assemble source. 汇编中必要的控制指令输出至汇编程序源同样也经由 C 编译器控制。

NODEBUGA

不能在目标模块文件中添加汇编源文件调试信息。

[格式描述]

```
[Δ]$[Δ]NODEBUGA
```

[功能]

- NNODEBUGA 控制指令通知汇编器将汇编程序源调试信息添加到目标模块文件中。

[使用]

- 使用 NODEBUGA 控制指令当 :
- 缺少汇编源程序调试将无法执行
- 只是目标是需要的 （同样也是为 PROM 的赋值）

[说明]

- DEBUGA 或 NODEBUGA 控制指令只能在选择的源模块文件的头部进行定义。
- 如果 BEBUGA 或者 NODEBUGA 控制指令省略，那么汇编器会默认为 DEBUGA 控制指令已经进行了定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 在启动命令行中，使用汇编程序选项 （-ga/-nga) 可以定义另外汇编程序源调试信息。
- 如果在源模块文件中的控制指令规格不同于启动命令行中的规格，则在启动命令行中的规格的优先级较高。
- 即使定义了汇编程序中的 （-nga）选项，汇编程序仍可以通过 DEBUGA 或者 NODEBUGA 控制指令来执行语法检查。
- 如果汇编调试信息输出的 C 编译器 , do not describe the debug information output control instructions when assembling the output assemble source. 汇编中必要的控制指令输出至汇编程序源同样也经由 C 编译器控制。

4.3.4　交叉引用列表输出规范控制指令

交叉引用列表输出规范控制指令指定源模块文件中的交叉引用列表输出。
有下列交叉引用列表输出规范控制指令。

控制指令	概要
XREF	输出交叉引用列表到汇编列表文件中。
NOXREF	不输出交叉引用列表到汇编列表文件中。
SYMLIST	输出符号列表到列表文件中。
NOSYMLIST	不输出符号列表到列表文件中。

XREF

输出交叉引用列表到汇编列表文件中。

[格式描述]

```
[Δ]$[Δ]XREF
[Δ]$[Δ]XR                ; 缩写形式
```

[功能]

- XREF 控制指令通知汇编器将交叉引用列表输出到汇编列表文件。

[使用]

- 当用户想获得关于源模块文件中定义的每个符号是否被引用或者在源模块文件有多少这样定义的符号的信息时，可以使用 XREF 控制指令输出交叉引用列表。
- 如果在每次汇编操作时都必须设定交叉引用列表中的输出和非输出项的话，您可通过在源模块文件中定义 XREF 以及 NOXREF 控制指令来节约时间和工作量。

[说明]

- XREF 或 NOXREF 控制指令只能在源模块文件的头部进行定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 也可在启动命令行中，通过汇编程序选项 （-kx/-nkx）指定交叉引用列表中的输出项以及非输出项。
- 如果在源模块文件中的控制指令规格不同于启动命令行中汇编程序选项的规范，则在启动命令行中规格的优先级要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编程序会在 XREF/NOXREF 控制指令中执行语法检查。

NOXREF

不输出交叉引用列表到汇编列表文件中。

[格式描述]

```
[Δ]$[Δ]NOXREF        ; 默认设定
[Δ]$[Δ]NOXR          ; 缩写形式
```

[功能]

- NOXREF 控制指令通知汇编程序不要将交叉引用列表输出到汇编列表文件。

[使用]

- 当用户想获得关于源模块文件中定义的每个符号是否被引用或者在源模块文件有多少这样定义的符号的信息时，可以使用 XREF 控制指令输出交叉引用列表。
- 如果在每次汇编操作时都必须设定交叉引用列表中的输出和非输出项的话，您可通过在源模块文件中定义 XREF 以及 NOXREF 控制指令来节约时间和工作量。

[说明]

- XREF 或 NOXREF 控制指令只能在源模块文件的头部进行定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 也可在启动命令行中，通过汇编程序选项 （-kx/-nkx）指定交叉引用列表中的输出项以及非输出项。
- 如果在源模块文件中的控制指令规格不同于启动命令行中汇编程序选项的规范，则在启动命令行中规格的优先级要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编程序会在 XREF/NOXREF 控制指令中执行语法检查。

SYMLIST

输出符号列表到列表文件中

[格式描述]

```
[Δ]$[Δ]SYMLIST
```

[功能]

- SYMLIST 控制指令通知汇编程序输出符号列表到列表文件中。

[使用]

- 使用 SYMLIST 控制指令输出符号列表。

[说明]

- SYMLIST 或 NOSYMLIST 控制指令只能在源模块文件的头部进行定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 符号列表输出也可通过启动命令行中的汇编程序选项 （-ks/-nks) 进行指定。
- 如果在源模块文件中的控制指令规格不同于启动命令行中汇编程序选项的规范，则在启动命令行中规格的优先级要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编程序会在 SYMLIST/NOSYMLIST 控制指令中执行语法检查。

NOSYMLIST

不输出符号列表到列表文件中。

[格式描述]

```
[Δ]$[Δ]NOSYMLIST          ; 默认设定
```

[功能]

- NOSYMLIST 控制指令通知汇编程序不输出符号列表到列表文件中。

[使用]

- 使用 NOSYMLIST 控制指令将不输出符号列表。

[说明]

- SYMLIST 或 NOSYMLIST 控制指令只能在源模块文件的头部进行定义。
- 如果定义了两个或者更多的控制指令，最后一个定义的控制指令的优先级将最高。
- 符号列表输出也可通过启动命令行中的汇编程序选项 （-ks/-nks) 进行指定。
- 如果在源模块文件中的控制指令规格不同于启动命令行中汇编程序选项的规范，则在启动命令行中规格的优先级要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编程序会在 SYMLIST/NOSYMLIST 控制指令中执行语法检查。

4.3.5　Include 控制指令

在源模块中需要引用其他源模块文件时可以使用 Include 控制指令
有效地使用 include 控制指令，可以减少用以描述源的时间。
有下列 include 控制指令。

控制指令	概要
INCLUDE	从另一个源模块文件中引用一系列语句。

INCLUDE

从另一个源模块文件中引用一系列语句。

[格式描述]

```
[Δ]$[Δ]INCLUDE[Δ]([Δ]filename[Δ])
[Δ]$[Δ]IC[Δ]([Δ]filename[Δ])              ; Abbreviated format
```

[功能]

- INCLUDE 控制指命令汇编器，插入并扩展指定文件的内容，该指定文件从用于汇编的源程序中开始。

[使用]

- 当两个或两个以上的源模块共享一些较大的成组语句时可以将他们合并至单一文件作为一个 INCLUDE 文件。如果在每个源模块中都必须使用成组的语句，使用 INCLUDE 控制指令可以指定这些需要生成 INCLUDE 文件的名字。
  得益于这些控制指令用户可以在描述源模块时节约大量的时间和人力。

[说明]

- INCLUDE 控制指令只能在普通源程序中进行声明。
- INCLUDE 文件的路径名和驱动器明可以通过汇编器选项 (-I) 进行指定。
- 汇编器通过以下顺序搜索 INCLUDE 文件的读取路径 :

(1) 当 INCLUDE 文件没有以路径名规格指定时

(a) 源文件所在的路径

(b) 通过汇编器选项 (-I) 指定的路径

(c) 通过环境变量 INC78K0R 指定的路径

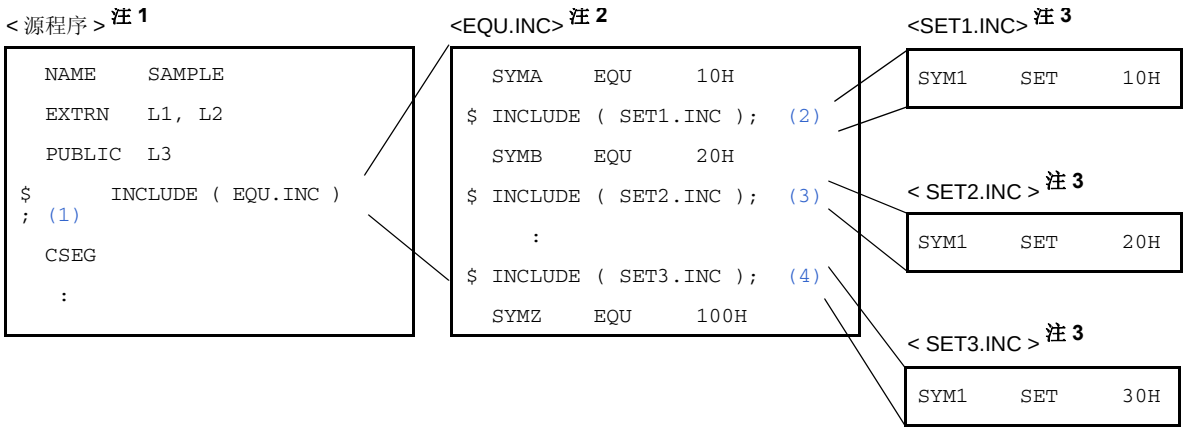
(2) 当指定的 INCLUDE 文件具有路径名时

如果是以反斜线符号 (\) 开头的驱动器名称或路径名指定 INCLUDE 文件， INCLUDE 文件指定的路径会在 INCLUDE 文件名之前 . 如果以相对路径 （不是以 \ 开头）指定 INCLUDE 文件，路径名将以上述 (1) 中描述的顺序添加在 INCLUDE 文件名之前。

- INCLUDE 文件的嵌套共有七层。换言之，汇编列表中显示的 INCLUDE 文件的嵌套共有 8 层 （? 嵌套 ? 在这里指的是在一个 INCLUDE 文件中包含一个或更多的 INCLUDE 文件）。
- 在 INCLUDE 文件中不需要描述 END 命令。
- 如果指定的 INCLUDE 文件不能打开时，汇编器会终止操作。
- INCLUDE 文件必须通过 IF 或者 _IF 控制指令结束，该指令可能和 ENDIF 控制指令成对出现在 INCLUDE 文件中。如果位于 INCLUDE 文件展开的入口位置的 IF 层与 INCLUDE 文件展开之后紧接着的那个 IF 层不相关的话，汇编器将输出错误信息并强制 IF 层返回到 INCLUDE 文件展开的入口位置的那一层。

- 在 INCLUDE 文件中定义宏时，宏的定义必须在 INCLUDE 文件中完成。如果在 INCLUDE 文件中意外出现了 ENDM 命令 （没有相应的 MACRO 命令），就会输出一个错误信息并且 ENDM 命令将被忽略。如果在 INCLUDE 文件中，丢失了用于 MACRO 命令的 ENDM 命令时，汇编器会输出错误信息但是会假设已描述了相应的 ENDM 命令并继续处理宏定义。
- 在 include 文件中不能定义两个或两个以上段。如果定义，将输出错误。

[应用示例]



(1) 这条控制指令设定 "EQU.INC" 作为 INCLUDE 文件。

(2) 这条控制指令设定 "SET1.INC" 作为 INCLUDE 文件。

(3) 这条控制指令设定 "SET2.INC" 作为 INCLUDE 文件。

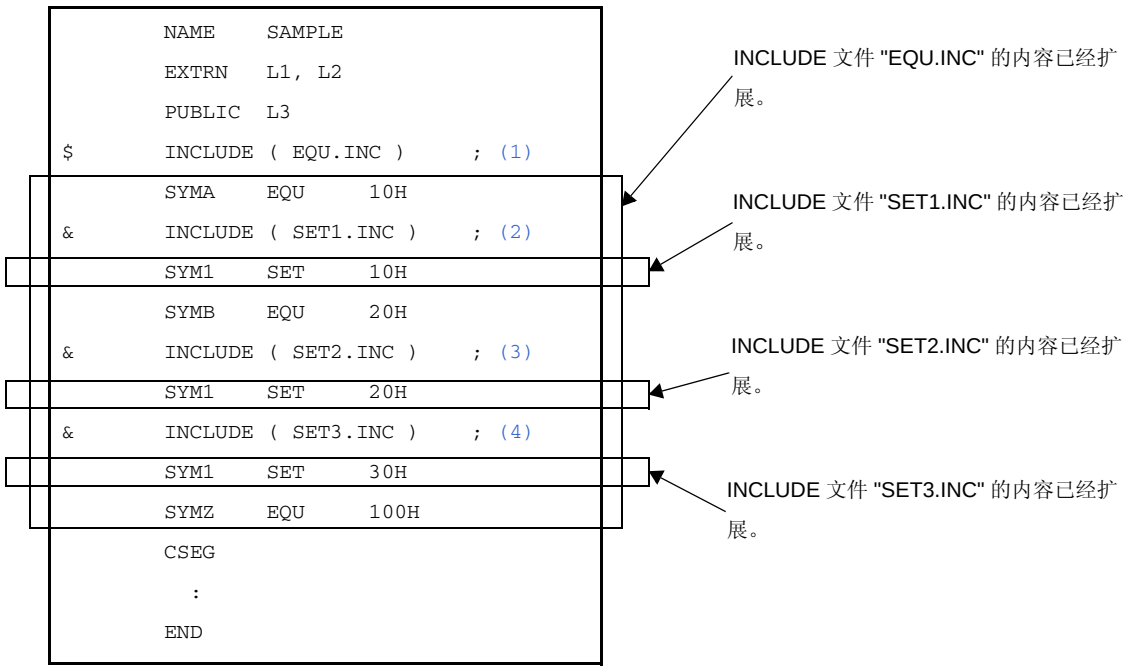
(4) 这条控制指令设定 "SET3.INC" 作为 INCLUDE 文件。

注 **1.** 在源文件中，可以指定两条或两条以上的 $IC 控制指令。可以不止一次指定同一个 INCLUDE 文件。

2. 为 INCLUDE 文件 "EQU.INC" 指定两条或两条以上的 $IC 控制指令。

3. "SET1.INC"、 "SET2.INC" 和 "SET3.INC"INCLUDE 文件均不能指定 $IC 控制指令。

当这个源程序进行汇编时， INCLUDE 文件的内容扩展如下：



4.3.6　汇编列表控制指令

在源模块文件中使用汇编列表控制指令来控制其输出格式例如换页、禁止列表输出以及副标题输出。
有下列汇编列表控制指令。

控制指令	概要
EJECT	说明汇编列表分页符。
LIST	说明汇编列表输出的起始位置。
NOLIST	说明汇编列表输出的终止位置。
GEN	输出宏定义行、引用行以及宏扩展行到汇编列表。
NOGEN	不输出宏定义行、引用行以及宏扩展行到汇编列表。
COND	输出条件汇编器中有效以及无效部分到汇编列表。
NOCOND	不输出条件汇编中有效以及无效部分到汇编列表。
TITLE	在每页汇编列表的头部 TITLE 列中打印字符串，符号列表以及交叉引用列表。
SUBTITLE	在汇编列表头部的 SUBTITLE 列中打印字符串。
FORMFEED	在列表文件的最后插入换页码。
NOFORMFEED	不在列表文件的最后输出格式回馈。
WIDTH	指定在列表文件中一行最大的字符数。
LENGTH	指定列表文件每页行数。
TAB	指定列表文件标签的字符数。

EJECT

说明汇编列表分页符。

[格式描述]

```
[Δ]$[Δ]EJECT
[Δ]$[Δ]EJ              ; Abbreviated format
```

[默认设定]

- EJECT 控制指令不用指定。

[功能]

- EJECT 控制指令导致汇编器处理汇编列表的跳页 （换页）工作。

[使用]

- 在汇编列表需要换页的源模块中，在指定行中声明 EJECT 控制指令。

[说明]

- EJECT 控制指令只能在普通源程序中进行声明。
- EJECT 控制指令镜像输出后汇编列表的换页才可以执行。
- 如果在启动命令行中指定汇编器选项 (-np) 或 (-llo)，或者汇编列表输出被其他控制指令禁止时， EJECT 控制指令将变得无效。
  参阅 " 建立 78K0R"，了解更多汇编器选项。
- 如果 EJECT 控制指令中有非法描述，那么汇编器将会输出错误信息。

[应用示例]

```
          :
        MOV     [DE+], A
        BR      $$
$       EJECT                                   ; (1)
          :
        CSEG
          :
        END
```

(1) 换页由 EJECT 控制指令执行。

上述汇编列表的应用示例显示如下。

```
          :
        MOV     [DE+], A
        BR      $$
$       EJECT                                   ; (1)
-----------------------------------------------page ejection-----------------------
          :
        CSEG
          :
        END
```

LIST

说明汇编列表输出的起始位置。

[格式描述]

```
[Δ]$[Δ]LIST          ; Default assumption
[Δ]$[Δ]LI            ; Abbreviated format
```

[功能]

- LIST 控制指令在汇编器中指出汇编列表输出必须开始的那一行。

[使用]

- 使用 LIST 控制指令取消由 NOLIST 控制指令指定的被输出禁止的汇编列表。
  通过 NOLIST 和 LIST 的结合使用，用户可以控制汇编列表的数量同样还可以控制列表的内容。

[说明]

- LIST 控制指令只能在普通源程序中进行描述。
- 如果 LIST 控制指令在 NOLIST 控制指令之后指定，则在 LIST 控制指令之后描述的语句将会再次在汇编列表上输出。 LIST 和 NOLIST 控制指令的镜像也同样会被输出到汇编列表中。
- 如果 LIST 和 NOLIST 控制指令都没有指定，在源模块中的所有语句将全部输出到汇编列表中。

[应用示例]

```
        NAME    SAMP1
$       NOLIST          ; (1)
DATA1   EQU     10H     ; The statement will not be output to the assembly list.
DATA2   EQU     11H     ; The statement will not be output to the assembly list.
          :             ; The statement will not be output to the assembly list.
DATAX   EQU     20H     ; The statement will not be output to the assembly list.
DATAY   EQU     20H     ; The statement will not be output to the assembly list.
$       LIST            ; (2)
        CSEG
          :
        END
```

(1) 由于 NOLIST 控制指令在这里没有进行指定，"$ NOLIST" 之后的语句直到 （2）中的 LIST 控制指令在汇编列表中将不会输出。
NOLIST 控制指令的镜像将被输出到汇编列表中。

(2) 由于 LIST 控制指令在此处进行了定义，在控制指令之后的语句将再次输出到汇编列表中。
LIST 控制指令的镜像也将被输出到汇编列表中。

NOLIST

说明汇编列表输出的终止位置。

[格式描述]

```
[Δ]$[Δ]NOLIST
[Δ]$[Δ]NOLI            ; Abbreviated format
```

[功能]

- NOLIST 控制指令说明了汇编器中的汇编列表必须禁止输出的行。
  在 NOLIST 控制指令之后进行描述的源语句将进行编译，但是直到源程序中的 LIST 控制指令出现后才会在汇编列表中进行输出。

[使用]

- 使用 NOLIST 控制指令可以限制汇编列表的输出量。
- 使用 LIST 控制指令取消由 NOLIST 控制指令指定的被输出禁止的汇编列表。
  通过 NOLIST 和 LIST 的结合使用，用户可以控制汇编列表的数量同样还可以控制列表的内容。

[说明]

- NOLIST 控制指令只能在普通源程序中进行描述。
- NOLIST 控制指令具有可以禁止汇编列表的输出并且不打断汇编处理的功能。
- 如果 LIST 控制指令在 NOLIST 控制指令之后指定，则在 LIST 控制指令之后描述的语句将会再次在汇编列表上输出。 LIST 和 NOLIST 控制指令的镜像也同样会被输出到汇编列表中。
- 如果 LIST 和 NOLIST 控制指令都没有指定，在源模块中的所有语句将全部输出到汇编列表中。

[应用示例]

```
        NAME    SAMP1
$       NOLIST          ; (1)
DATA1   EQU     10H     ; The statement will not be output to the assembly list.
DATA2   EQU     11H     ; The statement will not be output to the assembly list.
          :             ; The statement will not be output to the assembly list.
DATAX   EQU     20H     ; The statement will not be output to the assembly list.
DATAY   EQU     20H     ; The statement will not be output to the assembly list.
$       LIST            ; (2)
        CSEG
          :
        END
```

(1) 由于 NOLIST 控制指令在这里没有进行指定 , "$ NOLIST" 之后的语句直到 （2）中的 LIST 控制指令在汇编列表中将不会输出。
NOLIST 控制指令的镜像将被输出到汇编列表中。

(2) 由于 LIST 控制指令在此处进行了定义，在控制指令之后的语句将再次输出到汇编列表中。
LIST 控制指令的镜像也将被输出到汇编列表中。

GEN

输出宏定义行、引用行以及宏扩展行到汇编列表。

[格式描述]

```
[Δ]$[Δ]GEN              ; Default assumption
```

[功能]

- GEN 控制指令通知汇编器输出宏定义行、引用行以及宏扩展行到汇编列表。

[使用]

- 使用 GEN 控制指令可以限制汇编列表的输出量。

[说明]

- GEN 控制指令只能在普通源程序中进行描述。
- 如果 GEN 和 NOGEN 控制都没有进行指定，宏定义行、引用行以及宏扩展行将输出到汇编列表中。
- 指定列表控制将发生在 GEN 或者 NOGEN 控制指令镜像打印输出至汇编列表完成后。
- 如果 GEN 控制指令在 NOGEN 控制指令之后进行定义，那么汇编器会继续进行宏扩展行的输出。

[应用示例]

```
        NAME    SAMP
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        END
```

上述汇编列表的应用示例显示如下。

```
        NAME    SAMP1
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        MOV     A, #10H                 ; The macro-expanded lines will not be output.
        AUD     A, #20H                 ; The macro-expanded lines will not be output.
        END
```

(1) 由于 NOGEN 控制指令已经进行指定，宏扩展行将不会输出到汇编列表中。

NOGEN

不输出宏定义行、引用行以及宏扩展行到汇编列表。

[格式描述]

```
[Δ]$[Δ]NOGEN
```

[功能]

- NOGEN 控制指令通知汇编器输出宏定义行和引用行，但是将禁止输出宏扩展行。

[使用]

- 使用 NOGEN 控制指令可以限制汇编列表的输出量。

[说明]

- NOGEN 控制指令只能在普通源程序中进行声明。
- 如果 GEN 和 NOGEN 控制都没有进行指定，宏定义行、引用行以及宏扩展行将输出到汇编列表中。
- 指定列表控制将发生在 GEN 或者 NOGEN 控制指令镜像打印输出至汇编列表完成后。
- 即使由 NOGEN 控制指令控制列表的输出，汇编器仍然继续他的处理并累加语句数量计数。
- 如果 GEN 控制指令在 NOGEN 控制指令之后进行定义，那么汇编器会继续进行宏扩展行的输出。

[应用示例]

```
        NAME    SAMP
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        END
```

上述汇编列表的应用示例显示如下。

```
        NAME    SAMP1
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        MOV     A, #10H                 ; The macro-expanded lines will not be output.
        AUD     A, #20H                 ; The macro-expanded lines will not be output.
        END
```

(1) 由于 NOGEN 控制指令已经进行指定，宏扩展行将不会输出到汇编列表中。

COND

输出条件汇编器中有效以及无效部分到汇编列表。

[格式描述]

```
[Δ]$[Δ]COND          ; Default assumption
```

[功能]

- COND 控制指令通知汇编器输出符合与不符合条件汇编器的输出行到汇编列表中。

[使用]

- 使用 COND 控制指令可以限制汇编列表的输出量。

[说明]

- COND 控制指令只能在普通源程序中进行描述。
- 如果 COND 和 NOCOND 控制指令都没有进行指定，汇编器将会输出符合与不符合条件汇编器的输出行到汇编列表中。
- 指定列表控制将发生在 COND 或者 NOCOND 控制指令镜像打印输出至汇编列表完成后。
- 如果 COND 在 NOCOND 控制指令的之后进行指定，汇编器将会重新开始输出不匹配条件汇编的行以及在 IF/_IF, ELSEIF/_ELSEIF, ELSE, 和 ENDIF 中指定的行。

[应用示例]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                      ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #1H
$       ELSE                            ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #0H          ; This part, though assembled, will not be outout
                                        ; to the list.
$       ENDIF                           ; This part, though assembled, will not be outout
                                        ; to the list.
        END
```

NOCOND

不输出条件汇编器中有效以及无效部分到汇编列表。

[格式描述]

```
[Δ]$[Δ]NOCOND
```

[功能]

- NOCOND 控制指令通知汇编器仅输出符合条件汇编的行到汇编列表中。不匹配条件汇编的行以及在 IF/_IF, ELSEIF/_ELSEIF, ELSE, 和 ENDIF 中指定的行将禁止输出。

[使用]

- 使用 NOCOND 控制指令可以限制汇编列表的输出量。

[说明]

- NOCOND 控制指令只能在普通源程序中进行声明。
- 如果 COND 和 NOCOND 控制指令都没有进行指定，汇编器将会输出符合与不符合条件汇编器的输出行到汇编列表中。
- 指定列表控制将发生在 COND 或者 NOCOND 控制指令镜像打印输出至汇编列表完成后。
- 即使列表在 NOCOND 控制指令的控制下输出，汇编器仍将增加 ALNO 和 STNO 的总数。
- 如果 COND 在 NOCOND 控制指令的之后进行指定，汇编器将会重新开始输出不匹配条件汇编的行以及在 IF/_IF, ELSEIF/_ELSEIF, ELSE, 和 ENDIF 中指定的行。

[应用示例]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                      ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #1H
$       ELSE                            ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #0H          ; This part, though assembled, will not be outout
                                        ; to the list.
$       ENDIF                           ; This part, though assembled, will not be outout
                                        ; to the list.
        END
```

TITLE

在每页汇编列表的头部 TITLE 列中打印字符串，符号列表以及交叉引用列表。

[格式描述]

```
[Δ]$[Δ]TITLE[Δ]([Δ]'title-string'[Δ])
[Δ]$[Δ]TT[Δ]([Δ]'title-string'[Δ])          ; Abbreviated format
```

[默认设定]

- 当 TITLE 控制指令没有进行指定时，汇编列表头部的 TITLE 列将会留空。

[功能]

- TITLE 控制指令指定在每页汇编列表的头部 TITLE 列中打印字符串、符号列表以及交叉引用列表。

[使用]

- 使用 TITLE 控制指令在列表的每页中打印标题以便列表中的内容可以简单地被识别。
- 如果用户需要在每个汇编时间使用汇编器选项指定标题，那么在源模块文件的开始汇编器中通过描述这个控制指令可以帮助用户节省时间和工作。

[说明]

- TITLE 控制指令只能在选择的源模块文件的头部定义。
- 如果同时有两个或者更多的 TITLE 控制指令进行了指定，那么汇编器只接受最后一个定义的 TITLE 控制指令为有效。
- 标题字符串最多可以指定 60 个字符。如果在标题字符中包含有 61 或者更多的字符，那么汇编器只接受先前的 60 个字符为有效的。
  不过，如果汇编列表文件中的每行字符长度指定 （量 "X") 在 119 或更少字符， "X-60 字符 " 将视为有效。
- 如果在标题字串部分中需要使用单引用符 (')，则需要连续声明这个单引号两次。
- 如果不需要指定标题字串 （标题字串的字符数 =0），那么汇编器将留空给 TITLE 列。
- 如果在指定的标题字串中包含有在 "(2)　字符集 " 中所没有的任何字符时，汇编器将会输出 "!" 来代替在 TITLE 列中非法的字符。
- 汇编列表中的标题同样还可以在汇编器的启动命令行中通过汇编器选项 (-lh) 来进行指定。

[应用示例]

```
$ PROCESSOR ( f1166a0 )
$       TITLE ( 'THIS IS TITLE' )
        NAME    SAMPLE
        CSEG
        MOV     A , B
        END
```

上述汇编列表的应用示例显示如下。（每页的行数定义为 72）。

```
78K0R Assembler Vx.xx    THIS IS TITLE    Date:xx xxx xx  Page:1


Command :       -ll72 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn


         Assemble list


ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT


 1       1                             $       PROCESSOR ( f1166a0 )
 2       2                             $       TITLE ( 'THIS IS TITLE' )
 3       3                                    NAME    SAMPLE
 4       4      ----                           CSEG
 5       5      00000   63                     MOV     A, B
 6      6                                    END


Segment information :


ADRS  LEN     NAME


00000   00001H  ?CSEG


Target chip : uPD78F1166_A0
Device file : Vx.xx
Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

SUBTITLE

在汇编列表头部的 SUBTITLE 列中打印字符串。

[格式描述]

```
[Δ]$[Δ]SUBTITLE[Δ]([Δ]'title-string'[Δ])
[Δ]$[Δ]ST[Δ]([Δ]'title-string'[Δ])          ; Abbreviated format
```

[默认设定]

- 当 SUBTITLE 控制指令没有进行指定时，汇编列表头部的 SUBTITLE 列将会留空。

[功能]

- SUBTITLE 控制指令指定了汇编列表中每页头部 SUBTITLE 部分中将要打印的字符串。

[使用]

- 使用 SUBTITLE 控制指令在汇编列表的每页中打印副标题以便汇编列表中的内容可以简单地被识别。
  副标题中的字符串可以针对每页进行改变。

[说明]

- SUBTITLE 控制指令只能在普通源程序中进行描述。
- 副标题字符串最多可以指定 72 个字符。
  如果在标题字符中包含有 73 或者更多的字符，那么汇编器只接受先前的 72 个字符为有效的。 2 字节的字符算作为两个字符，标签算作为一个字符。
- 在 SUBTITLE 控制指令已经指定之后，通过 SUBTITLE 控制指令指定的字符串将会打印在此页的 SUBTITLE 这部分。不过，如果该控制指令在页的顶部 （第一行）中进行指定，副标题将会在该页上打印。
- 如果 SUBTITLE 控制指令没有进行指定，汇编器将在 SUBTITLE 部分留空。
- 如果在字符串部分中需要使用单引用符 (')，则需要连续声明这个单引号两次。
- 如果在 SUBTITLE 部分中字符串为 0，那么 SUBTITLE 列将会留空。
- 如果在指定的副标题字串中包含有在 "(2)　字符集 " 中所没有的任何字符时，汇编器将会输出 "!" 来代替在 SUBTITLE 列中非法的字符。如果 CR (0DH）进行了指定，在汇编列表中将会出错，并且不会打印任何东西。
  如果 00H 进行了指定，该指定指向结尾的单引用记号 (') 中的任意部分就不会输出。

[应用示例]

```
        NAME    SAMP
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 1' )       ; (1)
$       EJECT                                   ; (2)
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 2' )       ; (3)
$       EJECT                                   ; (4)
$       SUBTITLE ( 'THIS IS SUBTITLE 3' )       ; (5)
        END
```

(1) 此条控制指令指定了字符串 "THIS IS SUBTITLE 1"。

(2) 此条控制指令指定了换页。

(3) 此条控制指令指定了字符串 "THIS IS SUBTITLE 2"。

(4) 此条控制指令指定了换页。

(5) 此条控制指令指定了字符串 "THIS IS SUBTITLE 3"。

此汇编列表示列显示如下（每页指定的行数为 80）。

```
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:1

Command :       -cf1166a0 -ll80 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn
        Assemble list

ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT

 1       1                                  NAME    SAMP
 2     2    -----                CSEG
 3     3                      $   SUBTITLE ( 'THIS IS SUBTITLE 1' )   ; (1)
 4     4                      $   EJECT                               ; (2)
-------------------------------------------page ejection-------------------------
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:2

THIS IS SUBTITLE 1

ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT

 5     5    -----                CSEG
 6     6                      $   SUBTITLE ( 'THIS IS SUBTITLE 2' )   ; (3)
 7     7                      $   EJECT                               ; (4)
```

```
----------------------------------------------page ejection-------------------------
78K0R Assembler Vx.xx                              Date:xx xxx xx  Page:3


THIS IS SUBTITLE 2


ALNO  STNO ADRS   OBJECT   M I  SOURCE STATEMENT


 8     8                     $   SUBTITLE ( 'THIS IS SUBTITLE 3' )   ; (5)
 9      9                                    END


Segment informations :


ADRS  LEN     NAME


00000 00000H  ?CSEG


Target chip : uPD78F1166_A0
Device file : Vx.xx
Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

FORMFEED

在列表文件的最后插入换页码。

[格式描述]

```
[Δ]$[Δ]FORMFEED
```

[功能]

- FORMFEED 控制指令通知汇编器在汇编列表的最后输出 FORMFEED 代码。

[使用]

- 当用户想在打印汇编列表文件内容之后开始开始新页时，可以使用 FORMFEED 控制指令。

[说明]

- FORMFEED 控制指令只能在选择的源模块文件的头部定义。
- 在汇编列表进行打印时，在页的中间结束打印时列表的最后页可能无法显示。
  这个情况下，通过使用 FORMFEED 控制指令或者汇编器选项 (-lf) 在汇编列表的最后添加一个 FORMFEED 代码。
  在很多情况下， FORMFEED 代码在文件的最后将会被输出。出于这个原因，如果 FORMFEED 代码存在在列表文件的最后，可能会换到用户不想看到的白页上。为了防止这个情况，可以设定 NOFORMFEED 控制指令或者汇编器选项 (-nlf)，并 默认为有效。
- 如果同时有两个或者更多的 FORMFEED/NOFORMFEED 控制指令进行了指定，仅仅最后一个定义的控制指令为有效。
- 通过汇编器中启动命令行的选项 (-lf) 或者 (-nlf) 同样可以指定是否输出一个格式回馈代码。
- 如果在源模块文件中控制指令指定规格 (FORMFEED/NOFORMFEED) 不同于启动命令行中指定规格 (-lf/-nlf)，则在启动命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中进行了指定，汇编器仍会在 FORMFEED 或者 NOFORMFEED 控制指令中执行语法检查。

NOFORMFEED

不在列表文件的最后输出格式回馈。

[格式描述]

```
[Δ]$[Δ]NOFORMFEED      ; Default assumption
```

[功能]

- NOFORMFEED 控制指令通知汇编器在汇编列表的最后不输出 FORMFEED 代码。

[使用]

- 当用户想在打印汇编列表文件内容之后开始开始新页时，可以使用 FORMFEED 控制指令。

[说明]

- NOFORMFEED 控制指令只能在选择的源模块文件的头部定义。
- 在汇编列表进行打印时，在页的中间结束打印时列表的最后页可能无法显示。
  这个情况下，通过使用 FORMFEED 控制指令或者汇编器选项 (-lf) 在汇编列表的最后添加一个 FORMFEED 代码。
  在很多情况下， FORMFEED 代码在文件的最后将会被输出。出于这个原因，如果 FORMFEED 代码存在在列表文件的最后，可能会换到用户不想看到的白页上。为了防止这个情况，可以设定 NOFORMFEED 控制指令或者汇编器选项 (-nlf)，并 默认为有效。
- 如果同时有两个或者更多的 FORMFEED/NOFORMFEED 控制指令进行了指定，仅仅最后一个定义的控制指令为有效。
- 通过汇编器中启动命令行的选项 (-lf) 或者 (-nlf) 同样可以指定是否输出一个格式回馈代码。
- 如果在源模块文件中控制指令指定规格 (FORMFEED/NOFORMFEED) 不同于启动命令行中指定规格 (-lf/-nlf)，则在启动命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中进行了指定，汇编器仍会在 FORMFEED 或者 NOFORMFEED 控制指令中执行语法检查。

WIDTH

指定在列表文件中一行最大的字符数。

[格式描述]

```
[Δ]$[Δ]WIDTH[Δ]([Δ]columns-per-line[Δ])
```

[默认设定]

$WIDTH (132)

[功能]

- WIDTH 控制指令指定了列表文件中每行列 （字符）的数量。
  " 每行列数量 " 在范围从 72 至 260 为有效。

[使用]

- 当用户希望在列表文件中改变表每行中列数量大小时可以使用 WIDTH 控制指令。

[说明]

- WIDTH 控制指令只能在选择的源模块文件的头部定义。
- 如果同时有两个或者更多的 WIDTH 控制指令进行了指定，仅最后一个定义的控制指令为有效。
- 列表文件中每行的列数量同样可以要通过汇编器启动命令行中的选项 (-lw) 来进行指定。
- 如果在源模块文件中控制指令指定规格 (WIDTH) 不同于启动命令行中指定规格 (-lw)，则在命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编器仍会在 WIDTH 控制指令中执行一个语法检查。

LENGTH

指定列表文件中每页的行数

[格式描述]

```
[Δ]$[Δ]LENGTH[Δ]([Δ]lines-per-page[Δ])
```

[默认设定]

$LENGTH (66)

[功能]

- LENGTH 控制指令指定列表文件中每页的行数。
  " 每页行数 " 可能为 "0" 或是在范围从 20 至 32767 为有效。

[使用]

- 当用户希望在列表文件中改变表每页行数量大小时可以使用 LENGTH 控制指令。

[说明]

- LENGTH 控制指令只能在选择的源模块文件的头部定义。
- 如果同时有两个或者更多的 LENGTH 控制指令进行了指定，仅最后一个定义的控制指令为有效。
- 列表文件中每行的列数量同样可以要通过汇编器启动命令行中的选项 (-ll) 来进行指定。
- 如果在源模块文件中控制指令指定规格 (LENGTH) 不同于启动命令行中指定规格 (-ll)，则在命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编器仍会在 LENGTH 控制指令中执行一个语法检查。

TAB

指定列表文件标签的字符数。

[格式描述]

```
[Δ]$[Δ]TAB[Δ]([Δ]number-of-columns[Δ])
```

[默认设定]

$TAB (8)

[功能]

- TAB 控制指令指定了列表文件中用于制表位列的数量。
  " 制表位列数量 " 在范围从 0 至 8 为有效。
- TAB 控制指令通过在源模块列表中使用几个空格字符来替换使用 HT （水平制表）代码，它指定了成为基础制表处理的列数量。

[使用]

- 当任意列表中每行的字符数量通过使用 TAB 控制指令来减少时，可以使用 HT 代码来减少空格数量。

[说明]

- TAB 控制指令只能在选择的源模块文件的头部定义。
- 如果同时有两个或者更多的 TAB 控制指令进行了指定，仅最后一个定义的控制指令为有效。
- 制表位的数量同样可以在汇编器启动命令行通过选项 (-lt) 来进行指定。
- 如果在源模块文件中控制指令指定规格 (TAB) 不同于启动命令行中指定规格 (-lt)，则在命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-np）已经在启动命令行中设定，汇编器仍会在 TAB 控制指令中执行一个语法检查。

4.3.7　条件汇编控制指令

使用条件汇编切换符来选择条件汇编控制指令，决定是否将源模块中的一系列语句编译成汇编目标。
如果条件汇编指令是有效的，那么可在没有无用语句以及不改变源模块的情况下进行汇编。
有下列条件汇编控制指令。

控制指令	概要
IF	设定条件来限制汇编目标源语句。
_IF	
ELSEIF	
_ELSEIF	
ELSE	
ENDIF	
SET	通过指定 IF/ELSEIF 控制指令来设定选择名称。
RESET	

IF

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 IF 控制指令以及 ENDIF 控制指令中描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名为真 （除 00H 以外），则在这个 IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 IF 条件为假 （00H），则在这个 IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3)　符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

- 示例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF                           ; (2)
          :
        END
```

(1)　如果选择名称 "SW1" 的值为真， "text1" 中的语句将进行汇编。
如果选择名称 "SW1" 的值为假， "text1" 中的语句将不会进行汇编。
选择名称 "SW1" 的值已经由 "text0" 中描述的 SET 或者 RESET 控制指令来设定真或者假。

(2)　该指令说明了源条件汇编语句范围的终端。

- 示例 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF                           ; (3)
          :
        END
```

(1) 选择名称 "SW1" 的值已经由 "text0" 中描述的 SET 或者 RESET 控制指令来设定真或者假。
如果选择名称 "SW1" 的值为真， "text1" 中的语句将进行汇编， "text2" 中的语句将不会进行汇编。

(2) 如果在 (1) 中开关符名称 "SW1" 的值为假，那么 "text1" 中的语句将不会进行汇编， "text2" 中的语句将会进行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

- 示例 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE            ; (4)
                text4
$       ENDIF                           ; (5)
          :
        END
```

(1) 选择名称 "SW1","SW2", 和 "SW3" 的值已经由 "text0" 中描述的 SET 或者 RESET 控制指令来设定真或者假。
如果选择名称 "SW1" 或者 "SW2" 的值为真， "text1" 中的语句将进行汇编，"text2", "text3", 和 "text4" 中的语句将不会进行汇编。
如果选择名称 "SW1" 和 "SW2" 的值为假，那么 "text1" 中的语句将不会进行汇编， (2) 之后的语句将会有条件选择性的进行汇编。

(2) 如果在 (1) 中的选择名称 "SW1" 和 "SW2" 的值为非并且选择名称 "SW3" 的值为真时， "text2" 中的语句将会进行汇编， "text1","text3" 和 "text4" 中的语句将不会进行汇编。

(3) 如果在 (1) 中的选择名称 "SW1" 和 "SW2" 的值和 （2）中的 "SW3" 的值都为假时， "text4" 中的选择语句将会进行汇编， "text1","text2" 和 "text3" 中的语句将不会进行汇编。

(4) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值和 （2）中的 **"SW3"** 的值以及 （3）中的 **"SW4"** 的值都为假时， **"text4"** 中的语句将会进行汇编， **"text1","text2"** 和 **"text3"** 中的语句将不会进行汇编。

(5) 该指令说明了源条件汇编语句范围的终端。

_IF

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 _IF 控制指令和 ENDIF 控制指令之间描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 _IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名字为真 （并不是 00H），则在这个 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 _IF 条件为假 （00H），则在这个 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3)　符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

```
        text0
$       _IF ( SYMA )              ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                     ; (3)
          :
        END
```

(1) 选择名称 "SYMA" 的值由 "text0" 中 EQU 或者 SET 描述指令来进行定义。
如果符号名称 "SYMA" 为真 （非 "0"） ,"text1" 中的语句将执行汇编操作，而 "text2" 中将不会执行。

(2) 如果符号名称 "SYMA" 值和 "SYMA" 和 "SYMB" 的值均为 "0" 时， "text2" 中的语句将执行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

ELSEIF

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 IF 或者 _IF 控制指令至 ENDIF 控制指令中所描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 IF 或者 _IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名称为真 （除 00H 以外），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 IF 或者 _IF 条件为假 （00H），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3) 符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE            ; (4)
                text4
$       ENDIF                           ; (5)
          :
        END
```

(1) 选择名称 "SW1","SW2", 和 "SW3" 的值已经由 "text0" 中描述的 SET 或者 RESET 控制指令来设定真或者假。
如果选择名称 "SW1" 或者 "SW2" 的值为真， "text1" 中的语句将进行汇编，"text2", "text3", 和 "text4" 中的语句将不会进行汇编。
如果选择名称 "SW1" 和 "SW2" 的值为假，那么 "text1" 中的语句将不会进行汇编， (2) 之后的语句将会有条件选择性的进行汇编。

(2) 如果在 (1) 中的选择名称 "SW1" 和 "SW2" 的值为非并且选择名称 "SW3" 的值为真时， "text2" 中的语句将会进行汇编， "text1","text3" 和 "text4" 中的语句将不会进行汇编。

(3) 如果在 (1) 中的选择名称 "SW1" 和 "SW2" 的值和 （2）中的 "SW3" 的值都为假时， "text4" 中的选择语句将会进行汇编， "text1","text2" 和 "text3" 中的语句将不会进行汇编。

(4) 如果在 (1) 中的选择名称 "SW1" 和 "SW2" 的值和 （2）中的 "SW3" 的值以及 （3）中的 "SW4" 的值都为假时， "text4" 中的语句将会进行汇编， "text1","text2" 和 "text3" 中的语句将不会进行汇编。

(5) 该指令说明了源条件汇编语句范围的终端。

_ELSEIF

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 IF 或者 _IF 控制指令至 ENDIF 控制指令中所描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 IF 或者 _IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名称为真 （除 00H 以外），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 IF 或者 _IF 条件为假 （00H），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3)　符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。

- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                           ; (3)
          :
        END
```

(1) 选择名称 "SYMA" 的值由 "text0" 中 EQU 或者 SET 描述指令来进行定义。
如果符号名称 "SYMA" 为真 （非 "0"） ,"text1" 中的语句将执行汇编操作，而 "text2" 中将不会执行。

(2) 如果符号名称 "SYMA" 值和 "SYMA" 和 "SYMB" 的值均为 "0" 时， "text2" 中的语句将执行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

ELSE

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
or[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
or[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 IF 或者 _IF 控制指令至 ENDIF 控制指令中所描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 IF 或者 _IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名称为真 （除 00H 以外），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 IF 或者 _IF 条件为假 （00H），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3)　符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

- 示例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF                           ; (3)
          :
        END
```

(1) 选择名称 "SW1" 的值已经由 "text0" 中描述的 SET 或者 RESET 控制指令来设定真或者假。
如果选择名称 "SW1" 的值为真， "text1" 中的语句将进行汇编， "text2" 中的语句将不会进行汇编。

(2) 如果在 (1) 中开关符名称 "SW1" 的值为假，那么 "text1" 中的语句将不会进行汇编， "text2" 中的语句将会进行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

- 示例 2

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE            ; (4)
                text4
$       ENDIF                           ; (5)
          :
        END
```

(1) 选择名称 **"SW1","SW2"**, 和 **"SW3"** 的值已经由 **"text0"** 中描述的 **SET** 或者 **RESET** 控制指令来设定真或者假。如果选择名称 **"SW1"** 或者 **"SW2"** 的值为真， **"text1"** 中的语句将进行汇编，**"text2"**, **"text3"**, 和 **"text4"** 中的语句将不会进行汇编。
如果选择名称 **"SW1"** 和 **"SW2"** 的值为假，那么 **"text1"** 中的语句将不会进行汇编， **(2)** 之后的语句将会有条件选择性的进行汇编。

(2) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值为非并且选择名称 **"SW3"** 的值为真时， **"text2"** 中的语句将会进行汇编， **"text1","text3"** 和 **"text4"** 中的语句将不会进行汇编。

(3) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值和 （2）中的 **"SW3"** 的值都为假时， **"text4"** 中的选择语句将会进行汇编， **"text1","text2"** 和 **"text3"** 中的语句将不会进行汇编。

(4) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值和 （2）中的 **"SW3"** 的值以及 （3）中的 **"SW4"** 的值都为假时， **"text4"** 中的语句将会进行汇编， **"text1","text2"** 和 **"text3"** 中的语句将不会进行汇编。

(5) 该指令说明了源条件汇编语句范围的终端。

ENDIF

设定条件来限制汇编目标源语句。

[格式描述]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
or[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
or[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[功能]

- 控制指令设定条件用以限制在汇编过程中的源语句。
  在 IF 或者 _IF 控制指令至 ENDIF 控制指令中所描述的源语句是受条件汇编所控制的。
- 如果条件表达式的值或者由 IF 或者 _IF 控制指令 （例如 IF 或者 _IF 指令）指定的选择名称为真 （除 00H 以外），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 IF 或者 _IF 条件为假 （00H），则在这个 IF 或者 _IF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将不会执行汇编。
- 仅当在所有条件汇编控制在指令 ELSEIF 或者 _ELSEIF 控制指令之前描述的条件不匹配时 （例如，赋值为假）， ELSEIF 或 _ELSEIF 控制指令才会检查真 / 假状态。
  如果条件表达式的值或者由 ELSEIF 或者 _ELSEIF 控制指令 （例如 ELSEIF 或者 _ELSEIF 条件）指定的选择名为真，则在这个 ELSEIF 或者 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令（ELSEIF/_ELSEIF,ELSE, 或者 ENDIF）中所描述的源语句将执行汇编。为了后续的汇编处理，汇编器处理 ENDIF 控制指令之后的语句。
  如果 ELSEIF 或者 _ELSEIF 条件为假，则在这个 ELSEIF 或 _ELSEIF 控制指令之后直到在源程序中出现的下一个汇编条件控制指令 （ELSEIF/_ELSEIF,ELSE, 或 ENDIF）中所描述的源语句将不会执行汇编。
- 如果所有的 IF/_IF 和 ELSEIF/_ELSEIF 控制指令的条件都是在 ELSE 控制指令不匹配之前进行描述的 （例如，所有的选择名称都为假），那么在这个 ELSE 控制指令之后直到源程序中出现 ENDIF 控制指令中所描述的源语句将执行汇编。
- ENDIF 控制指令表明在汇编器中受条件汇编控制的源语句的终止。

[使用]

- 得益于这些条件汇编控制指令，需要进行汇编的源语句将无需太多的修改便能成为源程序。
- 如果只有在源程序中描述的程序开发使得调试语句变得有必要时，调试语句是否需要进行汇编 （转换成机器语言）可以通过对条件汇编设定选择来指定。

[说明]

- IF 和 ELSEIF 控制指令是用来配合选择名称来判断条件的真 / 假，然而 _IF 和 _ELSEIF 控制指令是配合条件表达式来判断条件的真 / 假的。
  IF/ELSEIF 和 _IF/_ELSEIF 可以联合一起使用。换言之 , ELSEIF/_ELSEIF 可以配合 IF 或者 _IF 以及 ENDIF 一同使用。
- 描述用以条件表达式中的绝对表达式。
- 描述选择名称的规则与常规符号描述规则是一样的 （关于细节，参阅 (3)　符号区域）。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 如果由 IF 或者 ELSEIF 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开。
  每个模块中最多可以使用五个选择名称
- 当通过 IF 或者 ELSEIF 控制指令指定两个或者更多的选择名称时，只要有一个选择名称为真，那么 IF 或者 ELSEIF 条件将会判断为匹配的。
- 每个由 IF 或者 ELSEIF 控制指令指定的选择名称值必须由 SET/RESET 控制指令来进行定义。
  所以，如果通过 IF 或者 ELSEIF 控制指令指定的选择名称值并不是在源模块中由 SET/RESET 控制指令来预先进行定义，需要被重新设定。
- 如果在指定的选择名称或是条件表达式中包含有非法描述，汇编器将会输出一个错误信息并且确定值为假。
- 当描述 IF 或者 _IF 控制指令时 ,IF 或者 _IF 控制指令必须配合 ENDIF 控制指令一同使用。
- 如果在宏体中描述 IF-ENDIF 块并且通过 EXITM 将控制从宏所在层传回，那么汇编器将强制 IF 层以返回宏体入口所在层。在这种情况下，不会发生错误。
- 在一个 IF-ENDIF 块中描述另一个 IF-ENDIF 块，可以参考 IF 控制指令的嵌套。 IF 控制指令的嵌套最多允许 8 层。
- 在条件汇编中，没有进行汇编语句的目标代码将不会生成，但是这些语句在不会对汇编列表产生改变的情况下会被输出。如果用户不想输出这些语句，可以使用 $NOCOND 控制指令。

[应用示例]

- 示例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF                           ; (2)
          :
        END
```

(1) 如果选择名称 **"SW1"** 的值为真， **"text1"** 中的语句将进行汇编。
如果选择名称 **"SW1"** 的值为假， **"text1"** 中的语句将不会进行汇编。
选择名称 **"SW1"** 的值已经由 **"text0"** 中描述的 **SET** 或者 **RESET** 控制指令来设定真或者假。

(2) 该指令说明了源条件汇编语句范围的终端。

- 示例 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF                           ; (3)
          :
        END
```

(1) 选择名称 **"SW1"** 的值已经由 **"text0"** 中描述的 **SET** 或者 **RESET** 控制指令来设定真或者假。
如果选择名称 **"SW1"** 的值为真， **"text1"** 中的语句将进行汇编， **"text2"** 中的语句将不会进行汇编。

(2) 如果在 **(1)** 中开关符名称 **"SW1"** 的值为假，那么 **"text1"** 中的语句将不会进行汇编， **"text2"** 中的语句将会进行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

- 示例 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE            ; (4)
                text4
$       ENDIF                           ; (5)
          :
        END
```

(1) 选择名称 **"SW1"**,**"SW2"**, 和 **"SW3"** 的值已经由 **"text0"** 中描述的 **SET** 或者 **RESET** 控制指令来设定真或者假。
如果选择名称 **"SW1"** 或者 **"SW2"** 的值为真， **"text1"** 中的语句将进行汇编，**"text2"**, **"text3"**, 和 **"text4"** 中的语句将不会进行汇编。
如果选择名称 **"SW1"** 和 **"SW2"** 的值为假，那么 **"text1"** 中的语句将不会进行汇编， **(2)** 之后的语句将会有条件选择性的进行汇编。

(2) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值为非并且选择名称 **"SW3"** 的值为真时， **"text2"** 中的语句将会进行汇编， **"text1","text3"** 和 **"text4"** 中的语句将不会进行汇编。

(3) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值和 （2）中的 **"SW3"** 的值都为假时， **"text4"** 中的选择语句将会进行汇编， **"text1","text2"** 和 **"text3"** 中的语句将不会进行汇编。

(4) 如果在 **(1)** 中的选择名称 **"SW1"** 和 **"SW2"** 的值和 （2）中的 **"SW3"** 的值以及 （3）中的 **"SW4"** 的值都为假时， **"text4"** 中的语句将会进行汇编， **"text1","text2"** 和 **"text3"** 中的语句将不会进行汇编。

(5) 该指令说明了源条件汇编语句范围的终端。

- 示例 4

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                           ; (3)
          :
        END
```

(1) 选择名称 **"SYMA"** 的值由 **"text0"** 中 **EQU** 或者 **SET** 描述指令来进行定义。
如果符号名称 **"SYMA"** 为真 （非 **"0"**） ,**"text1"** 中的语句将执行汇编操作，而 **"text2"** 中将不会执行。

(2) 如果符号名称 **"SYMA"** 值和 **"SYMA"** 和 **"SYMB"** 的值均为 **"0"** 时， **"text2"** 中的语句将执行汇编。

(3) 该指令说明了源条件汇编语句范围的终端。

SET

通过指定 IF/ELSEIF 控制指令来设定选择名称。

[格式描述]

```
[Δ]$[Δ]SET[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
```

[功能]

- SET 控制指令赋值给每个选择名称，并由 IF 或者 ELSEIF 控制指令进行定义。
- SET 控制指令再其指定过程中赋给每个选择名称为真值 （0FFH）。

[使用]

- 通过 SET 控制指令的描述来赋值给每个选择名称真值 （0FFH），并由 IF 或者 ELSEIF 控制指令进行定义。

[说明]

- 使用 SET 或者 RESET 控制指令，必须指定至少一个选择名称。
  关于选择名称描述的定义与符号描述一致 (参阅 "(3) 符号区域 ")。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 定义的选择名称可能与用户自定义的符号相同，但与保留字和其他选择名称不同。
- 如果由 SET 或者 RESET 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开 . 每个模块中最多可以使用一千个选择名称。
- 选择名称可以由 SET 控制指令设定一次为 " 真 "，可以由 RESET 控制指令来改变成为 " 假 "；反之亦然。
- 在描述 IF 或者 ELSEIF 控制指令之前，通过其指令指定的选择名称必须由在源模块中的 SET 或者 RESET 控制指令定义一次。
- 选择名称不会输出到交叉引用列表中。

[应用示例]

```
$       SET ( SW1 )                     ; (1)
          :
$       IF ( SW1 )      ; (2)
                text1
$       ENDIF                           ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )      ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE            ; (7)
                text4
$       ENDIF                           ; (8)
          :
        END
```

(1) 该指令赋给选择名称 **"SW1"** 真值 （**0FFH**）。

(2) 由于在上述 （**1**）中的选择名称已经赋于了真值， **"text1"** 中的语句将执行汇编。

(3) 该指令说明了从 （**2**）开始的源条件汇编语句范围的结束。

(4) 该指令分别赋给选择名称 **"SW1"** 和 **"SW2"** 假值 （**00H**）。

(5) 由于在上述 （**4**）中的选择名称 **"SW1"** 已经赋于了假值， **"text2"** 中的语句将不会执行汇编。

(6) 由于在上述 （**4**）中的选择名称 **"SW2"** 已经赋于了假值， **"text3"** 中的语句将不会执行汇编。

(7) 由于在上述 （**5**）和 （**6**）中的选择名称 **"SW1"** 和 **"SW2"** 都已经赋于了假值， **"text4"** 中的语句将执行汇编。

(8) 该指令说明了从 （**5**）开始的源条件汇编语句范围的结束。

RESET

通过指定 IF/ELSEIF 控制指令来设定选择名称。

[格式描述]

```
[Δ]$[Δ]RESET[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ...[Δ])
```

[功能]

- RESET 控制指令赋值给每个选择名称，并由 IF 或者 ELSEIF 控制指令进行定义。
- RESET 控制指令再其指定过程中赋给每个选择名称为假值 （00H）。

[使用]

- 通过 RESET 控制指令的描述来赋值给每个选择名称假值 （00H），并由 IF 或者 ELSEIF 控制指令进行定义。

[说明]

- 使用 SET 或者 RESET 控制指令，必须指定至少一个选择名称。
  关于选择名称描述的定义与符号描述一致 (参阅 "(3) 符号区域 ")。
  不过，可以用作选择名称被识别的最大字符数总是为 31。
- 定义的选择名称可能与用户自定义的符号相同，但与保留字和其他选择名称不同。
- 如果由 SET 或者 RESET 控制指令指定了两个或者更多的选择名称时，使用冒号 （:）将其相互分开 . 每个模块中最多可以使用一千个选择名称。
- 选择名称可以由 SET 控制指令设定一次为 " 真 "，可以由 RESET 控制指令来改变成为 " 假 "；反之亦然。
- 在描述 IF 或者 ELSEIF 控制指令之前，通过其指令指定的选择名称必须由在源模块中的 SET 或者 RESET 控制指令定义一次。
- 选择名称不会输出到交叉引用列表中。

[应用示例]

```
$       SET ( SW1 )                     ; (1)
          :
$       IF ( SW1 )      ; (2)
                text1
$       ENDIF                           ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )      ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE            ; (7)
                text4
$       ENDIF                           ; (8)
          :
        END
```

(1) 该指令赋给选择名称 **"SW1"** 真值 （**0FFH**）。

(2) 由于在上述 （**1**）中的选择名称已经赋于了真值， **"text1"** 中的语句将执行汇编。

(3) 该指令说明了从 （**2**）开始的源条件汇编语句范围的结束。

(4) 该指令分别赋给选择名称 **"SW1"** 和 **"SW2"** 假值 （**00H**）。

(5) 由于在上述 （**4**）中的选择名称 **"SW1"** 已经赋于了假值， **"text2"** 中的语句将不会执行汇编。

(6) 由于在上述 （**4**）中的选择名称 **"SW2"** 已经赋于了假值， **"text3"** 中的语句将不会执行汇编。

(7) 由于在上述 （**5**）和 （**6**）中的选择名称 **"SW1"** 和 **"SW2"** 都已经赋于了假值， **"text4"** 中的语句将执行汇编。

(8) 该指令说明了从 （**5**）开始的源条件汇编语句范围的结束。

4.3.8　Kanji 代码控制指令

这是一种用来解释在注释中所描述的 Kanji 字符的字符代码控制指令。
有下列 Kanji 代码控制指令。

控制指令	概要
KANJICODE	为指定在注释中所描述的 Kanji 字符解释其 Kanji 字符代码。

KANJICODE

为指定在注释中所描述的 Kanji 字符解释其 Kanji 字符代码。

[格式描述]

```
[Δ]$[Δ]KANJICODE[Δ]kanji-code
```

[默认设定]

$KANJICODE SJIS

[功能]

- 为指定在注释中所描述的 Kanji 字符解释其 Kanji 字符代码。
- Kanji 字符可以描述 SJIS/EUC/NONE。

  SJIS ：翻译为 Shift JIS 代码。

  EUC ：翻译为 EUC 代码。

  NONE ：不翻译为汉字。

[使用]

- 用以指定在注释行中 kanji （2 字节字符）的 kanji 代码 （2 字节代码）的解释说明。

[说明]

- KANJI 代码控制指令只能在选择的源模块文件的头部定义。
- 如果同时有两个或者更多的 KANJI 代码控制指令在源模块文件的头部进行了指定，仅最后一个定义的控制指令有效。
- Kanji 代码规格的停用同样可以在汇编器启动命令行通过汇编器选项 (-zs/-ze/-zn) 来进行指定。
- 如果在源模块文件中控制指令指定规格 (KANJICODE) 不同于启动命令行中指定规格 (-zs/-ze/-zn)，则在命令行中指定规格的优先权要比源模块文件中高。
- 即使汇编程序中选项 （-zs/-ze/-zn）已经在启动命令行中设定，汇编程序会在 KANJICODE 控制指令中执行一个语法检查。

4.3.9 其他控制指令

以下显示的控制指令是以高级程序如 C 编译器输出的特殊控制指令。

- $TOL_INF
- $DGS
- $DGL

4.4 宏

这部分解释了如何使用宏。

当你需要在一个源程序中反复使用一系列语句时，宏将非常有帮助。

4.4.1 概要

宏使得在源程序中不断重复一系列语句变得十分简单。

在宏本体中包含了一连串指令，定义的在 MACRO 命令和 ENDM 命令之间。这些指令可插入于源程序中引用宏的任何位置。

宏使得编写源程序更为简单。他们不同于子程序。

关于宏与子程序的区别解释如下。为了有效地使用，根据具体的目的来选择宏或子程序。

(1) 子程序

- 在一个程序中，子程序往往处理那些需反复执行的进程。子程序将由汇编器转换为机器语言。
- 要使用子程序，只需简单地使用子程序调用指令来调用该子程序。（通常，在调用这些子程序之前也需要设定一些参数，并后续对其进行调整。）

  对子程序的有效使用，便能高效地利用程序存储器。
- 子程序对编程结构也很重要。把程序分为多个功能块，可起到使整个程序结构明朗化的效果，这将便于程序的理解。这有利于程序的设计、代码编写和维护。

(2) 宏

- 宏的基本功能是用一个宏名称取代一连串的指令。

  宏的定义是，在 MACRO 命令和 ENDM 命令之间的一连串指令。当源代码中出现宏名称时，指令就插入该位置。汇编器会用实际参数取代宏当中的形式参数并把指令转换为机器语言。
- 宏可以含有参数。

  例如，如果一组指令具有相同的处理程序，但描述的数据操作数不同，那么就可以通过分配形式参数给数据的方法定义一个宏。通过在宏引用时间内描述宏名称和实际参数，汇编器可同时处理多组指令，这些指令仅在语句描述中不同。

子程序主要用于减少对存储器的需求和阐明程序结构。使用宏可使得程序更易描述和理解。

4.4.2 使用宏

(1) 宏定义

使用 MACRO 和 ENDM 命令定义宏。

(a) 格式

符号区域	助记符区域	操作数区域	注释区域
macro-name	MACRO	[*formal-parameter*[, ...]]	[; comment]
	:		
	ENDM		[; *comment*]

(b) 功能

定义宏，在 MACRO 命令和 ENDM 命令之间，把在符号区域指定的宏名称分配到一系列语句（称为宏本体）中。

(c) 示例

```
ADMAC   MACRO   PARA1, PARA2
        MOV     A, #PARA1
        ADD     A, #PARA2
        ENDM
```

上述为一个简单示例，把 PARA1 和 PARA2 的值相加，然后把结果存放至寄存器 A。宏的名称是 ADMAC。PARA1 和 PARA2 为形式参数。

详情请参阅 "4.2.8　宏指令 "。

(2) 参考宏

为了引用已定义的宏，将宏名称写入助记符区域。

(a) 格式

符号区域	助记符区域	操作数区域	注释区域
[*label*:]	*macro-name*	[*actual-parameter*[, ...]]	[; comment]

(b) 功能

引用指定名称的宏本体。

(c) 使用

使用上述格式进行宏本体引用。

(d) 说明

- 助记符区域中指定的宏名称必须在宏引用之前定义。
- 用逗号（,）分隔实际参数。在同一行中，最多可指定 16 个实际参数。
- 在实际参数的字符串中不可出现空格。
- 如果实际参数的字符串中包含逗号（,）、分号（;）、空白或 tab 键时，需要在字符两端使用单引号（'）。
- 根据在宏定义中的顺序，形式参数由左边开始逐一转换为实际参数。
  实际参数的数量必须与形式参数的数量一致。若不一致，系统将发出警告。

(e) 示例

```
        NAME    SAMPLE
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        CSEG
          :
        ADMAC   10H, 20H
          :
        END
```

引用了已定义宏指 DMAC
10H 和 20H 为实际参数。

(3) 宏扩展

汇编器对宏进行如下处理：

- 当遇到宏名称时，汇编器在宏名称所在的位置插入相应的宏本体。
- 然后，插入的宏本体与其他语句一样进行汇编处理。

(4) 示例

当上述 "(2) 参考宏 " 中显示的宏被引用时，其展开如下所示。

```
        NAME    SAMPLE

        ; Macro definition
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        ; Source code
        CSEG
          :

        ; Macro expansion
        ADMAC   10H, 20H        ; (a)
        MOV     A, #10H
        ADD     A, #20H
        ; Source code
          :
        END
```

(a) 当宏名称被引用时，就会插入宏本体。
在宏本体中，实际参数将取代形式参数。

4.4.3 宏内部的符号

在宏的内部，可定义两种类型的符号：全局符号和局部符号。

(1) 全局符号

- 全局符号是指源程序中任何语句均可引用的符号。
  因此，当展开一连串语句时，宏定义的全局符号被引用超过一次以上的话，系统将产生重复定义的错误。
- 不能使用 LOCAL 命令进行定义的符号为全局符号。

(2) 局部符号

- 局部符号是由 LOCAL 命令 （请参阅 ?4.2.8 宏指令 ?）定义的符号。
- 只有在宏中声明该符号为 LOCAL 时，才能引用局部符号。
- 不可在宏以外声明引用局部符号。

< 应用示例 >

```
        NAME    SAMPLE
          ; Macro definition
MAC1    MACRO
        LOCAL   LLAB            ; (a)
LLAB :                          ; (b)
          :
GLAB :                          ; (c)
        BR      LLAB            ; (d)
        BR      GLAB            ; (e)
        ENDM
          :
          ; Source code
REF1 :  MAC1                    ; (f) <- Macro reference
          :
        BR      LLAB            ; (g) <- Error

REF2 :  MAC1                    ; (h) <- Macro reference
          :
GLAB :                          ; (i) <- Error
          :
        END
```

(a) 这里声明 LLAB 标签为局部符号。

(b) 这里定义 LLAB 标签为局部符号。

(c) 这里定义 GLAB 标签为全局符号。

(d) 这里引用宏 MAC1 内部的局部符号 LLAB。

(e) 这里引用宏 MAC1 内部定义的全局符号 GLAB。

(f) 这里引用宏 MAC1。

(g) 这里引用宏 MAC1 外部定义的局部符号 LLAB。
当程序进行汇编时，出现错误。

(h) 这里引用宏 MAC1。
同一个宏被引用两次。

(i) 这里定义 GLAB 标签为全局符号。
已定义过相同的标签，因此当程序进行汇编时会出错。

< 上述应用示例的汇编列表 >

```
        NAME    SAMPLE
          :
REF1 :  MAC1
          ; Macro expansion
??RA0000 :
          :
GLAB :                          <- Error
        BR      ??RA0000
        BR      GLAB
          ; Source code
          :
        BR      !LLAB           <- Error
        BR      !GLAB
          :
REF2 :  MAC1
          ; Macro expansion
??RA0001 :
          :
GLAB :                          <- Error
        BR      ??RA0001
        BR      GLAB
          ; Source code
          :
        END
```

宏 MAC1 定义全局符号 GLAB。
宏 MAC1 被引用了两次。当宏第二次展开时将产生错误，因为全局符号 GLAB 被定义了两次。

4.4.4 宏运算符

有两种宏运算符？

(1) & (级联)

- 在宏本体中，“&”把两个字符串相级联。
  在宏展开时间，“&”左边的字符串与右边的字符串进行级联。在字符串级联之后，“&”符号本身将消失。
- 在宏定义时间，“&”之前和之后的字符串字符被识别为局部字符和形式参数。在宏展开时间，“&”之前和之后的字符串字符作为局部字符和形式参数进行运算，并级联至独立符号。
- 在单引号内的 “&”符号可简单的作为数据处理。
- 连续的两个 “&”符号作为一个 “&”符号处理。

下列为一个应用示例。

(a) 宏定义

```
MAC     MACRO   P
LAB&P :                 <- 形式参数 P 被识别
        D&B     10H
        DB      'P'
        DB      P
        DB      '&P'
        ENDM
```

(b) 宏引用

```
        MAC     1H
LAB1H :
        DB      10H     <- D&B 级联为 DB
        DB      'P'
        DB      1H
        DB      '&P'    <- 引号中的 '&' 只是数据
```

(2) '（单引号）

- 在宏引用行或 IPP 命令中，如果位于单引号内的字符串出现在实际参数的开始或出现在分界符之后时，字符串会翻译为实际参数。在没有单引号引用的情况下，字符串作为实际参数传送。
- 如果出现在宏本体中的字符串由一对单引号引用时，该字符串作为数据处理。
- 如果单纯的要在文本中使用单引号，应把单引号连续写两遍。

< 应用示例 >

```
        NAME    SAMP
MAC1    MACRO   P
        IRP     Q, <P>
                MOV     A, #Q
        ENDM
ENDM


        MAC1    '10, 20, 30'
```

当上述示例中的源代码进行汇编时，宏 MAC1 将展开如下。

```
IRP     Q, <10, 20, 30>
        MOV     A, #Q
ENDM
        MOV     A, #10          ; IRP 展开
        MOV     A, #20          ; IRP 展开
        MOV     A, #30          ; IRP 展开
```

4.5 保留字

保留字是指预存在汇编器中的字符串。保留字包括机器语言指令、命令、控制指令、运算符、寄存器名称和 sfr 符号。保留字只能用于特定目的的应用。

下表说明了保留字可在源代码语句中出现的位置，并列出了汇编器中的保留字

表 4-22. 保留字可出现的区域

区段	说明
符号区域	此区域不能出现保留字。
助记符区域	只有机器语言指令和命令可出现在该区域。
操作数区域	只有运算符、 sfr 符号和寄存器名称可出现在该区域。
注释区域	所有的保留字均能出现在该区域。

表 4-23. 保留字列表

类型	保留字
运算符	AND, BITPOS, DATAPOS, EQ (=), GE (>=), GT (>), HIGH, HIGHW， LE (<=), LOW, LOWW， LT (<), MASK, MOD, NE (< >), NOT, OR, SHL, SHR, XOR
伪指令	AT, BASE, BASEP, BR, BSEG, CALL, CALLT0, CSEG, DB, DBIT, DG, DS, DSEG, DSPRAM, DW, END, ENDM, ENDS, EQU, EXITM, EXTBIT, EXTRN, FIXED, IHRAM, IRP, IXRAM, LOCAL, LRAM, MACRO, MIRRORP, NAME, OPT_BYTE, ORG, PAGE64KP, PUBLIC, REPT, SADDR, SADDRP, SECUR_ID, SET, UNIT, UNIT64KP, UNITP
控制指令	COND, NOCOND, DEBUG, NODEBUG, DEBUGA [DG], NODEBUGA [NODG], EJECT [EJ], FORMFEED, NOFORMFEED, GEN, NOGEN, IF, _IF, ELSEIF, _ELSEIF, ELSE, ENDIF, INCLUDE [IC], KANJICODE, LENGTH, LIST [LI], NOLIST [NOLI], PROCESSOR [PC], SET, RESET, SUBTITLE [ST], SYMLIST, NOSYMLIST, TAB, TITLE [TT], WIDTH, XREF[XR], NOXREF [NOXR]
其他	DGL, DGS, SFR, SFRP, TOL_INF

备注 控制指令后面方括号中的内容为缩写形式。

参阅目标器件的用户手册，以获取 sfr 名称列表、中断请求源列表以及机器语言指令和寄存器名称列表。

4.6 指令

该部分说明了 78K0R 单片机指令的功能。

4.6.1 与 78K0 单片机的区别 （用于汇编器用户）

- 新流水线架构减少了用于所有指令的时钟周期数。现有的程序必须重新计算。
- 改变了所有的指令代码图。必须重新汇编程序。在进行重新汇编时，代码长度通常是增加的；但是在一些情况下，如果新指令取代了旧指令的话，总的代码长度将缩水。
- 存储器空间由 64 KB 变为 1 MB，允许更大的堆栈深度。如果汇编程序操作的 RAM 可进行堆栈指针访问的话，则需要进行地址的改变。根据多重 CALL 和多重中断深度，堆栈大小应比正常需求的稍大些。
- CALLT 表从 [0040H 至 007FH] 移动到 [0080H 至 00BFH]。对于 CALLT 表的引用地址必须改变。
- 如果使用了 78K0 单片机的库切换机制，那么必须重写汇编器程序。
- 当使用扩展 RAM 时，需改变地址。更新这些地址。
- 如果程序从扩展 RAM 中执行指令的话，将受到存储器空间地址改变的影响。把 BR !addr16 的语句更改到 BR !!addr20 中，把 CALL !addr16 语句改变为 CALL !!addr20 语句。
- 没有 IMS 或 IXS 寄存器 （用于设置存储器空间）。除非使用外部存储器，否则使用这些寄存器的代码必须被删除。如果使用外部存储器，注意 MM/MEM 寄存器 （用于设置存储器空间）的规范已更改。关于详细信息，请参考目标单片机的用户手册。
- 下列指令已被删除且输出交替码，结果导致代码长度增加。这些指令仍然可用来指定 -compati 选项，但是汇编器将使用替代码自动取代这些指令。

指令	操作数	备注
DIVUW	C	在切换时，交替指令会分开，这增加了执行时间。 已经添加了切换指令，因此推荐将该指令改变为新的切换指令。
ROR4	[HL]	交替指令的执行时间较长。 已经添加了切换指令，因此推荐将该指令改变为新的切换指令。
ROL4	[HL]	交替指令的执行时间较长。 已经添加了切换指令，因此推荐将该指令改变为新的切换指令。
ADJBA	None	交替指令的执行时间较长。没有等效的附加指令。
ADJBS	None	交替指令的执行时间较长。没有等效的附加指令。
CALLF	!addr11	CALLF 自动更改为 3 个字节的 CALL !addr16 指令。 无需修改，该指令可继续使用。
DBNZ	B, $addr16 C, $addr16 saddr, $addr16	该指令分裂为下列指令：DEC B / DEC C / DEC saddr 和 BNZ $addr20。无需修改，该指令可继续使用。

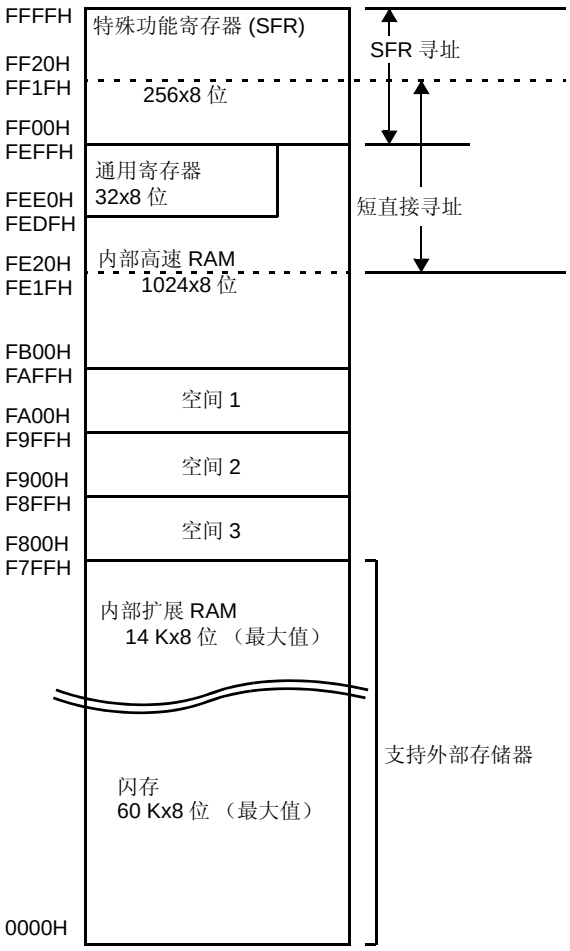
4.6.2 存储器空间

(1) 存储器空间

78K0 单片机的存储器空间限制在 64 KB 以内，但是在 78K0R 单片机中扩展到了 1 MB。

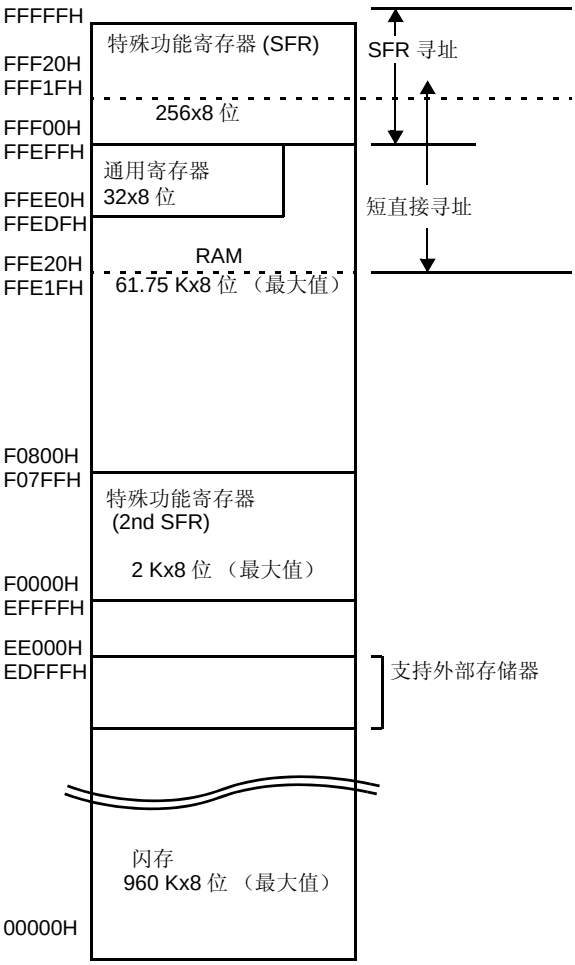
图 4-8. 78K0 和 78K0R 单片机的存储器分布图

< 78K0 单片机 >



- 程序存储器空间为 60 KB （最大值）。
- 内部高速 RAM 空间为 1 KB （最大值）（可堆栈）。
  内部扩展 RAM 空间为 14 KB （最大值）（可读取）。
- 空间 1、空间 2 和空间 3 的地址为 F800H 至 FAFFH （固定）。
- 支持外部扩展存储器。

< 78K0R 单片机 >



- 程序存储器空间为 960 KB （最大值）。
- RAM 空间为 61.75 KB （最大值）（允许堆栈和读取）。
- 2nd SFR 空间 （名称已改变）为 2 KB （最大值），从 F0000H 至 F07FFH。
- 支持外部扩展存储器。
  外部扩转存储器空间位置从器件自带的闪存区域至 EDFFFH。

(2) 内部程序存储器空间

在 78K0R 单片机中，程序存储器空间的地址范围从 00000H 至 EFFFFH。

关于内部 ROM 闪存的最大容量，请参考目标单片机的用户手册。

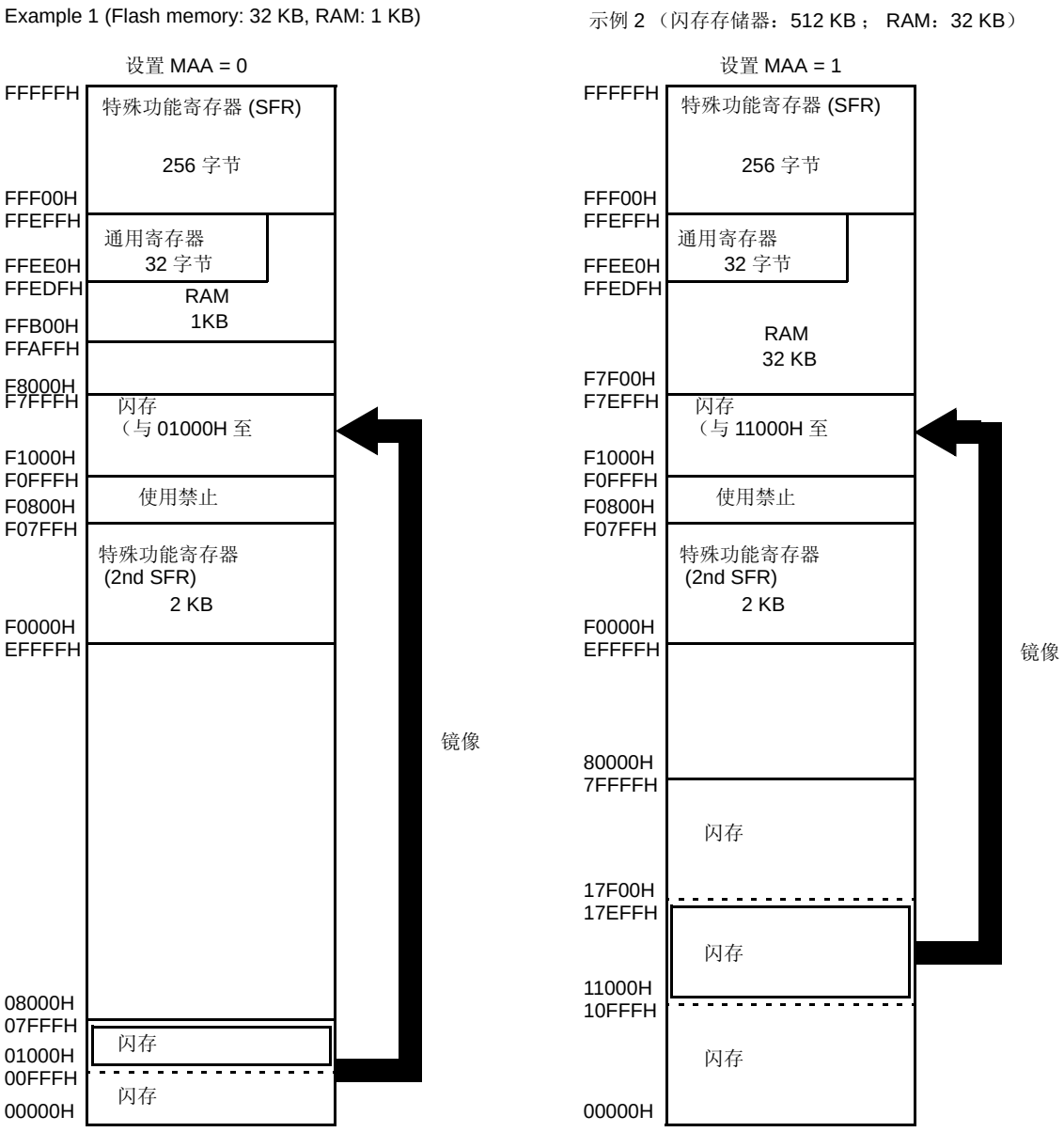
(a) 镜像空间

在 78K0R 单片机中，存储一般数据的区域为 00000H 至 0FFFFH （当 MAA = 0 时）以及从 10000H 至 1FFFFH （当 MAA = 1 时）的地址，均被镜像到地址 F0000H 至 FFFFFH。存储一般数据的内容可由较短的代码从地址 F0000H 至 FFFFFH 进行读取。但是，存储一般数据的区域不会镜像到 SFR、 2nd SFR、 RAM 和使用禁止区域。

镜像区域为只读区域，不能进行指令的拿取。

下图显示了两个示例。每个产品的规格不同，因此关于详细信息请参考目标单片机的用户手册。

图 4-9.　镜像区域示例



备注　　MAA：处理器模式控制寄存器 （PMC）的位 0。（详情请参阅 "(a)　处理器模式控制寄存器（PMC） "。）

(b) 向量表区域

在 78K0R 单片机中，从 0000H 至 007FH 的 128 字节区域保留为向量表区域。中断数量为 61（最大值）+ RESET 向量 + 片上调试向量 + 软件断点向量。由于向量只是两个字节的代码，因此中断分支的目标地址范围是从 00000H 至 0FFFFH 的 64 KB 区间。在 78K0 单片机中，从 0040H 至 007FH 的地址用于 CALLT 表；但在 78K0R 单片机中改变了向量地址。

(c) CALLT 指令表区域

在 78K0R 单片机中，从 0080H 至 00BFH 的 64 字节区域保留为 CALLT 指令表区域。

在 78K0 单片机中， CALL 指令为 1 个字节，但 78K0R 单片机具有 2 个字节的 CALL 指令。地址已经被改变。

由于地址代码只为 2 个字节的长度，因此中断分支目标地址为 00000H 至 0FFFFH 的 64 KB 区间。

(3) 内部数据存储器 （内部 RAM）空间

78K0 单片机具有内部高速 RAM 和内部扩展 RAM。内部高速 RAM 可用于堆栈，内部扩展 RAM 可用于读取。相对而言， 78K0R 单片机只用一个 RAM 区域就可进行堆栈和读取操作。

地址上限固定在 FFEFFH，而地址下限可根据产品的 RAM 容量向下扩展。最大容量为 61.75 KB。更多关于下限的详细信息，请参考目标单片机的用户手册。

78K0 系列和 78K0R 系列具有相同地址的通用寄存器区域和 saddr 空间 （FFEE0H 至 FFEFFH）。堆栈可位于 RAM 中的任何位置。

(4) 特殊功能寄存器 (SFR）区域

与通用寄存器不同， SFR 具有特殊功能。

SFR 空间位于 FFF00H 至 FFFFFH 区域。

虽然 SFR 规范与 78K0 单片机规范一致，但是在 78K0 系列中的一些改变会影响一些寄存器的固定地址。关于详细信息，请参考目标单片机的用户手册。

(5) 扩展 SFR （2nd SFR）区域

与通用寄存器不同， 2nd SFR 具有特殊功能。

2nd SFR 空间为 F0000H 至 F07FFH。在 SFR 区域 （FFF00H 至 FFFFFH）以外的空间被分配至该扩展 SFR 空间。然而，访问扩展 SFR 区域的指令比访问 SFR 区域的指令长 1 个字节。

(6) 外部存储器空间

外部存储器空间可通过设置存储器扩展模式寄存器进行访问。该存储器空间可从闪存存储器扩展至 EDFFFH。

在独立模式下，可获得 28 个外部引脚 （A19 至 A0 和 D7 至 D0）。在多重模式下，可获得 20 个外部引脚 （A19 至 A0）。

使用外部存储器进行引脚设定时，请参考目标器件用户手册中描述端口功能的章节。

注意事项 为了从外部存储器区域读取指令，应从 RAM 中的分支指令 （CALL 或 BR）开始并从外部存储器区域中的返回指令 （RET、 RETB 或 RETI）结束。

虽然闪存存储器区域和外部存储器区域相连，但程序不可横跨这两个区域。

4.6.3 寄存器

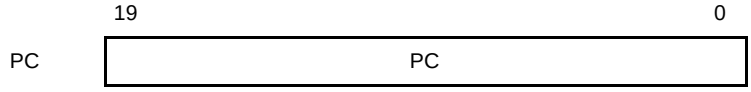
(1) 控制寄存器

控制寄存器是指具有控制程序顺序、程序状态和堆栈存储器的特殊功能寄存器。其中包括程序计数器、程序状态字和堆栈指针。

(a) 程序计数器 （PC）

程序计数器时一个 20 位的寄存器，用于存放下一个执行程序的地址。

图 4-10. 程序计数器的配置



(b) 程序状态字 （PSW）

程序状态字是一个 8 位寄存器，内部存放指令执行的设置和复位标志。

ISP1 标志添加作为产品的位 2，用于支持 4 个中断级。

当中断请求发生时，程序状态字会自动保存在堆栈中并执行 PUSH PSW 指令，当执行 RETB 或 RETI 指令和 POP PSW 指令时，程序状态字会自动恢复。

在复位信号输入端 PSW 设为 06H。

图 4-11. 程序状态字的配置

	7	6	5	4	3	2	1	0
PSW	IE	Z	RBS1	AC	RBS0	ISP1	ISP0	CY

- 中断使能标志 （IE）

  该标志用于控制 CPU 的中断请求确认。

  当 IE = 0 时，中断禁止 （DI），并且除了非屏蔽中断之外的所有中断均不能使用。

  当 IE = 1 时，允许中断 （EI），由中断屏蔽标志和优先级指定标志通过中断请求确认对各种中断源进行控制 。

  该标志通过执行 DI 指令或中断请求确认进行复位 （0）。通过执行 EI 指令置 （1）。

- 零标志 （Z）

  当运算结果为零时，该标志置 （1）。其他情况下均复位 （0）。

- 寄存器库选择标志 （RBS0 和 RBS1）

  这两个 1 比特的标志可用来选择 4 个寄存器库。

  这些标志的 2 比特大小的信息表示通过执行 SBL RBn 指令进行库选择。

- 辅助进位标志 （AC）

  当运算从 bit 3 进位或向 bit 3 借位时该标志置 （1）。其他情况下均复位 （0）。

- 服务优先级标志 （ISP0 和 ISP1）

  这些标志管理可确认的屏蔽向量中断的优先级。对于优先级低于 ISP0 和 ISP1 的值的向量中断请求将禁止确认，这由优先级指定寄存器 （PR）进行指定。实际的中断请求确认由中断使能标志 （IE）的状态控制。

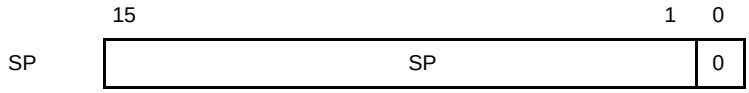
- 进位标志 （CY）

  通过执行加 / 减指令，该标志存储溢出或下溢。其存放执行循环移位指令而产生的移出值，并在位操作指令执行时起到位累加器的功能。

(c) 堆栈指针 （SP）

该 16 位寄存器存放堆栈的起始地址。该堆栈可位于内部 RAM 中。

图 4-12. 堆栈指针的配置



SP 指针在写入 （存储）堆栈之前减少，并在读取 （恢复）堆栈之后增加。

RESET 输入将使 SP 指针的内容变为未定义，因此必须在指令执行前对 SP 指针进行初始化。需指定 SP 指针为偶数地址。若指定了奇数地址， LSB 将设置为固定值 0。

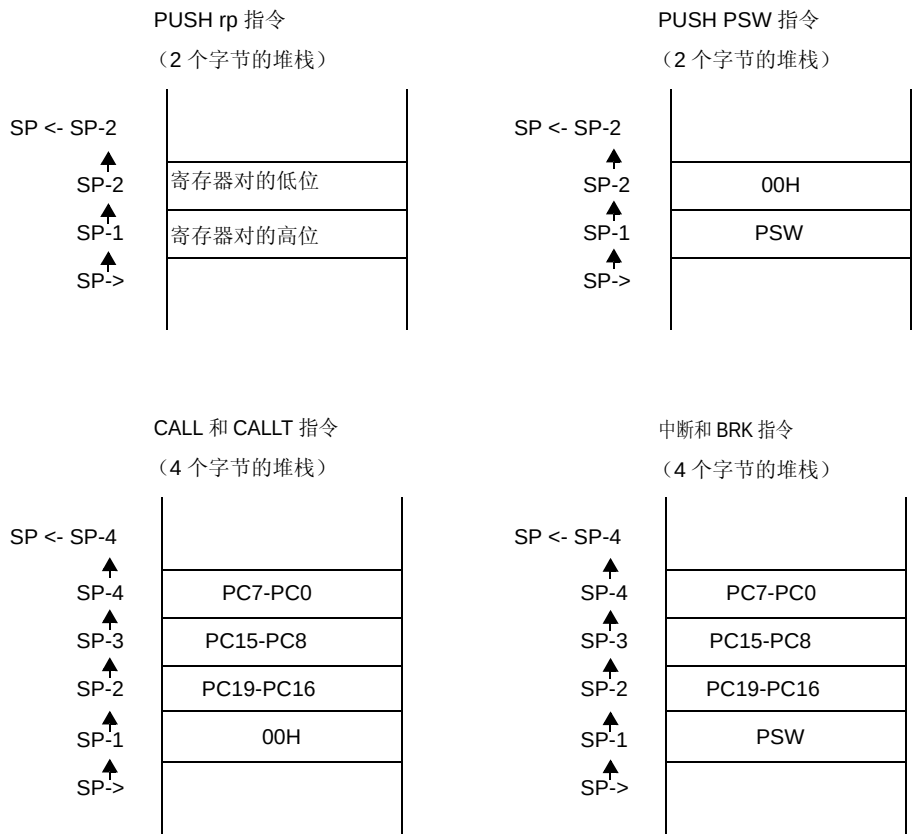
由于扩展了 78K0R 单片机，用于调用指令和中断的堆栈地址长度将增加 1 个字节。 16 位宽的堆栈 RAM 使得堆栈数据大小为 2 个字节或 4 个字节。（查看下列 " 表 4-24. 78K0 和 78K0R 单片机的堆栈数据大小的区别 "。）

表 4-24. 78K0 和 78K0R 单片机的堆栈数据大小的区别

存指令	恢复指令	78K0 单片机的堆栈数据大小	78K0R 单片机的堆栈数据大小
PUSH rp	POP rp	2 字节	2 字节
PUSH PSW	POP PSW	1 byte	2 字节
CALL, CALLT	RET	2 字节	4 字节
中断	RETI	3 字节	4 字节
BRK	RETB	3 字节	4 字节

下图显示了在 78K0R 单片机中，通过堆栈操作进行数据存储。

图 4-13. 数据存放于堆栈存储器



堆栈指针可能只指向内部 RAM。可在 F0000H 至 FFFFFH 范围内指定地址，因此注意不要超出内部 RAM 的存储器空间。若指定了内部 RAM 空间以外的地址，那么对于该地址的写操作将被忽略，并且将返回未定义值到读操作。

(2) 通用寄存器

片上通用寄存器分布于 RAM 中 FFEE0H 至 FFEFFH 地址范围。4 个寄存器库，每个库由 8 个 8 位寄存器（X、A、C、B、E、D、L 和 H）组成。通过 CPU 控制指令揸 EL RBn

每个寄存器可作为一个 8 位寄存器使用，两个 8 位寄存器组可作为 16 位寄存器使用。

程序可根据寄存器的功能名称（X、A、C、B、E、D、L、H、AX、BC、DE 和 HL）或绝对名称（R0 至 R7，RP0 至 RP3）指定所需的寄存器。

注意事项 禁止使用通用寄存器空间（FFEE0H 至 FFEFFH）作为指令读取区域或堆栈区域。

表 4-25. 通用寄存器列表 （78K0 兼容）

库名称	寄存器				绝对地址
	功能名称		绝对名称		
	16 位	8 位	16 位	8 位	
BANK0	HL	H	RP3	R7	FFEFFH
		L		R6	FFEFEH
	DE	D	RP2	R5	FFEFDH
		E		R4	FFEFCH
	BC	B	RP1	R3	FFEFBH
		C		R2	FFEFAH
	AX	A	RP0	R1	FFEF9H
		X		R0	FFEF8H
BANK1	HL	H	RP3	R7	FFEF7H
		L		R6	FFEF6H
	DE	D	RP2	R5	FFEF5H
		E		R4	FFEF4H
	BC	B	RP1	R3	FFEF3H
		C		R2	FFEF2H
	AX	A	RP0	R1	FFEF1H
		X		R0	FFEF0H
BANK2	HL	H	RP3	R7	FFEEFH
		L		R6	FFEEEH
	DE	D	RP2	R5	FFEEDH
		E		R4	FFEECH
	BC	B	RP1	R3	FFEEBH
		C		R2	FFEEAH
	AX	A	RP0	R1	FFEE9H
		X		R0	FFEF8H
BANK3	HL	H	RP3	R7	FFEE7H
		L		R6	FFEE6H
	DE	D	RP2	R5	FFEE5H
		E		R4	FFEE4H
	BC	B	RP1	R3	FFEE3H
		C		R2	FFEE2H
	AX	A	RP0	R1	FFEE1H
		X		R0	FFEE0H

(3) ES 和 CS 寄存器

78K0R 单片机添加了 ES 和 CS 寄存器。 ES 寄存器指定数据指令的高位地址， CS 寄存器指定分支指令的高位地址。查看 “(2) 数据地址寻址”以获得更多关于如何使用 ES 寄存器的信息，查看 “(1) 指令地址的寻址？以获得更多关于如何使用 CS 寄存器的信息。

复位时， ES 设置为 0FH， CS 设置为 00H

图 4-14. ES 和 CS 寄存器配置

	7	6	5	4	3	2	1	0
ES	0	0	0	0	ES3	ES2	ES1	ES0

	7	6	5	4	3	2	1	0
CS	0	0	0	0	CS3	CP2	CP1	CP0

(4) 特殊功能寄存器 (SFR)

下表列出了 78K0R 单片机中 SFR 的固定地址。

表 4-26. SFR 固定地址列表

地址	寄存器名称
FFFF8H	SPL
FFFF9H	SPH
FFFFAH	PSW
FFFFBH	Reserve
FFFFCH	CS
FFFFDH	ES
FFFFEH	PMC
FFFFFH	MEM

(a) 处理器模式控制寄存器 （PMC）

处理器模式控制寄存器为 8 位寄存器。详情请参阅 "(2) 内部程序存储器空间 "。

复位时， PMC 设置为 00H。

图 4-15. 处理器模式控制寄存器的配置

地址：FFFFEH 复位：00H R/W

符号	7	6	5	4	3	2	1	0
PMC	0	0	0	0	0	0	0	MAA

MAA	Flash 存储器镜像到区域 F0000H 至 FFFFFH 注
0	00000H 至 0FFFFH 地址，镜像到 F0000H 至 FFFFFH
1	10000H 至 1FFFFH 地址，镜像到 F0000H 至 FFFFFH

注 SFR 和 RAM 区域也位于 F0000H 至 FFFFFH 地址范围。若区域重叠的话，其优先级较高。

注意事项 1. 当首次初始化时，只能设置处理器模式控制一次。初始化结束后，禁止写入 PMC。
2. 设置 PMC 之后，需至少等待一个指令的时间才能访问镜像区域。

4.6.4 寻址

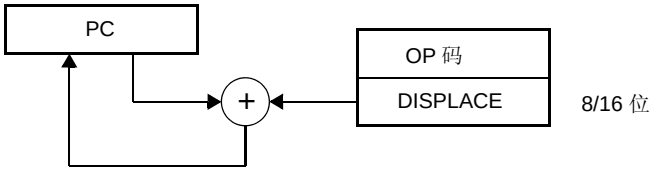
有两种寻址类型：数据地址的寻址和程序地址的寻址。该部分描述了两种地址类型的寻址模式。

(1) 指令地址的寻址

(a) 相对寻址

相对寻址从指令字中添加了一个位移值（带符号的补充数据：-128 至 +127 或 -32768 至 +32767）到程序计数器（PC），并把求和的值存入程序计数器。执行分支指令用于指定程序地址。
相对寻址只用于分支指令。

图 4-16. 相对寻址如何工作



(b) 立即寻址

立即寻址通过把程序字中的立即数据存放于程序计数器的方法来指定分支目标程序地址。
有两种类型的立即寻址：CALL !!addr20 或 BR !!addr20 指定的 20 位地址，以及 CALL !addr16 或 BR !addr16 指定的 16 位地址。当指定 16 位地址时，高 4 位设置为 0000。

图 4-17. CALL !!addr20/BR !!addr20 寻址的示例

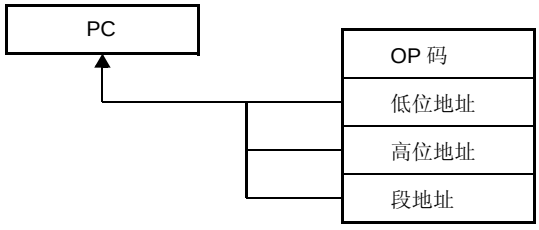
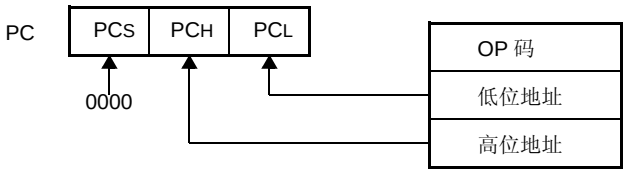


图 4-18. CALL !addr16/BR !addr16 寻址的示例

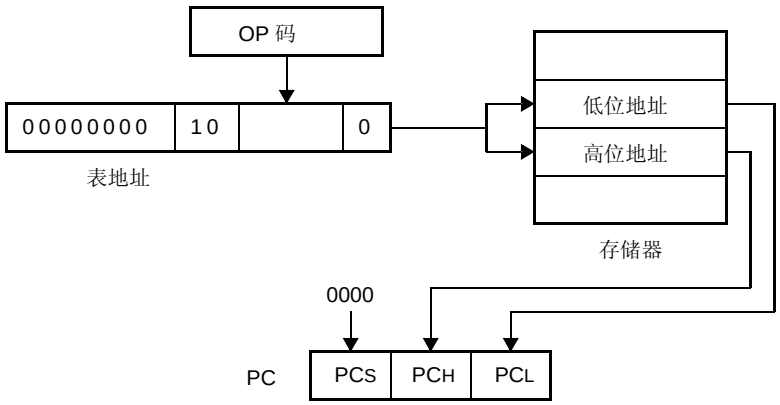


(c) 表间接寻址

表间接寻址通过指令字中的 5 位立即数据来指定 CALLT 表区域 （0080H 至 00BFH）中的地址。其中存放 CALLT 表地址中的内容以及下一 CALLT 表地址，作为程序计数器中的 16 位数据。这指定了被调用的程序地址。表间接寻址仅应用于 CALLT 指令。

在 78K0R 单片机中，只有从 00000H 至 0FFFFH 的 64 KB 空间内允许分支。

图 4-19. 表间接寻址如何工作

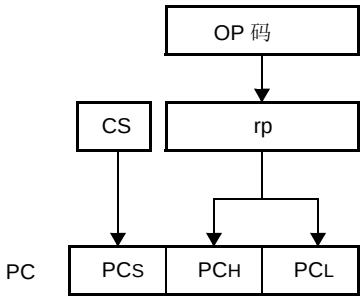


(d) 寄存器直接寻址

在寄存器直接寻址中程序指令字在当前寄存器库中指定一个通用寄存器对 （AX/BC/DE/HL）。该寄存器对中的内容和当前 CS 寄存器中的内容作为 20 位数据存放于程序计数器中。这指定了调用或分支程序地址。

寄存器直接寻址仅应用于 CALL AX/BC/DE/HL 指令和 BR AX 指令。

图 4-20. 寄存器直接寻址如何工作



(2) 数据地址寻址

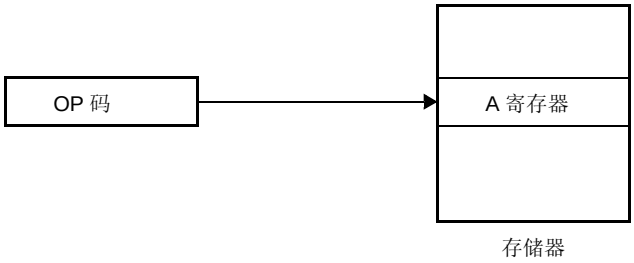
(a) 隐含寻址

隐含寻址用于指令访问带有特殊功能的寄存器，如累加器。指令字无需包含特殊区域来指令寄存器。寄存器规范由指令字本身隐含。

由于寄存器规范是隐含的，因此指令不带有操作数。

在 78K0R 单片机中，隐含寻址仅用于 MULU X 指令。

图 4-21. 隐含寻址如何工作



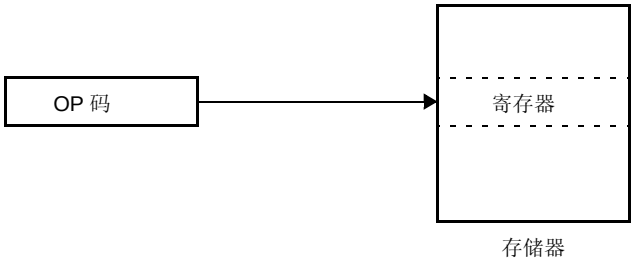
(b) 寄存器寻址

寄存器地址使用通用寄存器操作数对存储器进行访问。使用 3 位指令字指定 8 位寄存器，或 2 位指令字指定 16 位寄存器。

操作数格式如下所示。

格式	说明
r	X, A, C, B, E, D, L, H
rp	AX, BC, DE, HL

图 4-22. 寄存器寻址如何工作



(c) 直接寻址

直接寻址采用指令字中的立即数据作为操作数。可直接指定目标地址。

操作数格式如下所示。

格式	说明
ADDR16	标签或 16 位立即数据 （仅 F0000H 至 FFFFFH）
ES:ADDR16	标签或 16 位立即数据 （由 ES 寄存器指定高 4 位地址）

图 4-23. ADDR16 寻址的示例

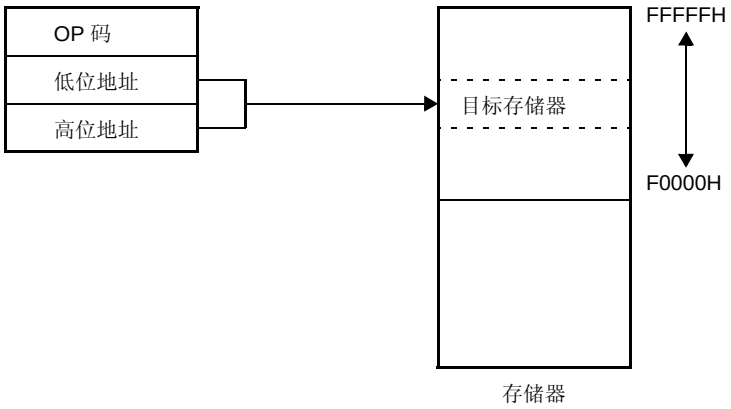
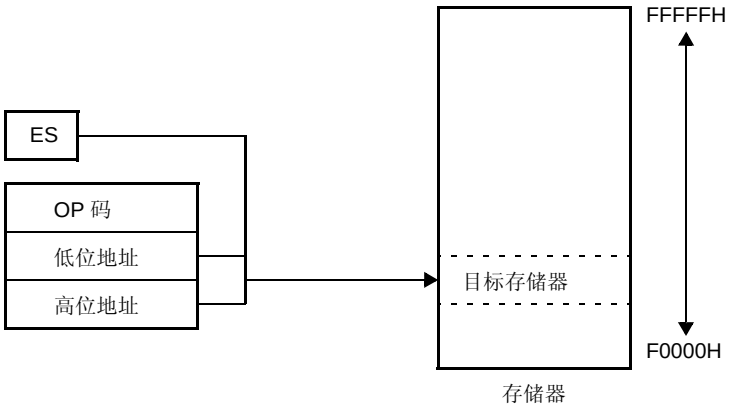


图 4-24. ES:ADDR16 寻址的示例



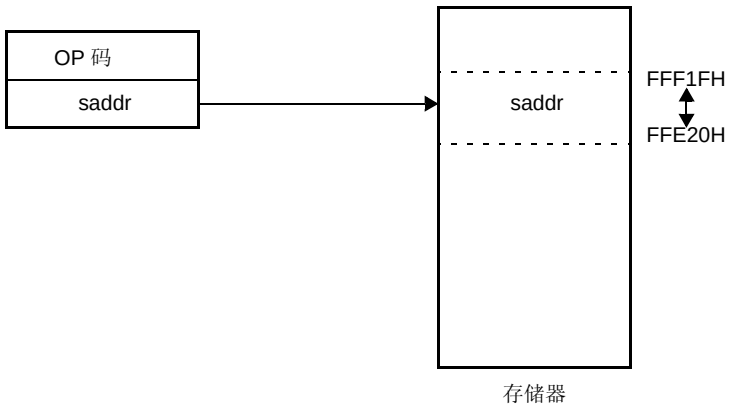
(d) 短直接寻址

短直接寻址采用指令字中的 8 位数据直接指定目标地址。该类型的寻址仅应用于 FFE20H 至 FFF1FH 的地址范围。

操作数格式如下所示。

格式	说明
SADDR	标签或立即数据的地址范围是 FFE20H 至 FFF1FH，或者立即数据的地址范围是 0FE20H 至 0FF1FH （受 FFE20H 至 FFF1FH 空间的限制）
SADDRP	标签或立即数据的地址范围是 FFE20H 至 FFF1FH，或者立即数据的地址范围是 0FE20H 至 0FF1FH （仅偶数地址） （受 FFE20H 至 FFF1FH 空间的限制）

图 4-25. 短直接寻址如何工作



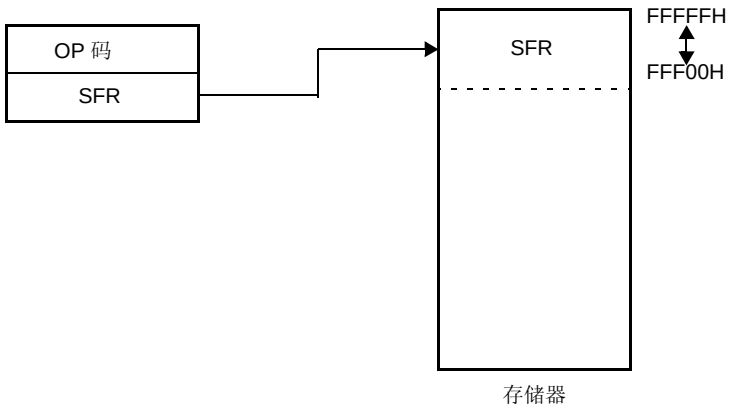
备注 SADDR 或 SADDRP 规范 （省略了地址的高 4 位）可以指定 FE20H 至 FF1FH 为 16 位立即数据或 FFE20H 至 FFF1FH 为 20 位立即数据。
无论使用何种格式，指定的地址处于存储器空间内的 FFE20H 至 FFF1FH 地址范围。

(e) SFR 寻址

SFR 寻址采用指令字中的 8 位数据直接指定目标 SFR 地址。该类型的寻址仅应用于 FFF00H 至 FFFFFH 的地址范围。
操作数格式如下所示。

格式	说明
SFR	SFR 寄存器名称
SFRP	16 位 SFR 寄存器名称 （仅偶数地址）

图 4-26. SFR 寻址如何工作



(f) 寄存器间接寻址

寄存器间接寻址采用指令字中的数据来指定一组寄存器对。指定的寄存器对中的内容作为操作数对目标存储器地址进行指定。

操作数格式如下所示。

格式	说明
-	[DE], [HL] （仅 F0000H 至 FFFFFH）
-	ES:[DE], ES:[HL] （ES 寄存器指定高 4 位地址）

图 4-27. [DE] 和 [HL] 寻址的示例

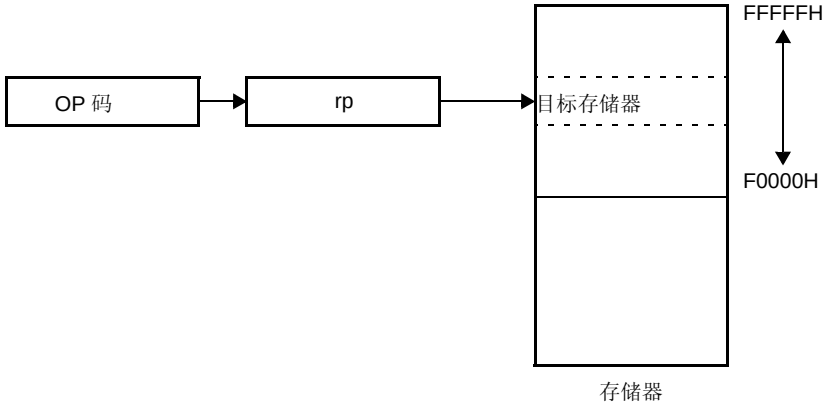
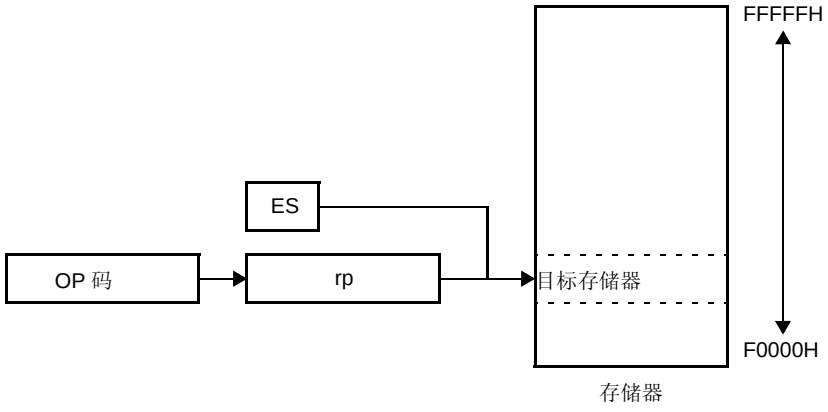


图 4-28. ES:[DE] 和 ES:[HL] 寻址的示例



(g) 基址寻址

在基址寻址中，指令字指定一组寄存器对和补偿。在基址寄存器对的内容中添加补偿（8 位或 16 位立即数据）来指定目标地址。

操作数格式如下所示。

格式	说明
-	[HL + byte], [DE + byte], [SP + byte] （仅 F0000H 至 FFFFFH）
-	word[B], word[C] （仅 F0000H 至 FFFFFH）
-	word [BC] （仅 F0000H 至 FFFFFH）
-	ES:[HL + byte], ES:[DE + byte] （ES 寄存器指定高 4 位地址）
-	ES:word[B], ES:word[C] （ES 寄存器指定高 4 位地址）
-	ES : word [BC] （ES 寄存器指定高 4 位地址）

图 4-29. [SP+byte] 寻址的示例

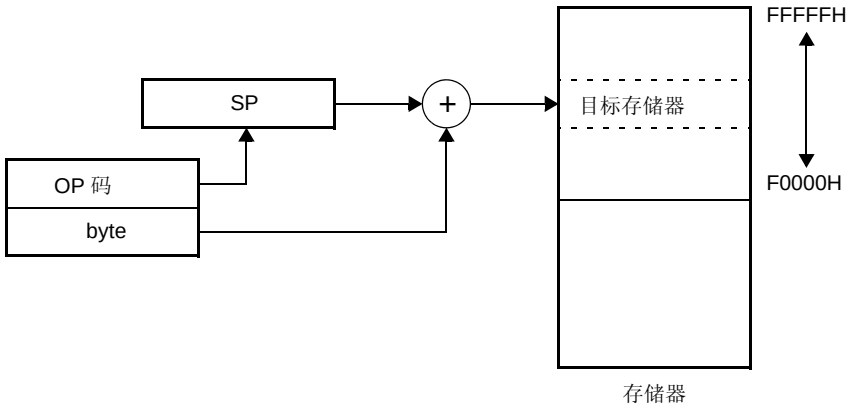


图 4-30. [HL+byte] 和 [DE+byte] 寻址的示例

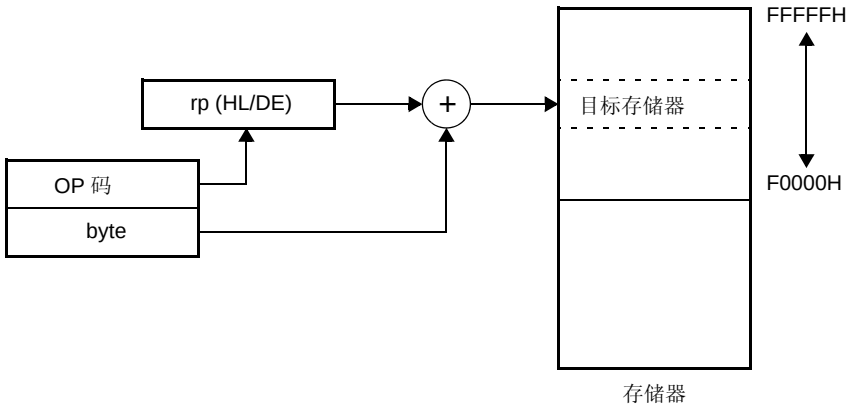


图 4-31. word[B] 和 word[C] 寻址的示例

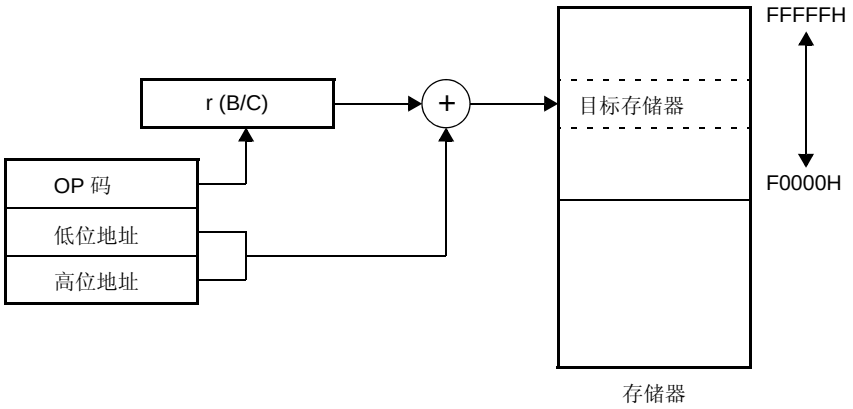


图 4-32. word[BC] 寻址的示例

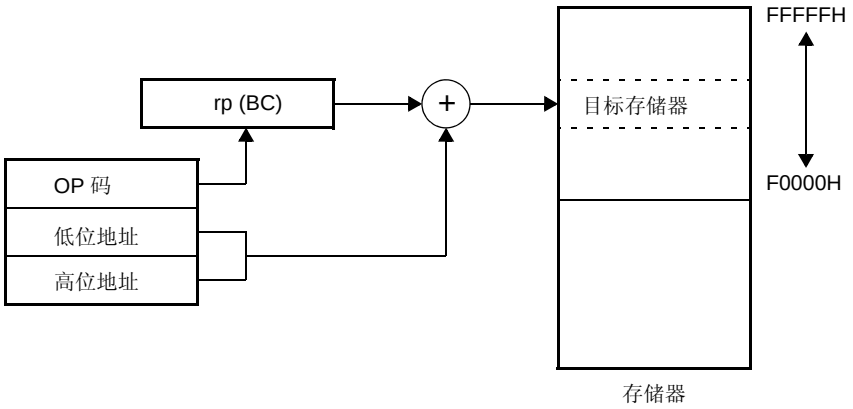


图 4-33. ES:[HL+byte] 和 ES:[DE+byte] 寻址的示例

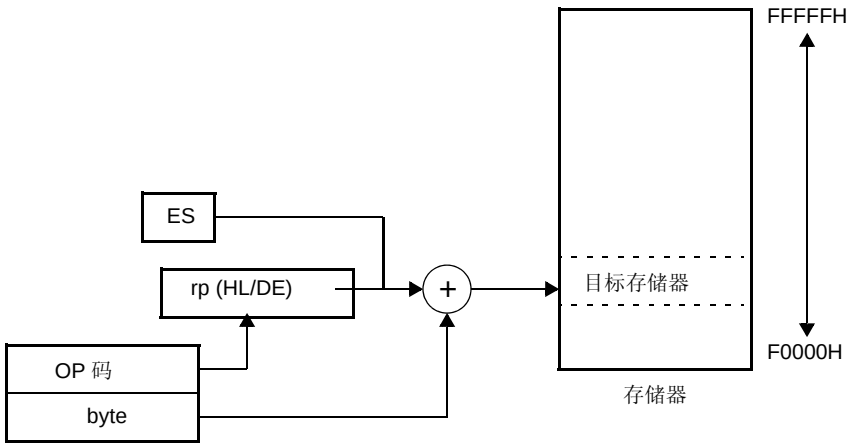


图 4-34. ES:word[B] 和 ES:word[C] 寻址的示例

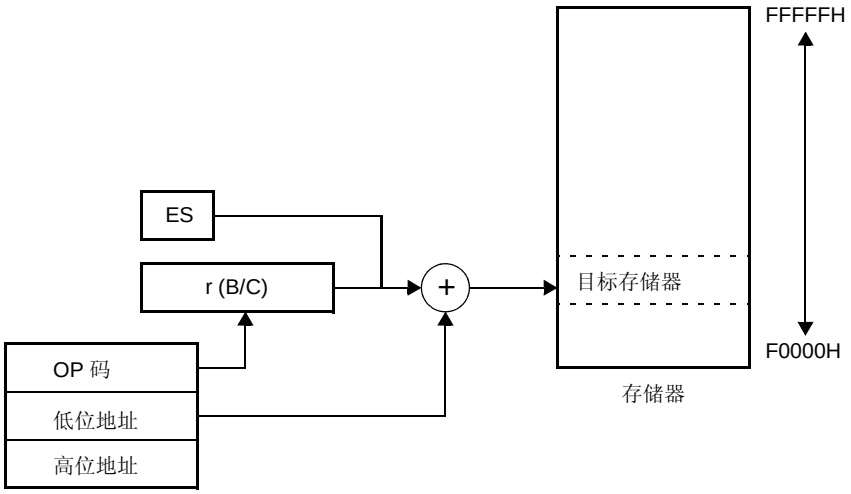
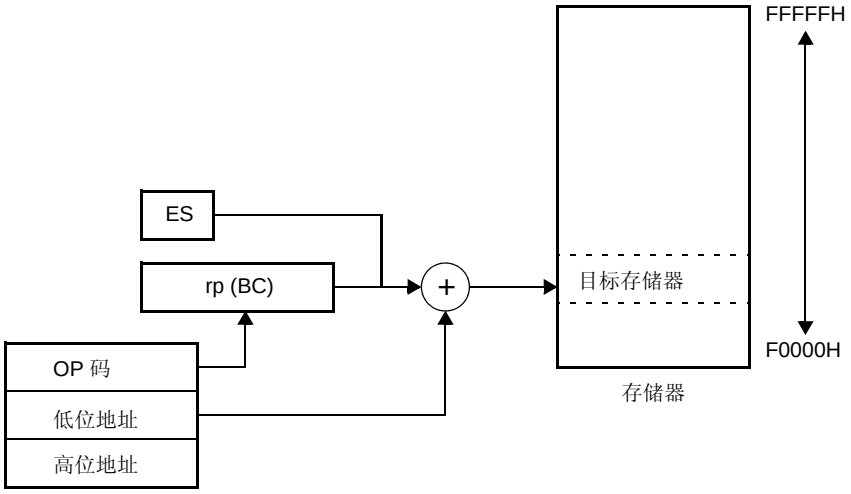


图 4-35. ES:word[BC] 寻址的示例



(h) 基址索引寻址

在基址索引寻址中，指令字指定一组寄存器对作为基址寄存器，以及 B 或 C 寄存器作为补偿寄存器。把基址寄存器中的内容添加至补偿寄存器从而指定目标地址。

操作数格式如下所示。

格式	说明
-	[HL + B], [HL + C] （仅 F0000H 至 FFFFFH）
-	ES:[HL + B], ES:[HL + C] （ES 寄存器指定高 4 位地址）

图 4-36. [HL+B] 至 [HL+C] 寻址的示例

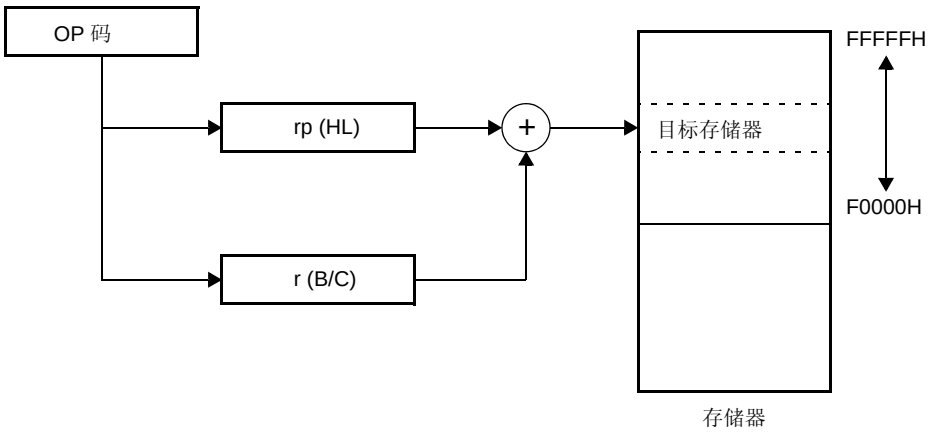
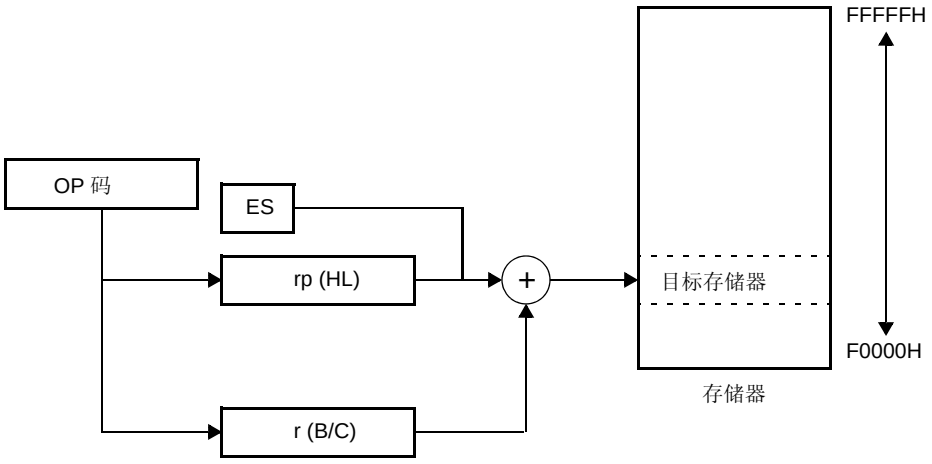


图 4-37. ES:[HL+B] 和 ES:[HL+C] 寻址的示例



(i) 堆栈寻址

堆栈寻址采用堆栈指针 （SP）的内容间接访问堆栈。当执行 PUSH 和 POP 指令、执行子程序调用和返回指令和由中断请求而产生的寄存器保存和恢复时，该类型的寻址会自动进行。

堆栈寻址仅应用于内部 RAM 区域。

操作数格式如下所示。

格式	说明
-	PUSH AX/BC/DE/HL POP AX/BC/DE/HL CALL/CALLT RET BRK RETB （产生中断请求） RETI

4.6.5 指令集

本章节列出了 78K0R 单片机指令集中的指令。
这些指令通用于 78K0R 系列单片机中的所有单片机。

(1) 操作数的表达形式和描述方法

" 操作数 " 区域内的每个指令的操作数由各种操作数类型决定其表达形式。（详情请参考汇编器规范。）当在源代码中有两种或更多种方式指定操作数时，选择其中一种方式进行。字母字符由一个大写字母表示，符号 #、 !、 !!、 $、 $!、 [ ] 和 ES: 为关键字并以其本身形式表示。各符号具有以下含义。

#	立即数据规范
!	16 位绝对地址规范
!!	20- 位绝对地址规范
$	8 位相对地址规范
$!	16- 位相对地址规范
[ ]	间接地址规范
ES	扩展地址规范

采用一个合适的值或标签来指定立即数据。当指定标签时，必须包含 #、 !、 !!、 $、 $!、 [ ] 或 ES: 符号。
对于寄存器操作数，可采用寄存器功能名称（X、A、C 等）或寄存器绝对名称（R0、R1、R2 等，如下表括号中所示）取代 r 和 rp。

表 4-27. 操作数类型表达和源代码格式

格式	说明
r	X(R0), A(R1), C(R2), B(R3), E(R4), D(R5), L(R6), H(R7)
rp	AX(RP0), BC(RP1), DE(RP2), HL(RP3)
sfr	特殊功能寄存器名称 （SFR 名称）
sfrp	特殊功能寄存器名称 （16 位 SFR，仅偶数地址[注]）
saddr	FFE20H 至 FFF1FH：立即数据或标签
saddrp	FFE20H 至 FFF1FH：立即数据或标签 （仅偶数地址[注]）
addr20	00000H 至 FFFFFH：立即数据或标签
addr16	0000H 至 0FFFFH：立即数据或标签 （偶数地址仅用于 16 位数据发送指令[注]）
addr5	0080H 至 00BFH：立即数据或标签 （仅偶数地址）
字	16- 位立即数据或标签
byte	8- 位立即数据或标签
位	3- 位立即数据或标签
RBn	RB0, RB1, RB2, RB3

注 当指定奇数地址时， Bit 0 = 0。

(2) 操作区域符号

" 操作 " 区域使用下列符号来表示指令执行时发生的操作。

表 4-28. 操作区域符号

符号	功能
A	A 寄存器：8 位累加器
X	X 寄存器
B	B 寄存器
C	C 寄存器
D	D 寄存器
E	E 寄存器
H	H 寄存器
L	L 寄存器
ES	ES 寄存器
CS	CS 寄存器
AX	AX 寄存器对：16 位累加器
BC	BC 寄存器对
DE	DE 寄存器对
HL	HL 寄存器对
PC	程序计数器
SP	堆栈指针
PSW	程序状态字
CY	进位标志
AC	辅助进位标志
Z	零标志
RBS	寄存器库选择标志
IE	中断请求使能标志
()	存储器内容由地址或括号中的寄存器内容表示
XH, XL XS, XH, XL	16 位寄存器：XH = 高 8 位， XL = 低 8 位 20 位寄存器：XS = 位 19 至位 16， XH = 位 15 至位 8， XL = 位 7 至位 0
^	Logical AND
v	Logical OR
∀	异或
—	反相数据
addr16	16 位立即数据
addr20	20 位立即数据
jdisp8	带符号 8 位数据 （位移值）
jdisp16	带符号 16 位数据 （位移值）

(3) 标志区域符号

“操作”区域使用下列符号来表示指令执行时发生的标志变化。

表 4-29. 标志区域符号

符号	标志变化
(白)	不变
0	清 0
1	置 1
x	根据结果进行设置或清除
R	之前存储的值被恢复

(4) PREFIX 指令

一些指令显示时带有 ES: 前缀。附加前缀使得可访问数据空间从 64 KB[F0000H 至 FFFFFH] 扩展为 1 MB[00000H 至 FFFFFH]。这是通过把 ES 寄存器的值添加到地址规范来完成的。当 PREFIX 操作码作为前缀附加到目标指令时，只有紧跟在 PREFIX 操作码之后执行的指令，其地址才增加了 ES 寄存器的值。

表 4-30. 使用 PREFIX 指令的示例

指令	OP 码				
	1	2	3	4	5
MOV !addr16, #byte	CFH	!addr16		#byte	-
MOV ES:!addr16, #byte	11H	CFH	!addr16		#byte
MOV A, [HL]	8BH	-	-	-	-
MOV A, ES : [HL]	11H	8BH	-	-	-

注意事项 在执行 PREFIX 指令之前，通常先在 ES 寄存器中设置正确的值，如 MOV ES, A。

(5) 操作列表

(a) 8 位数据传送指令

表 4-31. 操作列表 （8- 位数据传送指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
MOV	r, #byte	2	1	-	r <- byte			
	saddr, #byte	3	1	-	(saddr) <- byte			
	sfr, #byte	3	1	-	sfr <- byte			
	!addr16, #byte	4	1	-	(addr16) <- byte			
	A, r [注 3]	1	1	-	A <- r			
	A, r [注 3]	1	1	-	r <- A			
	A, saddr	2	1	-	A <- (saddr)			
	saddr, A	2	1	-	(saddr) <- A			
	A, sfr	2	1	-	A <- sfr			
	sfr, A	2	1	-	sfr <- A			
	A, !addr16	3	1	4	A <- (addr16)			
	!addr16, A	3	1	-	(addr16) <- A			
	PSW, #byte	3	3	-	PSW <- byte	x	x	x
	A, PSW	2	1	-	A <- PSW			
	PSW, A	2	3	-	PSW <- A	x	x	x
	ES, #byte	2	1	-	ES <- byte			
	ES, saddr	3	1	-	ES <- (saddr)			
	A, ES	2	1	-	A <- ES			
	ES, A	2	1	-	ES <- A			
	CS, #byte	3	1	-	CS <- byte			
	A, CS	2	1	-	A <- CS			
	CS, A	2	1	-	CS <- A			
	A, [DE]	1	1	4	A <- (DE)			
	[DE], A	1	1	-	(DE) <- A			
	[DE+byte], #byte	3	1	-	(DE + byte) <- byte			
	A, [DE+byte]	2	1	4	A <- (DE + byte)			
	[DE+byte], A	2	1	-	(DE + byte) <- A			
	A, [HL]	1	1	4	A <- (HL)			
	[HL], A	1	1	-	(HL) <- A			
	[HL+byte], #byte	3	1	-	(HL + byte) <- byte			
	A, [HL+byte]	2	1	4	A <- (HL + byte)			
	[HL+byte], A	2	1	-	(HL + byte) <- A			
	A, [HL+B]	2	1	4	A <- (HL + B)			

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	[HL+B], A	2	1	-	(HL + B) <- A			
	A, [HL+C]	2	1	4	A <- (HL + C)			
	[HL+C], A	2	1	-	(HL + C) <- A			
	word[B], #byte	4	1	-	(B + word) <- byte			
	A, word[B]	3	1	4	A <- (B + word)			
	word[B], A	3	1	-	(B + word) <- A			
	word[C], #byte	4	1	-	(C + word) <- byte			
	A, word[C]	3	1	4	A <- (C + word)			
	word[C], A	3	1	-	(C + word) <- A			
	word[BC], #byte	4	1	-	(BC + word) <- byte			
	A, word[BC]	3	1	4	A <- (BC + word)			
	word[BC], A	3	1	-	(BC + word) <- A			
	[SP+byte], #byte	3	1	-	(SP + byte) <- byte			
	A, [SP+byte]	2	1	-	A <- (SP + byte)			
	[SP+byte], A	2	1	-	(SP + byte) <- A			
	B, saddr	2	1	-	B <- (saddr)			
	B, !addr16	3	1	4	B <- (addr16)			
	C, saddr	2	1	-	C <- (saddr)			
	C, !addr16	3	1	4	C <- (addr16)			
	X, saddr	2	1	-	X <- (saddr)			
	X, !addr16	3	1	4	X <- (addr16)			
	ES:!addr16, #byte	5	2	-	(ES, addr16) <- byte			
	A, ES:!addr16	4	2	5	A <- (ES, addr16)			
	ES:!addr16, A	4	2	-	(ES, addr16) <- A			
	A, ES:[DE]	2	2	5	A <- (ES, DE)			
	ES:[DE], A	2	2	-	(ES, DE) <- A			
	ES:[DE+byte], #byte	4	2	-	((ES, DE) + byte) <- byte			
	A, ES:[DE+byte]	3	2	5	A <- ((ES, DE) + byte)			
	ES:[DE+byte], A	3	2	-	((ES, DE) + byte) <- A			
	A, ES:[HL]	2	2	5	A <- (ES, HL)			
	ES:[HL], A	2	2	-	(ES, HL) <- A			
	ES:[HL+byte], #byte	4	2	-	((ES, HL) + byte) <- byte			
	A, ES:[HL+byte]	3	2	5	A <- ((ES, HL) + byte)			
	ES:[HL+byte], A	3	2	-	((ES, HL) + byte) <- A			
	A, ES:[HL+B]	3	2	5	A <- ((ES, HL) + B)			
	ES:[HL+B], A	3	2	-	((ES, HL) + B) <- A			
	A, ES:[HL+C]	3	2	5	A <- ((ES, HL) + C)			
	ES:[HL+C], A	3	2	-	((ES, HL) + C) <- A			

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	ES:word[B], #byte	5	2	-	((ES, B) + word) <- byte			
	A, ES:word[B]	4	2	5	A <- ((ES, B) + word)			
	ES:word[B], A	4	2	-	((ES, B) + word) <- A			
	ES:word[C], #byte	5	2	-	((ES, C) + word) <- byte			
	A, ES:word[C]	4	2	5	A <- ((ES, C) + word)			
	ES:word[C], A	4	2	-	((ES, C) + word) <- A			
	ES:word[BC], #byte	5	2	-	((ES, BC) + word) <- byte			
	A, ES:word[BC]	4	2	5	A <- ((ES, BC) + word)			
	ES:word[BC], A	4	2	-	((ES, BC) + word) <- A			
	B, ES:!addr16	4	2	5	B <- (ES, addr16)			
	C, ES:!addr16	4	2	5	C <- (ES, addr16)			
	X, ES:!addr16	4	2	5	X <- (ES, addr16)			
XCH	A, r [注 3]	1 (r = X) 2 (except r = X)	1	-	A <--> r			
	A, saddr	3	2	-	A <--> (saddr)			
	A, sfr	3	2	-	A <--> sfr			
	A, !addr16	4	2	-	A <--> (addr16)			
	A, [DE]	2	2	-	A <--> (DE)			
	A, [DE+byte]	3	2	-	A <--> (DE + byte)			
	A, [HL]	2	2	-	A <--> (HL)			
	A, [HL+byte]	3	2	-	A <--> (HL + byte)			
	A, [HL+B]	2	2	-	A <--> (HL + B)			
	A, [HL+C]	2	2	-	A <--> (HL + C)			
	A, ES:!addr16	5	3	-	A <--> (ES, addr16)			
	A, ES:[DE]	3	3	-	A <--> (ES, DE)			
	A, ES:[DE+byte]	4	3	-	A <--> ((ES, DE) + byte)			
	A, ES:[HL]	3	3	-	A <--> (ES, HL)			
	A, ES:[HL+byte]	4	3	-	A <--> ((ES, HL) + byte)			
	A, ES:[HL+B]	3	3	-	A <--> ((ES, HL) + B)			
	A, ES:[HL+C]	3	3	-	A <--> ((ES, HL) + C)			
	A	1	1	-	A <- 01H			
	X	1	1	-	X <- 01H			
	B	1	1	-	B <- 01H			
	C	1	1	-	C <- 01H			
	saddr	2	1	-	(saddr) <- 01H			
	!addr16	3	1	-	(addr16) <- 01H			
	ES:!addr16	4	2	-	(ES, addr16) <- 01H			

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
CLRB	A	1	1	-	A <- 00H			
	X	1	1	-	X <- 00H			
	B	1	1	-	B <- 00H			
	C	1	1	-	C <- 00H			
	saddr	2	1	-	(saddr) <- 00H			
	!addr16	3	1	-	(addr16) <- 00H			
	ES:!addr16	4	2	-	(ES,addr16) <- 00H			
MOVS	[HL+byte], X	3	1	-	(HL + byte) <- X	x		x
	ES:[HL+byte], X	4	2	-	(ES, HL + byte) <- X	x		x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. Except r = A.

4. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟。

2. 时钟给在内部 ROM （Flash 存储器）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(b) 16 位数据传送指令

表 4-32. 操作列表 （16 位数据传送指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
MOVW	rp, #word	3	1	-	rp <- word			
	saddrp, #word	4	1	-	(saddrp) <- word			
	sfrp, #word	4	1	-	sfrp <- word			
	AX, saddrp	2	1	-	AX <- (saddrp)			
	saddrp, AX	2	1	-	(saddrp) <- AX			
	AX, sfrp	2	1	-	AX <- sfrp			
	sfrp, AX	2	1	-	sfrp <- AX			
	AX, rp [Note 3]	1	1	-	AX <- rp			
	rp, AX [注 3]	1	1	-	rp <- AX			
	AX, !addr16	3	1	4	AX <- (addr16)			
	!addr16, AX	3	1	-	(addr16) <- AX			
	AX, [DE]	1	1	4	AX <- (DE)			
	[DE], AX	1	1	-	(DE) <- AX			
	AX, [DE+byte]	2	1	4	AX <- (DE + byte)			
	[DE+byte], AX	2	1	-	(DE + byte) <- AX			
	AX, [HL]	1	1	4	AX <- (HL)			
	[HL], AX	1	1	-	(HL) <- AX			
	AX, [HL+byte]	2	1	4	AX <- (HL + byte)			
	[HL+byte], AX	2	1	-	(HL + byte) <- AX			
	AX, word[B]	3	1	4	AX <- (B + word)			
	word[B], AX	3	1	-	(B + word) <- AX			
	AX, word[C]	3	1	4	AX <- (C + word)			
	word[C], AX	3	1	-	(C + word) <- AX			
	AX, word[BC]	3	1	4	AX <- (BC + word)			
	word[BC], AX	3	1	-	(BC + word) <- AX			
	AX, [SP+byte]	2	1	-	AX <- (SP + byte)			
	[SP+byte], AX	2	1	-	(SP + byte) <- AX			
	BC, saddrp	2	1	-	BC <- (saddrp)			
	BC, !addr16	3	1	4	BC <- (addr16)			
	DE, saddrp	2	1	-	DE <- (saddrp)			
	DE, !addr16	3	1	4	DE <- (addr16)			
	HL, saddrp	2	1	-	HL <- (saddrp)			
	HL, !addr16	3	1	4	HL <- (addr16)			
	AX, ES:!addr16	4	2	5	AX <- (ES, addr16)			
	ES:!addr16, AX	4	2	-	(ES, addr16) <- AX			
	AX, ES:[DE]	2	2	5	AX <- (ES, DE)			

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	ES:[DE], AX	2	2	-	(ES, DE) <- AX			
	AX, ES:[DE+byte]	3	2	5	AX <- ((ES, DE) + byte)			
	ES:[DE+byte], AX	3	2	-	((ES, DE) + byte) <- AX			
	AX, ES:[HL]	2	2	5	AX <- (ES, HL)			
	ES:[HL], AX	2	2	-	(ES, HL) <- AX			
	AX, ES:[HL+byte]	3	2	5	AX <- ((ES, HL) + byte)			
	ES:[HL+byte], AX	3	2	-	((ES, HL) + byte) <- AX			
	AX, ES:word[B]	4	2	5	AX <- ((ES, B) + word)			
	ES:word[B], AX	4	2	-	((ES, B) + word) <- AX			
	AX, ES:word[C]	4	2	5	AX <- ((ES, C) + word)			
	ES:word[C], AX	4	2	-	((ES, C) + word) <- AX			
	AX, ES:word[BC]	4	2	5	AX <- ((ES, BC) + word)			
	ES:word[BC], AX	4	2	-	((ES, BC) + word) <- AX			
	BC, ES:!addr16	4	2	5	BC <- (ES, addr16)			
	DE, ES:!addr16	4	2	5	DE <- (ES, addr16)			
	HL, ES:!addr16	4	2	5	HL <- (ES, addr16)			
XCHW	AX, rp [Note 3]	1	1	-	AX <--> rp			
ONEW	AX	1	1	-	AX <- 0001H			
	BC	1	1	-	BC <- 0001H			
CLRW	AX	1	1	-	AX <- 0000H			
	BC	1	1	-	BC <- 0000H			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 除 rp = AX

4. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区 , 当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(c) 8 位 操作指令

表 4-33. 操作列表 （8 位操作指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
ADD	A, #byte	2	1	-	A, CY <- A + byte	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) + byte	x	x	x
	A, r [注 3]	2	1	-	A, CY <- A + r	x	x	x
	r, A	2	1	-	r, CY <- r + A	x	x	x
	A, saddr	2	1	-	A, CY <- A + (saddr)	x	x	x
	A, !addr16	3	1	4	A, CY <- A + (addr16)	x	x	x
	A, [HL]	1	1	4	A, CY <- A + (HL)	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A + (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A + (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A + (HL + C)	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A + (ES, addr16)	x	x	x
	A, ES:[HL]	2	2	5	A,CY <- A + (ES, HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A,CY <- A + ((ES, HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A,CY <- A + ((ES, HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A,CY <- A + ((ES, HL) + C)	x	x	x
ADDC	A, #byte	2	1	-	A, CY <- A + byte + CY	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) + byte + CY	x	x	x
	A, r [注 3]	2	1	-	A, CY <- A + r + CY	x	x	x
	r, A	2	1	-	r, CY <- r + A + CY	x	x	x
	A, saddr	2	1	-	A, CY <- A + (saddr) + CY	x	x	x
	A, !addr16	3	1	4	A, CY <- A + (addr16) + CY	x	x	x
	A, [HL]	1	1	4	A, CY <- A + (HL) + CY	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A + (HL + byte) + CY	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A + (HL + B) + CY	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A + (HL + C) + CY	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A + (ES, addr16) + CY	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A + (ES, HL) + CY	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A + ((ES, HL) + byte) + CY	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A + ((ES, HL) + B) + CY	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A + ((ES, HL) + C) + CY	x	x	x
SUB	A, #byte	2	1	-	A, CY <- A - byte	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) - byte	x	x	x
	A, r [注 3]	2	1	-	A, CY <- A - r	x	x	x

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	r, A	2	1	-	r, CY <- r - A	x	x	x
	A, saddr	2	1	-	A, CY <- A - (saddr)	x	x	x
	A, !addr16	3	1	4	A, CY <- A - (addr16)	x	x	x
	A, [HL]	1	1	4	A, CY <- A - (HL)	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A - (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A - (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A - (HL + C)	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A - (ES:addr16)	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A - (ES:HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A - ((ES:HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A - ((ES:HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A - ((ES:HL) + C)	x	x	x
SUBC	A, #byte	2	1	-	A, CY <- A - byte - CY	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) - byte - CY	x	x	x
	A, r [注 3]	2	1	-	A, CY <- A - r - CY	x	x	x
	r, A	2	1	-	r, CY <- r - A - CY	x	x	x
	A, saddr	2	1	-	A, CY <- A - (saddr) - CY	x	x	x
	A, !addr16	3	1	4	A, CY <- A - (addr16) - CY	x	x	x
	A, [HL]	1	1	4	A, CY <- A - (HL) - CY	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A - (HL + byte) - CY	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A - (HL + B) - CY	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A - (HL + C) - CY	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A - (ES:addr16) - CY	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A - (ES:HL) - CY	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A - ((ES:HL) + byte) - CY	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A - ((ES:HL) + B) - CY	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A - ((ES:HL) + C) - CY	x	x	x
AND	A, #byte	2	1	-	A <- A ^ byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) ^ byte	x		
	A, r [注 3]	2	1	-	A <- A ^ r	x		
	r, A	2	1	-	r <- r ^ A	x		
	A, saddr	2	1	-	A <- A ^ (saddr)	x		
	A, !addr16	3	1	4	A <- A ^ (addr16)	x		
	A, [HL]	1	1	4	A <- A ^ (HL)	x		
	A, [HL+byte]	2	1	4	A <- A ^ (HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A ^ (HL + B)	x		

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	A, [HL+C]	2	1	4	A <- A ^ (HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A ^ (ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A ^ (ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A ^ ((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A ^ ((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A ^ ((ES:HL) + C)	x		
OR	A, #byte	2	1	-	A <- A v byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) v byte	x		
	A, r [注 3]	2	1	-	A <- A v r	x		
	r, A	2	1	-	r <- r v A	x		
	A, saddr	2	1	-	A <- A v (saddr)	x		
	A, !addr16	3	1	4	A <- A v (addr16)	x		
	A, [HL]	1	1	4	A <- A v (HL)	x		
	A, [HL+byte]	2	1	4	A <- A v (HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A v (HL + B)	x		
	A, [HL+C]	2	1	4	A <- A v (HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A v (ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A v (ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A v ((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A v ((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A v ((ES:HL) + C)	x		
XOR	A, #byte	2	1	-	A <- A ∀byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) ∀byte	x		
	A, r [注 3]	2	1	-	A <- A ∀r	x		
	r, A	2	1	-	r <- r ∀A	x		
	A, saddr	2	1	-	A <- A ∀(saddr)	x		
	A, !addr16	3	1	4	A <- A ∀(addr16)	x		
	A, [HL]	1	1	4	A <- A ∀(HL)	x		
	A, [HL+byte]	2	1	4	A <- A ∀(HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A ∀(HL + B)	x		
	A, [HL+C]	2	1	4	A <- A ∀(HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A ∀(ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A ∀(ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A ∀((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A ∀((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A ∀((ES:HL) + C)	x		

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
CMP	A, #byte	2	1	-	A - byte	x	x	x
	saddr, #byte	3	1	-	(saddr) - byte	x	x	x
	A, r [注 3]	2	1	-	A - r	x	x	x
	r, A	2	1	-	r - A	x	x	x
	A, saddr	2	1	-	A - (saddr)	x	x	x
	A, !addr16	3	1	4	A - (addr16)	x	x	x
	A, [HL]	1	1	4	A - (HL)	x	x	x
	A, [HL+byte]	2	1	4	A - (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A - (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A - (HL + C)	x	x	x
	!addr16, #byte	4	1	4	(addr16) - byte	x	x	x
	A, ES:!addr16	4	2	5	A - (ES:addr16)	x	x	x
	A, ES:[HL]	2	2	5	A - (ES:HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A - ((ES:HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A - ((ES:HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A - ((ES:HL) + C)	x	x	x
	ES:!addr16, #byte	5	2	5	(ES:addr16) - byte	x	x	x
CMP0	A	1	1	-	A - 00H	x	x	x
	X	1	1	-	X - 00H	x	x	x
	B	1	1	-	B - 00H	x	x	x
	C	1	1	-	C - 00H	x	x	x
	saddr	2	1	-	(saddr) - 00H	x	x	x
	!addr16	3	1	4	(addr16) - 00H	x	x	x
	ES:!addr16	4	2	5	(ES:addr16) - 00H	x	x	x
CMPS	X, [HL+byte]	3	1	4	X - (HL + byte)	x	x	x
	X, ES:[HL+byte]	4	2	5	X - ((ES:HL) + byte)	x	x	x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. Except r = A.

4. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号在闪存的结束地址有映射（最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(d) 16- 位 运算指令

表 4-34. 操作列表 （16- 位运算指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
ADDW	AX, #word	3	1	-	AX, CY <- AX + word	x	x	x
	AX, AX	1	1	-	AX, CY <- AX + AX	x	x	x
	AX, BC	1	1	-	AX, CY <- AX + BC	x	x	x
	AX, DE	1	1	-	AX, CY <- AX + DE	x	x	x
	AX, HL	1	1	-	AX, CY <- AX + HL	x	x	x
	AX, saddrp	2	1	-	AX, CY <- AX + (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX, CY <- AX + (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX, CY <- AX + (HL + byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX, CY <- AX + (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX, CY <- AX + ((ES:HL) + byte)	x	x	x
SUBW	AX, #word	3	1	-	AX, CY <- AX - word	x	x	x
	AX, BC	1	1	-	AX, CY <- AX - BC	x	x	x
	AX, DE	1	1	-	AX, CY <- AX - DE	x	x	x
	AX, HL	1	1	-	AX, CY <- AX - HL	x	x	x
	AX, saddrp	2	1	-	AX, CY <- AX - (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX, CY <- AX - (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX, CY <- AX - (HL - byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX, CY <- AX - (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX, CY <- AX - ((ES:HL) + byte)	x	x	x
CMPW	AX, #word	3	1	-	AX - word	x	x	x
	AX, BC	1	1	-	AX - BC	x	x	x
	AX, DE	1	1	-	AX - DE	x	x	x
	AX, HL	1	1	-	AX - HL	x	x	x
	AX, saddrp	2	1	-	AX - (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX - (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX - (HL + byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX - (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX - ((ES:HL) + byte)	x	x	x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(e) 乘法指令

表 4-35. 操作列表 （乘法指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
MULU	X	1	1	-	AX <- A x X			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(f) 递增递减指令

表 4-36. 操作列表 （递增 - 递减指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
INC	r	1	1	-	r <- r + 1	x	x	
	saddr	2	2	-	(saddr) <- (saddr) + 1	x	x	
	!addr16	3	2	-	(addr16) <- (addr16) + 1	x	x	
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) + 1	x	x	
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) + 1	x	x	
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) + 1	x	x	
DEC	r	1	1	-	r <- r - 1	x	x	
	saddr	2	2	-	(saddr) <- (saddr) - 1	x	x	
	!addr16	3	2	-	(addr16) <- (addr16) - 1	x	x	
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) - 1	x	x	
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) - 1	x	x	
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) - 1	x	x	
INCW	rp	1	1	-	rp <- rp + 1			
	saddrp	2	2	-	(saddrp) <- (saddrp) + 1			
	!addr16	3	2	-	(addr16) <- (addr16) + 1			
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) + 1			
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) + 1			
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) + 1			
DECW	rp	1	1	-	rp <- rp - 1			
	saddrp	2	2	-	(saddrp) <- (saddrp) - 1			
	!addr16	3	2	-	(addr16) <- (addr16) - 1			
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) - 1			
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) - 1			
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) - 1			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(g) 移位指令

表 4-37. 操作列表 （移位指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
SHR	A, cnt	2	1	-	(CY <- A_0, A_{m-1} <- A_m, A_7 <- 0) x cnt			x
SHRW	AX, cnt	2	1	-	(CY <- AX_0, AX_{m-1} <- AX_m, AX_{15} <- 0) x cnt			x
SHL	A, cnt	2	1	-	(CY <- A_7, A_m <- A_{m-1}, A_0 <- 0) x cnt			x
	B, cnt	2	1	-	(CY <- B_7, B_m <- B_{m-1}, B_0 <- 0) x cnt			x
	C, cnt	2	1	-	(CY <- C_7, C_m <- C_{m-1}, C_0 <- 0) x cnt			x
SHLW	AX, cnt	2	1	-	(CY <- AX_{15}, AX_m <- AX_{m-1}, AX_0 <- 0) x cnt			x
	BC, cnt	2	1	-	(CY <- BC_{15}, BC_m <- BC_{m-1}, BC_0 <- 0) x cnt			x
SAR	A, cnt	2	1	-	(CY <- A_0, A_{m-1} <- A_m, A_7 <- A_7) xcnt			x
SARW	AX, cnt	2	1	-	(CY <- AX_0, AX_{m-1} <- AX_m, AX_{15} <- AX_{15}) x cnt			x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. cnt 是位移数。

4. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(h) 循环指令

表 4-38. 操作列表 （循环指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
ROR	A, 1	2	1	-	(CY, A_7 <- A_0, A_{m-1} <- A_m)x1			x
ROL	A, 1	2	1	-	(CY, A_0 <- A_7, A_{m+1} <- A_m)x1			x
RORC	A, 1	2	1	-	(CY <- A_0, A_7 <- CY, A_{m-1} <- A_m)x1			x
ROLC	A, 1	2	1	-	(CY <- A_7, A_0 <- CY, A_{m+1} <- A_m)x1			x
ROLWC	AX, 1	2	1	-	(CY <- AX_{15}, AX_0 <- CY, AX_{m+1} <- AX_m)x1			x
	BC, 1	2	1	-	(CY <- BC_{15}, BC_0 <- CY, BC_{m+1} <- BC_m)x1			x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(i) 位操作指令

表 4-39. 操作列表 （位操作指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
MOV1	CY, saddr.bit	3	1	-	CY <- (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- sfr.bit			x
	CY, A.bit	2	1	-	CY <- A.bit			x
	CY, PSW.bit	3	1	-	CY <- PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- (HL).bit			x
	saddr.bit, CY	3	2	-	(saddr).bit <- CY			
	sfr.bit, CY	3	2	-	sfr.bit <- CY			
	A.bit, CY	2	1	-	A.bit <- CY			
	PSW.bit, CY	3	4	-	PSW.bit <- CY	x	x	
	[HL].bit, CY	2	2	-	(HL).bit <- CY			
	CY, ES:[HL].bit	3	2	5	CY <- (ES, HL).bit			x
	ES:[HL].bit, CY	3	3	-	(ES, HL).bit <- CY			
AND1	CY, saddr.bit	3	1	-	CY <- CY ^ (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY ^ sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY ^ A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY ^ PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY ^ (HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY ^ (ES, HL).bit			x
OR1	CY, saddr.bit	3	1	-	CY <- CY v (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY v sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY v A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY v PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY v (HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY v (ES, HL).bit			x
XOR1	CY, saddr.bit	3	1	-	CY <- CY ∀(saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY ∀sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY ∀A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY ∀PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY ∀(HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY ∀(ES, HL).bit			x
SET1	saddr.bit	3	2	-	(saddr).bit <- 1			
	sfr.bit	3	2	-	sfr.bit <- 1			
	A.bit	2	1	-	A.bit <- 1			
	!addr16.bit	4	2	-	(addr16).bit <- 1			
	PSW.bit	3	4	-	PSW.bit <- 1	x	x	x
	[HL].bit	2	2	-	(HL).bit <- 1			

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
	ES:!addr16.bit	5	3	-	(ES, addr16).bit <- 1			
	ES:[HL].bit	3	3	-	(ES, HL).bit <- 1			
CLR1	saddr.bit	3	2	-	(saddr).bit <- 0			
	sfr.bit	3	2	-	sfr.bit <- 0			
	A.bit	2	1	-	A.bit <- 0			
	!addr16.bit	4	2	-	(addr16).bit <- 0			
	PSW.bit	3	4	-	PSW.bit <- 0	x	x	x
	[HL].bit	2	2	-	(HL).bit <- 0			
	ES:!addr16.bit	5	3	-	(ES, addr16).bit <- 0			
	ES:[HL].bit	3	3	-	(ES, HL).bit <- 0			
SET1	CY	2	1	-	CY <- 1			1
CLR1	CY	2	1	-	CY <- 0			0
NOT1	CY	2	1	-	CY <- $\overline{CY}$			x

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区 , 当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(j) 调用返回指令

表 4-40. 操作列表 （调用返回指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
CALL	rp	2	3	-	(SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PC <- CS, rp, SP <- SP - 4			
	$!addr20	3	3	-	(SP - 2) <- (PC + 3)S, (SP - 3) <- (PC + 3)H, (SP - 4) <- (PC + 3)L, PC <- PC + 3 + jdisp16, SP <- SP - 4			
	!addr16	3	3	-	(SP - 2) <- (PC + 3)S, (SP - 3) <- (PC + 3)H, (SP - 4) <- (PC + 3)L, PC <- 0000, addr16, SP <- SP - 4			
	!!addr20	4	3	-	(SP - 2) <- (PC + 4)S, (SP - 3) <- (PC + 4)H, (SP - 4) <- (PC + 4)L, PC <- addr20, SP <- SP - 4			
CALLT	[addr5]	2	5	-	(SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PCS <- 0000, PCH <- (000000000000, addr5 + 1), PCL <- (000000000000, addr5), SP <- SP - 4			
BRK	-	2	5	-	(SP - 1) <- PSW, (SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PCS <- 0000, PCH <- (0007FH), PCL <- (0007EH), SP <- SP - 4, IE <- 0			
RET	-	1	6	-	PCL <- (SP), PCH <- (SP + 1), PCS <- (SP + 2), SP <- SP + 4			

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
RETI	-	2	6	-	PC_L <- (SP), PC_H <- (SP + 1), PC_S <- (SP + 2), PSW <- (SP + 3), SP <- SP + 4	R	R	R
RETB	-	2	6	-	PC_L <- (SP), PC_H <- (SP + 1), PC_S <- (SP + 2), PSW <- (SP + 3), SP <- SP + 4	R	R	R

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(k) 堆栈操作指令

表 4-41. 操作列表 （堆栈操作指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
PUSH	PSW	2	1	-	(SP - 1) <- PSW, (SP - 2) <- 00H, SP <- SP - 2			
	rp	1	1	-	(SP - 1) <- rp_H, (SP - 2) <- rp_L, SP <- SP - 2			
POP	PSW	2	3	-	PSW <- (SP + 1), SP <- SP + 2	R	R	R
	rp	1	1	-	rp_L <- (SP), rp_H <- (SP + 1), SP <- SP + 2			
MOVW	SP, #word	4	1	-	SP <- word			
	SP, AX	2	1	-	SP <- AX			
	AX, SP	2	1	-	AX <- SP			
	HL, SP	3	1	-	HL <- SP			
	BC, SP	3	1	-	BC <- SP			
	DE, SP	3	1	-	DE <- SP			
ADDW	SP, #byte	2	1	-	SP <- SP + byte			
SUBW	SP, #byte	2	1	-	SP <- SP - byte			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(I) 绝对转移指令

表 4-42. 操作列表 （绝对转移指令）

记忆体单元的	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
BR	AX	2	3	-	PC <- CS, AX			
	$addr20	2	3	-	PC <- PC + 2 + jdisp8			
	$!addr20	3	3	-	PC <- PC + 3 + jdisp16			
	!addr16	3	3	-	PC <- 0000, addr16			
	!!addr20	4	3	-	PC <- addr20			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(m) 条件转移指令

表 4-43. 操作列表 （条件转移指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
BC	$addr20	2	2/4[注 3]	-	PC <- PC + 2 + jdisp8 if CY = 1			
BNC	$addr20	2	2/4[注 3]	-	PC <- PC + 2 + jdisp8 if CY = 0			
BZ	$addr20	2	2/4[注 3]	-	PC <- PC + 2 + jdisp8 if Z = 1			
BNZ	$addr20	2	2/4[注 3]	-	PC <- PC + 2 + jdisp8 if Z = 0			
BH	$addr20	3	2/4[注 3]	-	PC <- PC + 3 + jdisp8 if (Z v CY) = 0			
BNH	$addr20	3	2/4[注 3]	-	PC <- PC + 3 + jdisp8 if (Z v CY) = 1			
BT	saddr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 1			
	sfr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 1			
	A.bit, $addr20	3	3/5[注 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 1			
	PSW.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 1			
	[HL].bit, $addr20	3	3/5[注 3]	6/8	PC <- PC + 3 + jdisp8 if (HL).bit = 1			
	ES:[HL].bit, $addr20	4	4/6[注 3]	7/9	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 1			
BF	saddr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 0			
	sfr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 0			
	A.bit, $addr20	3	3/5[注 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 0			
	PSW.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 0			
	[HL].bit, $addr20	3	3/5[注 3]	6/8	PC <- PC + 3 + jdisp8 if (HL).bit = 0			
	ES:[HL].bit, $addr20	4	4/6[注 3]	7/9	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 0			
BTCLR	saddr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 1 然后复位 (saddr).bit			
	sfr.bit, $addr20	4	3/5[注 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 1 然后复位 sfr.bit			
	A.bit, $addr20	3	3/5[注 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 1 然后复位 sfr.bit			

记忆体单元	操作数	字节	时钟		操作	标志[注 4]		
			注 1	注 2		Z	AC	CY
	PSW.bit, $addr20	4	5/7 [注 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 1 然后复位 PSW.bitt	x	x	x
	[HL].bit, $addr20	3	3/5 [注 3]	-	PC <- PC + 3 + jdisp8 if (HL).bit = 1 然后复位 (HL).bit			
	ES:[HL].bit, $addr20	4	4/6 [注 3]	-	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 1 然后复位 (ES, HL).bit			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 时钟号显示满足条件或不满足条件。

4. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区 , 当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(n) 条件步进指令

表 4-44. 操作列表 (条件步进指令)

记忆体单元的	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
SKC	-	2	1	-	Next instruction skip if CY = 1			
SKNC	-	2	1	-	Next instruction skip if CY = 0			
SKZ	-	2	1	-	如果 Z = 1，跳过下个指令			
SKNZ	-	2	1	-	如果 Z = 0，跳过下个指令			
SKH	-	2	1	-	如果 (Z v CY) = 0，跳过下个指令			
SKNH	-	2	1	-	如果 (Z v CY) = 1，跳过下个指令			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变
0 ： 清除为 0
1 ： 设置为 1
x ： 根据结果设置或清除
R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

(o) CPU 控制指令

表 4-45. 操作列表 （CPU 控制指令）

记忆体单元	操作数	字节	时钟		操作	标志[注 3]		
			注 1	注 2		Z	AC	CY
SEL	RBn	2	1	-	RBS[1:0] <- n			
NOP	-	1	1	-	无操作			
EI	-	3	4	-	IE <- 1 (中断可用)			
DI	-	3	4	-	IE <- 0 (中断不可用)			
HALT	-	2	3	-	设置终止模式			
STOP	-	2	3	-	设置停止模式			

注 **1.** 当访问内部 RAM 区域或 SFR 区域，或使用无数据访问的指令时。

2. 当访问程序存储区域时。

3. 标记字段符号显示标记改变的同时执行指令。

Blank ： 不变

0 ： 清除为 0

1 ： 设置为 1

x ： 根据结果设置或清除

R ： 恢复以前保存的值

备注 **1.** 通过处理器时钟控制寄存器 （PCC)，选择 CPU 时钟 (fCLK) 的一个时钟即一个指令时钟

2. 时钟给在内部 ROM （闪存）区域中的程序编号。

3. n 是寄存器组号 （n=0 到 3）。

4. 产品存储在连接内部闪存区的外部存储区，当使用外部总线接口功能时，添加等待号到指令执行时钟号 - 在闪存的结束地址有映射 （最大 16 字节）。这是因为当进行指令码优先读取时，超过闪存空间并且访问外部存储空间时，输入外部存储等待。有关等待号，请参阅 "(b) 对作为数据的外部存储器内容进行访问 "。

4.6.6 解释指令

这部分解释了 78K0R 单片机中所使用的指令。

表 4-46. 汇编语言的指令列表

功能	指令
8 位数 据传送指令。	MOV, XCH, ONEB, CLRB, MOVS
16 位数据传送指令	MOVW, XCHW, ONEW, CLRW
8- 位 运算指令	ADD, ADDC, SUB, SUBC, AND, OR, XOR, CMP, CMP0, CMPS
16- 位 运算指令	ADDW, SUBW, CMPW
乘法指令	MULU
递增 / 递减指令	INC, DEC, INCW, DECW
移位指令	SHR, SHRW, SHL, SHLW, SAR, SARW
循环移位指令	ROR, ROL, RORC, ROLC, ROLWC
位操作指令	MOV1, AND1, OR1, XOR1, SET1, CLR1, NOT1
调用返回指令	CALL, CALLT, BRK, RET, RETI, RETB
堆栈操作指令	PUSH, POP, MOVW, ADDW, SUBW
绝对转移指令	BR
条件转移指令	BC, BNC, BZ, BNZ, BH, BNH, BT, BF, BTCLR
条件跳转指令	SKC, SKNC, SKZ, SKNZ, SKH, SKNH
CPU 控制指令	SEL, NOP, EI, DI, HALT, STOP

下列信息解释了独立指令。

[Instruction format]

显示了指令的基本书写格式。

[Operation]

指令操作由所使用的代码地址显示。

[Operand]

显示了可在该指令中指定的操作数。请查阅 “(2) 操作区域符号 ?，以获取关于每个操作数的描述。

[Flag]

表示由指令的执行而变化的标志操作。
在转换中会出现标志操作符。

符号	说明
Blank	不变
0	清 0
1	置 1
x	根据结果进行设置或清除
R	之前存储的值被恢复

[说明]

详细描述指令操作

[说明示例]

指令的说明示例

(1) 8 位数 据传送指令。

以下的 8 位传送指令可用。

指令	概要
MOV	字节数据传送
XCH	字节数据交换
ONEB	设置字节数据为 01H
CLRB	清除字节数据
MOVS	传送字节数据且改变 PSW

MOV

字节数据传送

[指令格式]

MOV dst, src

[运算符]

dst <- src

[操作数]

操作数 (dst, src)
r, #byte
saddr, #byte
sfr, #byte
!addr16, #byte
A, r 注
r, A 注
A, saddr
saddr, A
A, sfr
sfr, A
A, !addr16
!addr16, A
PSW, #byte
A, PSW
PSW, A
ES, #byte
ES, saddr
A, ES
ES, A
CS, #byte
A, CS
CS, A
A, [DE]
[DE], A
[DE+byte], #byte
A, [DE+byte]
[DE+byte], A
A, [HL]
[HL], A

操作数 (dst, src)
[HL+byte], #byte
A, [HL+byte]
[HL+byte], A
A, [HL+B]
[HL+B], A
A, [HL+C]
[HL+C], A
word[B], #byte
A, word[B]
word[B], A
word[C], #byte
A, word[C]
word[C], A
word[BC], #byte
A, word[BC]
word[BC], A
[SP+byte], #byte
A, [SP+byte]
[SP+byte], A
B, saddr
B, !addr16
C, saddr
C, !addr16
X, saddr
X, !addr16
ES:!addr16, #byte
A, ES:!addr16
ES:!addr16, A
A, ES:[DE]
ES:[DE], A
ES:[DE+byte], #byte
A, ES:[DE+byte]
ES:[DE+byte], A
A, ES:[HL]
ES:[HL], A
ES:[HL+byte], #byte
A, ES:[HL+byte]
ES:[HL+byte], A
A, ES:[HL+B]

操作数 (dst, src)
ES:[HL+B], A
A, ES:[HL+C]
ES:[HL+C], A
ES:word[B], #byte
A, ES:word[B]
ES:word[B], A
ES:word[C], #byte
A, ES:word[C]
ES:word[C], A
ES:word[BC], #byte
A, ES:word[BC]
ES:word[BC], A
B, ES:!addr16
C, ES:!addr16
X, ES:!addr16

注 Except r = A.

[标记]

(1) PSW, #byte 和 PSW, A 操作数

Z	AC	CY
x	x	x

x : 根据结果设置或清除

(2) 所有其他操作数组合

Z	AC	CY

Blank : 不变

[说明]

- 在第 2 个操作数中设定的源操作数 (src) 内容传送到第 1 个操作数中设定的目标操作数 (dst) 中。
- 在 MOV PSW 之间，不能响应中断，#byte 指令 /MOV PSW,A 指令和下条指令。

[说明举例]

```
MOV     A, #4DH         ;  (1)
```

(1) 传送 4DH 到 A 寄存器。

XCH

字节数据交换

[指令格式]

XCH dst, src

[运算符]

dst <--> src

[操作数]

操作数 (dst, src)
A, r 注
A, saddr
A, sfr
A, !addr16
A, [DE]
A, [DE+byte]
A, [HL]
A, [HL + byte]
A, [HL + B]
A, [HL + C]
A, ES:!addr16
A, ES:[DE]
A, ES:[DE+byte]
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 交换第 1 个操作数与第 2 个操作数的内容。

[说明举例]

```
XCH     A, FFEBCH       ; (1)
```

(1) 交换 A 寄存器和地址 FFEBCH 中的内容。

ONEB

设置字节数据为 01H

[指令格式]

ONEB dst

[运算符]

dst <- 01H

[操作数]

操作数 (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[标志]

Z	AC	CY

Blank ： 不变

[说明]

- 传送 01H 到第 1 个操作数中设定的目标操作数 (dst) 中。

[说明示例]

```
ONEB    A       ; (1)
```

(1) 传送 01H 到 A 寄存器。

CLRB

清除字节数据

[指令格式]

CLRB dst

[运算符]

dst <- 00H

[操作符]

操作数 (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 传送 00H 到第 1 个操作数中设定的目标操作数 (dst) 中。

[说明示例]

```
CLRB    A       ; (1)
```

(1) 传送 00H 到 A 寄存器。

MOVS

传送字节数据且改变 PSW

[指令格式]

MOVS dst, src

[运算符]

dst <- src

[操作数]

操作数 (dst, src)
[HL+byte], X
ES:[HL+byte], X

[标记]

Z	AC	CY
x		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 2 个操作数中设定的源操作数内容传送到第 1 个操作数中设定的目标操作数 (dst) 中。
- 如果 src 的值为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果寄存器 A 的值为 0 或如果 src 的值为 0， CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。

[说明示例]

```
MOVS    [HL+2H], X      ; (1)
```

(1) 当 HL = FE00H, X = 55H, A = 0H : 搠 = 55H?，并存储在地址 FE02H 上。
Z 标志 = 0 CY 标志 = 1 (由于 A 寄存器 = 0)

(2) 16 位数据传送指令

以下的 16 位传送指令可用。

指令	概要
MOVW	字数据传送
XCHW	字数据交换
ONEW	设置字数据 0001H
CLRW	清除字数据

MOVW

字数据传送

[指令格式]

MOVW dst, src

[运算符]

dst <- src

[操作数]

操作数 (dst, src)
rp, #word
saddrp, #word
sfrp, #word
AX, saddrp
saddrp, AX
AX, sfrp
sfrp, AX
AX, rp 注
rp, AX 注
AX, !addr16
!addr16, AX
AX, [DE]
[DE], AX
AX, [DE+byte]
[DE+byte], AX
AX, [HL]
[HL], AX
AX, [HL+byte]
[HL+byte], AX
AX, word[B]
word[B], AX
AX, word[C]
word[C], AX
AX, word[BC]
word[BC], AX
AX, [SP+byte]
[SP+byte], AX
BC, saddrp
BC, !addr16

操作数 (dst, src)
DE, saddrp
DE, !addr16
HL, saddrp
HL, !addr16
AX, ES:!addr16
ES:!addr16, AX
AX, ES:[DE]
ES:[DE], AX
AX, ES:[DE+byte]
ES:[DE+byte], AX
AX, ES:[HL]
ES:[HL], AX
AX, ES:[HL+byte]
ES:[HL+byte], AX
AX, ES:word[B]
ES:word[B], AX
AX, ES:word[C]
ES:word[C], AX
AX, ES:word[BC]
ES:word[BC], AX
BC, ES:!addr16
DE, ES:!addr16
HL, ES:!addr16

注 只当 rp = BC, DE 或 HL 时

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 在第 2 个操作数中设定的源操作数 (src) 内容传送到第 1 个操作数中设定的目标操作数 (dst) 中。

[说明示例]

```
MOVW    AX, HL          ;  (1)
```

(1) 传送 HL 寄存器中的内容到 AX 寄存器中。

[注意事项]

- 只能够指定偶数地址。能够指定奇数地址。

XCHW

字数据交换

[指令格式]

XCHW dst, src

[运算符]

dst <--> src

[操作数]

操作数 (dst, src)
AX, rp 注

注 只当 rp = BC, DE 或 HL 时

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 交换第 1 个操作数与第 2 个操作数的内容。

[说明示例]

```
XCHW    AX, BC          ;  (1)
```

(1) AX 寄存器中的存储内容与 BC 寄存器中的存储内容相交换。

ONEW

设置字数据 0001H

[指令格式]

ONEW dst

[运算符]

dst <- 0001H

[操作数]

操作数 (dst)
AX
BC

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 传送 0001H 到第 1 个操作数中设定的目标操作数 (dst) 中。

[说明示例]

```
ONEW    AX      ; (1)
```

(1) 传送 0001H 到 AX 寄存器中。

CLRW

清除字数据

[指令格式]

CLRW dst

[运算符]

dst <- 0000H

[操作数]

操作数 (dst)
AX
BC

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 传送 0000H 到第 1 个操作数中设定的目标操作数中。

[说明示例]

```
CLRW    AX      ; (1)
```

(1) 传送 0000H 到 AX 寄存器中。

(3) 8- 位 运算指令

以下的 8 位运算指令可用。

指令	概要
ADD	字节数据相加
ADDC	带进位的字节数据相加
SUB	字节数据相减
SUBC	带进位的字节数据相减
AND	字节数据 AND 运算
OR	字节数据 OR 运算
XOR	字节数据异 OR 运算
CMP	字节数据的比较
CMP0	字节数据零的比较
CMPS	字节数据的比较

ADD

字节数据相加

[指令格式]

ADD dst, src

[运算符]

dst, CY <- dst + src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相加，且结果保存在 CY 标志和目标操作数中 （dst)。
- 如果加法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果加法产生的进位超过第 7 位，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果加法产生了第 4 位的进位超出了第 3 位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明示例]

```
ADD     CR10, #56H      ;  (1)
```

(1) CR10 寄存器与 56H 相加的结果保存在 CR10 寄存器中。

ADDC

带进位的字节数据相加

[指令格式]

ADDC dst, src

[运算符]

dst, CY <- dst + src + CY

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 以及 CY 标志相加，且结果保存在 CY 标志和目标操作数中 （dst)。
  CY 标志与最低有效位相加。这条指令主要用于两个或更多字节的相加。
- 如果加法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果加法产生的进位超过第 7 位，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果加法产生了第 4 位的进位超出了第 3 位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明举例]

```
ADDC    A, [HL+B]       ;  (1)
```

(1) A 寄存器内容和地址 (HL 寄存器 + (B 寄存器)) 上的内容以及 CY 标志相加，其结果保存在 A 寄存器内。

SUB

字节数据相减

[指令格式]

SUB dst, src

[运算符]

dst, CY <- dst - src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 第 2 个操作数中设定的源操作数 （src) 与在第 1 个操作数中设定的目标操作数 (dst) 相减，且结果保存在目标操作数 （dst) 及 CY 标志中。
  由于源操作数 (src) 与目标操作数 （dst) 相等，目标操作数清除为 0。
- 如果减法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果减法生成 7 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果减法产生超出了第 4 位 - 第 3 位的借位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明举例]

```
SUB     D, A            ;  (1)
```

(1) D 寄存器减去 A 寄存器，其结果保存在 D 寄存器中。

SUBC

带进位的字节数据相减

[指令格式]

SUBC dst, src

[运算符]

dst, CY <- dst - src - CY

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 减去第 2 个操作数中设定的源操作数 （src) 和 CY 标志，且结果保存在目标操作数 （dst) 中。
  最低有效位减去 CY 标志。这条指令主要用于两个或更多字节的相减。
- 如果减法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果减法生成 7 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果减法产生超出了第 4 位 - 第 3 位的借位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明举例]

```
SUBC    A, [HL]          ;  (1)
```

(1) A 寄存器减去 (HL 寄存器) 地址内容和 CY 标志，其结果保存在 A 寄存器内。

AND

字节数据 AND 运算

[指令格式]

AND dst, src

[运算符]

dst <- dst ^ src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x		

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相乘的位元逻辑乘积运算，其结果保存在目标操作数中 （dst)。
- 如果逻辑乘积结果显示所有位都为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。

[说明举例]

```
AND     FFEBAH, #11011100B      ; (1)
```

(1) FFEBAH 的内容与 11011100B 进行的位元逻辑乘积运算，其结果保存在 FFEBAH。

OR

字节数据 OR 运算

[指令格式]

OR dst, src

[运算符]

dst <- dst v src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x		

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相乘的位元逻辑和运算，其结果保存在目标操作数中 （dst)。
- 如果逻辑和结果显示所有位都为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。

[说明举例]

```
OR      A, FFE98H       ; (1)
```

(1) A 寄存器和 FFE98H 进行的位元逻辑和运算，其结果保存在 A 寄存器内。

XOR

字节数据异 OR 运算

[指令格式]

XOR dst, src

[运算符]

dst <- dst ∀src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 Except r = A.

[标记]

Z	AC	CY
x		

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相乘的位元逻辑异和运算，其结果保存在目标操作数中 （dst)。
  对目标操作数 （dst) 所有位进行逻辑非运算，可能使用这条指令通过选择源操作数为 #0FFH。
- 如果逻辑异和结果显示所有位都为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。

[说明举例]

```
XOR     A, L            ;  (1)
```

(1) A 和 L 寄存器进行的位元逻辑异和运算，其结果保存在 A 寄存器内。

CMP

字节数据的比较

[指令格式]

CMP dst, src

[运算符]

dst - src

[操作数]

操作数 (dst, src)
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
!addr16, #byte
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]
ES:!addr16, #byte

注 Except r = A.

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 在目标操作数 (dst) 中设定的第 1 个操作数减去在源操作数 (src) 中设定的第 2 个操作数。
  相减的结果不能任意保存且只能改变 Z, AC 和 CY 标志。
- 如果减法结果为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果减法生成 7 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果减法产生超出了第 4 位 - 第 3 位的借位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明举例]

```
CMP     FFE38H, #38H            ; (1)
```

(1) 在地址 FFE38H 上的内容减去 38H，且只有标志发生改变。
(地址 FFE38H 上的内容与直接数据相比较)。

CMP0

字节数据零的比较

[指令格式]

CMP0 dst

[运算符]

dst - 00H

[操作数]

操作数 (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 减去 00H。
- 相减的结果不能任意保存且只有 Z, AC 和 CY 标志发生改变。
- 如果 dst 的值已为 00H，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- AC 和 CY 标志总是清零 （0）。

[说明示例]

```
CMP0    A               ; (1)
```

(1) 如果 A 寄存器值为 0，则设置 Z 标志。

CMPS

字节数据的比较

[指令格式]

CMPS dst, src

[运算符]

dst - src

[操作数]

操作数 (dst, src)
X, [HL+byte]
X, ES:[HL+byte]

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 在目标操作数 (dst) 中设定的第 1 个操作数减去在源操作数 (src) 中设定的第 2 个操作数。
  相减的结果不能任意保存且只能改变 Z, AC 和 CY 标志。
- 如果减法结果为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 当计算结果不为 0 或当无论寄存器 A 还是 dst 的值为 0，则设置 CY 标志为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 如果减法产生超出了第 4 位 - 第 3 位的借位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。

[说明示例]

```
CMPS    X, [HL+F0H]     ; (1)
```

(1) 当 HL = FD12H : X 的值与地址 FFE02H 的内容相比较 , 如果两个值匹配则设置 Z 标志。 X 的值与地址 FFE02H 的内容相比较 , 如果两个值不匹配则设置 CY 标志。
当寄存器 A 的值为 0 时，设置 CY 标志。当寄存器 X 的值为 0 时，设置 CY 标志。从第 4 位到第 3 位借位设置 AC 标志，与 CMP 指令类似。

(4) 16- 位 运算指令

以下的 16 位运算指令可用。

指令	概要
ADDW	字数据的加法
SUBW	字数据的减法
CMPW	字数据的比较

ADDW

字数据的加法

[指令格式]

ADDW dst, src

[运算符]

dst, CY <- dst + src

[操作数]

操作数 (dst, src)
AX, #word
AX, AX
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 在目标操作数 (dst) 中指定的第 1 个操作数与源操作数 （src) 中指定的第 2 个操作数相加，且结果保存在目标操作数中 （dst)。
- 如果加法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清除 （0）。
- 如果加法生成 15 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 根据加法的结果， AC 标志为未定义。

[说明示例]

```
ADDW    AX, #ABCDH      ;  (1)
```

(1) AX 寄存器的内容加上 ABCDH，且结果保存在 AX 寄存器中。

SUBW

字数据的减法

[指令格式]

SUBW dst, src

[运算符]

dst, CY <- dst - src

[操作数]

操作数 (dst, src)
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 源操作数 （src) 中指定的第 2 个操作数与在目标操作数 (dst) 中指定的第 1 个操作数相减，且结果保存在目标操作数 （dst) 及 CY 标志中。
- 如果减法结果显示 dst 为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果减法生成 15 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 根据减法的结果， AC 标志为未定义。

[说明示例]

```
SUBW    AX, #ABCDH      ; (1)
```

(1) AX 寄存器的内容减去 ABCDH，且结果保存在 AX 寄存器中。

CMPW

字数据的比较

[指令格式]

CMPW dst, src

[运算符]

dst - src

[操作数]

操作数 (dst, src)
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[标记]

Z	AC	CY
x	x	x

x : 根据结果设置或清除

[说明]

- 在目标操作数 (dst) 中设定的第 1 个操作数减去在源操作数 (src) 中设定的第 2 个操作数。
  相减的结果不能任意保存且只能改变 Z, AC 和 CY 标志。
- 如果减法结果为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果减法生成 15 位的运行结果，则 CY 标志设置为 （1）。在其他所有情况下， CY 标志已清除 （0）。
- 根据减法的结果， AC 标志为未定义。

[说明示例]

```
CMPW    AX, #ABCDH              ;  (1)
```

(1) AX 寄存器中的内容减去 ABCDH，且只有标志发生变化。
(比较 AX 寄存器和中间数据)

(5) 乘法指令

以下乘法指令可用。

指令	概要
MULU	无符号数据的乘法。

MULU

无符号数据的乘法。

[指令格式]

MULU src

[运算符]

AX <- A x src

[操作数]

操作数 (src)
X

[标记]

Z	AC	CY

Blank： 不变

[说明]

- A 寄存器内容和源操作数 （src) 数据以无符号数据相乘，并存储结果在 AX 寄存器中。

[说明举例]

```
MULU    X       ;  (1)
```

(1) A 寄存器内容和 X 寄存器内容相乘，并存储结果在 AX 寄存器中。

≥

(6) 递增 / 递减指令

以下递增 / 递减指令可用。

指令	概要
INC	字节数据递增
DEC	字节数据递减
INCW	字数据递增
DECW	字数据递减

INC

字节数据递增

[指令格式]

INC dst

[运算符]

dst <- dst + 1

[操作数]

操作数 (src)
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES: [HL+byte]

[标记]

Z	AC	CY
x	x	

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 目标操作数 (dst) 逐个递增。
- 如果递增结果为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果递增产生的运行结果由 3 位变 4 位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。
- 因为这条指令反复进行计数器和变址寻址偏移寄存器的递增 , CY 标志内容不变 (在多字节操作中保持 CY 标志的内容)。

[说明示例]

```
INC     B               ;  (1)
```

(1) B 寄存器递增。

DEC

字节数据递减

[指令格式]

DEC dst

[运算符]

dst <- dst - 1

[操作数]

操作数 (src)
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES: [HL+byte]

[标记]

Z	AC	CY
x	x	

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 目标操作数 (dst) 逐个递减。
- 如果递减结果为 0，标志 Z 设置为 （1）。在其他所有情况下， Z 标志已清零 （0）。
- 如果递减产生的运行结果由 4 位变 3 位，则 AC 标志设置为 （1）。在其他所有情况下， AC 标志已清零 （0）。
- 因为这条指令不断反复进行计数器的计数 , CY 标志不变 (在多字节操作中保持 CY 标志的内容)。
- 如果 dst 是 B 或 C 寄存器或 saddr，并且不希望改变 AC 和 CY 标志内容，可以使用 DBNZ 指令。

[说明示例]

```
DEC     FFE92H          ; (1)
```

(1) 在地址 FFE92H 上的内容递减。

INCW

字数据递增

[指令格式]

INCW dst

[运算符]

dst <- dst + 1

[操作数]

操作数 (src)
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES: [HL+byte]

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 目标操作数 (dst) 逐个递增。
- 因为这条指令对寻址寄存器 （指针）不断递增， Z， AC 和 CY 标志内容不变。

[说明示例]

```
INCW    HL              ;  (1)
```

(1) HL 寄存器递增。

DECW

字数据递减

[指令格式]

DECW dst

[运算符]

dst <- dst - 1

[操作数]

操作数 (src)
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES: [HL+byte]

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 目标操作数 (dst) 逐个递减。
- 因为这条指令对寻址寄存器 （指针）不断递减， Z， AC 和 CY 标志内容不变。

[说明示例]

```
DECW    DE              ;  (1)
```

(1) DE 寄存器递减。

(7) 移位指令

以下移位指令可用。

指令	概要
SHR	逻辑右移
SHRW	逻辑右移
SHL	逻辑左移
SHLW	逻辑左移
SAR	算术右移
SARW	算术右移

SHR

逻辑右移

[指令格式]

SHR dst, cnt

[运算符]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_7 <- 0) x cnt

[操作数]

操作数 (dst, cnt)
A, cnt

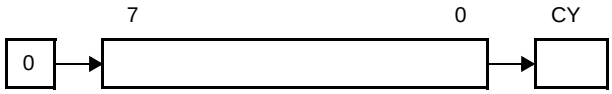
[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行右移位。
- 在 MSB(位 7）上输入的 ?
- cnt 能设定为任何从 1 到 7 的值。



[说明示例]

```
SHR     A, 3    ; (1)
```

(1) 当 A 寄存器的值为 F5H 时， A=1EH 且 CY=1。

```
A = F5H  CY = 0
A = 7AH  CY = 1  1 time
A = 3DH  CY = 0  2 times
A = 1EH  CY = 1  3 times
```

SHRW

逻辑右移

[指令格式]

SHRW dst, cnt

[运算符]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_{15} <- 0) x cnt

[操作数]

操作数 (dst, cnt)
AX, cnt

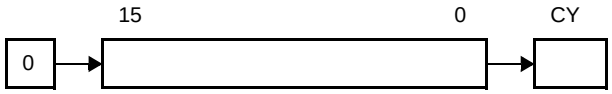
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行右移位。
- 在 MSB(位 15）上输入的 ?
- cnt 能设定为任何从 1 到 15 的值。



[说明示例]

```
SHRW    AX, 3   ; (1)
```

(1) 当 AX 寄存器的值为 AAF5H 时， AX=155EH 且 CY=1。

```
AX = AAF5H  CY = 0
AX = 557AH  CY = 1  1 time
AX = 2ABDH  CY = 0  2 times
AX = 155EH  CY = 1  3 times
```

SHL

逻辑左移

[指令格式]

SHL dst, cnt

[运算符]

(CY <- dst_7, dst_m <- dst_{m-1}, dst_0 <- 0) x cnt

[操作数]

操作数 (dst, cnt)
A, cnt
B, cnt
C, cnt

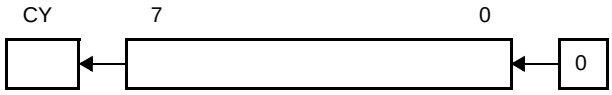
[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行左移位。
- 在 LSB(位 0） 上输入的 ?
- cnt 能设定为任何从 1 到 7 的值。



[说明示例]

```
SHL     A, 3    ; (1)
```

(1) 当 A 寄存器的值为 5DH 时， A=E8H 且 CY=0。

```
A = 5DH  CY = 0
A = BAH  CY = 0  1 time
A = 74H  CY = 1  2 times
A = E8H  CY = 0  3 times
```

SHLW

逻辑左移

[指令格式]

SHLW dst, cnt

[运算符]

(CY <- dst_{15}, dst_m <- dst_{m-1}, dst_0 <- 0) x cnt

[操作数]

操作数 (dst, cnt)
AX, cnt
BC, cnt

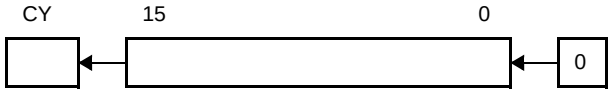
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行左移位。
- 在 LSB(位 0）上输入的 ?
- cnt 能设定为任何从 1 到 15 的值。



[说明示例]

```
SHLW    BC, 3   ; (1)
```

(1) 当 BC 寄存器的值为 C35DH 时， A=1AE8H 且 CY=0。

```
BC = C35DH  CY = 0
BC = 86BAH  CY = 1  1 time
BC = 0D74H  CY = 1  2 times
BC = 1AE8H  CY = 0  3 times
```

SAR

算术右移

[指令格式]

SAR dst, cnt

[运算符]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_7 <- dst_7) x cnt

[操作数]

操作数 (dst, cnt)
A, cnt

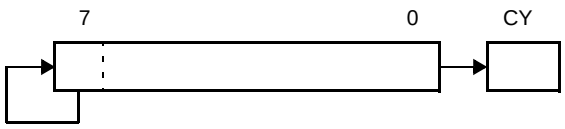
[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行右移位。
- 在 MSB(位 7）上保留相同的值，并持续值的移位从输入的 0 位到 CY。
- cnt 能设定为任何从 1 到 7 的值。



[说明示例]

```
SAR     A, 4    ; (1)
```

(1) 当 A 寄存器的值为 8CH 时， A=F8H 且 CY=1。

```
A = 8CH  CY = 0
A = C6H  CY = 0  1 time
A = E3H  CY = 0  2 times
A = F1H  CY = 1  3 times
A = F8H  CY = 1  4 times
```

SARW

算术右移

[指令格式]

SARW dst, cnt

[运算符]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_{15} <- dst_{15}) x cnt

[操作数]

操作数 (dst, cnt)
AX, cnt

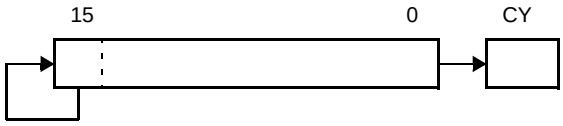
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 操作数 （dst) 中设定的目标操作数按照 cnt 中设定次数进行右移位。
- 在 MSB(位 15）上保留相同的值，并持续值的移位从输入的 0 位到 CY。
- cnt 能设定为任何从 1 到 15 的值。



[说明示例]

```
SAR     AX, 4   ; (1)
```

(1) 当 AX 寄存器的值为 A28CH 时， AX=FA28H 且 CY=1。

```
AX = A28CH  CY = 0
AX = D146H  CY = 0  1 time
AX = E8A3H  CY = 0  2 times
AX = F451H  CY = 1  3 times
AX = FA28H  CY = 1  4 times
```

(8) 循环移位指令

以下循环移位指令可用。

指令	概要
ROR	字节数据向右循环移位
ROL	字节数据向左循环移位
RORC	字节数据进位右循环移位
ROLC	字节数据进位左循环移位
ROLWC	字数据进位左循环移位

ROR

字节数据向右循环移位

[指令格式]

ROR dst, cnt

[运算符]

(CY, dst_7 <- dst_0, dst_{m-1} <- dst_m) x one time

[操作数]

操作数 (dst, cnt)
A, 1

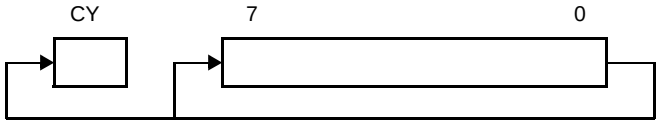
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 内容仅向右循环移位一次。
- 同时循环移位从 LSB(0 位）到 MSB(7 位）的内容并传送到 CY 标志内。



[说明示例]

```
ROR     A, 1            ;  (1)
```

(1) A 寄存器内容向右循环一位。

ROL

字节数据向左循环移位

[指令格式]

ROL dst, cnt

[运算符]

(CY, dst_0 <- dst_7, dst_{m+1} <- dst_m) x one time

[操作数]

操作数 (dst, cnt)
A, 1

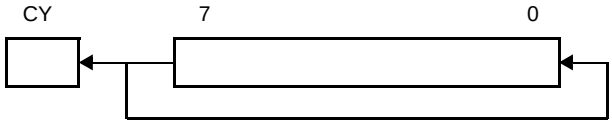
[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 内容仅向左循环移位一次。
- 同时循环移位从 MSB(7 位）到 LSB(0 位）的内容并传送到 CY 标志内。



[说明示例]

```
ROL     A, 1            ;  (1)
```

(1) A 寄存器内容向左循环一位。

RORC

字节数据进位右循环移位

[指令格式]

RORC dst, cnt

[运算符]

(CY <- dst_0, dst_7 <- CY, dst_{m-1} <- dst_m) x one time

[操作数]

操作数 (dst, cnt)
A, 1

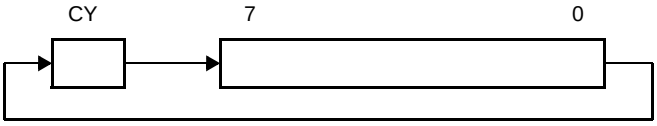
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 内容仅向右循环移位一次。



[说明示例]

```
RORC    A, 1            ;  (1)
```

(1) A 寄存器内容向右循环一位，包括 CY 标志。

ROLC

字节数据进位左循环移位

[指令格式]

ROLC dst, cnt

[运算符]

(CY <- dst_7, dst_0 <- CY, dst_{m+1} <- dst_m) x one time

[操作数]

操作数 (dst, cnt)
A, 1

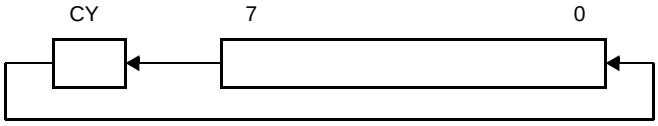
[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 内容仅向左循环移位一次带进位标志。



[说明示例]

```
ROLC    A, 1            ;  (1)
```

(1) A 寄存器内容向左循环一位，包括 CY 标志。

ROLWC

字数据进位左循环移位

[指令格式]

ROLWC dst, cnt

[运算符]

(CY <- dst_{15}, dst_0 <- CY, dst_{m+1} <- dst_m) x one time

[操作数]

操作数 (dst, cnt)
AX, 1
BC, 1

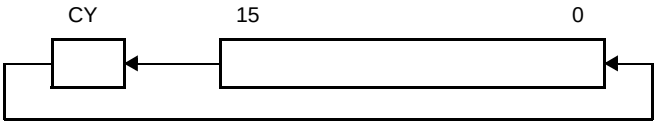
[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 在第 1 个操作数中设定的目标操作数 (dst) 内容仅向左循环移位一次带进位标志。



[说明示例]

```
ROLWC   BC, 1           ; (1)
```

(1) BC 寄存器内容向左循环一位，包括 CY 标志。

(9) 位操作指令

以下位操作指令可用。

指令	概要
MOV1	1 位数据传送
AND1	1 位数据的 AND 运算
OR1	1 位数据的 OR 运算
XOR1	1 位数据的异 OR 运算
SET1	1 位数据的设置
CLR1	1 位数据的清除
NOT1	1 位数据的逻辑非

MOV1

1 位数据的传送

[指令格式]

MOV1 dst, src

[运算符]

dst <- src

[操作数]

操作数 (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
saddr.bit, CY
sfr.bit, CY
A.bit, CY
PSW.bit, CY
[HL].bit, CY
CY, ES:[HL].bit
ES:[HL].bit, CY

[标记]

(1) dst = CY

Z	AC	CY
		x

Blank : 不变
x : 根据结果设置或清除

(2) dst = PSW.bit

Z	AC	CY
x	x	

Blank : 不变
x : 根据结果设置或清除

(3) 所有其他操作数组合

Z	AC	CY

Blank : 不变

[说明]

- 在第 2 个操作数中设定为位数据的源操作数 (src) 传送到第 1 个操作数中设定的目标操作数 (dst) 中。
- 当目标操作数 (dst) 为 CY 或 PSW.bit, 只有相应的标志会改变。

[说明示例]

```
MOV1    P3.4, CY    ;  (1)
```

(1) CY 标志内容传送到端口 3 的第 4 位上。

AND1

1 位数据的 AND 运算

[指令格式]

AND1 dst, src

[运算符]

dst <- dst ^ src

[操作数]

操作数 (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[标记]

Z	AC	CY
		x

Blank： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相乘的位数据逻辑乘积，其结果保存在目标操作数中 （dst)。
- 运算结果保存在 CY 标志中 (由于目标操作数 (dst))。

[说明示例]

```
AND1  CY, FFE7FH.3  ; (1)
```

(1) FFE7FH 中的第 3 位和 CY 标志的逻辑乘积，其结果保存在 CY 标志中。

OR1

1 位数据的 OR 运算

[指令格式]

OR1 dst, src

[运算符]

dst <- dst v src

[操作数]

操作数 (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相加的位数据逻辑和，其结果保存在目标操作数中 （dst)。
- 运算结果保存在 CY 标志中 (由于目标操作数 (dst))。

[说明示例]

```
OR1     CY, P2.5    ;  (1)
```

(1) FFE7FH 中的第 2 位和 CY 标志相加的逻辑和，其结果保存在 CY 标志中。

XOR1

1 位数据的异 OR 运算

[指令格式]

XOR1 dst, src

[运算符]

dst <- dst ∀src

[操作数]

操作数 (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- 第 1 个操作数中设定的目标操作数 (dst) 与第 2 个操作数中设定的源操作数 （src) 相加的位数据逻辑异和，其结果保存在目标操作数中 （dst)。
- 运算结果保存在 CY 标志中 (由于目标操作数 (dst))。

[说明示例]

```
XOR1  CY, A.7 ;  (1)
```

(1) FFE7FH 中的第 7 位和 CY 标志相加的逻辑异和，其结果保存在 CY 标志中。

SET1

1 位数据的设置

[指令格式]

SET1 dst

[运算符]

dst <- 1

[操作数]

操作数 (dst)
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[标记]

(1) dst = PSW.bit

Z	AC	CY
x	x	x

x : 根据结果设置或清除

(2) dst = CY

Z	AC	CY
		1

Blank : 不变
1 : 设置为 1

(3) 所有其他操作数组合

Z	AC	CY

Blank : 不变

[说明]

- 目标操作数 (dst) 设置为 (1)。
- 当目标操作数 (dst) 为 CY 或 PSW.bit, 只有相应的标志设置为 （1）。

[说明示例]

```
SET1    FFE55H.1        ; (1)
```

(1) FFE55H 地址的第 1 位设置为 (1)。

CLR1

1 位数据的清除

[指令格式]

CLR1 dst

[运算符]

dst <- 0

[操作数]

操作数 (dst)
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[标记]

(1) dst = PSW.bit

Z	AC	CY
x	x	x

x : 根据结果设置或清除

(2) dst = CY

Z	AC	CY
		0

Blank : 不变
0 : 清除为 0

(3) 所有其他操作数组合

Z	AC	CY

Blank : 不变

[说明]

- 目标操作数 (dst) 清零 (0)。
- 当目标操作数 (dst) 为 CY 或 PSW.bit, 只有相应的标志清零 （0）。

[说明示例]

```
CLR1    P3.7            ;  (1)
```

(1) 端口 3 的第 7 位清零 （0）。

NOT1

1 位数据的逻辑非

[指令格式]

NOT1 dst

[运算符]

dst <- $\overline{\text{dst}}$

[操作数]

操作数 (dst)
CY

[标记]

Z	AC	CY
		x

Blank ： 不变
x ： 根据结果设置或清除

[说明]

- CY 标志反向。

[说明示例]

```
NOT1    CY      ;  (1)
```

(1) CY 标志反向。

(10) 调用返回指令

以下调用返回指令可用。

指令	概要
CALL	子程序调用
CALLT	子程序调用 （引用调用表）
BRK	软件向量中断
RET	从子程序返回
RETI	从硬件向量中断返回
RETB	从软件中断返回

CALL

子程序调用

[指令格式]

CALL 目标

[运算符]

(SP - 2) <- (PC + n)S,
(SP - 3) <- (PC + n)H,
(SP - 4) <- (PC + n)L,
SP <- SP - 4,
PC <- 目标

备注 当使用 !!addr20 时，n 为 4。当使用 !addr16 或 $!addr20 时，n 为 3。当使用 AX,BC,DE, 或 HL 时，n 为 2。

[操作数]

操作数 (目标)
AX
BC
DE
HL
$!addr20
!addr16
!!addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这是使用 20/16- 位绝对地址或寄存器重定位地址的子程序调用。
- 下条指令的起始地址 (PC+n) 保存在堆栈中并转移到目标操作数中设定的地址 （目标）。

[说明举例]

```
CALL    !!3E000H        ; (1)
```

(1) 子程序调用到 3E000H。

CALLT

子程序调用 （引用调用表）

[指令格式]

CALLT [addr5]

[运算符]

(SP - 2) <- (PC + 2)$_S$,
(SP - 3) <- (PC + 2)$_H$,
(SP - 4) <- (PC + 2)$_L$,
PC$_S$ <- 0000,
PC$_H$ <- (000000000000, addr5 + 1),
PC$_L$ <- (000000000000, addr5),
SP <- SP - 4

[操作数]

操作数 ([addr5])
[addr5]

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 这是引用调用表的子程序调用。
- 下条指令的起始地址 (PC+2) 保存在堆栈中并转移到显示调用表字数据的地址上 (放置在 000000000000B 地址的高 12 位 , 以及显示的 8 位 addr5 上的低 5 位)。

[说明示例]

```
CALLT   [80H]           ; (1)
```

(1) 子程序调用子数据地址 00080H 和 00081H。

[备注]

- 只能设定偶数地址 （不能设定奇数地址）。
  addr5: 从 00080H 到 000BEH 的直接数据或标签 (只限偶数地址)

BRK

软件向量中断

[指令格式]

BRK

[运算符]

(SP - 1) <- PSW,
(SP - 2) <- (PC + 2)$_S$,
(SP - 3) <- (PC + 2)$_H$,
(SP - 4) <- (PC + 2)$_L$,
PC$_S$ <- 0000,
PC$_H$ <- (0007FH),
PC$_L$ <- (0007EH),
SP <- SP - 4,
IE<- 0

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这是软件中断指令。
- PSW 和下条指令地址 (PC+2) 保存到堆栈中。在那以后， IE 标志清零且保存的数据转移到向量地址的字数据地址上 (0007EH, 0007FH)。
  因为 IE 标志已清零 (0), 则禁止接下可屏蔽向量中断的使用。
- RETB 指令用于从此指令生成的软件向量中断中返回。

RET

从子程序返回

[指令格式]

RET

[运算符]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
SP <- SP + 4,

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这是条从 CALL 产生的子程序调用和 CALLT 指令中的返回指令。
- 保存在堆栈中的字数据返回到 PC, 且程序从子程序返回。

RETI

从硬件向量中断返回

[指令格式]

RETI

[运算符]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
PSW <- (SP + 3),
SP <- SP + 4

[操作数]

None

[标记]

Z	AC	CY
R	R	R

R : 恢复以前保存的值

[说明]

- 这是向量中断中的返回指令。
- 保存在堆栈中的数据返回到 PC 和 PSW，且从中断服务程序中返回程序。
- 这条指令不能用于从使用 BRK 指令的软件中断中返回。
- 在这条指令和下条指令执行期间，无中断应答。
- 由不可屏蔽的中断接受设置 NMIS 标志为 1，并由 RETI 指令清零。

[注意事项]

- 当由执行除 RETI 指令以外的指令后并从不可屏蔽中断返回时，因为 NMIS 标志没有清零，任何中断 （包括不可屏蔽中断）不能接受。

RETB

从软件中断返回

[指令格式]

RETB

[运算符]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
PSW <- (SP + 3),
SP <- SP + 4

[操作数]

None

[标记]

Z	AC	CY
R	R	R

R : 恢复以前保存的值

[说明]

- 这是从使用 BRK 指令产生的软件中断中返回的指令。
- 保存在堆栈中的数据返回到 PC 和 PSW，且从中断服务程序中返回程序。
- 在这条指令和下条指令执行期间，无中断应答。

(11) 堆栈操作指令

以下堆栈操作指令可用。

指令	概要
PUSH	压入
POP	弹出
MOVW	传送 Stck 指针和字数据
ADDW	堆栈指针加法
SUBW	堆栈指针减法

PUSH

压入

[指令格式]

PUSH src

[运算符]

(1) src = rp

(SP - 1) <- rp_H,
(SP - 2) <- rp_L,
SP <- SP - 2

(2) src = PSW

(SP - 1) <- PSW,
(SP - 2) <- 00H,
SP <- SP - 2

[操作数]

操作数 (src)
PSW
rp

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 设定保存在堆栈中的源操作数 （src) 为寄存器数据。

[说明示例]

```
PUSH    AX              ;  (1)
```

(1) AX 寄存器内容保存在堆栈中。

POP

弹出

[指令格式]

POP dst

[运算符]

(1) dst = rp

rp_L <- (SP),
rp_H <- (SP + 1),
SP <- SP + 2

(2) dst = PSW

PSW <- (SP + 1),
SP <- SP + 2

[操作数]

操作数 (dst)
PSW
rp

[标记]

(1) dst = rp

Z	AC	CY

Blank ： 不变

(2) dst = PSW

Z	AC	CY
R	R	R

R ： 恢复以前保存的值

[说明]

- 设定寄存器中从堆栈返回的数据为目标操作数 （dst)。
- 当操作数为 PSW，用堆栈数据替换每个标志。
- 在 POP PSW 指令和随后的指令间没有中断响应。

[说明示例]

```
POP     AX              ;  (1)
```

(1) 返回堆栈数据到 AX 寄存器。

MOVW

传送 Stck 指针和字数据

[指令格式]

MOVW dst, src

[运算符]

dst <- src

[操作数]

操作数 (dst, src)
SP, #word
SP, AX
AX, SP
HL, SP
BC, SP
DE, SP

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这是对堆栈指针内容进行操作的指令。
- 在源操作数 (src) 中设定的第 2 个操作数保存在目标操作数 (dst) 中设定的第 1 个操作数中。

[说明示例]

```
MOVW    SP, #FE1FH              ;  (1)
```

(1) FE1FH 存储在堆栈指针中。

ADDW

堆栈指针加法

[指令格式]

ADDW SP, src

[运算符]

SP <- SP + src

[操作数]

操作数 (SP, src)
SP, #byte

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 堆栈指针设定的第 1 个操作数与源操作数 (src) 中设定的第 2 个操作数相加，并且结果保存在堆栈指针中。

[说明示例]

```
ADDW    SP, #12H    ; (1)
```

(1) 堆栈指针或 12H 相加，且结果保存在堆栈指针中。

SUBW

堆栈指针减法

[指令格式]

SUBW SP, src

[运算符]

SP <- SP - src

[操作数]

操作数 (SP, src)
SP, #byte

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 堆栈指针中设定的第 1 个操作数减去源操作数 (src) 中设定的第 2 个操作数，且结果保存在堆栈指针中。

[说明示例]

```
SUBW    SP, #12H    ; (1)
```

(1) 堆栈指针减去 12H，且结果保存在堆栈指针中。

(12)绝对转移指令

以下绝对转移指令可用

指令	概要
BR	绝对转移指令

BR

绝对转移指令

[指令格式]

BR 目标

[运算符]

PC <- 目标

[操作数]

操作数 (目标)
AX
$addr20
$!addr20
!addr16
!!addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这是绝对转移指令。
- 转移目标地址中操作数 （目标）的子数据到 PC 并分支。

[说明示例]

```
BR      !!12345H        ; (1)
```

(1) 转移到地址 12345H。

(13)条件转移指令

以下条件转移指令可用

指令	概要
BC	带进位标志的条件转移 (CY = 1)
BNC	带进位标志的条件转移 (CY = 0)
BZ	带零标志的条件转移 (Z = 1)
BNZ	带零标志的条件转移 (Z = 0)
BH	以数值大小为条件的转移 ((Z v CY) = 0)
BNH	以数值大小为条件的转移 ((Z v CY) = 1)
BT	以测试位为条件的转移 （字节数据位 =1）
BF	以测试位为条件的转移 （字节数据位 =0）
BTCLR	以测试位为条件的转移和清零 （字节数据位 =1）

BC

带进位标志的条件转移 (CY = 1)

[指令格式]

BC $addr20

[运算符]

PC <- PC + 2 + jdisp8 if CY = 1

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 CY=1, 则数据转移到由操作数设定的地址上。
  当 CY = 0 时 , 没有进程在进行且执行下一条指令。

[说明示例]

```
BC      $00300H         ; (1)
```

(1) 当 CY=1, 数据传送到 00300H(在地址 0027FH 到 0037EH 范围内设置使用这条指令的起始）。

BNC

带进位标志的条件转移 (CY = 0)

[指令格式]

BNC $addr20

[运算符]

PC <- PC + 2 + jdisp8 if CY = 0

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 CY=0, 则数据转移到由操作数设定的地址上。
  当 CY = 1 时 , 没有进程在进行且执行下一条指令。

[说明示例]

```
BNC     $00300H         ; (1)
```

(1) 当 CY=0, 数据传送到 00300H(在地址 0027FH 到 0037EH 范围内设置使用这条指令的起始）。

BZ

带零标志的条件转移 (Z = 1)

[指令格式]

BZ $addr20

[运算符]

PC <- PC + 2 + jdisp8 if Z = 1

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 Z=1, 则数据转移到由操作数设定的地址上。
  当 Z = 0 时 , 没有进程在进行且执行下一条指令。

[说明示例]

```
DEC     B
BZ      $003C5H         ; (1)
```

(1) 当 B 寄存器为 0，则数据传送到 003C5H （在地址 00344H 到 00443H 范围内设置这条指令的起始位置）。

BNZ

带零标志的条件转移 (Z = 0)

[指令格式]

BNZ $addr20

[运算符]

PC <- PC + 2 + jdisp8 if Z = 0

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank : 不变

[说明]

- 当 Z=0, 则数据转移到由操作数设定的地址上。
  当 Z = 1 时 , 没有进程在进行且执行下一条指令。

[说明示例]

```
CMP     A, #55H
BNZ     $00A39H         ; (1)
```

(1) 如果 A 寄存器不是 55H, 则数据传送到 00A39H(在地址 009B8H 到 00AB7H 范围内设置这条指令的起始位置）。

BH

以数值大小为条件的转移 ((Z v CY) = 0)

[指令格式]

BH $addr20

[运算符]

PC <- PC + 3 + jdisp8 if (Z v CY) = 0

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 (Z v CY) = 0, 则数据转移到由操作数设定的地址上。
  当 (Z v CY) = 1, 没有进程在进行且执行下一条指令。
- 这条指令用于判断哪个无符号数据值更高。在这条指令之前直接使用 CMP 指令，它检测第 1 操作数是否比第 2 操作数更高。

[说明示例]

```
CMP     A, C
BH      $00356H     ; (1)
```

(1) 当 A 寄存器内容比 C 寄存器内容更大，则转移到地址 00356H （BH 指令的起始位置，然而，是在地址 002D4H 到 003D3H 内）。

BNH

以数值大小为条件的转移 ((Z v CY) = 1)

[指令格式]

BNH $addr20

[运算符]

PC <- PC + 3 + jdisp8 if (Z v CY) = 1

[操作数]

操作数 ($addr20)
$addr20

[标记]

Z	AC	CY

Blank : 不变

[说明]

- 当 (Z v CY) = 1, 则数据转移到由操作数设定的地址上。
  当 (Z v CY) = 0, 没有进程在进行且执行下一条指令。
- 这条指令用于判断哪个无符号数据值更高。在这条指令前直接使用 CMP 指令，它检测第一操作数是否没有第二操作数高 （第一操作数等于或小于第二操作数）。

[说明示例]

```
CMP     A, C
BNH     $00356H     ; (1)
```

(1) 当 A 寄存器内容等于或小于 C 寄存器内容，则转移到地址 00356H （BH 指令的起始位置，然而，是在地址 002D4H 到 003D3H 内）。

BT

以测试位为条件的转移 （字节数据位 =1）

[指令格式]

BT bit, $addr20

[运算符]

PC <- PC + b + jdisp8 if bit = 1

[操作数]

操作数 (bit, $addr20)	b (字节数)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 如果第 1 操作数 （位）内容已经设置为 （1), 则数据已转移到第 2 操作数设定的地址上 ($addr20).
  如果第 1 操作数 （位）内容已经设置为 （1），则没有进程在运行且指令下一条指令。

[说明示例]

```
BT      FFE47H.3, $0055CH   ; (1)
```

(1) 当在地址 FFE47H 上的第 3 位为 1，则数据转移到 0055CH （在地址 004DAH 到 005D9H 范围内设置这条指令的起始位置）。

BF

以测试位为条件的转移 （字节数据位 =0）

[指令格式]

BF bit, $addr20

[运算符]

PC <- PC + b + jdisp8 if bit = 0

[操作数]

操作数 (bit, $addr20)	b (字节数)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 如果第 1 操作数 （位）内容已经清零 （0), 则数据已转移到第 2 操作数设定的地址上 ($addr20).
  如果第 1 操作数 （位）内容已经清零 （0），则没有进程在运行且指令下一条指令。

[说明示例]

```
BF      P2.2, $01549H       ; (1)
```

(1) 当端口 2 的第 2 位为 0，则数据传送到地址 01549H （在地址 014C6H 到 015C5H 范围内设置这条指令的起始位置）。

BTCLR

以测试位为条件的转移和清零 （字节数据位 =1）

[指令格式]

BTCLR bit, $addr20

[运算符]

PC <- PC + b + jdisp8 if bit = 1, then bit <- 0

[操作数]

操作数 (bit, $addr20)	b (字节数)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[标记]

(1) bit = PSW.bit

Z	AC	CY
x	x	x

x : 根据结果设置或清除

(2) 所有其他操作数组合

Z	AC	CY

Blank : 不变

[说明]

- 如果第 1 操作数 （位）内容已经设置为 （1), 则它们已清零并转移到第 2 操作数设定的地址上。
  如果第 1 操作数 （位）内容已经设置为 （1），则没有进程在运行且指令下一条指令。
- 当第 1 操作数 （位）是 PSW.bit，则相应的标志内容清零 (0)。

[说明示例]

```
BTCLR PSW.0, $00356H    ; (1)
```

(1) 当 PSW 的第 0 位（CY 标志）为 1，则 CY 标志清除为 0 且转移到地址 00356H 上（在地址 002D4H 到 003D3H 范围内设置这条指令的起始位置）。

(14) 条件跳转指令

以下条件跳转指令可用。

指令	概要
SKC	进位标志的跳转 (CY = 1)
SKNC	进位标志的跳转 (CY = 0)
SKZ	零标志的跳转 (CY = 1)
SKNZ	零标志的跳转 (CY = 0)
SKH	数字大小的跳转 ((Z v CY) = 0)
SKNH	数字大小的跳转 ((Z v CY) = 1)

SKC

进位标志的跳转 (CY = 1)

[指令格式]

SKC

[运算符]

如果 CY = 1，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 CY = 1 时 , 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用掏 S:
- 当 CY = 0 时 , 执行下一条指令。

[说明示例]

```
MOV     A, #55H
SKC
ADD     A, #55H     ; (1)
```

(1) 当 CY = 0, 和 55H 当 CY = 1，则 A 寄存器的值 = AAH。

SKNC

进位标志的跳转 (CY = 0)

[指令格式]

SKNC

[运算符]

如果 CY = 0，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 CY = 0 时 , 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用掏 S:
- 当 CY = 1 时 , 执行下一条指令。

[说明示例]

```
MOV     A, #55H
SKNC
ADD     A, #55H     ; (1)
```

(1) 当 CY = 1, 和 55H 当 CY = 0，则 A 寄存器的值 = AAH。

SKZ

零标志的跳转 (CY = 1)

[指令格式]

SKZ

[运算符]

如果 Z = 1，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 当 Z = 1 时 , 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用 “ES:”标识），则花费了两个时钟的执行时间。
- 当 Z = 0 时 , 执行下一条指令。

[说明示例]

```
MOV     A, #55H
SKZ
ADD     A, #55H     ; (1)
```

(1) 当 Z = 0, 和 55H 当 Z = 1，则 A 寄存器的值 = AAH。

SKNZ

零标志的跳转 (CY = 0)

[指令格式]

SKNZ

[运算符]

如果 Z = 0，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 Z = 0 时 , 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用掏 S:
- 当 Z = 1 时 , 执行下一条指令。

[说明示例]

```
MOV     A, #55H
SKNZ
ADD     A, #55H     ; (1)
```

(1) 当 Z = 1, 和 55H 当 Z = 0，则 A 寄存器的值 = AAH。

SKH

数字大小的跳转 ((Z v CY) = 0)

[指令格式]

SKH

[运算符]

如果 (Z v CY) = 0，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 (Z v CY) = 0, 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用掏 S:
- 当 (Z v CY) = 1, 则执行下一条指令。

[说明示例]

```
CMP     A, #80H
SKH
CALL    !!TARGET        ; (1)
```

(1) 当 A 寄存器内容比 80H 高，则跳过调用指令并执行下一条指令。当 A 寄存器内容是 80H 或更低时，执行下一条调用指令并转移到目标地址中执行。

SKNH

数字大小的跳转 ((Z v CY) = 1)

[指令格式]

SKNH

[运算符]

如果 (Z v CY) = 1，跳过下个指令

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 当 (Z v CY) = 1, 跳过下一条指令。随后的指令是 NOP，且花费了一个时钟的执行时间。然而，如果随后的指令是 PREFIX 指令 （用 ES: 标识），则花费了两个时钟的执行时间。
- 当 (Z v CY) = 0, 则执行下一条指令。

[说明示例]

```
CMP     A, #80H
SKNH
CALL    !!TARGET        ; (1)
```

(1) 当 A 寄存器内容为 80H 或更低时，则跳过调用指令并执行下一条指令。当 A 寄存器内容是比 80H 高时，执行下一条调用指令并转移到目标地址中执行。

(15) CPU 控制指令

以下 CPU 控制指令可用。

指令	概要
SEL	选定寄存器组
NOP	无操作
EI	中断可用
DI	中断不可用
HALT	中止模式设置
STOP	停止模式设置

SEL

选定寄存器组

[指令格式]

SEL RBn

[运算符]

RBS0, RBS1 <- n ; (n = 0 to 3)

[操作数]

操作数 (RBn)
RBn

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 设定操作数的寄存器组 (RBn) 作为随后指令的寄存器组使用。
- RBn 范围从 RB0 到 RB3。

[说明举例]

```
SEL     RB2     ;  (1)
```

(1) 选中的寄存器组 2 作为随后指令的寄存器组使用。

NOP

无操作

[指令格式]

NOP

[运算符]

无操作

[操作数]

None

[标记]

Z	AC	CY

Blank： 不变

[说明]

- 不处理只消耗时间。

EI

中断可用

[指令格式]

EI

[运算符]

IE<- 1

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 设置可屏蔽中断的应答状态 (通过设置中断标志 (IE) 为 （1））。
- 在这条指令和下条指令间，无中断应答。
- 如果执行这条指令 , 能禁止对其他源程序中来的向量中断应答。详细内容 , 在每个产品的用户操作手册中参阅中断功能的说明。

DI

中断不可用

[指令格式]

DI

[运算符]

IE<- 0

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 通过禁止向量中断，可屏蔽中断应答 (中断使标志 （IE) 清零 （0））。
- 在这条指令和下条指令间，无中断应答。
- 有关中断服务的详细内容 , 参阅各产品用户手册中中断功能的内容。

HALT

中止模式设置

[指令格式]

HALT

[运算符]

设置终止模式

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

- 这条指令用于设置 HALT 模式来停止 CPU 运行时钟。通过将这个模式与正常运行模式绑定，间歇运行能减少整个系统的电能消耗量。

STOP

停止模式设置

[指令格式]

STOP

[运算符]

设置停止模式

[操作数]

None

[标记]

Z	AC	CY

Blank ： 不变

[说明]

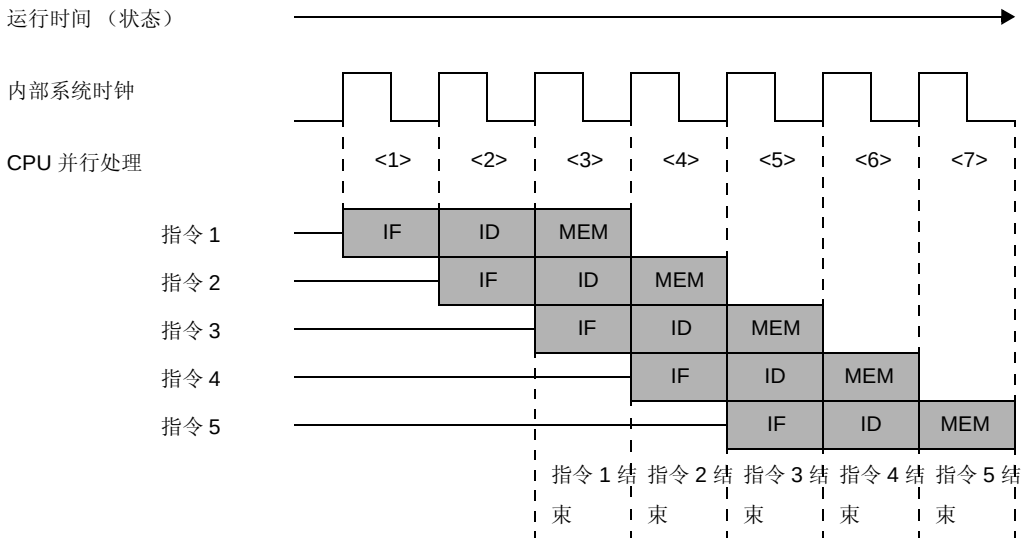
- 这条指令用于设置 STOP 模式来停止主系统时钟振荡器，以及停止整个系统。电力消耗量能最节省到只有漏电电流。

4.6.7 流水线

(1) 特性

78K0R 单片机采用三级流水线控制，使得几乎所有指令在一个周期内执行完毕。指令分为三个阶段执行：指令读取 （IF）、指令解码 （ID）和存储器访问 （MEM）。

图 4-38. 五种类型指令的流水线执行 （示例）



IF （指令读取）	指令被读取并且读取指针增加。
ID （指令解码）	指令被解码并且计算地址。
MEM （存储器访问）	执行已解码的指令并访问存储器内的目标地址。

(2) 工作时钟的数量

在一些单片机流水线架构中存在一个固有问题，即无法预测指令执行时所需的时钟数。然而， 78K0R 单片机解决了这个问题。指令总是在相同数量的时钟内执行，允许稳定的程序执行。

除下面列举的情况之外，执行的时钟数如 ?(5) 操作列表”中所示。

(a) 对作为数据的闪存内容进行访问

当把闪存中的内容作为数据进行访问时，说明流水线处于 MEM 阶段。这将显示在操作列表中的数量上再增加一定的时钟数。详情请参阅 “(5) 操作列表”。

(b) 对作为数据的外部存储器内容进行访问

当外部存储器的内容作为数据被访问时，将产生一个 CPU 等待。这将显示在操作列表中的数量上再增加一定的时钟数。

下表显示了增加的时钟数。

外部扩展时钟输出 （CLKOUT）选择时钟	等待周期
fCLK	3 个时钟
fCLK/2	5 或 6 个时钟
fCLK/3	7 至 9 个时钟
fCLK/4	9 至 12 个时钟

(c) 从 RAM 读取指令

当指令从 RAM 中被读取时，指令列队将为空，因为从 RAM 读取的时间较长。 CPU 必须等待知道指令列队就绪为止。此外，在指令从 RAM 读取期间，如果 RAM 被访问时， CPU 也必须等待。

(d) 从外部存储器读取指令

当指令从外部存储器中被读取时，指令列队将为空，因为从外部存储器读取的时间较长。 CPU 必须等待知道指令列队就绪为止。此外，在指令从外部存储器读取期间，如果外部存储器被访问时， CPU 也必须等待。下表显示了增加的时钟数

外部扩展时钟输出 （CLKOUT）选择时钟	等待周期
fCLK	3 个时钟
fCLK/2	5 或 6 个时钟
fCLK/3	7 至 9 个时钟
fCLK/4	9 至 12 个时钟

(e) 指令组合的危害

当数据写入寄存器之后，该寄存器立即被用于间接存储器访问时，将产生一个时钟的等待时间。

寄存器	上一个指令	下一指令的操作数或下一指令
DE	把指令写入 D 寄存器注 把指令写入 E 寄存器注 把指令写入 DE 寄存器注 SEL RBn	[DE], [DE+byte]
HL	把指令写入 H 寄存器注 把指令写入 L 寄存器注 把指令写入 HL 寄存器注 SEL RBn	[HL], [HL+byte], [HL+B], [HL+C], [HL].bit
B	把指令写入 B 寄存器注 SEL RBn	word[B], [HL+B]
C	把指令写入 C 寄存器注 SEL RBn	word[C], [HL+C]
BC	把指令写入 B 寄存器注 把指令写入 C 寄存器注 把指令写入 BC 寄存器注 SEL RBn	word[BC], [HL+B], [HL+C]

SP	MOVW SP, #word
	MOVW SP, AX
	ADDW SP, #byte
	SUBW SP, #byte
		[SP+byte]
		CALL 指令、 CALLT 指令、 BRK 指令、 SOFT 指令、 RET 指令、 RETI 指令、 RETB 指令、中断、 PUSH 指令、 POP 指令

CS	MOV CS, #byte MOV CS, A	CALL rp BR AX
AX	把指令写入 A 寄存器注 把指令写入 X 寄存器注 把指令写入 AX 寄存器注 SEL RBn	BR AX

寄存器	上一个指令	下一指令的操作数或下一指令
AX BC DE HL	把指令写入 A 寄存器[注] 把指令写入 X 寄存器[注] 把指令写入 B 寄存器[注] 把指令写入 C 寄存器[注] 把指令写入 D 寄存器[注] 把指令写入 E 寄存器[注] 把指令写入 H 寄存器[注] 把指令写入 L 寄存器[注] 把指令写入 AX 寄存器[注] 把指令写入 BC 寄存器[注] 把指令写入 DE 寄存器[注] 把指令写入 HL 寄存器[注] SEL RBn	CALL rp

注 写入指令到寄存器也需要一定的等待时间，用于直接寻址、短的直接寻址、寄存器间接寻址、基址寻址或基址索引寻址进行复写目标寄存器。

第 5 章 链接指令说明

本章介绍链接指令的必要条目以及如何编写指令文件。

5.1 编码方法

本节介绍链接指令的编码方法。

5.1.1 链接指令

链接指令 （这里指的是指令) 一组命令 , 用于在链接程序中发出不同的命令，如文件输入，可用存储区域及区段的分配。

有下列两种指令。

指令 类型	作用
存储指令	- 在存储中声明地址。 - 将存储分割为几个区域，并设定存储区域。 CALLT 区域 内部 ROM 外部 ROM SADDR 除 SADDR 以外的内部 RAM
段地址指令	- 设定段地址。 设定下列每段的内容。 绝对地址 只设定存储区域。

用编辑器来创建指令文件 (指令文件），且启动链接器时设定 -d 选项。

链接器从该文件读取指令文件并翻译来执行链接处理。

(1) 指令文件

指令文件中指定指令格式显示如下。

- 存储指令

```
MEMORY  memory-area-name: (start-address-value, size) [ /memory-space-name]
```

- 分配段指令

```
MERGE segment-name: [ ATΔ(Δstart-addressΔ) ] [ =memory-area-name-specification] [ /
memory-space-name]

MERGE  segment-name: [ merge-attribute ] [ =memory-area-name-specification] [ /
memory-space-name]
```

另外，在单个指令文件中能设定多重指令。

有关各指令的详细内容 , 参阅 "(2) 存储指令 " 和 "(3) 段地址指令 "。

(a) 符号

在指定段名称，存储区域名称和存储空间名称中要区分大小写字母。

(b) 数值

当在每条指令中设定条目为数字常量时，它们能设定为十进制数或十六进制数。
设定的数字与源代码中的相同，在十六进制数后面附上 "H" 。如果 A-F 出现在开头位置，则先加 "0"。
示例显示如下。

```
23H , 0FC80H
```

(c) 注释

指令文件中，当设定 ";" 或 "#" 时，把从此处字符到换行 （LF) 符为止之间的文本作为注释。另外，如果在换行符出现之前已经在指令文件末尾，则文件结尾以上的文本作为注释。
示例显示如下。
下划线部分为注释。

```
; DIRECTIVE FILE FOR 78F1166_A0
MEMORY MEM1 : ( 40000H, 10000H ) #SECOND MEMORY AREA
```

(2) 存储指令

存储指令是定义存储区域的指令 （有效存储地址和它的名称）。
通过使用这个名称的段地址指令能引用定义的存储区域。
可以定义多达 100 个存储区域，其中包含默认存储区域。

语法显示如下。

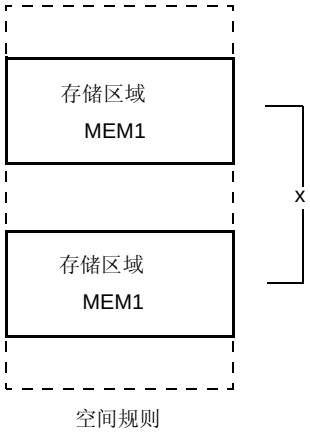
```
MEMORYΔmemory-area-nameΔ:Δ(Δstart-addressΔ,ΔsizeΔ)[/Δmemory-space-name]
```

(a) 存储区域名称

设定已定义存储区域的名称。
设定时需要满足如下条件。

- 在存储区域能使用的字符为 A - Z, a - z, 0 - 9, _, ?, @。
  然而 , 存储区域名称不能以 0 - 9 的数字开头。
- 用大小写字母来区分字符。
- 能同时混合使用大小写字母字符。
- 存储区域名称的长度最大 256 个字符。若使用
  257 个字符或更多字符会导致错误结果。
- 在同一个存储区名称不能用于不同的存储区域，哪怕这些存储区域位于不同的存储空间也不行。
  当存储空间相同或不同时，不允许在不同的存储区域使用相同的存储区域名称。

图 5-1. 不能设定存储区域名称的例子



(b) 起始地址

定义存储区域的起始地址。

在 0H-0FFFFFH 之间写入数字常量。

(c) 容量

定义存储区域的大小。

设定时需要满足如下条件。

- 数字常量为 1 或更大。
- 当重新定义存储区域的大小为默认链接器时，则对定义范围内有限制。
  关于为每个设备默认定义存储区域大小以及可以重定义的范围的内容 , 参阅 每个设备文件的 " 使用注意事项 "。

(d) 存储区域名称

用 64 kB 空间 REGULAR 来表示存储空间名称 .

当设定时需要满足的条件如下所示。

- 存储空间名称全部使用大写字母来设定。
- 当省略存储空间名称，则默认为 REGULAR。
- 如果在写了 "/" 之后省略了存储空间名称，则产生错误。

功能显示如下。

- 在指定存储空间名称的存储区域定义指定的存储空间。
- 用 1 条存储指令能定义 1 个存储区域。
- 可以多次指定存储指令。当在设定的指令上有多重定义，则结果会出错。
- 只要存储指令不引用相同存储区域 , 则默认存储区域有效。当省略存储指令，则链接器只能为每条指令设定默认存储区域。
- 当不使用默认存储空间且用不同的区域名称使用它，则设置默认区域的名称大小为 "0"。

用法示例显示如下。

- 定义存储空间 地址 0H 到 1FFH 作为存储区域 ROMA。

```
MEMORY ROMA : ( 0H, 200H )
```

(3) 段地址指令

段地址指令是用来在存储器的指定区域或特定地址分配一个指令区段。

语法显示如下。

```
MERGEΔsegment-nameΔ:Δ[ATΔ(Δstart-addressΔ)][Δ=Δmemory-area-name][Δ/Δmemory-space-name]
MERGEΔsegment-nameΔ:Δ[merge-attribute][Δ=Δmemory-area-name][Δ/Δmemory-space-name]
```

(a) 段名称

在输入到链接器中的目标模块文件中包含段名称。

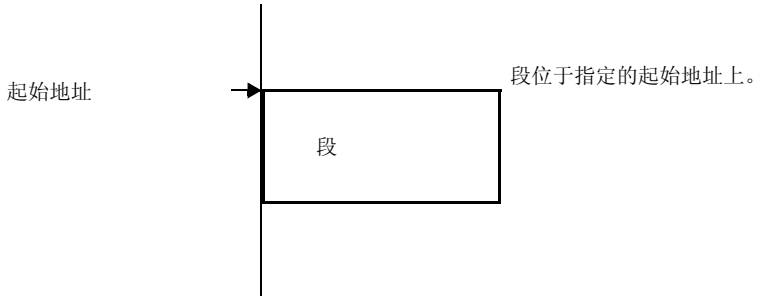
- 只有输入区段可以用区段名称进行指定。
- 在汇编程序源代码中必须设定特定的段名称。

(b) 起始地址

在 " 起始地址 " 的区域分配段。

- 保留字 AT 必须全部用大写字母字符或小写字母字符指定。不能混合大写和小写字母字符。
- 用数字常量来设定起始地址。

图 5-2.　设定起始地址和段地址



注意事项 1.　当区段分配到指定起始地址时，如果段的定位已经超过了存储区域范围，则产生错误结果。

2.　对于由区段指令的 AT 指令或由 ORG 指令指定了存储地址的区段来说，不能用链接指令来指定其起始地址。

(c) 合并属性

当在源代码中有多个同名的段，用指令 "COMPLETE" 设定不合并或用 "SEQUENT (默认)" 进行合并。

SEQUENT	按段出现的次序顺序合并，为了不留下空闲空间。 按段出现的顺序以位为单位合并 BSEG。
COMPLETE	如果有多个同名的段，则会产生错误。

示例显示如下。

```
MERGE DSEG1 : COMPLETE = RAM
```

(d) 存储区域名称

设定定位段的存储空间存储空间名称。

- 只有 REGULAR 能设定为存储空间名称。
- 存储空间名称全部使用大写字母来设定。
- 当省略存储空间名称时，则默认为 REGULAR。

段地址显示如下。

存储区域	存储空间	段位置
无指定	无指定	存储区域位于 REGULAR 默认空间
存储区域名称	无指定	在 REGULAR 空间指定存储区域

此表着重声明段地址的目标存储区域。此外，如果当决定实际位置地址时设定 "AT(起始地址） "，段将位于那个地址上。

例如，为 "CSEG BASE" 的段重定位属性 , 如果设定存储名称 "REGULAR", 则段位置会将会定位于在 0000H 到 FFFFH 内。

注意事项如下所示。

- 根据在汇编期间使用段定义指令设定重定位属性，来决定它们的定位地址，而使用段地址指令不能设定输入段。
- 如果为不存在的段设定段名称，则会产生错误结果。
- 如果为同一段设定多个段定位指令，则将产生错误结果。

5.2 保留字节

在指令文件中使用保留字显示如下。

在指令文件中的保留字不能用作其他目的 （段名称，存储区域名称，等等)。

保留字节	说明
MEMORY	设定存储指令
MERGE	设定段定位指令
AT	设定段定位指令的定位属性 （起始地址）
SEQUENT	设定段定位指令的合并属性 （合并段）
COMPLETE	设定段定位指令的合并属性 （不合并段）

注意事项 保留字能用大写字母或小写字母设定。但是不能混合大写和小写字母字符。

示例 MEMORY ： 可以使用
MEMORY ： 可以使用
MEMORY ： 不可以使用

5.3 编码举例

链接指令编码举例显示如下。

5.3.1 在设定链接指令时

- 为带段类型的段 SEG1 分配地址，重定位 "CSEG UNIT" 的属性。
  声明存储区域如下。

```
MEMORY ROM  : ( 0000H, 1000H )
MEMORY MEM1 : ( 1000H, 2000H )
```

- 当分配输入段 SEG1 到 ROM 区域内的 500H 上 （看以下图 （1））。

```
MERGE  SEG1 : AT ( 500H )
```

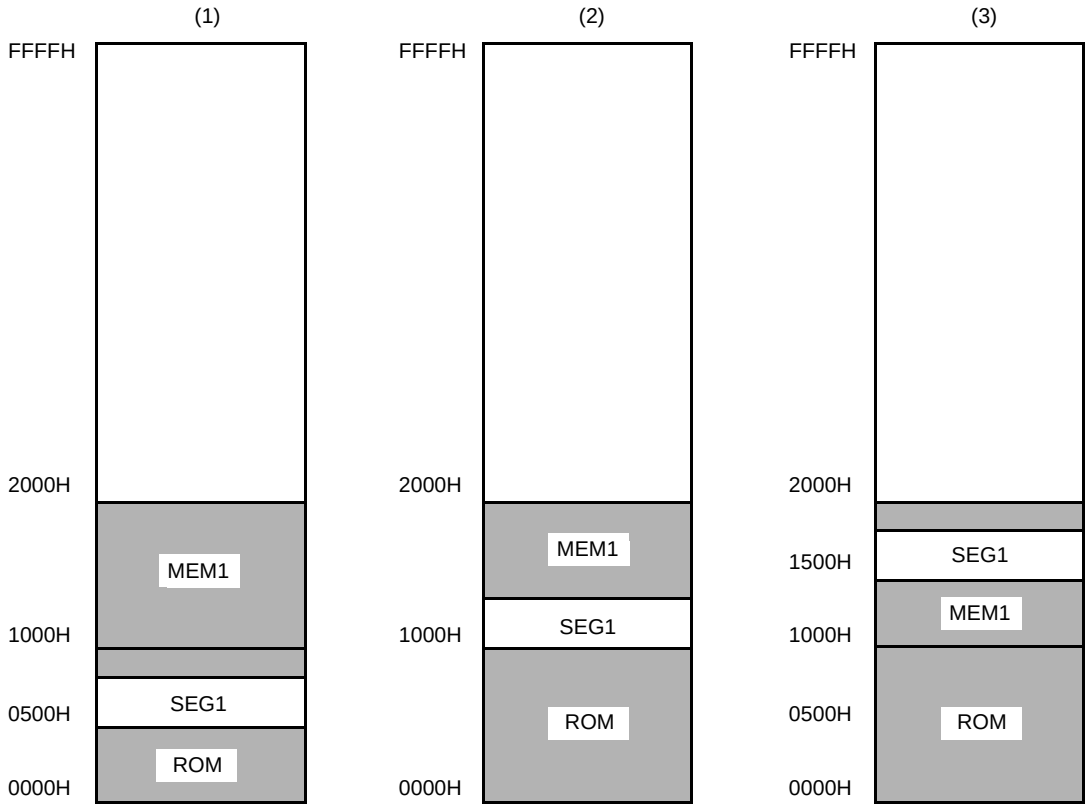
- 当分配输入段 SEG1 到 MEM1 存储区域内 （看以下图 （2））。

```
MERGE  SEG1 : = MEM1
```

- 当分配输入段 SEG1 到 MEM1 存储区域内的 1500H 上 （看以下图 （3））。

```
MERGE SEG1 : AT (1500H)=MEM178
```

图 5-3. 输入段 SEG1 的分配实例



5.3.2　在使用编译器时

本节介绍在使用编译器时，如何创建 链接指令文件。在链接时创建与实际目标系统相兼容的文件，并使用 -d 选项设定已创建的文件。

另外，在创建文件时，请注意以下注意事项。

- 78K0R C 编译器可以为特定目的使用短距离地址区域 (saddr 区域) 部分显示如下。
  特别在 FFEB4H 到 FFEDFH 的 44 个字节区域内。

(a) 在设定 -qr 选项时 , 寄存器变量 [FFEB4H 到 FFEC3H]

(b) norec 函数参数 , 自动变量 [FFEC4H 到 FFED3H]

(c) 段信息 [FFED4H 到 FFED7H]

(d) 运行时间库参数 [FFED8H 到 FFEDFH]

(e) 标准库操作 （区域 (a) 和 (b) 的部分）

注意事项　在用户不使用标准库时，区域 (e) 不可用。

以下显示用链接指令文件改变 RAM 大小的举例 (lk78k0r.dr)。

在改变存储大小时，小心存储器不要被其他区域覆盖。在发生改变时 , 查看使用目标设备的存储器映射。

```
                    起始地址大小
memory RAM :     ( 0FCF00h, 002F20h )     -> 使其大小更大。
memory SDR :     ( 0FFE20h, 000098h )        ( 也必须改变起始地址 )
merge @@INIS : = SDR                      -> 设定段地址。
merge @@DATS : = SDR                      -> 设定段地址。
merge @@BITS : = SDR                      -> 设定段位置
```

在想改变段位置时，添加合并状态。在使用改变编译器的输出节名称功能时，(有关详细内容，参阅 " 改变编译器输出区段名称 (#pragma section ...)")。

在没有足够空间定位段时，改变段地址的结果，改变相应的存储状态。

第 6 章 函数说明

在 C 语言中没有执行外部 (外围) 设备或装备输入 / 输出的命令。这是因为 C 语言的设计者希望这种函数的个数尽可能保持最少。然而，在实际系统开发中输入 / 输出指令是必须的。所以 , 在 78K0R C 编译器中必定含有输入 / 输出操作的库函数。

本章节介绍 78K0R C 编译器所带的库函数，并且这些库函数能够在仿真器中使用。

6.1 分配库

对 C 编译器中库的分配 78K0R 做如下介绍。

当在应用程序中使用标准库时，包含相关头文件以及使用的库函数。

运行时间库是标准库的一部分，通过 78K0R C 编译器自动调用例程 , 它们不是 C 语言和汇编语言源代码中描述的函数。

表 6-1. 分配库

库类型	已包含的函数
标准库	- 字符 / 字符串函数 - 程序控制函数 - 特殊函数 - 输入和输出函数 - 公用函数 - 字符串和存储函数 - 数学函数 - 诊断函数
运行时间库	- 递增 - 递减 - 符号颠倒 - 第一个附加物 - 逻辑非 - 乘法 - 除法 - 余数计算 - 加法 - 减法 - 向左移位 - 向右移位 - 比较 - 位与 - 位或 - 位异或 - 转换浮点数 - 转换为浮点数 - 位的转换 - 启动程序 - 函数预处理 / 后处理 - BCD- 类型转换 - 辅助

6.1.1　标准库

本节显示包含在标准库中的函数。
当设定了 -zf 选项时，则完全支持标准库。

(1) 字符 / 字符串函数

功能名称	作用	头文件	重返
isalpha	判断字符是否是字母符号 (A 到 Z, a 到 z)	ctype.h	OK
isupper	判断字符是否是大写字母符号 (A 到 Z)	ctype.h	OK
islower	判断字符是否是小写字母符号 (a 到 z)	ctype.h	OK
isdigit	判断字符是否是数字 (0 到 9)	ctype.h	OK
isalnum	判断字符是否是字母数字式字符 (0 到 9, A 到 Z, a 到 z)	ctype.h	OK
isxdigit	判断字符是否是十六进制数 (0 到 9, A 到 F, a 到 f)	ctype.h	OK
isspace	判断字符是否是空白字符 (空格 , 制表符 , 回车 , 换行 , 垂直制表符 , 换页)	ctype.h	OK
ispunct	判断字符是否是除空白字符或字母数字字符以外的打印字符。	ctype.h	OK
isprint	判断字符是否是打印字符	ctype.h	OK
isgraph	判断字符是否是除空白字符以外的打印字符	ctype.h	OK
iscntrl	判断字符是否是控制字符	ctype.h	OK
isascii	判断字符是否是 ASCII 码	ctype.h	OK
toupper	将小写字母字符转换为大写	ctype.h	OK
tolower	将大写字母字符转换为小写	ctype.h	OK
toascii	转换输入为 ASCII 码	ctype.h	OK
_toupper	从输入字符中去掉 "a" 并加上 "A"	ctype.h	OK
toup		ctype.h	OK
_tolower	从输入字符中去掉 "A" 并加上 "a"	ctype.h	OK
tolow		ctype.h	OK

OK ： 重返

(2) 程序控制函数

功能名称	作用	头文件	重返
setjmp	在调用时保存环境	setjmp.h	NG
longjmp	用 setjmp 恢复保存过的环境	setjmp.h	NG

NG ： 不重返

(3) 特殊函数

功能名称	作用	头文件	重返
va_start	为处理可变参数做设置	stdarg.h	OK
va_starttop	为处理可变参数做设置	stdarg.h	OK
va_arg	处理可变参数	stdarg.h	OK
va_end	显示处理可变参数的结束	stdarg.h	OK

OK ： 重返

(4) 输入和输出函数

功能名称	作用	头文件	重返
sprintf	根据格式在字符串中写入数据	stdio.h	Δ
sscanf	根据格式从输入字符中读取数据	stdio.h	Δ
printf	根据格式输出数据到 SFR 中	stdio.h	Δ
scanf	根据格式从 SFR 中读取数据	stdio.h	Δ
vprintf	根据格式输出数据到 SFR 中	stdio.h	Δ
vsprintf	根据格式在字符串中写入数据	stdio.h	Δ
getchar	从 SFR 中读取一个字符	stdio.h	OK
gets	读取字符串	stdio.h	OK
putchar	输出一个字符到 SFR 中	stdio.h	OK
puts	输出字符串	stdio.h	OK
__putc	输出一个字符到 opaque	stdio.h	OK

OK ： 重返
Δ　： 不支持浮点而重返的函数

(5) 公用函数

功能名称	作用	头文件	重返
atoi	十进制整数字符串转换为整数型	stdlib.h	OK
atol	十进制整数字符串转换为长型	stdlib.h	OK
strtol	转换字符串为长型	stdlib.h	OK
strtoul	转换字符串为无符号长型	stdlib.h	OK
calloc	分配数组区间并初始化为 0	stdlib.h	OK
free	释放已分配的内存块	stdlib.h	OK
malloc	分配块	stdlib.h	OK
realloc	重新分配块	stdlib.h	OK
中止	异常中止程序	stdlib.h	OK
atexit	在正常终止时，注册调用的函数	stdlib.h	NG
退出	终止程序	stdlib.h	NG
abs	获取整数型值的绝对值	stdlib.h	OK
labs	获取 long 型值的绝对值	stdlib.h	OK
div	进行 int 型除法运算，获取商数和余数	stdlib.h	NG
ldiv	进行 long 型除法运算，获取商数和余数	stdlib.h	NG
brk	设置中断值	stdlib.h	NG
sbrk	增加 / 减少中断值	stdlib.h	NG
atof	十进制整数字符串转换为 double 型	stdlib.h	NG
strtod	转换字符串为 double 型	stdlib.h	NG
itoa	转换 int 型 为字符串	stdlib.h	OK
ltoa	转换 long 型为字符串	stdlib.h	OK
ultoa	转换无符号 long 型为字符串	stdlib.h	OK
rand	生成伪随机号	stdlib.h	NG
srand	初始化伪随机号的生成状态	stdlib.h	NG
bsearch	二进制查找	stdlib.h	OK
qsort	快速排序	stdlib.h	OK
strbrk	设置中断值	stdlib.h	OK
strsbrk	增加 / 减少中断值	stdlib.h	OK
stritoa	转换 int 型 为字符串	stdlib.h	OK
strltoa	转换 long 型为字符串	stdlib.h	OK
strultoa	转换无符号 long 型为字符串	stdlib.h	OK

OK ： 重返
NG ： 不重返

(6) 字符串和存储函数

功能名称	作用	头文件	重返
memcpy	为指定数量的字符复制缓冲	string.h	OK
memmove	为指定数量的字符复制缓冲	string.h	OK
strcpy	复制字符串	string.h	OK
strncpy	从字符串的首字符开始，复制指定数量的字符。	string.h	OK
strcat	在字符串后追加字符串	string.h	OK
strncat	在字符串后追加指定字符数量的字符串	string.h	OK
memcmp	在两个缓冲中比较指定数量的字符	string.h	OK
strcmp	比较两个字符串	string.h	OK
strncmp	在两个字符串中比较指定数量的字符	string.h	OK
memchr	在缓冲区内指定数量的字符中查找指定字符串	string.h	OK
strchr	从字符串和首次返回发生事件的位置中查找指定字符串	string.h	OK
strrchr	从字符串和最后返回发生事件的位置中查找指定字符串	string.h	OK
strspn	从段起始处获取长度，该段包含的字符位于查找字符串内特定字符串中	string.h	OK
strcspn	从段起始处获取长度，该段包含的字符不在查找字符串内特定字符串中	string.h	OK
strpbrk	在字符串中查找特定字符串时，获取任意字符第一个出现的位置	string.h	OK
strstr	在字符串中查找特定字符串时，获取指定字符串第一个出现的位置	string.h	OK
strtok	分解字符串为不带分隔符的字符。	string.h	NG
memset	初始化指定字符的缓冲中指定数量的字符	string.h	OK
strerror	返回指针区域，该区域存储与错误信息相对应的指定错误号。	string.h	OK
strlen	获取字符串的长度	string.h	OK
strcoll	根据范围特定信息比较两个字符串	string.h	OK
strxfrm	根据范围特定信息变换两个字符串	string.h	OK

OK ： 重返
NG ： 不重返

(7) 数学函数

功能名称	作用	头文件	重返
acos	发现 acos	math.h	NG
asin	发现 asin	math.h	NG
atan	发现 atan	math.h	NG

功能名称	作用	头文件	重返
atan2	发现 atan2	math.h	NG
cos	发现 cos	math.h	NG
sin	发现 sin	math.h	NG
tan	发现 tan	math.h	NG
cosh	发现 cosh	math.h	NG
sinh	发现 sinh	math.h	NG
tanh	发现 tanh	math.h	NG
exp	发现 eNG 位函数	math.h	NG
frexp	发现小数和 NG 指数部分	math.h	NG
ldexp	发现 NG * 2 ^ eNGp	math.h	NG
log	发现自然对数	math.h	NG
log10	发现底为 10 的对数	math.h	NG
modf	发现小数和整数部分	math.h	NG
pow	发现 NG 的 y 次幂	math.h	NG
sqrt	发现平方根	math.h	NG
ceil	发现最小整数大于 NG	math.h	NG
fabs	发现浮点数 NG 的绝对值	math.h	NG
floor	发现最大整数小于 NG	math.h	NG
fmod	发现 NG/y 的余数	math.h	NG
matherr	获取库操作浮点数的 eNG 叠处理	math.h	NG
acosf	发现 acos	math.h	NG
asinf	发现 asin	math.h	NG
atanf	发现 atan	math.h	NG
atan2f	发现 y/NG 的 atan	math.h	NG
cosf	发现 cos	math.h	NG
sinf	发现 sin	math.h	NG
tanf	发现 tan	math.h	NG
coshf	发现 cosh	math.h	NG
sinhf	发现 sinh	math.h	NG
tanhf	发现 tanh	math.h	NG
expf	发现 eNG 位函数	math.h	NG
frexpf	发现小数和 NG 指数部分	math.h	NG
ldexpf	发现 NG * 2 ^ eNGp	math.h	NG
logf	发现自然对数	math.h	NG
log10f	发现底为 10 的对数	math.h	NG
modff	发现小数和整数部分	math.h	NG
powf	发现 NG 的 y 次幂	math.h	NG
sqrtf	发现平方根	math.h	NG
ceilf	发现最小整数大于 NG	math.h	NG

功能名称	作用	头文件	重返
fabsf	发现浮点数 NG 的绝对值	math.h	NG
floorf	发现最大整数小于 NG	math.h	NG
fmodf	发现 NG/y 的余数	math.h	NG

NG ： 不重返

(8) 诊断函数

功能名称	作用	头文件	重返
__assertfail	支持宏声明	assert.h	OK

OK ： 重返

6.1.2 运行时间库

本节显示包含在运行时间库中的函数。

通过在函数名称的开头附加 @@ 格式，调用这些操作指令。因此，通过在函数名称的开头附加 _@ 格式，调用 cstart, cstarte, cprep, 和 cdisp。

另外，指令不出现在以下没有支持库的表中。编译器进行内联展开。

long 型的加 / 减，与 / 或 / 异或，和移位也会经过内联展开。

(1) 递增

功能名称	作用
lsinc	递增符号 long 型
luinc	递增无符号 long 型
finc	递增浮点

(2) 递减

功能名称	作用
lsdec	递减符号 long 型
ludec	递减无符号 long 型
fdec	递减浮点

(3) 符号颠倒

功能名称	作用
lsrev	颠倒 long 型的符号
lurev	颠倒无符号 long 型的符号
frev	颠倒浮点的符号

(4) 第一个附加物

功能名称	作用
lscom	获取符号 long 的第一个附加物
lucom	获取无符号 long 的第一个附加物

(5) 逻辑非

功能名称	作用
lsnot	符号 long 型取反
lunot	无符号 long 型取反

(6) 乘法

功能名称	作用
csmul	在符号 char 数据间进行乘法
cumul	在无符号 char 数据间进行乘法
iumul	在无符号 int 数据间进行乘法
lsmul	在符号 long 数据间进行乘法
lumul	在无符号 long 数据间进行乘法
fmul	进行浮点数间的乘法

(7) 除法

功能名称	作用
csdiv	在符号 char 数据间进行除法
cudiv	在无符号 char 数据间进行除法
isdiv	在符号 int 数据间进行除法
iudiv	在无符号 int 数据间进行除法
lsdiv	在符号 long 数据间进行除法
ludiv	在无符号 long 数据间进行除法
fdiv	进行浮点数间的除法

(8) 余数计算

功能名称	作用
csrem	在符号 char 数据间进行余数计算
curem	在无符号 char 数据间进行余数计算
isrem	在符号 int 数据间进行余数计算
iurem	在无符号 int 数据间进行余数计算
lsrem	在符号 long 数据间进行余数计算
lurem	在无符号 long 数据间进行余数计算

(9)　加法

功能名称	作用
lsadd	在符号 long 数据间进行加法
luadd	在无符号 long 数据间进行加法
fadd	进行浮点数间的加法

(10) 减法

功能名称	作用
lssub	在符号 long 数据间进行减法
lusub	在无符号 long 数据间进行减法
fsub	进行浮点数间的减法

(11) 向左移位

功能名称	作用
lslsh	进行符号 long 数据的左移位
lulsh	进行无符号 long 数据的左移位

(12) 向右移位

功能名称	作用
lsrsh	进行符号 long 数据的右移位
lursh	进行无符号 long 数据的右移位

(13) 比较

功能名称	作用
cscmp	比较符号 char 型数据
iscmp	比较符号 int 型数据
lscmp	比较符号 long 型数据
lucmp	比较无符号 long 型数据
fcmp	比较浮点数

(14) 位与

功能名称	作用
lsband	在符号 long 数据间进行 AND(与) 操作
luband	在无符号 long 数据间进行 AND(与) 操作

(15)位或

功能名称	作用
lsbor	在符号 long 数据间进行 OR(或) 操作
lubor	在无符号 long 数据间进行 OR(或) 操作

(16)位异或

功能名称	作用
lsbNG 或	在符号 long 数据间进行 XOR(异或) 操作
lsbNG 或	在无符号 long 数据间进行 XOR(异或) 操作

(17)转换浮点数

功能名称	作用
ftols	从浮点数转换到符号 long 型
ftolu	从浮点数转换到无符号 long 型

(18)转换为浮点数

功能名称	作用
lstof	从符号 long 型转换到浮点数
lutof	从无符号 long 型转换到浮点数

(19)位的转换

功能名称	作用
btol	转换位为 long 型

(20)启动程序

功能名称	作用

cstart	启动模块
	- 获取 NG 功能寄存器函数的空间 (4 * 32 字节) 并在开头产生标签名 _@FNCTBL
	- 获取中断空间 （32 字节）并在开头产生标签名 _@MEMTOP 且这个空间的 NG 地址标签名称 _@MEMBTM
	- 按照下列方式和在指定启动模块的起始地址中定义复位向量表的段
	`@@VECT00 CSEG AT 0000H`
	`DW _@cstart`
	- 设置镜像范围
	- 设置寄存器组到 RB0
	- 设置变量 _errno 到 0, 它是用于输入错误号，
	- 设置变量 _@FNCENT 到 0, 它是用于输入带 ateNG 函数的已注册函数号 ,
	- 初始化中断值并设置 _@MEMTOP 的地址为变量 _@BRKADR

功能名称	作用
	- 设置 1 作为变量 _@SEED 的初始值，该变量是随机函数的伪随机号的来源 - 执行初始化数据的复制处理，并 eNGecute 0 - 清除无初始值 NG 外部数据。 - 调用主函数 （用户程序） - 调用带参数 0 的 eNGit 函数

(21) 函数预处理 / 后处理

功能名称	作用
cpre3	进行函数的预处理 （包含寄存变量 saddr 区域）
cprep3	进行函数的预处理 （包含寄存变量 saddr 区域）
cdis3e	进行函数的后处理 （包含寄存变量 saddr 区域）
cdisp3	进行函数的后处理 （包含寄存变量 saddr 区域）
cpre3e	进行函数的预处理 （包含寄存变量 saddr 区域）
cdis3e	进行函数的后处理 （包含寄存变量 saddr 区域）
hdwinit	在 CPU 复位后，立即进行外围设备 (sfr) 的初始化

(22) BCD- 类型转换

功能名称	作用
bcdtob	转换 1 字节 bcd 码为 1 字节 2 进制
btobcd	转换 1 字节 2 进制为 2 字节 bcd 码
bcdtow	转换 2 字节 bcd 码为 2 字节 2 进制
wtobcd	Converts 2-byte binary to 2-byte bcd
bbcd	转换 1 字节 2 进制为 2 字节 bcd 码

(23) 辅助

功能名称	作用
divuw	divuw 指令兼容
df1in	替换 fiNGed 指令样本
dn4in	替换 fiNGed 指令样本
dn4ip	替换 fiNGed 指令样本
df4in	替换 fiNGed 指令样本
df4ip	替换 fiNGed 指令样本
dn4ino	替换 fiNGed 指令样本
dn4ipo	替换 fiNGed 指令样本
df4ino	替换 fiNGed 指令样本
df4ipo	替换 fiNGed 指令样本
df1de	替换 fiNGed 指令样本
dn4de	替换 fiNGed 指令样本
dn4dp	替换 fiNGed 指令样本
df4de	替换 fiNGed 指令样本

功能名称	作用
df4dp	替换 fiNGed 指令样本
dn4deo	替换 fiNGed 指令样本
dn4dpo	替换 fiNGed 指令样本
df4deo	替换 fiNGed 指令样本
df4dpo	替换 fiNGed 指令样本
indao	替换 fiNGed 指令样本
ifdao	替换 fiNGed 指令样本
inado	替换 fiNGed 指令样本
ifado	替换 fiNGed 指令样本
lnd0	替换 fiNGed 指令样本
lfd0	替换 fiNGed 指令样本
ln0d	替换 fiNGed 指令样本
lf0d	替换 fiNGed 指令样本
lnd0o	替换 fiNGed 指令样本
lfd0o	替换 fiNGed 指令样本
ln0do	替换 fiNGed 指令样本
lf0do	替换 fiNGed 指令样本

6.2　函数间的接口

用函数调用来使用库函数。通过调用指令来完成函数调用。参数在堆栈上传递，返回值通过寄存器传递。
但是，如果可能 , 第一个参数也通过寄存器传递。

6.2.1　参数

标准库和常规函数一样的函数接口 （传递参数，存储返回值）
详情参阅 "3.4.2　普通函数调用接口 "。

6.2.2　返回值

返回值最小为 16 位并通过寄存器 BC 到 DE 的 16 位单元，从低位开始顺序存储。当返回结构时 , 在 BC, DE 中存储结构的起始地址。
详情参阅 "3.4.1　返回值 "。

6.2.3　通过分隔库来保存已用的寄存器

库使用 HL 保存寄存器用于堆栈的区域。
库使用 saddr 区域保存 saddr 中用于堆栈的区域。
库也通过使用堆栈区域来使用工作区域。
传递参数过程和返回值的实例显示如下 (小模式 , 中模式)。
以下显示函数调用。

```
"long func ( int a, long b, char *c );"
```

(1) 在堆栈上压入参数 （函数调用源）

用指令 c 在堆栈上压入参数， b 使用高 16 位， b 使用低 16 位。在 AX 寄存器中传递 a。

(2) 使用调用指令调用 func （函数调用源）

在 b 使用低 16 位后，在堆栈上压入返回地址，控制转移给了函数 func。

(3) 保存在函数中使用的寄存器 （函数调用目标）

当使用 HL 时，在堆栈上压入 HL。

(4) 通过在堆栈上的寄存器压入第一个传递参数 （函数调用目标）

(5) 进行函数 func 的处理，在寄存器中保存返回值 （函数调用目标）

在 BC 中存储返回值 "long" 的低 16 位，在 DE 中存储高 16 位。

(6) 恢复存储的第一个参数 （函数调用目标）

(7) 恢复保存的寄存器 （函数调用目标）

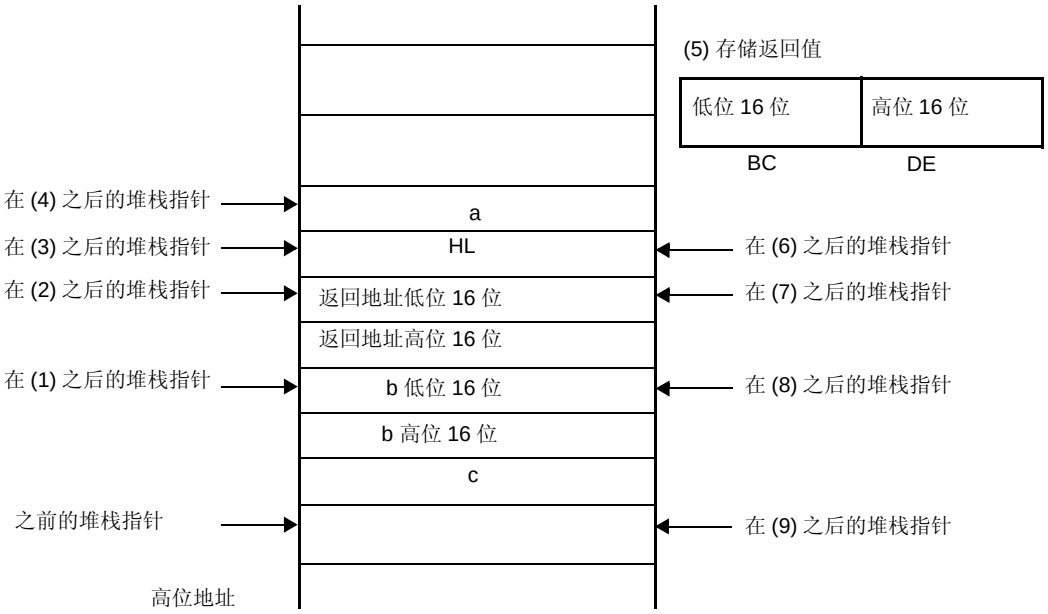
(8) 使用 ret 指令返回控制给调用函数 （函数调用目标）

(9) 在堆栈上清除参数 （函数调用源）

多个参数字节 （2 字节 单位）添加为堆栈指针。

6 为添加堆栈指针。

图 6-1. 在函数调用期间的堆栈区域



6.3 头文件

在 78K0R C 编译器中有 13 个头文件并且它们定义和声明标准库函数，类型名称，和宏名称。
78K0R C 编译器头文件显示如下。

6.3.1 ctype.h

ctype.h 定义字符 / 字符串 函数。
在 ctype.h 中，定义了下列函数。
因此，当编译选项 -za （此选项用于关闭不符合 ANSI 标准的函数并打开部分符合 ANSI 标准的函数，则没有定义的 _toupper 和 _tolower，在它们的位置上定义 tolow 和 toup。当没有设定 -za 时，则没有定义 tolow 和 toup。声明函数也要根据选项和特定的模式而变化。

```
isalpha, isupper, islower, isdigit, isalnum, isxdigit, isspace, ispunct, isprint, isgraph,
iscntrl, isascii, toupper, tolower, toascii, _toupper/toup, _tolower/tolow
```

6.3.2 setjmp.h

setjmp.h 定义程序控制函数。
在 setjmp.h 中，定义了下列函数。另外，声明函数也要根据选项和特定的模式而变化。

```
setjmp, longjmp
```

在 setjmp.h 中 , 声明下列对象。

- int 型数组 "jmp_buf" 的声明

```
typedef int     jmp_buf[12]
```

6.3.3 stdarg.h

stdarg.h 定义特殊函数。
在 stdarg.h 中，定义了下列函数。

```
va_start, va_starttop, va_arg, va_end
```

在 stdarg.h 中，定义了下列对象。

- 声明指针型 "va_list" 为 char 型

```
typedef char    *va_list  ;
```

6.3.4 stdio.h

stdio.h 定义输入 / 输出函数。在 stdio.h 中，定义了下列函数。
然而，声明函数要根据选项和特定的模式而变化。

```
sprintf, sscanf, printf, scanf, vprintf, vsprintf, getchar, gets, putchar, puts, __putc
```

声明下列宏名称。

```
#define  EOF     ( -1 )
```

6.3.5 stdlib.h

stdlib.h 定义 字符 / 字符串函数，存储函数，程序控制，数学函数 , 和特殊函数。在 stdlib.h 中，定义了下列函数。

因此，当编译选项 -za （此选项用于关闭不符合 ANSI 标准的函数并打开部分符合 ANSI 标准的函数），则没有定义的 brk, sbrk, itoa, ltoa, 和 ultoa，在它们的位置上定义 strbrk, strsbrk, stritoa, strltoa, 和 strultoa。当没有设定 -za 时，则这些函数不可以定义。

```
atoi, atol, strtol, strtoul, calloc, free, malloc, realloc, abort, atexit, exit, abs, labs,
div, ldiv, brk, sbrk, atof, strtod, itoa, ltoa, ultoa, rand, srand, bsearch, qsort, strbrk,
strsbrk, stritoa, strltoa, strultoa
```

在 stdlib.h 中，声明了下列对象。

- 对带有 int 型成员 "quot" 和 "rem" 的结构体类型 "div_t" 做声明。

```
typedef struct {
        int     quot ;
        int     rem ;
} div_t ;
```

- 定义宏名称 "RAND_MAX"

```
#define RAND_MAX        32767
```

- 宏名称的声明

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

6.3.6 string.h

string.h 定义了 字符 / 字符串函数，存储函数，和特殊函数。

在 string.h 中，定义了下列函数。

然而，声明函数要根据选项和特定的模式而变化。

```
memcpy, memmove, strcpy, strncpy, strcat, strncat, memcmp, strcmp, strncmp, memchr, strchr,
strrchr, strspn, strcspn, strpbrk, strstr, strtok, memset, strerror, strlen, strcoll,
strxfrm
```

6.3.7 error.h

error.h 包含 errno.h。

6.3.8 errno.h

声明或定义下列对象

- 宏名称 "EDOM", "ERANGE", "ENOMEM" 的定义

```
#define EDOM    1
#define ERANGE  2
#define ENOMEM  3
```

- 可变 int 型外部变量 "errno" 的声明

```
extern volatile int errno ;
```

6.3.9 limits.h

在 limits.h 中 , 定义了下列宏名称。

```
#define CHAR_BIT        8
#define CHAR_MAX        +127
#define CHAR_MIN        -128
#define INT_MAX         +32767
#define INT_MIN         -32768
#define LONG_MAX        +2147483647
#define LONG_MIN        -2147483648

#define SCHAR_MAX       +127
#define SCHAR_MIN       -128
#define SHRT_MAX        +32767
#define SHRT_MIN        -32768
#define UCHAR_MAX       255U
#define UINT_MAX        65535U
#define ULONG_MAX       4294967295U
#define USHRT_MAX       65535U

#define SINT_MAX        +32767
#define SINT_MIN        -32768
#define SSHRT_MAX       +32767
#define SSHRT_MIN       -32768
```

然而，当设定了 -qu 选项时，则认为是不可修改的无符号 char 型，按照以下方式通过编译器和已声明的宏 __CHAR_UNSIGNED__ 来声明 CHAR_MAX 和 CHAR_MIN。

```
#define CHAR_MAX        ( 255U )
#define CHAR_MIN        ( 0 )
```

6.3.10 stddef.h

在 stddef.h 中 , 声明或定义下列对象。

- int 型 "ptrdiff_t 的声明

```
typedef int     ptrdiff_t ;
```

- 无符号 int 型 "size_t" 的声明

```
typedef unsigned int    size_t ;
```

- 定义宏名称 "NULL"

```
#define NULL    ( void * ) 0 ;
```

- 定义宏名称 "offsetof"

```
#define offsetof ( type, member )  ( (size_t) & (((type*)0) -> member) )
```

备注 offsetof (*类型*, *指定成员*)
扩展常用整数常量表达式类型 size_t, 从结构体 （设定类型）的开始到结构体成员 （设定指定成员），那个值是字节单位的偏移值。
当成员指定说明符已经对 t 作了静态类型的声明，评价表达式的结果 & （(t. 成员指定说明符）必须是地址常数。当设定的成员是位字段时，则没有操作保证。

6.3.11 math.h

在 math.h 中，定义了下列函数。

```
acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10,
modf, pow, sqrt, ceil, fabs, floor, fmod, matherr, acosf, asinf, atanf, atan2f, cosf, sinf,
tanf, coshf, sinhf, tanhf, expf, frexpf, ldexpf, logf, log10f, modff, powf, sqrtf, ceilf,
fabsf, floorf, fmodf
```

下列对象的定义。

- 定义宏名称 "HUGE_VAL"

```
#define HUGE_VAL        DBL_MAX
```

6.3.12 float.h

在 float.h 中，定义了下列对象。

当 double 型的大小为 32 位时，依据由编译器声明的宏 __DOUBLE_IS_32BITS__ 分隔定义的宏。

```
#ifndef _FLOAT_H

#define FLT_ROUNDS      1
#define FLT_RADIX       2

#ifdef  __DOUBLE_IS_32BITS__
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    24
#define LDBL_MANT_DIG   24

#define FLT_DIG         6
#define DBL_DIG         6
#define LDBL_DIG        6

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -125
#define LDBL_MIN_EXP    -125

#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -37
#define LDBL_MIN_10_EXP -37

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +128
#define LDBL_MAX_EXP    +128

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +38
#define LDBL_MAX_10_EXP +38

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         3.40282347E+38F
#define LDBL_MAX        3.40282347E+38F

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     1.19209290E-07F
#define LDBL_EPSILON    1.19209290E-07F

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         1.17549435E-38F
#define LDBL_MIN        1.17549435E-38F

#else   /* __DOUBLE_IS_32BITS__ */
```

```
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    53
#define LDBL_MANT_DIG   53

#define FLT_DIG         6
#define DBL_DIG         15
#define LDBL_DIG        15

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -1021
#define LDBL_MIN_EXP    -1021

#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -307
#define LDBL_MIN_10_EXP -307

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +1024
#define LDBL_MAX_EXP    +1024

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +308
#define LDBL_MAX_10_EXP +308

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         1.7976931348623157E+308
#define LDBL_MAX        1.7976931348623157E+308

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     2.2204460492503131E-016
#define LDBL_EPSILON    2.2204460492503131E-016

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         2.225073858507201E-308
#define LDBL_MIN        2.225073858507201E-308
#endif  /* __DOUBLE_IS_32BITS__ */

#define _FLOAT_H
#endif  /* !_FLOAT_H */
```

6.3.13 assert.h

在 assert.h 中，定义了下列函数。

```
__assertfail
```

在 assert.h 中，定义了下列对象。

```
#ifdef  NDEBUG
#define assert ( p )    ( ( void ) 0 )
#else
extern  int     __assertfail ( char *__msg, char *__cond, char *__file, int__line );
#define assert ( p )    ( ( p ) ?( void ) 0 : ( void ) __assertfail (
                            "Assertion failed : %s, file %s, line %d\n",
                        #p, __FILE__, __LINE__ ) )
#endif  /* NDEBUG */
```

然而， assert.h 头文件引用一个或更多宏，在 assert.h 头文件中没有定义 NDEBUG。当在源文件中包含 assert.h 时，如果 NDEBUG 作为宏来定义， 按如下显示对 assert 宏做简单定义 , 并且 __assertfail 也没有定义。

```
#define assert ( p )    ( ( void ) 0 )
```

6.4 重返

重返是种状态，它表示其它程序成功调用程序中函数是可能的。

78K0R C 编译器标准库具有重返功能而不是用静态区域。。因此，通过调用其它程序不会损坏在存储区域中通过函数来使用的数据。

因此 , 小心下列函数不能重返。

- 不能重返的函数。

```
setjmp, longjmp, atexit, exit
```

- 通过启动程序使用获取区域的函数。

```
div, ldiv, brk, sbrk, rand, srand, strtok
```

- 处理浮点指针号的函数

```
sprintf, sscanf, printf, scanf, vprintf, vsprintf 注
atof, strtod, all math functions
```

注 sprintf, sscanf, printf, scanf, vprintf, and vsprintf, 这些函数不支持重返的浮点数。

6.5 使用适合标准库的参数 / 返回值

根据存储模式，链接标准库中为参数 / 返回值指定指针的函数到相应的库中。

处理指针不是默认存储模式，通过调用以下带标准函数名称的函数，那个指针可能与相应的库相链接。

< 函数名称 >_n : : 总是作为 near 方式处理指针。

< 函数名称 >_n : : 总是作为 far 方式处理指针。

例如：当选中小模式时，far 指针能够用来设定 strcmp 函数的参数。

例子显示如下。

```
#include <string.h>


__far char * sf1;
__far char * sf2;


void main ( void ) {
          :
        r = strcmp_f ( sf1, sf2 );
          :
}
```

注意事项显示如下。

- 当设定了小模式和中模式时，输入 / 输出函数 sprintf/printf/vprintf/vsprintf/sscanf/scanf 的指针参数处理变量参数时，可以用 near 指针来处理。不能使用函数指针。
  当使用函数指针时，或使用 far 指针时，使用 printf_f 时必须定义所有变量参数指针为 far 指针。当设定了大模式时，输入 / 输出函数 sprintf/printf/vprintf/vsprintf/sscanf/scanf 的指针参数处理变量参数时，可以用 far 指针来处理
  。
- 当设定了小模式和中模式时，特殊函数 va_start/va_starttop/va_arg/va_end 的指针参数处理变量参数时，可以用 near 指针来处理。不能使用函数指针。

6.6　字符 / 字符串函数

以下的字符 / 字符串函数可用。

功能名称	作用
isalpha	判断字符是否是字母符号 (A 到 Z, a 到 z)
isupper	判断字符是否是大写字母符号 (A 到 Z)
islower	判断字符是否是小写字母符号 (a 到 z)
isdigit	判断字符是否是数字 (0 到 9)
isalnum	判断字符是否是字母数字式字符 (0 到 9, A 到 Z, a 到 z)
isxdigit	判断字符是否是十六进制数 (0 到 9, A 到 F, a 到 f)
isspace	判断字符是否是空白字符 (空格 , 制表符 , 回车 , 换行 , 垂直制表符 , 换页)
ispunct	判断字符是否是除空白字符或字母数字字符以外的打印字符。
isprint	判断字符是否是打印字符
isgraph	判断字符是否是除空白字符以外的打印字符
iscntrl	判断字符是否是控制字符
isascii	判断字符是否是 ASCII 码
toupper	将小写字母字符转换为大写
tolower	将大写字母字符转换为小写
toascii	转换输入为 ASCII 码
_toupper	从输入字符中去掉 "a" 并加上 "A"
toup	
_tolower	从输入字符中去掉 "A" 并加上 "a"
tolow	

isalpha

判断 *c* 是否是字母字符 (A 到 Z, a 到 z)

[语法]

```
#include <ctype.h>
int         isalpha ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在字母字符中 (A 到 Z 或 a 到 z) : 1 如果字符 *c* 不包含在字母字符中 (A 到 Z 或 a 到 z) : 0

[说明]

- 如果字符 *c* 包含在字母字符中 (A 到 Z 或 a 到 z), 则返回 1。
  其它情况下，返回 0。

isupper

判断 *c* 是否是大写字母字符 (A 到 Z)

[语法]

```
#include <ctype.h>
int         isupper ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在大写字母中 (A 到 Z) : 1 如果字符 *c* 不包含在大写字母中 (A 到 Z) : 0

[说明]

- 如果字符 *c* 包含在大写字母字符中 (A 到 Z)，则返回 1。
  其它情况下，返回 0。

islower

判断 *c* 是否是小写字母字符 (a 到 z)

[语法]

```
#include <ctype.h>
int         islower ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在小写字母中 (a 到 z) : 1 如果字符 *c* 不包含在小写字母中 (a 到 z) : 0

[说明]

- 如果字符 *c* 包含在小写字母中 (a 到 z)，则返回 1。
  其它情况下，返回 0。

isdigit

判断 *c* 是否是数字 (0 到 9)

[语法]

```
#include <ctype.h>
int         isdigit ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在数字字符中 (0 到 9) : 1 如果字符 *c* 不包含在数字字符中 (0 到 9) : 0

[说明]

- 如果字符 *c* 包含在数字字符中 (0 到 9), 则返回 1。
  其它情况下，返回 0。

isalnum

判断 *c* 是否是含字母和数字的字符 (0 到 9, A 到 Z, a 到 z)

[语法]

```
#include <ctype.h>
int         isalnum ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在含字母和数字的字符中 (0 到 9 和 A 到 Z 或 a 到 z) : 1 如果字符 *c* 不包含在含字母和数字的字符中 (0 到 9 和 A 到 Z 或 a 到 z) : 0

[说明]

- 如果字符 *c* 包含在含字母和数字的字符中 (0 到 9 和 A 到 Z 或 a 到 z), 则返回 1。
  其它情况下，返回 0。

isxdigit

判断 *c* 是否是十六进制数 (0 到 9, A 到 F, a 到 f)

[语法]

```
#include <ctype.h>
int         isxdigit ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在十六进制数中 (0 到 9 和 A 到 F 或 a 到 f) : 1 如果字符 *c* 不包含在十六进制数中 (0 到 9 和 A 到 F 或 a 到 f) : 0

[说明]

- 如果字符 *c* 包含在十六进制数中 (0 到 9 和 A 到 F 或 a 到 f), 则返回 1。
  其它情况下，返回 0。

isspace

判断 *c* 是否是空白字符 (空格 , 制表符 , 回车 , 换行 , 垂直制表符 , 换页)

[语法]

```
#include <ctype.h>
int         isspace ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在空白字符中 : 1 如果字符 *c* 不包含在空白字符中 : 0

[说明]

- 如果字符 *c* 包含在空白字符中 (空格 , 制表符 , 回车 , 换行 , 垂直制表符 , 换页)，则返回 1。
  其它情况下，返回 0。

ispunct

判断 *c* 是否是除空白字符或字母数字字符以外的打印字符

[语法]

```
#include <ctype.h>
int         ispunct ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在除空白字符或字母数字字符以外的打印字符中 : 1 如果字符 *c* 不包含在除空白字符或字母数字字符以外的打印字符中 : 0

[说明]

- 如果字符 *c* 包含在除空白字符或字母数字字符以外的打印字符中 , 则返回 1。
  其它情况下，返回 0。

isprint

判断 *字符 c* 是否是打印字符

[语法]

```
#include <ctype.h>
int         isprint ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在打印字符中 : 1 如果字符 *c* 不包含在打印字符中 : 0

[说明]

- 如果字符 *c* 包含在打印字符中，则返回 1。
  其它情况下，返回 0。

isgraph

判断 *c* 是否是除空白字符以外的打印字符

[语法]

```
#include <ctype.h>
int         isgraph ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在可打印的非空字符中 : 1 如果字符 *c* 不包含在可打印的非空字符中 : 0

[说明]

- 如果字符 *c* 包含在可打印的非空字符中，则返回 1。
  其它情况下，返回 0。

iscntrl

判断 *c* 是否是控制字符

[语法]

```
#include <ctype.h>
int         iscntrl ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在控制字符中 : 1 如果字符 *c* 不包含在控制字符中 : 0

[说明]

- 如果字符 *c* 包含在控制字符中，则返回 1。
  其它情况下，返回 0。

isascii

判断 *c* 是否是 ASCII 码

[语法]

```
#include <ctype.h>
int         isascii( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 判断字符	如果字符 *c* 包含在 ASCII 码中 : 1 如果字符 *c* 不包含在 ASCII 码中 : 0

[说明]

- 如果字符 *c* 包含在 ASCII 码中，则返回 1。
  其它情况下，返回 0。

toupper

将小写字母字符转换为大写

[语法]

```
#include <ctype.h>
int         toupper ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	如果 *c* 是可变换字符 : 与 *c* 的大写相当 如果没有变换： c

[说明]

- toupper 函数核对查看参数是否为小写字母且如果是，则变换字母到对应的大写字母。

tolower

将大写字母字符转换为小写

[语法]

```
#include <ctype.h>
int         tolower ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	如果 *c* 是可变换字符 : *c* 对应的大写字母 如果没有变换： c

[说明]

- tolower 函数检查参数是否为大写字母，如果是，则将该字母转换成对应的小写字母。

toascii

转换输入为 ASCII 码

[语法]

```
#include <ctype.h>
int         toascii ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	转换各位直到超出 "*c*" 到 0 的 ASCII 码范围，并获取该值。

[说明]

- toascii 函数转换 "*c*" 的各位 (第 7 到 15 位) 直到超过 "*c*" (第 0 到 6 位) 到 "0" 的 ASCII 码范围，并返回转化后的位值。

_toupper

"*c*" 减去 "a" 并加 "A" 的结果
(_toupper 与 toup 非常相似)

备注 a: 小写字母 ; A: 大写字母

[语法]

```
#include <ctype.h>
int     _toupper ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	"*c*" - "a" 的减法结果加上 "A"，获取该值。

备注 a: 小写字母 ; A: 大写字母

[说明]

- _toupper 函数与 toup 相似，除了其不能测试查看参数是否是小写字母。

toup

"*c*" 减去 "a" 并加 "A" 的结果
(_toupper 与 toup 非常相似)

备注　　a: 小写字母 ; A: 大写字母

[语法]

```
#include <ctype.h>
int         toup ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	"*c*" - "a" 的减法结果加上 "A"，获取该值。

备注　　a: 小写字母 ; A: 大写字母

[说明]

- toup 函数与 _toupper 相似，除了其能测试查看参数是否是小写字母。

_tolower

"*c*" 减去 "A" 并加 "a" 的结果
(_tolower 与 tolow 非常相似)

备注 a: 小写字母 ; A: 大写字母

[语法]

```
#include <ctype.h>
int     _tolower ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	"*c*" - "A" 的减法结果加上 "a"，获取该值。

备注 a: 小写字母 ; A: 大写字母

[说明]

- _tolower 函数与 tolow 相似，除了其不能测试查看参数是否是大写字母。

tolow

"*c*" 减去 "A" 并加 "a" 的结果
(_tolower 与 tolow 非常相似)

备注　　a: 小写字母 ; A: 大写字母

[语法]

```
#include <ctype.h>
int         tolow ( int c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 转换字符	"*c*" - "A" 的减法结果加上 "a"，获取该值。

备注　　a: 小写字母 ; A: 大写字母

[说明]

- tolow 函数与 _tolower 相似，除了其能测试查看参数是否是大写字母。

6.7　程序控制函数

以下程序控制函数可用。

功能名称	作用
setjmp	在调用时保存环境
longjmp	用 setjmp 恢复保存过的环境

setjmp

在调用时保存环境

[语法]

#include <setjmp.h>
int setjmp (jmp_buf *env*) ;

[参数 / 返回值]

参数	返回数值
env : 保存数组的环境信息	如果直接调用： 0 如果从相应的 longjmp 返回： 如果 "*val* " 为 0，值为 1 或由 "*val*" 给定。

[说明]

- 当直接调用 setjmp 时 , 保存 saddr 区域 , SP 以及用作 HL 寄存器的函数返回地址或 *env 寄存变量* ，并返回 0。

longjmp

用 setjmp 恢复保存过的环境

[语法]

#include <setjmp.h>
void longjmp (jmp_buf *env* , int *val*) ;

[参数 / 返回值]

参数	返回数值
env : 由 setjmp 保存数组的环境信息 *val* : 返回值到 setjmp	longjmp 将不返回，因为程序执行重新恢复 stjmp 后的状态并保存环境到 "*env*"。

[说明]

- longjmp 恢复保存过的环境到 *env* (HL 寄存器 , saddr 区域和 当作寄存变量使用的 SP)。程序执行继续就如相应的 setjmp 返回 *val* (然而 , 如果 *val* 为 0, 返回 1)。

6.8　特殊函数

以下特殊函数可用。

功能名称	作用
va_start	为处理可变参数做设置
va_starttop	为处理可变参数做设置
va_arg	处理可变参数
va_end	显示处理可变参数的结束

va_start

为处理可变参数做设置 (宏)

[语法]

#include <stdarg.h>
void va_start (va_list *ap* , *parmN*) ;

备注 在 stdarg.h 中使用 typedef 定义 va_list。

[参数 / 返回值]

参数	返回数值
ap : 为了在 va_arg 和 va_end 中使用，初始化变量 *parmN* : 在可变参数之前的参数	None

[说明]

- 在 va_start 宏里 , 它的参数 *ap* 必须是 va_list 类型 (char* type) 对象。
- 指针指向的下一个参数 *parmN* 存储在 *ap* 中。
- *parmN* 是在函数原型中设定的最后 (最右) 一个参数的名称。
- 如果 *parmN* 是寄存器存储类 , 则不能保证这个函数的正确运行。
- 如果 *parmN* 是第一个参数 , 通常这个函数不可以运行 （使用 va_starttop 代替）。

va_starttop

为处理可变参数做设置 (宏)

[语法]

```
#include <stdarg.h>
void  va_starttop ( va_list ap , parmN ) ;
```

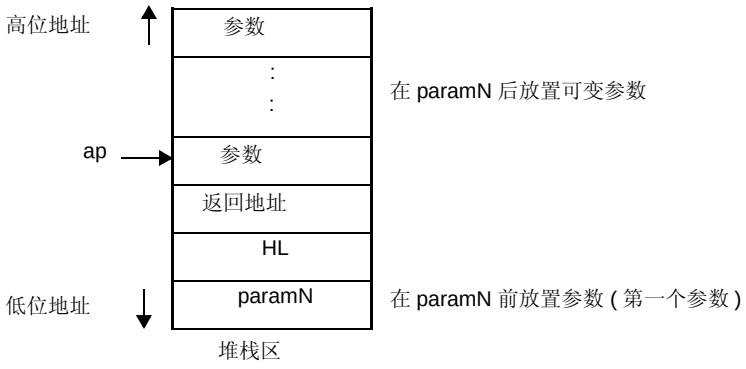
备注　　在 stdarg.h 中使用 typedef 定义 va_list。

[参数 / 返回值]

参数	返回数值
ap : 为了在 va_arg 和 va_end 中使用，初始化变量 *parmN* : 在可变参数之前的参数	None

[说明]

- *ap* 必须是 va_list 类型对象。
- 指针指向的下一个参数 *parmN* 存储在 *ap* 中。
- *parmN* 是在函数原型中设定的最右第一个参数的名称。
- 如果 *parmN* 是寄存器存储类 , 通常这个函数不能运行。
- 如果 *parmN* 是除第一个参数以外的参数，通常这个函数不能运行。



va_arg

处理可变参数 (宏)

[语法]

#include <stdarg.h>
type va_arg (va_list *ap* , type) ;

备注 在 stdarg.h 中使用 typedef 定义 va_list。

[参数 / 返回值]

参数	返回数值
ap : 处理参数列表的变量 type : 输入指向可变参数的相关位置 (输入可变长度的类型 ; 例如 , 如果 va_arg (va_list *ap*, int) 或如果用 long 型描述 va_arg (va_list *ap*, long))	正常情况： 在可变参数相关位置的值 如果 *ap* 是空指针 : 0

[说明]

- 在 va_arg 宏里 , 它的参数 *ap* 一定与 va_list 和 va_start 初始化的类型对象相同 (不能保证其他正常运行)。
- va_arg 返回在输入类型为可变参数中的相关位置上的数值。
  第一个可变参数的相关位置放在 va_start 之后并在每个 va_arg 中继续进行。
- 如果参数指针 *ap* 是空指针 , 则 va_arg 返回 0 (类型输入)。
- 使用 78K0R C 编译器 , 在设定用作参数列表的指针时 , 如果使用中模式，则必须设定 near 数据指针 (2- 字节长度), 并且当使用中模式时，当使用长模式时，必须设定 far 数据指针 （4 字节长度）。
  在两个模式中函数指针长度固定为 4 个字节，但当设定指针用作参数列表时，在每个模式中的指针长度必须设定为 2 或 4 个字节长度。

va_end

显示处理可变参数的结束 (宏)

[语法]

```
#include <stdarg.h>
void  va_end ( va_list ap ) ;
```

备注　　在 stdarg.h 中使用 typedef 定义 va_list。

[参数 / 返回值]

参数	返回数值
ap : 变量用于处理可变参数数量	None

[说明]

- va_end 宏用参数指针 *ap* 设置空指针，并通知宏处理程序 - 已经处理了在可变参数中所有的参数。

6.9 输入和输出函数

以下输入和输出函数可用。

功能名称	作用
sprintf	根据格式在字符串中写入数据
sscanf	根据格式从输入字符中读取数据
printf	根据格式输出数据到 SFR 中
scanf	根据格式从 SFR 中读取数据
vprintf	根据格式输出数据到 SFR 中
vsprintf	根据格式在字符串中写入数据
getchar	从 SFR 中读取一个字符
gets	读取字符串
putchar	输出一个字符到 SFR 中
puts	输出字符串
__putc	输出一个字符到 opaque

sprintf

根据格式在字符串中写入数据

[语法]

#include <stdio.h>
int sprintf (char *s , const char *format , ...) ;

[参数 / 返回值]

参数	返回数值
s : 指向字符串指针输出的写法 格式 : 指向字符串指针的格式命令表示。 ... : 转换 0 或多个参数	在 s 中写入多个字符 (不对终止符计数。)

[说明]

- 如果实参数比格式数少，则不能保证正常运行。尽管格式失效，实参仍保留，在这种情况下，只能评估并忽略多余的实参。
- 按照由格式命令设定的格式， sprintf 转换 0 或更多参数并把它们写 (复制) 到字符串内。
- 可以使用零个或更多格式命令。照样输出常规字符 (除在格式命令开头加 % 字符以外) 到字符串 s 中。每个格式命令有零个或更多符合格式的参数 并输出它们到字符串 s 内。
- 每个格式命令的开头带有 % 字符且后面有如下这下：

(1) 零个或更多标志 （以后会介绍）修改格式命令的含义。

(2) 用任意十六进制来设定最小的字段宽度

如果在转换以后输出宽度小于最小字段宽度，则这个标识符在输出区域的左侧填有零个空格。 (如果在 % 符号后面接着左对齐标志 "-" (减), 在输出区的右侧填满零。) 默认用空格填充。如果输出区是用多个 0 来填充，则在字段宽度标识符前放 0。如果数字或字符的宽度要比最小字段宽度要宽，则在超出最小值时全数打印它。

- 任意精度 （多个小数位）说明 （整数）
  用 d, i, o, u, x, 和 X 类型标识符 , 设定最小的数字个数。
  用 s 类型标识符 , 设定最大字符数 (最大字段宽度)。
  输出多个带小数点的数字，指定用 e, E, 和 f 转换来设定小数点。使用 g 和 G 转换指定最大有效数字。
  这个精度规范必须以 (.integers) 形式构成。如果省略整数部分 , 假设已经指定为 0。
  使用这个精度规范的大量填充结果取代使用字段宽度规范的填充。
- 任意 h, l 和 L 修饰符
  h 修饰符命令 sprintf 函数进行 d, i, o, u, x, 或 X 类型转换，在 short int 或 unsigned short int 类型上附加这个修饰符。 h 修饰符命令 sprintf 函数进行 n 类型转换，在指向 short int 类型的指针上附加这个修饰符。
  l 修饰符命令 sprintf 函数进行 d, i, o, u, x, 或 X 类型转换，在 long int 或 unsigned long int 类型上附加这个修饰符。 h 修饰符命令 sprintf 函数进行 n 类型转换，在指向 long int 类型的指针上附加这个修饰符。
  如果有其他类型修饰符 , 则忽略 h, l 或 L 修饰符。

- 指定转换的字符 (以后会介绍)

在最小字段宽度中或精度 (多个小数位) 规范 , * 可以用在整数型字符串的位置上。在这种情况下 , 通过 int 参数传递整数值 (在转换参数之前)。

任何负数字段宽度是在 -(减) 标志之后加上正数字段的结果。将忽略所有负数的精度。

以下标志用于修改格式命令：

标志	内容
-	转换的结果是在字段内右对齐。
+	符号转换的结果总是在开头加上 + 或 - 符号。
空格	如果符号转换的结果没有符号，则在输出区前置空格。 如果同时设定 +（加）标志和空格标志，则忽略空格标志。
#	结果转换在 " 分配表格 " 里。 在 o 类型转换中，为了第 1 个数字为 0，提高精度。 在 x 或 X 类型转换中，在非零结果前前置 0x 或 0X。 在 e, E, 和 f 类型转换中，小数点强制插入到所有输出值中 （默认时不含 #，在有数值在后面时，只显示小数点）。 在 g 和 G 类型转换中 , 小数点强制插入到所有输出值中 , 且不允许舍去 0(默认时不含 #，在有数值在后面时，只显示小数点。将舍去接下去的 0)。 在所有其他转换中，忽略 # 标志。
0	结果是在左边用零代替空格填充。 当 0 标志同 "-" 标志一起设定，忽略 0 标志。 在设定精度时，在 in d, i, o, u, x, 和 X 转换中忽略 0 标志。

输出转换规范的格式代码如下：

格式代码	内容
d, i	转换 int 参数为带符号十进制格式。
o	转换 int 参数为带符号十进制格式。
u	转换 int 参数为无符号八进制格式。
x	转换 int 参数为无符号十六进制格式 （用小写字母 abcdef)。
X	转换 int 参数为无符号十六进制格式 （用大写字母 ABCDEF)。

用 d, i, o, u, x, 和 X 类型标识符 , 设定结果数字的最小个数 (最小字段宽度）。如果输出区比最小字段宽度更短，则用零来填充。

如果没有设定精度，则假定已经设定 1。

如果 0 精度转换 0，则没有东西不会出现。

代码精度	内容
f	转换双参数为带 [-] *dddd*.*dddd* 格式的符号值。 *dddd* 是一个或多个十进制数 (s)。数字的绝对值由小数点前数字的个数决定，所需的精度由小数点后的数字个数决定。 当省略精度时，则它认为是 6。
e	转换双参数为带 [-] *d*.*dddd* e [sign] *ddd* 格式的符号值。 d 是一个十进制数 , 且 *dddd* 是一个或多个十进制数 (s)。 *ddd* 是正好 3 个数字的十进制数 , 且符号为 + 或 -。 当省略精度时，则它认为是 6。
E	与 e 的格式相同，除在指数之前添加 E 来代替 e 以外。
g	当基于指定精度的双参数转换时，使用 f 的无论哪个 shorter 方法或 e 格式。 e 格式只能使用在数值的幂小于 -4 或大于指定精度的数字。 把以下的 0 去尾，只有在一个或更多数字接着的时候显示小数点。
G	与 g 的格式相同，除在指数之前添加 E 来代替 e 以外。
c	转换 int 参数为无符号 char，且使用单字符编写结果。
s	关联参数是指向字符串的指针，且写字符串的字符直到终止符 （而不是包含在输出中）。 如果指定精度，将超过最大字段宽度的字符串截断末尾。 当没有设定精度或大于数组，则数组必须包含空字符。
p	关联参数是指向 void 的指针，且用无符号十六进制 4 位数显示指针指 (少于 4 位数的指针变量前面加 0)。 使用 无符号十六进制 8 位数显示大模式 (在 2- 位数中填充 0 且在少于 6 位数的指针值前加 0)。 如果省略精度设定。
n	关联参数是指向 integer 的指针，且以 far 方式在字符串 "s" 中写入字符数量。 不进行转换。
%	Prints a % sign. 不转换关联参数 (但标志和最小字段宽度的设定有效）。

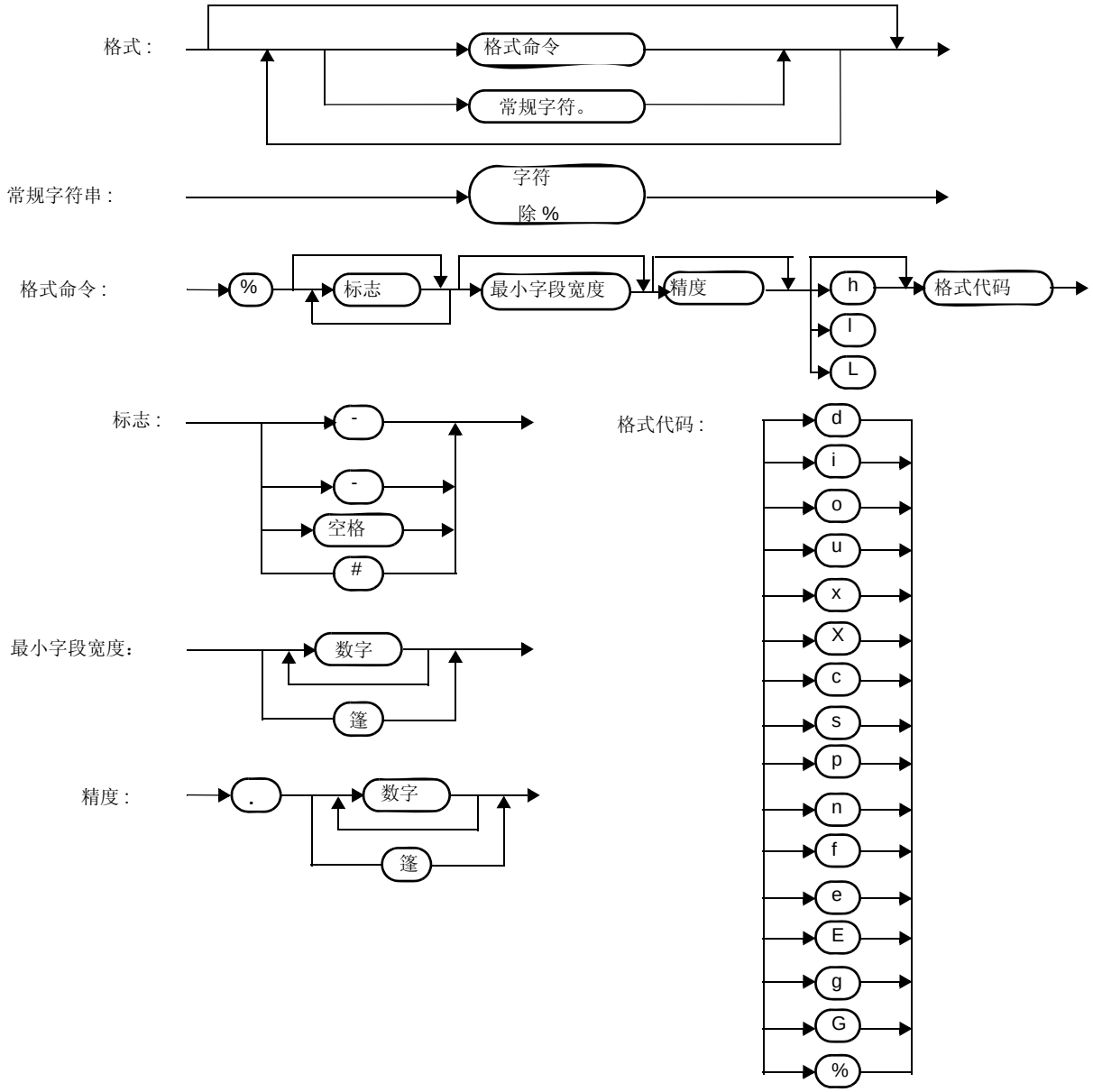
- 不能保证无效转换指令操作。
- 当实参数十联合体 union 或结构体 structure，或指向显示它们的指针 （除用 % 进行转换的字符类型数组或用 %p 转换的指针），不能保证其操作。
- 即使在无字段宽度或字段宽度小的情况下，转换结果将不能截断。换句话说，在转换结果的字符数量大于字段宽度时，该字段扩展宽度，使其宽度适合包含转换结果。
- 用 %f, %e, %E, %g, %G 转换的特殊输出字符串格式显示如下。

  non-numeric -> "(NaN)"
  + -> "(+INF)"
  - -> "(-INF)"

  sprintf 在字符串 *s* 末尾写入空字符。 (这个字符串包含在返回值的计数中。)

*格式命令*的语法描述如下。

图 6-2. 格式命令*的语法*



- 使用 78K0R C 编译器 , 在使用中模式时， near 数据指针 (2 字节长度) 必须设定 , 且在使用大模式为转换指令 s, p, 和 n 设定参数指针时， far 数据指针 (4 字节长度) 必须设定。
  在两个模式中函数指针长度固定为 4 个字节，但当使用指针作为参数时，在每个模式中的指针长度必须设定为 2 或 4 个字节长度。

sscanf

根据格式从输入字符中读取数据

[语法]

```
#include <stdio.h>
int         sscanf ( const char *s , const char *format , ... ) ;
```

[参数 / 返回值]

参数	返回数值
s : 指向输入字符串的指针 *格式* : 指向字符串指针显示输入格式命令。 ... : 指向目标的指针转换值以及零或更多参数保存在目标中	如果字符串 *s* 是空的 : -1 如果字符串 *s* 不是空的 : 已分配的输入数据条目数

[说明]

- sscanf 输入数据到 *s* 指向的字符串。*格式指定* 允许输入的输入字符串指向的字符串。
  在格式用作指向目标指针之后，零个或更多参数。从输入字符串中如何转换为指定格式的数据。
- 如果没有足够的参数与指向格式命令的格式相匹配 , 不能保证编译器正常操作。多余参数，表达式将进行计算且无数据输入。
- *格式的指向控制字符串指针* 包含零个或更多格式命令，格式命令分为以下 3 种类型 :
  1　: 空白字符 (一个或更多字符时 isspace 为 true)
  2　: 非空白字符 (除 % 以外)
  3　: 格式指令
- 各开头带有 % 字符的格式指令以及这些内容接在后面：

(1) 可选 * 字符禁止分配数据到相应的参数中

(2) 可选十进制整数来设定最大的字段宽度

(3) 可选 h, l 或 L 修饰符来表示接收端的目标大小

如果 h 在格式指令 d, i, o, 或 x 之前 , 则参数是指向 short int 的指针而不是指向 int。如果 l 在这些格式指令之前 , 则参数是指向 long int 的指针。

同样如果格式指令 h 在 u 之前 , 则参数是指向无符号 short int 的指针。同样如果格式指令 l 在 u 之前 , 则参数是指向无符号 long int 的指针。

如果 l 在转换指令 e, E, f, g, G 之前 , 则参数是指向 double 的指针 (不使用 l 时默认为指向 float 的指针)。如果 L 优先 , 则忽略它。

备注　　转换指令：用相应转换类型表示字符 （后面会提到）

- sscanf 按 " *格式* " 顺序执行格式命令且如果任何格式命令失败，则中止该函数。

(1) 在控制字符串中的空白字符使 sscanf 读取空白字符数 （包括零）直到第一个非空白字符 （将不读它)。如果它没有遇到任何非空白字符，则这个空白字符串命令失败。

(2) 非空白字符使 sscanf 读取和丢弃匹配字符。如果没有找到指定的字符，则这条命令失败。

(3) 格式命令用各类型指令定义输入流的集合 （后面会详细介绍）。
按照以下步骤执行格式命令：

(a) 跳过输入空白字符 （由 isspace 来设定），除在类型指令为 [, c, or n。

(b) 从字符串 "s" 中读取输入数据项 , 除类型指令为 n 时。
类型指令定义输入数据项为字符串中第 1 部分流的最长输入流。接着的字符解释为还没有读取的输入数据项。
如果输入数据项的长度为 0，则格式命令执行失败。

(c) 通过除 % 以外的类型指令来转换输入数据项 （带类型指令 n 的输入字符个数）为指定类型。
如果输入数据项与指定类型不匹配，则命令执行失败。
除非赋值受到 * 的限制，转换结果存储在第一个参数指向的目标里，并符合 " *格式* " 且仍没有接受到转换的结果。

以下类型指令可用。

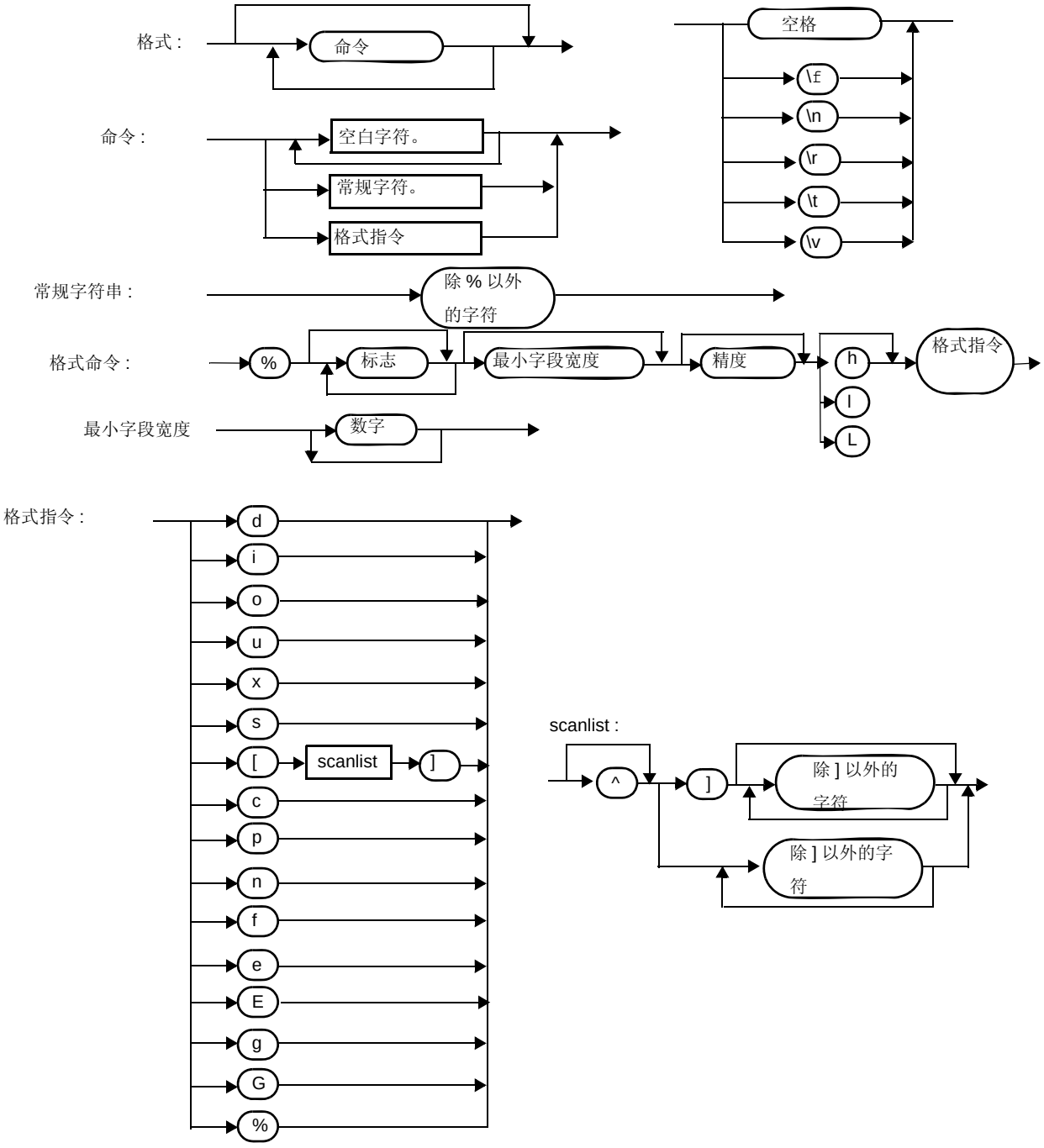
转换指令	内容
d	转换十进制整数 （可以有符号）。 相应参数必须指向整数。
I	转换整数 （可以有符号）。 如果数字前置了 0x 或 0X，则数字解释为十六进制整数。如果数字前置 0, 则数字解释为八进制整数。其他数字作为十进制整数。 相应参数必须指向整数。
o	转换八进制整数 （可以有符号）。 相应参数必须指向整数。
u	转换无符号十进制整数。 相应参数必须指向无符号整数。
x	转换十六进制整数 （可以有符号）。
e, E, f, g, G	浮点数包含可选符号 （+ 或 -），一个或更多个连续的十进制数包含小数点，可选指数 （e 或 E)，以及以下可选的符号整数值。 在转换结果产生溢出时 , 或在转换结果产生下溢时 ± , 转换结果会变为非正常化数字或 ± 0。 相应参数指向浮点数。

转换指令	内容
s	输入包含非空白字符串的字符串。 相应参数指向整数。 0x 或 0X 能分配在第一个十六进制整数上。 相应参数必须是指向数组的指针，其有足够的大小存放这个字符串和 null 中止符。 将自动添加 null 中止符。
[	输入包含需要字符组的字符串 （调用 scanset)。 相应参数必须是指向数组第 1 字符串的指针，其有足够的大小存放这个字符串和 null 中止符。 将自动添加 null 中止符。 从这个字符开始直到关闭方括号 （）的格式命令继续。 . 这个字符串 (调用 scanlist) 包含在方括号中组成 scanset，除在字符直接在打开的方括号之后为音调符。当字符是音调符 , 除 scanlist 在音调符和关闭的方括号间以外的所有字符串组成 scanset。然而，在 scanlist 开头带 [ ] 或 [^], 这个闭方括弧包含在 scanlist 中，且下一个闭方括弧出现在 scanlist 的末尾。 连字符 (-) 在不同于 scanlist 的左边或右边末尾，其当作连字符标点符号标记，如果在字符范围左侧设定连字符 （-）不小于字符右侧的 ASCII 码值。
c	输入的字符串包含由字段宽度来指定的字符数。（如果省略设定的字段宽度，则假设为 1。） 相应参数必须是指向数组第 1 字符串的指针，其有足够的大小存放这个字符串。 将不添加 null 中止符。
p	读取无符号十六进制整数。 相应参数必须指向 void 指针。
n	不从字符串 *s* 中输入。 相应参数必须指向整数。目前从 "*s*" 中通过这个函数读取的多个字符存储在这个指针指向的目标中。 %n 格式名不包含于返回赋值计数。
%	读取 % 符号。 转换和赋值两者都不发生。

如果格式设定无效，则格式命令执行失败。
如果在输入流中出现 null 中止符，则 sscanf 将中止。
如果在整数转换中产生溢出 （使用 d, i, o, u, x, or p 格式指令），在转换后根据数据类型的位数来截取高位。

*格式命令*的语法描述如下。

图 6-3. 格式命令*的语法*



- 使用 78K0R C 编译器 , 在使用中模式时， near 数据指针 (2 字节长度) 必须设定 , 且在使用大模式为转换指令 s, p, 和 n 设定参数指针时， far 数据指针 (4 字节长度) 必须设定。
  在两个模式中函数指针长度固定为 4 个字节，但当使用指针作为参数时，在每个模式中的指针长度必须设定为 2 或 4 个字节长度。

printf

根据格式输出数据到 SFR 中

[语法]

#include <stdio.h>
int printf (const char *_format_ , ...) ;

[参数 / 返回值]

参数	返回数值
格式 : 指针指向的字符串表示输出 ... : 转换 0 或多个参数	字符数输出到 *s* (末尾的空字符不计算在内)

[说明]

- (0 个或更多) 转换带有*格式*的参数并使用 putchar 函数输出 , 根据转换设定的格式输出。
- 输出转换的设定为 0 或更多指令。使用 putchar 函数输出常规字符 (除转换设定的开头带 %) 通过提取和转换以下 （0 个或多个）参数并使用 puthcar 函数输出转换设定。
- 各转换的规范与 sprintf 函数相同。

scanf

根据格式从 SFR 中读取数据

[语法]

```
#include <stdio.h>
int         scanf ( const char *format , ... ) ;
```

[参数 / 返回值]

参数	返回数值
格式 : 指针指向的字符串表示输入转换格式。 ... : 分配转换值给指针 （0 或多个）指向目标的参数	在字符串 *s* 不为空时 : 分配输入条目数

[说明]

- 使用 getchar 函数进行输入。通过字符串格式设定允许的输入字符串 。在 *带格式的指针指向目标后使用参数* 。*格式设定如何对输入的字符串进行转换*。
- 在没有足够带格式的参数时 , 不能保证正常操作。在参数过多时，将计算表达式而不输入。
- *格式*包含 0 条或更多指令。指令如下。
  1 ： 一个或更多空字符 （字符使 isspace 为真）
  2 ： 空白字符 (除 % 以外)
  3 ： 转换的表示
- 如果转换输入字符的末尾，这样的转换会与输入字符产生冲突，冲突的输入字符被舍去。转换的表示与 sscanf 函数的表示相同。

vprintf

根据格式输出数据到 SFR 中

[语法]

#include <stdio.h>
int vprintf (const char **format* , va_list *p*) ;

[参数 / 返回值]

参数	返回数值
格式 : 指针指向的字符串表示输出转换的格式 *p* : 指向参数列表的指针	输出字符数 (末尾的空字符不计算在内)

[说明]

- 指针指向的参数列表表示已转换的参数，且使用 putchar 函数根据输出转换的格式输出。
- 各转换格式与 sprintf 函数相同。

vsprintf

根据格式在字符串中写入数据

[语法]

#include <stdio.h>
int vsprintf (char *s , const char *format , va_list p) ;

[参数 / 返回值]

参数	返回数值
s : 指针指向的字符串写入输出区 *格式* : 指针指向的字符串表示输出转换的格式 *p* : 指向参数列表的指针	字符数输出到 *s* (末尾的空字符不计算在内)

[说明]

- 指向参数列表的指针表示写入参数到字符串 *s* 中，并根据指定格式的转换格式后输出。
- 输出格式与 sprintf 函数相同。

getchar

从 SFR 中读取一个字符

[语法]

```
#include <stdio.h>
int         getchar ( void ) ;
```

[参数 / 返回值]

参数	返回数值
None	从 SFR 中读取字符

[说明]

- 从 SFR 符号 P0(端口 0）中读取返回值。
- 错误检验与不能读取有关。
- 更改 SFR 为读取，既有必要在库中对源代码进行再注册，也有必要由用户来创建新的 getchar 函数。

gets

读取字符串

[语法]

```
#include <stdio.h>
char  *gets ( char *s ) ;
```

[参数 / 返回值]

参数	返回数值
s : 指向输入字符串	正常 : s 如果在文件的末尾没有读取字符 : 空指针

[说明]

- 使用 getchar 函数读取字符串并存储在 *s* 表示俄数组中。
- 在检测文件末尾时 (getchar 函数返回 -1) 或在读取到换行符时，结束字符串的读取。禁止读取换行符，且最后编写在字符串末尾的空字符存储在数组中。
- 当返回值正常时 , 它返回 *s*。
- 当检测文件的结尾，且在数组中没有读取字符，则数组的内容保持不变，且返回空指针。

putchar

输出一个字符到 SFR 中

[语法]

```
#include <stdio.h>
int         putchar ( int  c ) ;
```

[参数 / 返回值]

参数	返回数值
c : 输出字符	已经输出的字符

[说明]

- *c* 指定的字符写入到 SFR 符号 P0 (端口 0) 中 (转换为无符 char 型)。
- 错误检验与不能写入有关。
- 更改 SFR 为写入，既有必要在库中对源代码进行再注册，也有必要由用户来创建新的 putchar 函数。

puts

输出字符串

[语法]

```
#include <stdio.h>
int         puts ( const char  *s ) ;
```

[参数 / 返回值]

参数	返回数值
s : 指向输出字符串的指针	正常 : 0 当 putchar 函数返回 -1 时： -1

[说明]

- 使用 putchar 函数写入由 *s* 表示的字符串 , 在输出区的末尾添加换行符。
- 不在字符串的末尾写入空字符。
- 在返回值正常时，则返回 0，且在 putchar 函数返回 -1 时，返回 -1。

__putc

输出一个字符到 opaque

[语法]

```
#include <stdio.h>
int         __putc ( int c , void *opaque ) ;
```

[参数 / 返回值]

参数	返回数值
c : 输出字符 *opaque :* 输出指针指向的字符串到目标	已经输出的字符

[说明]

- __putc 函数写由 *c* 设定的字符串 (通过转换它为无符号 char 型) 到 *opaque* 表示的目标中。由 *opaque* 表示的目标增加 1 个字节。
- 如果 *opaque* 为 0，则调用 putchar 函数并返回 putchar 函数的返回值。

6.10　公用函数

以下公用函数可用。

功能名称	作用
atoi	十进制整数字符串转换为整数型
atol	十进制整数字符串转换为长型
strtol	转换字符串为长型
strtoul	转换字符串为无符号长型
calloc	分配数组区间并初始化为 0
free	释放已分配的内存块
malloc	分配块
realloc	重新分配块
中止	异常中止程序
atexit	在正常终止时，注册调用的函数
退出	终止程序
abs	获取整数型值的绝对值
labs	获取 long 型值的绝对值
div	进行 int 型除法运算，获取商数和余数
ldiv	进行 long 型除法运算，获取商数和余数
brk	设置中断值
sbrk	增加 / 减少中断值
atof	十进制整数字符串转换为 double 型
strtod	转换字符串为 double 型
itoa	转换 int 型 为字符串
ltoa	转换 long 型为字符串
ultoa	转换无符号 long 型为字符串
rand	生成伪随机号
srand	初始化伪随机号的生成状态
bsearch	二进制查找
qsort	快速排序
strbrk	设置中断值
strsbrk	增加 / 减少中断值
stritoa	转换 int 型 为字符串
strltoa	转换 long 型为字符串
strultoa	转换无符号 long 型为字符串

atoi

十进制整数字符串转换为整数型

[语法]

#include <stdlib.h>
int atoi (const char **nptr*) ;

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串	如果适当地转换： int 型值 如果产生正溢出： INT_MAX (32,767) 如果产生负溢出： INT_MIN (-32,768) 如果字符串无效： 0

[说明]

- atoi 函数转换由指向 int 型值的指针 *nptr* 指向字符串的第一部分。
  atoi 函数从字符串的开头跳过零或多个空白字符 （当 isspace 变成 true 时）以及转换字符串从下一个跳过的空白字符到整数 （直到数字和空字符出现在字符串以外）。如果在字符串中没有发现可转换的数字，函数返回 0。
  如果产生溢出，由于正溢出函数返回 INT_MAX (32,767) 以及由于负溢出函数返回 INT_MIN (-32,768)。

atol

十进制整数字符串转换为长型

[语法]

```
#include <stdlib.h>
long int atol ( const char *nptr ) ;
```

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串	如果适当地转换： long int 值 如果产生正溢出： LONG_MAX (2,147,483,647) 如果产生负溢出： LONG_MIN (-2,147,483,648) 如果字符串无效： 0

[说明]

- atol 函数转换由指向 long int 型值的指针 *nptr* 指向字符串的第一部分。
  atol 函数从字符串的开头跳过零或多个空白字符 （它的 isspace 变成 true）以及转换字符串从下一个跳过的空白字符到整数 （直到数字和空字符出现在字符串以外）。如果在字符串中没有发现可转换的数字，函数返回 0。
  如果产生溢出，由于正溢出函数返回 LONG_MAX (2,147,483,647) 以及由于负溢出函数返回 LONG_MIN (-2,147,483,648) 。

strtol

转换字符串为长型

[语法]

#include <stdlib.h>
long int strtol (const char **nptr* , char ***endptr* , int *base*) ;

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串 *endptr* : 指针存储指向不可识别块的指针。 *基* : 在字符串中表示的基数。	如果适当地转换： long int 值 如果产生正溢出： LONG_MAX (2,147,483,647) 如果产生负溢出： LONG_MIN (-2,147,483,648) 如果没有转换： 0

[说明]

- strtol 函数通过指针 *nptr* 指向以下 3 个部分分解字符串的指向：

(1) 字符串中的空白字符可以为空 （用 isspace 来设定）

(2) 通过以基数值决定的 *基来表示整数*

(3) 字符串中一个或更多字符不能识别 （包含 null 终止符）

备注 strtol 函数将字符串中的部分转换为整数并返回这个整数值。

- *以* 0 为基数表示基准点应该从字符串的首个数字算起 。以 0x 或 0X 开头表示十六进制数 ; 以 0 开头表示八进制数 ; 否则该数字解释为十六进制数。 (在这种情况下 , 数字可以带正负号)。
- 如果 *基数为* 2 到 36, 设置从 a 到 z 或 A 到 Z 字母成为数字的一部分 (它们可以带正负号)，这样这些基数可以用来表示 10 到 35。
  如果基数为 16 则省略开头的 0x 或 0X。
- 如果 *endptr* 不是空指针 , 通过 *endptr* 指向字符串的部分 (3) 保存在指向的对象中。
- 如果修正值导致溢出，根据正负符号和在 ERANGE (2) 中设置的 errno, 函数返回正溢出的 LONG_MAX (2,147,483,647) 或负溢出的 LONG_MIN (-2,147,483,648)。
- 如果字符串 (2) 为空或字符串中首个非空白字符不适合于给定基数的整数 , 则函数不进行转换并返回 0。在这种情况下，通过 endptr 字符串的值 *nptr* 存储在指向对象中 (如果它不是空字符串)。基数 0 和 2 到 36 保持为真。

strtoul

转换字符串为无符号长型

[语法]

#include <stdlib.h>
unsigned long int strtoul (const char **nptr* , char ***endptr* , int *base*) ;

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串 *endptr* : 指针存储指向不可识别块的指针。 *基* : 在字符串中表示的基数。	如果适当地转换： 无符号 long 型 如果产生溢出 : ULONG_MAX (4,294,967,295U) 如果没有转换： 0

[说明]

- strtoul 函数通过指针 *nptr* 指向以下 3 个部分分解字符串的指向：

(1) 字符串中的空白字符可以为空 （用 isspace 来设定）

(2) 通过以基数值决定的 *基来表示整数*

(3) 字符串中一个或更多字符不能识别 （包含 null 终止符）

备注 strtoul 函数将字符串中的部分 (2) 转换为无符号整数并返回这个无符号整数值。

- *以* 0 为基数表示基准点应该从字符串的首个数字算起 。以 0x 或 0X 开头表示十六进制数 ; 以 0 开头表示八进制数 ; 否则该数字解释为十六进制数。
- 如果 *基数为* 2 到 36, 设置从 a 到 z 或 A 到 Z 字母成为数字的一部分 (它们可以带正负号)，这样这些基数可以用来表示 10 到 35。如果基数为 16 则省略开头的 0x 或 0X。
- 如果 *endptr* 不是空指针 , 通过 *endptr* 指向字符串的部分 (3) 保存在指向的对象中。
- 如果修正值导致溢出，函数返回 ULONG_MAX (4,294,967,295U) 并设置 errno 为 ERANGE (2)。
- 如果字符串 (2) 为空或字符串 (2) 中首个非空白字符不适合于给定基数的整数 , 则函数不进行转换并返回 0。在这种情况下，通过 endptr 字符串的值 *nptr* 存储在指向对象中 (如果它不是空字符串)。基数 0 和 2 到 36 保持为真。

calloc

分配数组区间并初始化为 0

[语法]

#include <stdlib.h>
void * calloc (size_t *nmemb* , size_t *size*) ;

[参数 / 返回值]

参数	返回数值
nmemb : 数组中的多个成员 *大小* : 每个成员的大小	如果分配所需的大小： 指针指向分配区域的起始位置 如果没有分配所需的大小： 空指针

[说明]

- calloc 函数为包含 n 个成员的数组分配空间 （由 *nmemb* 来设定), 设定每个空间拥有几个字节的大小并初始化空间 （数组成员）为零。
- 如果分配所需的大小 ， 返回指针指向分配区域的起始位置。
- 如果没有分配所需的大小，则返回空指针。
- 通过中断值开始分配内存，并在相邻分配空间的地址中将产生新的中断值。如果新中断值是奇数， calloc 函数纠正它为偶数。有关存储函数 brk 的中断值设定 , 参阅 " brk"。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 calloc_n 和 calloc_f 不可用。

free

释放已分配的内存块

[语法]

#include <stdlib.h>
void free (void *_ptr_) ;

[参数 / 返回值]

参数	返回数值
ptr : 释放指向块起始位置的指针	None

[说明]

- free 函数释放 ptr 指向的已分配空间 (在中断值前)。 (在 free 之后调用 malloc, calloc, 或 realloc ， 将给你更早释放的空间。)
- 如果 *ptr* 不指向已分配空间， free 将不操作。通过设置 *ptr* 为新中断值，进行释放已分配空间。)
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 free_n 和 free_f 不可用。

malloc

分配块

[语法]

#include <stdlib.h>
void *malloc (size_t *size*) ;

[参数 / 返回值]

参数	返回数值
大小 : 分配存储块的大小	如果分配所需的大小： 指针指向分配区域的起始位置 如果没有分配所需的大小： 空指针

[说明]

- malloc 函数按大小设定的字节数分配存储块，并返回指向分配区域的第 1 个字节的指针。
- 如果不能分配存储体，则函数返回空指针。
- 通过中断值开始分配内存，并在相邻分配区域的地址中将产生新的中断值。如果新中断值是奇数，malloc 函数纠正它为偶数。有关存储函数 brk 的中断值设定 , 参阅 " brk"。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 malloc_n 和 malloc_f 不可用。

realloc

重新分配块

[语法]

#include <stdlib.h>
void * realloc (void **ptr* , size_t *size*) ;

[参数 / 返回值]

参数	返回数值
ptr : 指向以前分配的块开头的指针 *大小* : 给这个块指定新的大小	如果重新分配所需的大小： 指针指向重新分配空间的开头 如果 *ptr* 是空指针 : 指针指向分配空间的开头 如果不能重新分配需要 *大小* 或 "*ptr*" 不是空指针 : 空指针

[说明]

- realloc 函数改变 ptr 指向的已分配空间大小 (在中断值之前) 为 *size* 设定的大小。如果 *size* 的值要比分配的空间大小要大 , 则分配空间的内容最多达原来的大小保留不变。 realloc 函数值分配增加的空间。如果 *size* 的值小于分配空间的大小 , 则函数将释放减少已分配空间。
- 如果 *ptr* 是空指针 , realloc 函数将按设定的大小重新分配存储块 (和 malloc 相同)。
- 如果 *ptr* 没有指向以前分配的存储块或如果没有存储体可以分配 , 函数不执行且返回空指针。
- 将按照 *ptr* 设置的地址加上 *size* 设定的字节数，作为新的中断值来重新进行分配。如果新中断值是奇数，realloc 函数纠正它为偶数。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 realloc_n 和 realloc_f 不可用。

中止

异常中止程序

[语法]

```
#include <stdlib.h>
void abort ( void ) ;
```

[参数 / 返回值]

参数	返回数值
None	不返回到它的调用方。

[说明]

- 循环中止函数且能不返回它的调用方。
- 用户必须创建中止处理程序。

atexit

在正常终止时，注册调用的函数

[语法]

```
#include <stdlib.h>
int atexit ( void ( *func ) ( void ) ) ;
```

[参数 / 返回值]

参数	返回数值
func : 指向注册函数的指针	如果注册函数为包裹函数： 0 如果不能注册函数 : 1

[说明]

- atexit 函数注册 *func* 指向的包裹函数，为了通过从 main 调用退出或返回，且在正常程序中止前它能被调用而不带参数。
- 可以建立多达 32 个包裹函数。如果能注册包裹函数，则 atexit 返回 0。如果不能注册再多的包裹函数，那是因为 32 个包裹函数已经注册，则函数返回 1。

退出

终止程序

[语法]

#include <stdlib.h>
void exit (int *status*) ;

[参数 / 返回值]

参数	返回数值
状态: 状态值标志中止	退出绝不返回。

[说明]

- exit 函数导致程序直接地，正常的中止。
- 这个函数按照 atexit 注册它们时的顺序的逆向顺序调用包裹函数。
- exit 函数循环且绝不可能返回它的调用方。
- 用户必须创建退出处理程序。

abs

获取整数型值的绝对值

[语法]

#include <stdlib.h>
int abs (int *j*) ;

[参数 / 返回值]

参数	返回数值
j : 获取任一带符号整数的绝对值	如果 *j* 属于 -32,767 <= *j* <= 32,767 : *j* 的绝对值 如果 *j* 为 -32,768 : -32,768 (0x8000)

[说明]

- abs 返回其 int 型参数的绝对值。
- 如果 *j* 为 -32,768, 函数返回 -32,768。

labs

获取 long 型值的绝对值

[语法]

#include <stdlib.h>
long int labs (long int *j*) ;

[参数 / 返回值]

参数	返回数值
j : 获取任一带符号整数的绝对值	如果 *j* 属于 -2,147,483,647 <= *j* <= 2,147,483,647 : *j* 的绝对值 如果 *j* 的值 为 -2,147,483,648 : -2147483,648 (0x80000000)

[说明]

- labs 返回其 long 型参数的绝对值。
- 如果 *j* 的值为 -2,147,483,648, 函数返回 -2,147,483,648。

div

进行 int 型除法运算，获取商数和余数

[语法]

#include <stdlib.h>
div_t div (int *numer* , int *denom*) ;

[参数 / 返回值]

参数	返回数值
数: 除法的分子 *分母*: 除法的分母	div_t 类型成员为 quot 元素商和 rem 元素的余数

[说明]

- div 函数进行分子除分母的整数除法。
- 商的绝对值定义为不大于分子的绝对值除分母的绝对值的最大整数。余数总是和除法的结果有相同的符号 (另外如果*分子* 和分母 有相同的符号 ; 不同于减)。
- 余数为*分子* - *分母* * 商的值。
- 如果*分母*为 0, 则商为 0 且余数为*分子*。
- 如果*分子*为 -32,768, 分母为 -1，则商为 -32,768 且余数为 *0*。

ldiv

进行 long 型除法运算，获取商数和余数

[语法]

#include <stdlib.h>

ldiv_t ldiv (long int *numer* , long int *denom*) ;

[参数 / 返回值]

参数	返回数值
数: 除法的分子 *分母*: 除法的分母	ldiv_t 类型成员为 quot 元素商和 rem 元素的余数

[说明]

- ldiv_t 函数进行分子除分母的 long 整数除法。
- 商的绝对值定义为不大于分子的绝对值除分母的绝对值的最大 long 整数。余数总是和除法的结果有相同的符号 (另外如果*分子*和分母 有相同的符号 ; 不同于减)。
- 余数为*分子* - *分母* * 商的值。
- 如果*分母*为 0, 则商为 0 且余数为*分子*。
- 如果*分子*为 -2,147,483,648, 分母为 -1，则商为 -2,147,483,648 且余数为 *0*。

brk

设置中断值

[语法]

#include <stdlib.h>
int brk (char **endds*) ;

[参数 / 返回值]

参数	返回数值
endds : 设置中断值，释放块。	如果适当地转换： 0 如果不能改变中断值 : -1

[说明]

- brk 函数按 *endds* 给定的值设置中断值 (地址接下去到分配的存储块的结束地址）。
- 如果 *endds* 超出了允许地址范围 , 函数不设置中断值且设置 errno 为 ENOMEM (3)。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 brk_n 和 brk_f 不可用。

sbrk

增加 / 减少中断值

[语法]

#include <stdlib.h>
char *sbrk (int *incr*) ;

[参数 / 返回值]

参数	返回数值
incr : 设置中断值 （字节）增加 / 减少。	如果适当地转换： 老的中断值 如果老中断值不能增加或减少： -1

[说明]

- sbrk 函数增加或减少设置中断值是由 *incr* 设定的字节数量来决定。 (由 *incr* 的加号或减号来决定增加或减少。)
- 如果 *incr* 设定为奇数 , 则 sbrk 函数校正它为偶数。
- 如果增加或减少的中断值超过了允许的地址范围，函数不能改变原始的中断值，并设置 errno 为 ENOMEM (3)。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 sbrk_n 和 sbrk_f 不可用。

atof

十进制整数字符串转换为 double 型

[语法]

```
#include <stdlib.h>
double atof ( const char *nptr ) ;
```

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串	如果适当地转换： 转换值 如果产生正溢出： HUGE_VAL (溢出值符号) 如果产生负溢出： 0 如果字符串无效： 0

[说明]

- atof 函数转换 *nptr* 指针指向的字符串为 double 型数值。
  atof 函数从字符串的开头跳过零或多个空白字符 （当 isspace 变成 true 时）以及转换字符串从下一个跳过的空白字符到浮点数 （直到数字和空字符出现在字符串以外）。
- 当转换正确时，返回浮点指针号。
- 如果转换中产生溢出，返回带符号的溢出值 HUGE_VAL，并设置 errno 为 ERANGE。
- 如果由于上溢出或溢出导致有效数字的删除，分别返回非正常数字和 +0，且设置 errno 为 ERANGE。
- 如果不能进行转换，返回 0。

strtod

转换字符串为 double 型

[语法]

#include <stdlib.h>
double strtod (const char **nptr* , char ***endptr*) ;

[参数 / 返回值]

参数	返回数值
nptr : 转换字符串 *endptr* : 指针存储指向不可识别块的指针。	如果适当地转换： 转换值 如果产生正溢出： HUGE_VAL (溢出值符号) 如果产生负溢出： 0 如果字符串无效： 0

[说明]

- strtod 函数转换 *nptr* 指针指向的字符串为 double 型数值。
  strtod 函数从字符串的开头跳过零或多个空白字符 （当 isspace 变成 true 时）以及转换字符串从下一个跳过的空白字符到浮点数 （直到数字和空字符出现在字符串以外）。
  如果字符串开头的字符不符合这个格式，则中止扫描。如果 *endptr* 不是空指针 , 指针开头的字符可能是存储在 endptr 中的空格。
- 当转换正确时，返回浮点指针号。
- 如果转换中产生溢出，返回带符号的溢出值 HUGE_VAL，并设置 errno 为 ERANGE。
- 如果由于上溢出或溢出导致有效数字的删除，分别返回非正常数字和 +0，且设置 errno 为 ERANGE。此外 , *endptr* 在那时存储下一字符串的指针。
- 如果不能进行转换，返回 0。

itoa

转换 int 型为字符串

[语法]

#include <stdlib.h>

char *itoa (int *value* , char **string* , int *radix*) ;

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串为整数 *字符串* : 指向转换结果的指针 *radix* : 输出字符串的基	如果适当地转换： 指向转换字符串的指针 如果没有正确转换： 空指针

[说明]

- itoa, ltoa, ultoa 函数都是转换 *value* 设定的整数值为它相当的字符串，用空字符来中止且存储结果在 "*string*" 指向的区域。
- 由 radix 决定输出字符串的*基* , 它必须在范围 2 到 36 之间。各函数进行转换依靠指定的 *radix* ，且返回指向转换字符串的指针。如果设定的 *radix* 超出了范围 2 到 36, 则函数不进行转换且返回空指针。

ltoa

转换 long 型为字符串

[语法]

#include <stdlib.h>
char *ltoa (long *value* , char **string* , int *radix*) ;

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串为整数 *字符串* : 指向转换结果的指针 *radix* : 输出字符串的基	如果适当地转换： 指向转换字符串的指针 如果没有正确转换： 空指针

[说明]

- itoa, ltoa, ultoa 函数都是转换 *value* 设定的整数值为它相当的字符串，用空字符来中止且存储结果在 "*string*" 指向的区域。
- 由 radix 决定输出字符串的*基* , 它必须在范围 2 到 36 之间。各函数进行转换依靠指定的 *radix* ，且返回指向转换字符串的指针。如果设定的 *radix* 超出了范围 2 到 36, 则函数不进行转换且返回空指针。

ultoa

转换无符号 long 型为字符串

[语法]

#include <stdlib.h>

char *ultoa (unsigned long *value* , char **string* , int *radix*) ;

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串为整数 *字符串* : 指向转换结果的指针 *radix* : 输出字符串的基	如果适当地转换： 指向转换字符串的指针 如果没有正确转换： 空指针

[说明]

- itoa, ltoa, ultoa 函数都是转换 *value* 设定的整数值为它相当的字符串，用空字符来中止且存储结果在 "*string*" 指向的区域。
- 由 radix 决定输出字符串的*基* , 它必须在范围 2 到 36 之间。各函数进行转换依靠指定的 *radix* ， 且返回指向转换字符串的指针。如果设定的 *radix* 超出了范围 2 到 36, 则函数不进行转换且返回空指针。

rand

生成伪随机号

[语法]

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

[参数 / 返回值]

参数	返回数值
None	在 0 到 RAND_MAX 范围内的伪随机

[说明]

- 每次调用 rand 函数 , 它返回在 0 到 RAND_MAX 范围内的伪随机整数。

srand

初始化伪随机号的生成状态

[语法]

#include <stdlib.h>
void srand (unsigned int *seed*) ;

[参数 / 返回值]

参数	返回数值
种子数 : 随机数生成器的起始值	None

[说明]

- srand 函数为随机数队列设置起始值。 *种子数*是用于为随机数的进程设置起始点，在调用 rand 时返回该随机数。如果使用相同的 *种子* 数值 , 则在 srand 再次被调用时，随机数的队列是相同的。
- 在 srand 用于设置与调用 rand 相同的 *种子*之前，在 srand 已经调用时用的*种子数* = 1 之后，调用 rand 。(默认种子数为 1。)

bsearch

二进制查找

[语法]

```
#include <stdlib.h>
void *bsearch ( const void *key , const void *base , size_t nmemb , size_t size ,
                                                    int ( *compare ) ( const void *, const void * ) ) ;
```

[参数 / 返回值]

参数	返回数值
主键 : 为查找建立主键指针。 *基* : 指向已排序数组包含查找信息 *nmemb* : 数据的元素个数 *大小* : 数组的大小 *比较* : 指向函数的指针用于比较 2 个主键	如果数组包含主键： 指向第 1 个成员的指针与 " *主键* " 相匹配 如果在数组中不包含主键： 空指针

[说明]

- bsearch 函数在 base 指向的已排序数组中进行二进制查找且返回与 *key* 指向的主键相匹配的第 1 个成员的指针。 *base* 指向的数据必须是数据，且包含由 *size* 设定的大小的 nmemb 的多个成员，以及必须按升序存储。
- 通过比较有 2 个参数指向的函数 (*第 1 个参数作为主键*且第 2 个参数作为数据元素), 比较 2 个参数 , 并返回 :
  如果第 1 个参数小于第 2 个参数，则返回负值。
  如果两个参数相等，则返回 0
  如果第 1 个参数大于第 2 个参数，则返回正值。

qsort

快速排序

[语法]

#include <stdlib.h>

void qsort (void *_base_ , size_t _nmemb_ , size_t _size_ , int (*_compare_) (const void * , const void *)) ;

[参数 / 返回值]

参数	返回数值
基: 排序指针指向的数组 *nmemb* : 数组中的多个成员 *大小*: 数组成员的大小 *比较*: 指向函数的指针用于比较 2 个主键	None

[说明]

- qsort 函数对 *base* 指向区域的成员进行升序排序。
  base 指向的区域包含 size 设定的各 nmemb 成员数的大小 。
- 指向比较拥有 2 个参数的函数 （数组元素 1 和 2），比较 2 个参数，并返回：
- 数据元素 1 作为第 1 个参数，且数组元素 2 作为第 2 个参数
  如果第 1 个参数小于第 2 个参数，则返回负值。
  如果两个参数相等，则返回 0
  如果第 1 个参数大于第 2 个参数，则返回正值。
- 如果 2 个数组元素相等，则最接近数组顶部的元素排在前面。

strbrk

设置中断值

[语法]

#include <stdlib.h>
int strbrk (char *endds) ;

[参数 / 返回值]

参数	返回数值
endds : 设置中断值	正常 : 0 当不能改变中断值时 : -1

[说明]

- 按 *endds* 给定的值设置中断值 (地址接着分配区域的结束地址后面）。
- 在 *endds* 超过允许范围时，不改变中断值。设置 errno 为 ENOMEM(3)，且返回 -1。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 strbrk_n 和 strbrk_f 不可用。

strsbrk

增加 / 减少中断值

[语法]

#include <stdlib.h>
char *strsbrk (int *incr*) ;

[参数 / 返回值]

参数	返回数值
incr : 合计增加 / 减少中断值	正常 : 老的中断值 当不能增加 / 减少中断值时 : -1

[说明]

- *incr* 增加字节 / 减少中断值 (由 *incr* 的符号决定)。
- 当在增加 / 减少之后中断值超过允许范围，则中断值不改变。设置 errno 为 ENOMEM(3) ，且返回 -1。
- 由于通过存在内部 RAM 中的 C 编译器分配空间 78K0R, 参数 *ptr* 总是为 near 指针。因此函数 strsbrk_n 和 strsbrk_f 不可用。

stritoa

转换 int 型 为字符串

[语法]

#include <stdlib.h>
char *stritoa (int *value* , char **string* , int *radix*) ;

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串 *字符串* : 指向转换结果的指针 *radix* : 设定基数	正常 : 指向转换字符串的指针 其他 空指针

[说明]

- 转换指定的数字值 *value* 为字符串，并在结尾加上空字符 , 且结果保存在字符串的指定区域。根据 设定*的基数* 进行转换 , 且返回指向已转换字符串的指针。
- *基数*必须在 2 到 36 的数值范围内。在其他情况下，不进行转换且返回空指针。

strltoa

转换 long 型为字符串

[语法]

```
#include <stdlib.h>
char *strltoa ( long value , char *string , int radix ) ;
```

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串 *字符串* : 指向转换结果的指针 *radix* : 设定基数	正常 : 指向转换字符串的指针 其他 空指针

[说明]

- 转换指定的数字值 *value* 为字符串，并在结尾加上空字符 , 且结果保存在字符串的指定区域。根据 设定*的基数* 进行转换 , 且返回指向已转换字符串的指针。
- *基数*必须在 2 到 36 的数值范围内。在其他情况下，不进行转换且返回空指针。

strultoa

转换无符号 long 型为字符串

[语法]

#include <stdlib.h>

char *strultoa (unsigned long *value* , char **string* , int *radix*) ;

[参数 / 返回值]

参数	返回数值
数值 : 转换字符串 *字符串* : 指向转换结果的指针 *radix* : 设定基数	正常 : 指向转换字符串的指针 其他 空指针

[说明]

- 转换指定的数字值 *value* 为字符串，并在结尾加上空字符 , 且结果保存在字符串的指定区域。根据 设定*的基数* 进行转换 , 且返回指向已转换字符串的指针。
- *基数*必须在 2 到 36 的数值范围内。在其他情况下，不进行转换且返回空指针。

6.11　字符串和存储函数

以下字符串和存储函数可用。

功能名称	作用
memcpy	为指定数量的字符复制缓冲
memmove	为指定数量的字符复制缓冲
strcpy	复制字符串
strncpy	从字符串的首字符开始，复制指定数量的字符。
strcat	在字符串后追加字符串
strncat	在字符串后追加指定字符数量的字符串
memcmp	在两个缓冲中比较指定数量的字符
strcmp	比较两个字符串
strncmp	在两个字符串中比较指定数量的字符
memchr	在缓冲区内指定数量的字符中查找指定字符串
strchr	从字符串和首次返回发生事件的位置中查找指定字符串
strrchr	从字符串和最后返回发生事件的位置中查找指定字符串
strspn	从段起始处获取长度，该段包含的字符位于查找字符串内特定字符串中
strcspn	从段起始处获取长度，该段包含的字符不在查找字符串内特定字符串中
strpbrk	在字符串中查找特定字符串时，获取任意字符第一个出现的位置
strstr	在字符串中查找特定字符串时，获取指定字符串第一个出现的位置
strtok	分解字符串为不带分隔符的字符。
memset	初始化指定字符的缓冲中指定数量的字符
strerror	返回指针区域，该区域存储与错误信息相对应的指定错误号。
strlen	获取字符串的长度
strcoll	根据范围特定信息比较两个字符串
strxfrm	根据范围特定信息变换两个字符串

memcpy

为指定数量的字符复制缓冲

[语法]

#include <string.h>

void *memcpy (void **s1*, const void **s2* , size_t *n*) ;

[参数 / 返回值]

参数	返回数值
s1 : 复制指向目标指针的数据 *s2* : 复制指向目标指针包含的数据。 *n* : 复制多个字符	*s1* 的数值

[说明]

- memcpy 函数复制 s2 指向目标的 *n* 个连续字节 到 s1 指向的目标。
- 如果 *s2* < *s1* < *s2* + *n* (*s1* 和 *s2* 重叠), memcpy 的存储器复制不能保证运行 (因为从区域的起始按顺序开始复制）。

memmove

复制特定数量字符的缓冲区 （即使缓冲区重叠，函数仍能正确执行存储复制）

[语法]

#include <string.h>

void *memmove (void **s1* , const void **s2* , size_t *n*) ;

[参数 / 返回值]

参数	返回数值
s1 : 复制指向目标指针的数据 *s2* : 复制指向目标指针包含的数据。 *n* : 复制多个字符	*s1* 的数值

[说明]

- memmove 函数也可以从 s2 指向目标中复制 *n* 个连续字节 到 s1 指向的目标。
- 即使 *s1* 和 *s2* 重叠 , 函数仍能正确执行存储复制。

strcpy

复制字符串

[语法]

#include <string.h>
char *strcpy (char **s1* , const char **s2*) ;

[参数 / 返回值]

参数	返回数值
s1 : 复制目标数组指针 *s2* : 复制源数组指针	*s1* 的数值

[说明]

- strcpy 函数复制指向 s2 中的字符串内容到指向的 s1 数组中 (包含终止符)。
- 如果 *s2* < *s1* < (*s2* + 字符长度已复制), 不能保证 strcpy 行为的运行 (从开始按顺序开始复制，而不是从指定字符串开始）。

strncpy

从字符串的首字符开始，复制指定数量的字符。

[语法]

#include <string.h>

char *strncpy (char **s1* , const char **s2* , size_t *n*) ;

[参数 / 返回值]

参数	返回数值
s1 : 复制目标数组指针 *s2* : 复制源数组指针 *n* : 复制多个字符	*s1* 的数值

[说明]

- strncpy 函数从指向 s2 的字符串中复制多达 n 个字符到指向 s1 的数组中。。
- 如果 *s2* < *s1* < (*s2* + 字符长度已复制或 *s2* + *n* - 1 的最小值), 不能保证 strncpy 行为的运行 (从开始按顺序开始复制，而不是从指定字符串开始）。
- 如果指向 *s2* 的字符串小于由 n 设定的字符数 , strncpy 函数复制字符串直到终止空字符，且添加空字符直到达到复制字符串的数量 *n*。

strcat

在字符串后追加字符串

[语法]

#include <string.h>
char *strcat (char **s1* , const char **s2*) ;

[参数 / 返回值]

参数	返回数值
s1 : 另一个字符串的副本 (*s2*) 接在指针指向的字符串后面 *s2* : 指针指向的字符串与另一个字符串的副本 (*s1*) 相连接	*s1* 的数值

[说明]

- strcat 函数连接指向 *s2* 的字符串副本 (包含空中止符) 到指向 *s1* 的字符串上。一般空中止符结束后，第 1 个字符串 s2 将覆盖 *s1* 。
- 当在交错的目标中相互进行复制，则不能保证其操作。

strncat

在字符串后追加指定字符数量的字符串

[语法]

#include <string.h>

char *strncat (char **s1* , const char **s2* , size_t *n*) ;

[参数 / 返回值]

参数	返回数值
s1 : 另一个字符串的副本 (*s2*) 接在指针指向的字符串后面 *s2* : 指针指向的字符串与另一个字符串的副本 (*s1*) 相连接 *n* : 连接多个字符	*s1* 的数值

[说明]

- strncat 函数只是连接 *s2* 指向的字符串中 n 个指定字符 (不包含空中止符) 到 *s1* 指向的字符串中。一般空中止符结束后，第 1 个字符串 s2 将覆盖 *s1* 。
- 如果 *s2* 指向的字符串小于设定的 n 个字符 , 则 strncat 函数连接的字符串包含空中止符。如果有超过设定的 *n* 字符数时 , 则开始从头连接 *n* 个字符段。
- 总要连接空中止符。
- 当在交错的目标中相互进行复制，则不能保证其操作。

memcmp

在两个缓冲中比较指定数量的字符

[语法]

#include <string.h>

int memcmp (const void *s1 , const void *s2 , size_t n) ;

[参数 / 返回值]

参数	返回数值
s1 : 比较指向 2 个数据目标指针 *s2* : 比较指向 2 个数据目标指针 *n* : 比较字符个数	如果比较 s1 和 s2 中的 n 个字符且发现相同 : 0 如果比较 s1 和 s2 中的 n 个字符且发现不同 : 数值差异则转换初始的不同字符为 int (*s1* 字母 - *s2* 字母))

[说明]

- memcmp 函数使用 *n* 个字符去比较 *s1* 和 *s2* 两者表示的目标。
- memcmp 函数返回 0，在比较 s1 和 s2 中的 n 个字符且发现相同时。
- memcmp 函数返回数值差异 (*s1* 字母 - *s2* 字母)，转换初始的不同字符为 int，如果比较 *s1* 和 *s2* 两个中的 n 个字符且发现不同。

strcmp

比较两个字符串

[语法]

```
#include <string.h>
int strcmp ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 比较指针指向的一个字符串 *s2* : 比较指针指向的其它字符串	如果 *s1* 等于 *s2* : 0 如果 *s1* 小于或大于 *s2* : 数值差异则转换初始的不同字符为 int (*s1* 字母 - *s2* 字母))

[说明]

- strcmp 函数用于比较 *s1* 和 *s2* 两者表示的字符串。
- 如果 *s1* 等于 *s2* : 函数返回 0。如果 *s1* 小于或大于 *s2*, strcmp 函数返回数值差异 (*s1* 字母 - *s2* 字母)，且转换初始的不同字符为 int。

strncmp

在两个字符串中比较指定数量的字符

[语法]

#include <string.h>

int strncmp (const char *s1 , const char *s2 , size_t n) ;

[参数 / 返回值]

参数	返回数值
s1 : 比较指针指向的一个字符串 *s2* : 比较指针指向的其它字符串 *n* : 比较字符个数	如果比较 s1 和 s2 中的 n 个字符且发现相同 : 0 如果比较 s1 和 s2 中的 n 个字符且发现不同 : 数值差异则转换初始的不同字符为 int (*s1* 字母 - *s2* 字母))

[说明]

- strncmp 函数使用 *n* 个字符去比较 *s1* 和 *s2* 两者表示的目标。
- strncmp 函数返回 0，在比较 s1 和 s2 中的 n 个字符且发现相同时。 strncmp 函数返回数值差异 (s1 *字母* - s2 *字母*)，*转换初始的不同字符为 int，如果比较 s1 和* s2 *两个中的 n 个字符且发现不同。*

memchr

在缓冲区内指定数量的字符中查找指定字符串

[语法]

```
#include <string.h>
void *memchr ( const void *s , int c , size_t n ) ;
```

[参数 / 返回值]

参数	返回数值
s : 在存储器中查找指针指向的目标 *c* : 查找字符 *n* : 查找字节数	如果发现 *c*: 指向 *c* 的第一个事件 如果没有发现 *c*: 空指针

[说明]

- memchr 函数首先转换 *c* 设定的字符为无符号 char 型，且返回指针指向这个字符的第一个事件，在从 s 指向的目标开头的字节数内。
- 如果没有发现字符，则函数返回空指针。

strchr

从字符串和首次返回发生事件的位置中查找指定字符串

[语法]

```
#include <string.h>
char *strchr ( const char *s , int c ) ;
```

[参数 / 返回值]

参数	返回数值
s : 查找指针指向的字符串 *c* : 查找指定字符	如果在 s 中发现了 *c* ： 在字符串 s 中指针表示 *c* 中的第一个事件 如果在 s 中没有发现了 *c* ： 空指针

[说明]

- strchr 函数查找 *s* 指向的字符串中由 *c* 指定的字符，且返回在字符串中指向 *c* 第一个事件的指针 (转换为 char 类型)。
- 空中止符作为字符串的一部分。
- 如果在字符串中没有发现指定字符，则函数返回空指针。

strrchr

从字符串和最后返回发生事件的位置中查找指定字符串

[语法]

```
#include <string.h>
char *strrchr ( const char *s , int c ) ;
```

[参数 / 返回值]

参数	返回数值
s : 查找指针指向的字符串 *c* : 查找指定字符	如果在 s 中发现了 *c* : 在字符串 s 中指针表示 *c* 中的最后一个事件 如果在 s 中没有发现了 *c* : 空指针

[说明]

- strrchr 函数查找 *s* 指向的字符串中由 *c* 指定的字符，且返回在字符串中指向 *c* 最后一个事件的指针 (转换为 char 类型)。
- 空中止符作为字符串的一部分。
- 如果发现不匹配，则函数返回空指针。

strspn

从段起始处获取长度，该段包含的字符位于查找字符串内特定字符串中

[语法]

```
#include <string.h>
size_t strspn ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 查找指针指向的字符串 *s2* : 比较指向字符串指定的字符	字符串 *s1* 子字符串的长度仅有包含在字符串 *s2* 中的那些字符构成。

[说明]

- strspn 函数返回 *s1* 指向字符串的子字符串的长度，该长度由包含在 *s2* 指向字符串的那些字符构成。换句话说，这个函数返回字符串 *s1* 中第 1 字符的索引值，它与字符串 *s2* 中的任何字符不匹配。
- *s2* 的空中止符不作为 *s2* 的一部分。

strcspn

从段起始处获取长度，该段包含的字符不在查找字符串内特定字符串中

[语法]

```
#include <string.h>
size_t strcspn ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 查找指针指向的字符串 *s2* : 比较指向字符串指定的字符	字符串 *s1* 子字符串的长度由不包含在字符串 *s2* 中的那些字符构成。

[说明]

- strcspn 函数返回 *s1* 指向字符串的子字符串的长度，该长度由不包含在 *s2* 指向字符串的那些字符构成。换句话说 , 这个函数返回字符串 *s1* 中第 1 字符的索引值，它与字符串 *s2* 中的任何字符匹配。
- *s2* 的空中止符不作为 *s2* 的一部分。

strpbrk

在字符串中查找特定字符串时，获取任意字符第一个出现的位置

[语法]

```
#include <string.h>
char *strpbrk ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 查找指针指向的字符串 *s2* : 比较指向字符串指定的字符	如果发现任何匹配： 字符串 *s1* 中指针指向的第 1 个字符与字符串 *s2* 中的任一字符匹配。 如果没有发现匹配： 空指针

[说明]

- strpbrk 函数返回 *s1* 指向的字符串中指向第 1 个字符串的指针，其与 *s2* 指向的字符串中的任一字符匹配。
- 如果在字符串 s1 没有发现在字符串 *s2* 中的字符 , 函数返回空指针。

strstr

在字符串中查找特定字符串时，获取指定字符串第一个出现的位置

[语法]

```
#include <string.h>
char *strstr ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 查找指针指向的字符串 *s2* : 指向指定字符串的指针	如果在 s1 中发现了 *s2* ： 指向在字符串 s2 的 字符串 *s1* 中的第一个出现的字符 如果在 s1 中没有发现 *s2* ： 空指针 如果 *s2* 是空指针 : *s1* 的数值

[说明]

- strstr 函数返回 s2 指向的字符串的 *s1* 指向的字符串中指向的第 1 个出现的字符 (除 s2 中的空中止符以外)。
- 如果在字符串 s1 中没有发现字符串 s2，则函数返回空指针。
- 如果字符串 *s2* 是空字符串 , 则函数返回 *s1* 的值。

strtok

获取字符串的长度

[语法]

#include <string.h>
char *strtok (char **s1* , const char **s2*) ;

[参数 / 返回值]

参数	返回数值
s1 : 从指向的字符串中获取记号或空指针 *s2* : 指向的字符串包含标记分隔符	如果发现它 : 指向标记的第 1 个字符 如果没有返回标记 : 空指针

[说明]

- 标记是包含除在字符串中设定的分隔符以外的字符。
- 如果 *s1* 是空指针 , 则分解在以前 strtok 调用中由保存指针指向的字符串。然而，如果保存指针是空指针，则函数不做任何事返回空指针。
- 如果 *s1* 不是空指针 , 则分解 *s1* 指向的字符串。
- strtok 函数查找 *s1* 指向的字符串内的任一字符是否包含在 *s2* 指向的字符串中。如果没有发现字符，则函数更改保存指针为空指针，并返回它。如果发现任一字符，则字符成为标记的首字符。
- 如果发现标记的首字符，则在标记的第 1 个字符后函数查找任一包含在 s2 字符串中的字符。如果没有发现字符，则函数更改保存指针为空指针。如果发现任一字符，则空字符覆盖该字符，且保存指针指向的下一个字符。
- 函数返回指向标记的第 1 个字符。

memset

初始化指定字符的缓冲中指定数量的字符

[语法]

#include <string.h>
void *memset (void *s , int c , size_t n) ;

[参数 / 返回值]

参数	返回数值
s : 初始化在存储器中指针指向的目标 *c* : 为字符的值分配每一字节 *n* : 初始化字节数	*n* 的值

[说明]

- memset 函数首先转换 *c* 设定的字符为无符号 char 型，且分配这个字符的值到从 s 指向的目标开头的字节数 n 中。

strerror

返回指针区域，该区域存储与错误信息相对应的指定错误号。

[语法]

```
#include <string.h>
char *strerror ( int errnum ) ;
```

[参数 / 返回值]

参数	返回数值
errnum : 出错号	如果消息与存在的错误编号相关： 指向字符串的指针描述错误消息 如果消息与存在的错误编号不相关： 空指针

[说明]

- strerror 函数返回以下与 *errnum* 值有关的值。

errnum 的值	返回数值
0	指向字符串 " 错误 0" 的指针
1 (EDOM)	指向字符串 " 参数太大 " 的指针
2 (ERANGE)	指向字符串 " 结果太大 " 的指针
3 (ENOMEM)	指向字符串 " 内存不足 " 的指针
其他	空指针

- 错误消息字符串分配在 far 区域，因此返回值总为 far 指针。这是没有 strerror_n/strerror_f 函数的原因。

strlen

获取字符串的长度

[语法]

```
#include <string.h>
size_t strlen ( const char *s ) ;
```

[参数 / 返回值]

参数	返回数值
s : 指向字符串的指针	字符串 *s* 的长度

[说明]

- strlen 函数返回 *s* 指向的空中止字符串的长度。

strcoll

根据范围特定信息比较两个字符串

[语法]

```
#include <string.h>
int         strcoll ( const char *s1 , const char *s2 ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 指向比较字符串的指针 *s2* : 指向比较字符串的指针	当字符串 *s1* 和 *s2* 相等 : 0 当字符串 *s1* 和 *s2* 不相等 : 有差异的两个值的第 1 个不相同的字符串转换为 int (*s1* 字符 - *s2* 字符)

[说明]

- 78K0R C 编译器不支持特定于公共空间的操作。
  操作方法与 strcmp 相同。

strxfrm

根据范围特定信息变换两个字符串

[语法]

```
#include <string.h>
size_t  strxfrm ( char  *s1 , const char  *s2 , size_t  n ) ;
```

[参数 / 返回值]

参数	返回数值
s1 : 指向已比较字符串的指针 *s2* : 指向已比较字符串的指针 *n* : *s1* 为字符的最大个数	返回转换结果字符串的长度 （不包含表示结束的字符串）。 如果返回值为 *n* 或更多 , *s1* 表示的数组内容未定义。

[说明]

- 78K0R C 编译器不支持特定于公共空间的操作。

操作方法与以下函数相同。

```
strncpy ( s1 , s2 , c ) ;
return ( strlen ( s2 ) ) ;
```

6.12　数学函数

以下数学函数可用。

功能名称	作用
acos	计算 acos
asin	计算 asin
atan	计算 atan
atan2	计算 atan2
cos	计算 cos
sin	计算 sin
tan	计算 tan
cosh	计算 cosh
sinh	计算 sinh
tanh	计算 tanh
exp	计算指数函数
frexp	计算小数和指数部分
ldexp	计算 x * 2 ^ exp
log	计算自然对数
log10	计算底为 10 的对数
modf	计算小数和整数部分
pow	计算 x 的 y 次幂
sqrt	计算平方根
ceil	计算最小整数而不比 x 小
fabs	计算浮点数 x 的绝对值
floor	计算最大整数而不比 x 大
fmod	计算 x/y 的余数
matherr	获取库操作浮点数的异常处理
acosf	计算 acos
asinf	计算 asin
atanf	计算 atan
atan2f	计算 y/x 的 atan
cosf	计算 cos
sinf	计算 sin
tanf	计算 tan
coshf	计算 cosh
sinhf	计算 sinh
tanhf	计算 tanh
expf	计算指数函数
frexpf	计算小数和指数部分
ldexpf	计算 x * 2 ^ exp
logf	计算自然对数

功能名称	作用
log10f	计算底为 10 的对数
modff	计算小数和整数部分
powf	计算 x 的 y 次幂
sqrtf	计算平方根
ceilf	计算最小整数而不比 x 小
fabsf	计算浮点数 x 的绝对值
floorf	计算最大整数而不比 x 大
fmodf	计算 x/y 的余数

acos

计算 acos

[语法]

```
#include <math.h>
double  acos ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 -1 <= *x* <= 1 : *x* 的 acos 当 *x* < -1, 1 < *x*, *x* = NaN : NaN

[说明]

- 计算 x 的 acos(范围在 0 和 π 间)。
- 在 *x* < -1, 1 < *x* 定义的区域错误的情况下 , 返回 NaN 且设置 EDOM。
- 当 *x* 是个非数字值时 , 返回 NaN。

asin

计算 asin

[语法]

```
#include <math.h>
double  asin ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 -1 <= *x* <= 1 : *x* 的 acos 当 *x* < -1, 1 < *x*, *x* = NaN : NaN 当 *x* = -0 : -0 当产生下溢出时： 非正常数字

[说明]

- 计算 x 的 asin (范围在 $-\pi/2$ 和 $+\pi/2$ 间) 。
- 在 *x* < -1, 1 < *x* 定义的区域错误的情况下 , 返回 NaN 且设置 EDOM 为 errno。
- 当 *x* 是个非数字值时 , 返回 NaN。
- 当 *x* 为 -0 时 , 返回 -0。
- 如果转换结果中产生下溢出，返回非正常数字。

atan

计算 atan

[语法]

```
#include <math.h>
double  atan ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 atan 当 *x* = NaN : NaN 当 *x* = -0 : -0 当产生下溢出时： 非正常数字

[说明]

- 计算 x 的 atan (范围在 $-\pi/2$ 和 $+\pi/2$ 间) 。
- 当 *x* 是个非数字值时 , 返回 NaN。
- 当 *x* 为 -0 时 , 返回 -0。
- 如果转换结果中产生下溢出，返回非正常数字。

atan2

计算 y/x 的 atan

[语法]

```
#include <math.h>
double  atan2 ( double y , double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 进行数字值运算	正常 : *y*/*x* 的 atan 当 *x* 和 *y* 为 0 或 不能表达 y/x 的值时 , 或无论 *x* 或 *y* 为 NaN 以及 *x* 和 *y* 为 ± ? 时 : NaN 当产生下溢出时： 非正常数字

[说明]

- 计算 y/x 的 atan (范围在 -π 和 +π 间)。
- 当 *x* 和 *y* 为 0 或 不能表达 y/x 的值时 , 或当 *x* 和 *y* 为无穷时，返回 NaN 并设置 EDOM 为 errno。
- 无论 *x* 还是 *y* 是否是个非数字值 , 返回 NaN。
- 如果操作结果中产生下溢出，返回非正常数字。

cos

计算 cos

[语法]

```
#include <math.h>
double  cos ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 cos 当 *x* = NaN, 当 *x* 为无穷 : NaN

[说明]

- 计算 *x* 的 cos
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

sin

计算 sin

[语法]

```
#include <math.h>
double  sin ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 sin 当 *x* = NaN 时 , 当 *x* 为无穷时 : NaN 当产生下溢出时： 非正常数字

[说明]

- 计算 *x* 的 sin
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

tan

计算 tan

[语法]

```
#include <math.h>
double  atan ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 tan 当 *x* = NaN, *x* = ± ∞ : NaN 当产生下溢出时： 非正常数字

[说明]

- 计算 *x* 的 tan。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

cosh

计算 cosh

[语法]

```
#include <math.h>
double  cosh ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 cosh 当 *x* = NaN : NaN 当 *x* = ± ∞: + ∞ 当产生溢出时 HUGE_VAL (溢出值符号)

[说明]

- 计算 *x* 的 cosh
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回正的无穷值。
- 如果运行结果中产生溢出，返回带正号的 HUGE_VAL，并设置 ERANGE 为 errno。

sinh

计算 sinh

[语法]

```
#include <math.h>
double  sin ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 sinh 当 *x* = NaN : NaN 当 *x* = ± ∞: ± ∞ 当产生溢出时 : HUGE_VAL (溢出值符号) 当产生下溢出时： ± 0

[说明]

- 计算 *x* 的 sinh。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± ∞, 返回 ± ∞。
- 如果运行结果中产生溢出，返回带溢出值符号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果运行结果中产生下溢出，返回 +0。

tanh

计算 tanh

[语法]

```
#include <math.h>
double  atan ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 tanh 当 *x* = NaN : NaN 当 *x* = ± ∞: ± 1 当产生下溢出时： ± 0

[说明]

- 计算 *x* 的 tanh。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± ∞, 返回 ± ∞。
- 如果运行结果中产生下溢出，返回 ± 0。

exp

计算指数函数

[语法]

```
#include <math.h>
double  exp ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 指数函数 当 *x* = NaN : NaN 当 *x* = +∞ : + 当 *x* = -∞ : +0 当产生下溢出时： 非正常数字 当由于下溢出产生有效数字丢失时： +0 当产生溢出时 : HUGE_VQAL (带正号)

[说明]

- 计算 *x* 指数函数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 +∞, 返回 +∞ 。
- 如果 *x* 为 -∞, 返回 +0。
- 如果运行结果中产生下溢出，返回非正常数字。
- 如果运行结果由于产生下溢出，有效数字丢失时，则返回 +0。
- 如果运行结果中产生溢出，返回带正号的 HUGE_VAL，并设置 ERANGE 为 errno。

frexp

计算小数和指数部分

[语法]

```
#include <math.h>
double  frexp ( double x , int *exp ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *exp* : 指针指向存储指数部分	正常 : *x* 的小数部分 当 *x* = NaN, *x* = ± ∞ : NaN 当 *x* = ± 0 : ± 0

[说明]

- 分隔浮点数 *x* 为小数部分 m 和指数部分 n 例如：*x* = m * 2 ^ n 并返回小数部 m。
- 指数 n 存储在显示 *exp* 指针的地方。然而绝对值 m 为 0.5 或 更大且小于 1.0。
- 如果 *x* 是非数字 , 返回 NaN 和 **exp* 为 0 的值。
- 如果 *x* 为无穷数 , 返回 NaN, 并设置 EDOM 为 errno 带 **exp* 为 0 的值。
- 如果 *x* 为 ± 0, 返回 ± 0 和 **exp* 为 0 的值。

ldexp

计算 x * 2 ^ exp

[语法]

#include <math.h>
double ldexp (double *x* , int *exp*) ;

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *exp* : 求幂	正常 : *x* * 2 ^ *exp* 当 *x* = NaN : NaN 当 *x* = ± ∞ : ± 当 *x* = ± 0 : ± 0 当产生溢出时 : HUGE_VAL (溢出值符号) 当产生下溢出时 : 非正常数字 当由于下溢出产生有效数字丢失时 : ± 0

[说明]

- 计算 x * 2 ^ exp。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± ?, 返回 ± ∞。
- 如果 *x* 为 ± 0, 返回 ± 0。
- 如果运行结果中产生溢出，返回溢出值 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果操作结果中产生下溢出，返回非正常数字。
- 如果运行结果由于产生下溢出，有效数字丢失时，返回 ± 0。

log

计算自然对数

[语法]

```
#include <math.h>
double  log ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的自然对数 当 *x* < 0 : NaN 当 *x* = 0 : - 当 *x* = NaN : NaN 当 *x* 为无穷时 : +∞

[说明]

- 计算 *x* 的自然对数。
- 在 *x* < 0 区域错误情况下 , 返回 NaN, 设置 EDOM 为 errno。
- 如果 *x* = 0, 返回 -∞, 并设置 ERANGE 为 errno。
- 如果 *x* 是非数字 , 返回 NaN。
- 如果 *x* 为 +∞, 返回 +∞。

log10

计算底为 10 的对数

[语法]

```
#include <math.h>
double  log10 ( double x  ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : ithm， x 为 10 作为底 当 *x* < 0 : NaN 当 *x* = 0 : - ∞ 当 *x* = NaN : NaN 当 *x* 为无穷时 : +∞

[说明]

- 计算 x 为 10 作为底*的对数*。
- 在 *x* < 0 区域错误情况下 , 返回 NaN, 设置 EDOM 为 errno。
- 如果 *x* = 0, 返回 -∞并设置 ERANGE 为 errno。
- 如果 *x* 是非数字 , 返回 NaN。
- 如果 *x* 为 +∞, 返回 +∞。

modf

计算小数和整数部分

[语法]

#include <math.h>
double modf (double *x* , double **iptr*) ;

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *iptr* : 指针指向整数部分	正常 : *x* 的分数部分 当 *x* 为非数字或无穷时 : NaN 当 *x* 为 ± 0 : ± 0

[说明]

- 将浮点数 *x* 分为分数部分和整数部分
- 返回分数部分 - 与 *x* 同号 , 通过指针 *iptr* 在显示的地址上存储整数部分。
- 如果 *x* 是非数字 , 返回 NaN 并通过指针 *iptr* 存储在显示地址上。
- 如果 *x* 是无穷的 , 返回 NaN 并通过指针 *iptr* 存储在显示地址上，并设置 EDOM 为 errno。
- 如果 *x* = ± 0, 通过指针 *iptr* 存储 ± 0 在显示地址处。

pow

计算 x 的 y 次幂

[语法]

```
#include <math.h>
double  pow ( double x , double y ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 乘数	正常 : *x* ^ *y* 无论 *x* = NaN 或 *y* = NaN 时 , *x* = +∞ 和 *y* = 0, *x* < 0 和 *y* ≠ 整数 , *x* < 0 andy = ± ∞, *x* = 0 和 *y* <= 0 : NaN 当产生溢出时 : HUGE_VAL (溢出值符号) 当产生下溢出时： 非正常数字 当由于下溢出产生有效数字丢失时： ± 0

[说明]

- 计算 x ^ y.
- 无论 *x* = NaN 或 *y* = NaN 时 , 返回 NaN。
- Either 当 *x* = ± ∞和 *y* = 0, *x* < 0 and *y* ≠ 整数 , *x* < 0 和 *y* = ± ∞或 *x* = 0 和 *y* <= 0, 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生溢出，返回带溢出值符号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果产生下溢出，返回非正常数字。
- 如果由于产生下溢出造成有效数字丢失，返回 ± 0。

sqrt

计算平方根

[语法]

```
#include <math.h>
double  sqrt ( double x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 *x*>=0 : *x* 的平方根 当 *x* < 0 : 0 当 *x* = NaN : NaN 当 *x* = ± 0 : ± 0

[说明]

- 计算 *x* 的平方根。
- 在 *x* <0 区域错误情况下 , 返回 0 并设置 EDOM 为 errno。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± 0, 返回 ± 0。

ceil

计算最小整数而不比 x 小

[语法]

#include <math.h>
double ceil (double *x*) ;

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : 不小于 *x* 的最小整数 当 *x* 为非数字或当 *x* 为无穷时 : NaN 当 *x* = -0 : +0 当不能表达大于 *x* 的最小整数 : x

[说明]

- 计算不小于 *x* 的最小整数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果 *x* 为 -0, 返回 +0。
- 如果不能表达大于 *x* 的最小整数，返回 *x* 。

fabs

计算浮点数 x 的绝对值

[语法]

```
#include <math.h>
double  fabs ( double  x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 计算数字值的绝对值	正常 : *x* 的绝对值 当 *x* = NaN : NaN 当 *x* = -0 : +0

[说明]

- 计算 *x* 的绝对值
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 -0, 返回 +0。

floor

计算最大整数而不比 x 大

[语法]

#include <math.h>
double floor (double *x*) ;

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : 不大于 *x* 的最大整数 当 *x* 为非数字或当 *x* 为无穷时 : NaN 当 *x* = -0 : +0 当不能表达不大于 *x* 的最大整数 : x

[说明]

- 计算不大于 *x* 的最大整数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果 *x* 为 -0, 返回 +0。
- 如果不能表达小于 *x* 的最大整数，返回 *x* 。

fmod

计算 x/y 的余数

[语法]

#include <math.h>
double fmod (double *x* , double *y*)

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 进行数字值运算	正常 : x/y 的余数 当 *x* 为非数字或当 *y* 为非数字时， 当 *y* 为 ± 0, 当 *x* 为 ±∞: NaN 当 *x* ≠ ∞和 *y* = ± ∞: x

[说明]

- 用 x - i * y 表达式计算 x/y 的余数。 i 为整数。
- 如果 *y* ≠ 0, 返回值有和 *x* 相同的符号且绝对值小于 *y*。
- 如果 *x* 为非数字或当 *y* 为非数字时，返回 NaN。
- 如果 *y* 为 ± 0 或 *x* = ± ∞, 返回 NaN 且设置 EDOM 为 errno。
- 如果 *y* 为无穷 , *返回 x* 除非 *x* 为无穷。

matherr

获取库操作浮点数的异常处理

[语法]

```
#include <math.h>
void  matherr ( struct exception *x ) ;
```

[参数 / 返回值]

参数	返回数值

`struct  exception {`
`int  type ;`
`char  *name ;`
`}`
 	
type :
数值显示算数异常
name :
函数名称
	None

[说明]

- 产生异常时，自动调用标准库和运行时间库中的 matherr 来处理浮点数。
- 从标准库中调用时，设置 EDOM 和 ERANGE 为 errno。
  以下显示算数异常类型和 errno 的关系。

类型	算数异常	值设置为 errno
1	下溢出	范围错误
2	丢失	范围错误
3	溢出	范围错误
4	除数为零	EDOM
5	不能操作	EDOM

  通过更改或创建 matherr，对原有错误进行处理。
- 参数总是 near 指针，因为在内部 RAM 中它指向异常结构体。这是没有 matherr_n/matherr_f 函数的原因。

acosf

计算 acos

[语法]

```
#include <math.h>
float  acosf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 -1 <= *x* <= 1 : *x* 的 acos 当 *x* < -1, 1 < *x*, *x* = NaN : NaN

[说明]

- 计算 x 的 acos(范围在 0 和 π 间)。
- 在 *x* < -1, 1 < *x* 定义区域错误的情况下 , 返回 NaN 且设置 EDOM 为 errno。
- 如果 *x* 是个非数字值时 , 返回 NaN。

asinf

计算 asin

[语法]

```
#include <math.h>
float  asinf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 -1 <= *x* <= 1 : *x* 的 acos 当 *x* < -1, 1 < *x*, *x* = NaN : NaN 当 *x* = -0 : -0 当产生下溢出时： 非正常数字

[说明]

- 计算 x 的 asin (范围在 $-\pi/2$ 和 $+\pi/2$ 间) 。
- 在 *x* < -1, 1 < *x* 定义区域错误的情况下 , 返回 NaN 且设置 EDOM 为 errno。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* =-0, 返回 -0。
- 如果运行结果中产生下溢出，返回非正常数字。

atanf

计算 atan

[语法]

```
#include <math.h>
float  atanf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 atan 当 *x* = NaN : NaN 当 *x* = -0 : -0 当产生下溢出时： 非正常数字

[说明]

- 计算 x 的 atan (范围在 -π/2 和 + π/2)。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* =-0, 返回 -0。
- 如果运行结果中产生下溢出，返回非正常数字。

atan2f

计算 y/x 的 atan

[语法]

```
#include <math.h>
float  atan2f ( float y , float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 进行数字值运算	正常 : y/x 的 atan 当 *x* 和 *y* 都为 0 或 不能表达 y/ *x* *的值* , 或无论 *x* 或 *y* 为 NaN, 则 *x* 和 *y* 都为无穷 : NaN 当产生下溢出时: 非正常数字

[说明]

- 计算 y/x 的 atan (范围在 -π 和 +π 间)。
- 当 *x* 和 *y* 都为 0 或 不能表达 y/x 的值时 , 或当 *x* 和 *y* 都为无穷时，返回 NaN 并设置 EDOM 为 errno。
- 无论 *x* 还是 *y* 是非数字值时 , 返回 NaN。
- 如果运行结果中产生下溢出，返回非正常数字。

cosf

计算 cos

[语法]

```
#include <math.h>
float  cosf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 cos 当 *x* = NaN 时 ,*x* 为无穷 : NaN

[说明]

- 计算 *x* 的 cos
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

sinf

计算 sin

[语法]

```
#include <math.h>
float  sinf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 sin 当 *x* = NaN 时 ,*x* 为无穷 : NaN 当产生下溢出时： 非正常数字

[说明]

- 计算 *x* 的 sin
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

tanf

计算 tan

[语法]

```
#include <math.h>
float  tanf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 tan 当 *x* = NaN 时 ,*x* 为无穷 : NaN 当产生下溢出时： 非正常数字

[说明]

- 计算 *x* 的 tan。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 如果 *x* 的绝对值特别大 , 操作结果就无意义了。

coshf

计算 cosh

[语法]

```
#include <math.h>
float  coshf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 cosh 当 *x* = NaN : NaN 当 *x* 为无穷时 : + 当产生溢出时 : HUGE_VQAL (带正号)

[说明]

- 计算 *x* 的 cosh
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回正的无穷值。
- 如果运行结果中产生溢出，返回带正号的 HUGE_VAL，并设置 ERANGE 为 errno。

sinhf

计算 sinh

[语法]

```
#include <math.h>
float  sinhf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 sinh 当 *x* = NaN : NaN 当 *x* = ± ∞: ± ∞ 当产生溢出时 : HUGE_VAL (溢出值符号) 当产生下溢出时： ± 0

[说明]

- 计算 *x* 的 sinh。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± ∞, 返回 ± ∞。
- 如果运行结果中产生溢出，返回带溢出值符号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果运行结果中产生下溢出，返回 +± 0。

tanhf

计算 tanh

[语法]

```
#include <math.h>
float  tanhf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的 tanh 当 *x* = NaN : NaN 当 *x* = ± ∞: ± 1 当产生下溢出时： ± 0

[说明]

- 计算 *x* 的 tanh。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± ∞ 返回 ± 1。
- 如果运行结果中产生下溢出，返回 ± 0。

expf

计算指数函数

[语法]

```
#include <math.h>
float  expf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 指数函数 当 *x* = NaN : NaN 当 *x* = +∞: + 当 *x* = -∞: +0 当产生溢出时 : HUGE_VQAL (带正号) 当产生下溢出时： 非正常数字 当由于下溢出产生有效数字丢失时： +0

[说明]

- 计算 *x* 指数函数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 +∞, 返回 +∞。
- 如果 *x* 为 -∞, 返回 +0。
- 如果运行结果中产生溢出，返回带正号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果操作结果中产生下溢出，返回非正常数字。
- 如果运行结果由于产生下溢出造成有效数字丢失，则返回 +0。

frexpf

计算小数和指数部分

[语法]

```
#include <math.h>
float  frexpf ( float  x , int *exp ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *exp* : 指针指向存储指数部分	正常 : *x* 的小数部分 当 *x* = NaN, *x* = ± ∞时： NaN 当 *x* = ± 0 : ± 0

[说明]

- 将浮点数 *x* 分为小数部分 m 和指数部分 n 例如：*x* = m * 2 ^ n 并返回小数部分 m。
- 指数 n 存储在显示 *exp* 指针的地方。然而绝对值 m 为 0.5 或 更大且小于 1.0。
- 如果 *x* 是非数字 , 返回 NaN 和 **exp* 为 0 的值。
- 如果 *x* 为 ± ∞, 返回 NaN, 并设置 EDOM 为 errno 带 **exp* 为 0。
- 如果 *x* 为 ± 0, 返回 ± 0 和 **exp* 为 0 的值。

ldexpf

计算 x * 2 ^ exp

[语法]

```
#include <math.h>
float  ldexpf ( float x , int exp ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *exp* : 求幂	正常 : *x* * 2 ^ *exp* 当 *x* = NaN : NaN 当 *x* = ± ∞: ± ∞ 当 *x* = ± 0 : ± 0 当产生溢出时 : HUGE_VAL (溢出值符号) 当产生下溢出时： 非正常数字 当由于下溢出产生有效数字丢失时： ± 0

[说明]

- 计算 x * 2 ^ exp。
- 如果 *x* 是个非数字值时 , 返回 NaN。如果 *x* 为 ± ∞ 返回 ± ∞。如果 *x* 为 ± 0, 返回 ± 0。
- 如果运行结果中产生溢出，返回带溢出值符号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果操作结果中产生下溢出，返回非正常数字。
- 如果运行结果由于产生下溢出，有效数字丢失时，返回 ± 0。

logf

计算自然对数

[语法]

```
#include <math.h>
float  logf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : *x* 的自然对数 当 *x* < 0 : NaN 当 *x* = 0 : - ∞ 当 *x* = NaN : NaN 当 *x* 为无穷时 : + ∞

[说明]

- 计算 *x* 的自然对数。
- 在 *x* < 0 区域错误情况下 , 返回 NaN, 设置 EDOM 为 errno。
- 如果 *x* = 0, 返回 -∞ 并设置 ERANGE 为 errno。
- 如果 *x* 是非数字 , 返回 NaN。
- 如果 *x* 为 +∞ 返回 +∞。

log10f

计算底为 10 的对数

[语法]

```
#include <math.h>
float  log10f ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : x 为 10 作为底*的对数* 当 *x* < 0 : NaN 当 *x* = 0 : - ∞ 当 *x* = NaN : NaN 当 *x* = -? 时 : + ∞

[说明]

- 计算 x 为 10 作为底*的对数*。
- 在 *x* < 0 区域错误情况下 , 返回 NaN, 设置 EDOM 为 errno。
- 如果 *x* = 0, 返回 -∞, 并设置 ERANGE 为 errno。
- 如果 *x* 是非数字 , 返回 NaN。
- 如果 *x* 为 +∞, 返回 +∞。

modff

计算小数和整数部分

[语法]

```
#include <math.h>
float  modff ( float x , float *iptr ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *iptr* : 指针指向整数部分	正常 : *x* 的分数部分 当 *x* = NaN 时 ,*x* 为无穷 : NaN 当 *x* = ± 0 : ± 0

[说明]

- 将浮点数 *x* 分为分数部分和整数部分。
- 返回分数部分 - 与 *x* 同号 , 通过指针 *iptr* 在显示地址上存储整数部分。
- 如果 *x* 是非数字 , 返回 NaN 并通过指针 *iptr* 存储在显示地址上。
- 如果 *x* 是无穷的 , 返回 NaN 并通过指针 *iptr* 存储在显示地址上，并设置 EDOM 为 errno。
- 如果 *x* = ± 0, 返回 ± 0 并通过指针 *iptr* 存储显示地址。

powf

计算 x 的 y 次幂

[语法]

```
#include <math.h>
float  powf ( float x , float y ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 乘数	正常 : *x* ^ *y* 当 *x* = NaN 或 *y* = NaN *x* = +∞和 *y* = 0 *x* < 0 和 *y* ≠ 整数 , *x* < 0 和 *y* = ± ∞ *x* = 0 和 *y* <= 0 : NaN 当产生溢出时 : HUGE_VAL(溢出值符号) 当产生下溢出时： 非正常数字 当由于下溢出产生有效数字丢失时： ± 0

[说明]

- 计算 x ^ y.
- 无论 *x* = NaN 或 *y* = NaN 时 , 返回 NaN。
- 要么当 *x* = +∞和 *y* = 0, *x* < 0 和 *y*≠ 整数 , *x* < 0 和 *y* = ± ∞, 或 *x* = 0 和 *y* <= 0, 返回 NaN 并设置 EDOM 为 errno。
- 如果运行结果中产生溢出，返回带溢出值符号的 HUGE_VAL，并设置 ERANGE 为 errno。
- 如果产生下溢出，返回非正常数字。
- 如果由于产生下溢出造成有效数字丢失，返回 ± 0。

sqrtf

计算平方根

[语法]

```
#include <math.h>
float  sqrtf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	当 *x*>=0 : *x* 的平方根 当 *x* < 0 : 0 当 *x* = NaN : NaN 当 *x* = ± 0 : ± 0

[说明]

- 计算 *x* 的平方根。
- 在 *x* <0 区域错误情况下 , 返回 0 并设置 EDOM 为 errno。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 ± 0, 返回 ± 0。

ceilf

计算最小整数而不比 x 小

[语法]

```
#include <math.h>
float  ceilf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : 不小于 *x* 的最小整数 当 *x* = NaN 时 ,*x* 为无穷 : NaN 当 *x* = -0 : +0 当不能表达大于 *x* 的最小整数 : *x*

[说明]

- 计算不小于 *x* 的最小整数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 -0, 返回 +0。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果不能表达大于 *x* 的最小整数，返回 *x* 。

fabsf

计算浮点数 x 的绝对值

[语法]

```
#include <math.h>
float  fabsf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 计算数字值的绝对值	正常 : *x* 的绝对值 当 *x* 为非数字时 : NaN 当 *x* = -0 : +0

[说明]

- 计算 *x* 的绝对值
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为 -0, 返回 +0。

floorf

计算最大整数而不比 x 大

[语法]

```
#include <math.h>
float  floorf ( float x ) ;
```

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算	正常 : 不大于 *x* 的最大整数 当 *x* = NaN 时 ,*x* 为无穷 : NaN 当 *x* = -0 : +0 当不能表达不大于 *x* 的最大整数 : x

[说明]

- 计算不大于 *x* 的最大整数。
- 如果 *x* 是个非数字值时 , 返回 NaN。
- 如果 *x* 为无穷 , 返回 NaN 并设置 EDOM 为 errno。
- 如果 *x* 为 -0, 返回 +0。
- 如果不能表达小于 *x* 的最大整数，返回 *x* 。

fmodf

计算 x/y 的余数

[语法]

#include <math.h>
float fmodf (float *x* , float *y*) ;

[参数 / 返回值]

参数	返回数值
x : 进行数字值运算 *y* : 进行数字值运算	正常 : x/y 的余数 当 *y* 为 ± 0 或 *x* 为 ± ?, 当 *x* 为非数字或当 *y* 为非数字时 : NaN 当 *x* ≠∞ 和 *y* = ± ∞: x

[说明]

- 用 x - i * y 表达式计算 x/y 的余数。 i 为整数。
- 如果 *y* ≠ 0, 返回值有和 *x* 相同的符号且绝对值小于 *y*。
- 如果 *y* 为 ± 0 或 *x* = ± ∞, 返回 NaN 且设置 EDOM 为 errno。
- 如果 *x* 为非数字或当 *y* 为非数字时，返回 NaN。
- 如果 *y* 为无穷 , *返回x* 除非 *x* 为无穷。

6.13 诊断函数

以下诊断函数可用。

功能名称	作用
__assertfail	支持宏声明

__assertfail

支持宏声明

[语法]

```
#include <assert.h>
int         __assertfail ( char *__msg , char *__cond , char *__file , int __line ) ;
```

[参数 / 返回值]

参数	返回数值
__msg : 指示字符串的指针输出格式转换说明，并传递给 printf 函数。 *__cond* : assert 宏的实参数 *__file* : 源文件名 *__line* : 源代码的行号	未定义

[说明]

- __assertfail 函数从 assert 宏处接收信息 (参见 6.3.13　assert.h), 调用 printf 函数 , 输出信息 , 并调用中止函数。
- assert 宏用于在程序中添加诊断函数。

  当执行 assert 宏，如果 p 为 false （等于 0）， assert 宏传递与特定调用有关的信息，该特定调用引起 false 值（实参数文本，源文件名称，和包含在信息中的源代码行数。其他两个是 __assertfail 函数的宏 __FILE__ 和 __LINE__ 各自地值。

6.14 库堆栈消耗表

本节介绍库中每个函数消耗堆栈的数量。

6.14.1 标准库

每个标准库中的堆栈函数消耗的堆栈数量显示在以下表中。

(1) ctype.h

功能名称	小模式共享 和中模式	大模式共享
isalpha	0	0
isupper	0	0
islower	0	0
isdigit	0	0
isalnum	0	0
isxdigit	0	0
isspace	0	0
ispunct	0	0
isprint	0	0
isgraph	0	0
iscntrl	0	0
isascii	0	0
toupper	0	0
tolower	0	0
toascii	0	0
_toupper	0	0
toup	0	0
_tolower	0	0
tolow	0	0

(2) setjmp.h

功能名称	小模式共享 和中模式	大模式共享
setjmp	4	4
longjmp	2	2

(3) stdarg.h

功能名称	小模式共享 和中模式	大模式共享
va_arg	0	0
va_start	0	0
va_starttop	0	0
va_end	0	0

(4) stdio.h

功能名称	小模式共享 和中模式	大模式共享
sprintf	58 (130) 注 1	58 (14030) 注 1
sscanf	294 (332) 注 1 (350) 注 2	294 (340) 注 1 (358) 注 2
printf	70 (128) 注 1	7 (4138) 注 1
scanf	308 (3302) 注 1 (348) 注 2	312 (338) 注 1 (356) 注 2
vprintf	70 (12830) 注 1	76 (14030) 注 1
vsprintf	58 (13030) 注 1	58 (14) 注 1
getchar	0	0
gets	8	14
putchar	0	0
puts	6	10
__putc	4	4

注　**1.** 括号中的值作为支持浮点数的版本使用。

2. 当支持浮点数的版本产生操作异常时，出现括号中的值。

(5) stdlib.h

功能名称	小模式共享 和中模式	大模式共享
atoi	4	4
atol	10	10
strtol	20	20
strtoul	20	20
calloc	12	12
free	8	8
malloc	6	6
realloc	12	12
abort	0	0
atexit	0	0
exit	6 + n 注 1	6 + n 注 1
abs	0	0

功能名称	小模式共享 和中模式	大模式共享
labs	0	0
div	6 (230) 注 2	6 (230) 注 2
ldiv	18 (830) 注 2	18 (830) 注 2
brk	0	0
sbrk	2	2
atof	46 (6430) 注 3	46 (6430) 注 3
strtod	46 (6430) 注 3	48 (6630) 注 3
itoa	10	10
ltoa	16	16
ultoa	16	16
rand	18 (14) 注 4	18 (14) 注 4
srand	0	0
bsearch	36 + n 注 5	40 + n 注 5
qsort	16 + n 注 6	18 + n 注 6
strbrk	0	0
strsbrk	2	2
stritoa	10	10
strltoa	16	16
strultoa	16	16

注 **1.** n 为通过 atexit 函数注册的外部函数消耗堆栈的总量。

2. 括号中的值只有在使用乘数 / 除数时使用。

3. 括号中的值只有在支持浮点数的版本产生操作异常时使用。

4. 当括号中的值作为乘数 / 除数的乘数使用。

5. n 是通过 bsearch 调用外部函数的堆栈消耗数。

6. n 是 (X+ (从 qsort 中调用外部函数的堆栈消耗量)) - (1 + (循环调用数量))。

当使用小模式和中模式的共享库时 : X = 38

当使用大模式的库共享时 : X = 40

(6) string.h

功能名称	小模式共享 和中模式	大模式共享
memcpy	4	8
memmove	4	6
strcpy	2	6
strncpy	4	10
strcat	2	6
strncat	4	8
memcmp	2	4
strcmp	2	2
strncmp	2	2

功能名称	小模式共享 和中模式	大模式共享
memchr	2	4
strchr	4	2
strrchr	4	6
strspn	4	6
strcspn	4	4
strpbrk	4	6
strstr	4	8
strtok	4	4
memset	4	6
strerror	0	0
strlen	0	0
strcoll	2	2
strxfrm	4	4

(7) math.h

功能名称	小模式共享 和中模式	大模式共享
acos	30 (48) 注	30 (48) 注
asin	30 (48) 注	30 (48) 注
atan	30 (48) 注	30 (48) 注
atan2	30 (48) 注	30 (48) 注
cos	28 (46) 注	28 (46) 注
sin	28 (46) 注	28 (46) 注
tan	34 (52) 注	34 (52) 注
cosh	34 (52) 注	34 (52) 注
sinh	34 (52) 注	34 (52) 注
tanh	40 (58) 注	40 (58) 注
exp	30 (48) 注	30 (48) 注
frexp	2 (16) 注	4 (16) 注
ldexp	0 (16) 注	0 (16) 注
log	30 (48) 注	30 (48) 注
log10	30 (48) 注	30 (48) 注
modf	2 (16) 注	4 (16) 注
pow	30 (48) 注	30 (48) 注
sqrt	22 (40) 注	22 (40) 注
ceil	2 (16) 注	2 (16) 注
fabs	0	0
floor	2 (16) 注	2 (16) 注
fmod	2 (16) 注	2 (16) 注

功能名称	小模式共享 和中模式	大模式共享
matherr	0	0
acosf	30 (48) 注	30 (48) 注
asinf	30 (48) 注	30 (48) 注
atanf	30 (48) 注	30 (48) 注
atan2f	30 (48) 注	30 (48) 注
cosf	28 (46) 注	28 (46) 注
sinf	28 (46) 注	28 (46) 注
tanf	34 (52) 注	34 (52) 注
coshf	34 (52) 注	34 (52) 注
sinhf	34 (52) 注	34 (52) 注
tanhf	40 (58) 注	40 (58) 注
expf	30 (48) 注	30 (48) 注
frexpf	2 (16) 注	4 (16) 注
ldexpf	0 (16) 注	0 (16) 注
logf	30 (48) 注	30 (48) 注
log10f	30 (48) 注	30 (48) 注
modff	2 (16) 注	4 (16) 注
powf	30 (48) 注	30 (48) 注
sqrtf	22 (40) 注	22 (40) 注
ceilf	2 (16) 注	2 (16) 注
fabsf	0	0
floorf	2 (16) 注	2 (16) 注
fmodf	2 (16) 注	2 (16) 注

注 因为产生操作异常时括号中才有值。

(8) assert.h

功能名称	小模式共享 和中模式	大模式共享
__assertfail	82 (140) 注	92 (156) 注

注 因为在使用支持浮点数的 printf 版本时，括号中才有值。

6.14.2　运行时间库

每个运行时间库中堆栈函数消耗的堆栈数量显示在以下表中。

(1) 递增

功能名称	堆栈消耗
lsinc	0
luinc	0
finc	16 (34) 注

注　因为产生操作异常时括号中才有值。

(2) 递减

功能名称	堆栈消耗
lsdec	0
ludec	0
fdec	16 (34) 注

注　因为产生操作异常时括号中才有值。

(3) 符号颠倒

功能名称	堆栈消耗
lsrev	2
lurev	2
frev	0

(4) 第一个附加物

功能名称	堆栈消耗
lscom	0
lucom	0

(5) 逻辑（非）NOT

功能名称	堆栈消耗
lsnot	0
lunot	0

(6) 乘

功能名称	堆栈消耗
csmul	0
cumul	0
iumul	4 (2) 注 1

功能名称	堆栈消耗
lsmul	8 (4) 注 1
lumul	8 (4) 注 1
fmul	8 (26) 注 2

注 **1.** 括号中的值只有使用 乘数 , 在乘数 / 除数中时使用。
2. 因为产生操作异常时，括号中才有值。

(7) 除

功能名称	堆栈消耗
csdiv	8 (10) 注 1
cudiv	2 (4) 注 1
isdiv	12 (8) 注 1
iudiv	6 (2) 注 1
lsdiv	12 (8) 注 1
ludiv	6 (2) 注 1
fdiv	8 (26) 注 2

注 **1.** 括号中的值只有在使用乘数 / 除数时使用。
2. 因为产生操作异常时，括号中才有值。

(8) 余数

功能名称	堆栈消耗
csrem	8 (10) 注
curem	2 (4) 注
isrem	12 (8) 注
iurem	6 (2) 注
lsrem	12 (8) 注
lurem	6 (12) 注

注 括号中的值只有在使用乘数 / 除数时使用 。

(9) 加

功能名称	堆栈消耗
lsadd	0
luadd	0
fadd	8 (26) 注

注 因为产生操作异常时，括号中才有值。

(10) 减

功能名称	堆栈消耗
lssub	2
lusub	2
fsub	8 (26) 注

注　　因为产生操作异常时，括号中才有值。

(11) 向左移位

功能名称	堆栈消耗
lslsh	4
lulsh	4

(12) 向右移位

功能名称	堆栈消耗
lsrsh	4
lursh	4

(13) 比较

功能名称	堆栈消耗
cscmp	0
iscmp	0
lscmp	2
lucmp	2
fcmp	4 (24) 注

注　　因为产生操作异常时，括号中才有值。

(14) 位与

功能名称	堆栈消耗
lsband	0
luband	0

(15) 位或

功能名称	堆栈消耗
lsbor	0
lubor	0

(16)位异或

功能名称	堆栈消耗
lsbxor	0
lubxor	0

(17)转换浮点数

功能名称	堆栈消耗
ftols	6
ftolu	6

(18) 转换为浮点数

功能名称	堆栈消耗
lstof	6
lutof	6

(19) 位的转换

功能名称	堆栈消耗
btol	0

(20) 启动程序

功能名称	堆栈消耗
cstart	4

(21) 函数的预处理和后处理

功能名称	堆栈消耗
cpre3e	基址指针的大小 + 第一个参数 + 寄存变量 + 自动变量
cprep3	基址指针的大小 + 第一个参数 + 寄存变量 + 自动变量
cdis3e	0
cdisp3	0
hdwinit	0

(22) BCD- 类型转换

功能名称	作用
bcdtob	6
btobcd	6
bcdtow	6
wtobcd	8
bbcd	6

(23) 辅助

功能名称	作用
bcdtob	6
btobcd	6
bcdtow	6
wtobcd	8
bbcd	6

功能名称	堆栈消耗
divuw	6 (2) 注
df1in	0
dn4in	0
dn4ip	4
df4in	0
df4ip	4
dn4ino	0
dn4ipo	4
df4ino	0
df4ipo	4
df1de	0
dn4de	0
dn4dp	4
df4de	0
df4dp	4
dn4deo	0
dn4dpo	4
df4deo	0
df4dpo	4
indao	0
ifdao	0
inado	0
ifado	0
lnd0	2
lfd0	2
ln0d	0
lf0d	0
lnd0o	2
lfd0o	2
ln0do	0
lf0do	0

注　括号中的值只有在使用乘 / 除时使用。

6.15 库中最大中断禁用时间列表

使用 乘数的库 , 乘数 / 除数 , 允许一段时间内中断禁用为了在中断期间操作内容没有破坏。
库中使用乘数 , 乘数 / 余数的最大中断禁用的次数显示如下。
在库中允许中断禁用期间没有一段时间，用来使用 乘数 , 乘数 / 余数。

表 6-2. 库的最大中断禁用时间 （时钟数）

分类	功能名称	最大中断禁止时间		备注
		当使用乘数时	当使用乘数时 / 余数时	
乘法	@@iumul	12	12	在无符号 int 数据间进行乘法
	@@lumul	24	24	在无符号 long 数据间进行乘法
	@@lsmul	24	24	在符号 long 数据间进行乘法
除法	@@cudiv	-	40	在无符号 char 数据间进行除法
	@@csdiv	-	40	在符号 char 数据间进行除法
	@@iudiv	-	39	在无符号 int 数据间进行除法
	@@isdiv	-	39	在符号 int 数据间进行除法
	@@ludiv	-	43	在无符号 long 数据间进行除法
	@@lsdiv	-	43	在符号 long 数据间进行除法
余数计算	@@curem	-	40	在无符号 char 数据间进行余数计算
	@@csrem	-	40	在符号 char 数据间进行余数计算
	@@iurem	-	39	在无符号 int 数据间进行余数计算
	@@isrem	-	39	在符号 int 数据间进行余数计算
	@@lurem	-	43	在无符号 long 数据间进行余数计算
	@@lsrem	-	43	在符号 long 数据间进行余数计算
辅助	@@divuw	?|	43	divuw 指令兼容
stdio.h	printf	?|	43 注	输出数据到 SFR
	sprintf	?|	43 注	写输入到字符串
	vprintf	?|	43 注	输出数据到 SFR
	vsprintf	?|	43 注	写输入到字符串

分类	功能名称	最大中断禁止时间		备注
		当使用乘数时	当使用乘数时 / 余数时	
stdlib.h	div	?|	41	进行 int 型除法
	ldiv	?|	46	进行 long 型除法
	rand	24	24	Uses @@lumul
	qsort	12	12	Uses @@iumul

注 括号中的值作为支持浮点数的版本使用。

6.16 用于启动程序升级的批处理文件和库函数。

78K0R C 编译器为标准库函数和启动程序部分的升级提供批处理文件。在 bat 文件夹中的批处理文件显示在以下表中。

表 6-3. 用于升级库函数的批处理文件

批处理文件	应用
mkstup.bat	升级启动程序 （cstart*.asm)。 当改变启动程序时，使用这个批处理文件进行汇编。
reprom.bat	升级固件 ROM 的结束程序 (rom.asm)。 当改变 rom.asm 时，使用这个批处理文件升级库。
repgetc.bat	升级 getchar 函数 默认设想设置 SFR 的 P0 为输入端口。当有必要改变这个设置时，改变定义在 getchar.asm 中的 PORT EQU 的值并且使用这个批处理文件升级库。
repputc.bat	升级 putchar 函数 默认设想设置 SFR 的 P0 为输出端口。当有必要改变这个设置时，改变定义在 getchar.asm 中的 PORT EQU 的值并且使用这个批处理文件升级库。
repputcs.bat	为了和 C 编译器兼容，升级 putchar 函数为 SM+ 78K0R。 当有必要校验 putchar 函数的输出时，使用 SM+ 为 78K0R 和 C 编译器 , 使用这个批处理文件升级库。
repselo.bat	保存 / 恢复 C 编译器 (_@KREGxx) 的保留区域，当作 setjmp 和 longjmp 函数中部分保存 / 恢复处理。 当设定 -qr 选项时，使用这个批处理文件升级库。
repselon.bat	不保存 / 恢复编译器的保留区域 (_@KREGxx), 当作 setjmp 和 longjmp 函数的部分保存 / 恢复处理。 当没有设定 -qr 选项时，使用这个批处理文件升级库。
repvect.bat	升级在闪存中分配的中断向量表的子表的地址值设置过程 (vect*.asm)。 默认预想设置闪存中分支表的开始地址为 2000H。当有必要改变这个设置时，改变定义在 vect.inc 中的 ITBLTOP EQU 的值并且使用这个批处理文件升级库。
repmul.bat	升级乘数库。
repmuldiv.bat	升级乘数 / 除数库。

6.16.1　使用批处理文件

使用在子文件夹 bat 中的批处理文件。

因为这些批处理文件用于激活汇编语言和库管理程序，绑定汇编语言等到 CubeSuite 中是必须的。在使用批处理文件之前，使用环境变量 PATH 设置包含 78K0R 汇编语言执行格式文件的文件夹。

为批处理 bat 文件创建同级子目录 (lib) 并且在这个子文件夹中放入汇编后的文件。当在与 bat 同级的子文件夹 lib 中安装 C 启动程序或库时，则会覆盖这些文件。

输出包含批处理文件的汇编文件到 Src\cc78k0\lib。在链接前，复制这些文件到 lib78k0 目录。

为了使用批处理文件，移动当前文件夹到子文件 bat 中并执行每个批处理文件。为了进行执行，下列参数是必须的。

产品类型 = 芯片类型 (目标芯片的分类)

f1166a0 ： uPD78F1166_A0 等。

以下是描述如何使用每个批处理文件。

(1) 启动程序

```
mkstup chiptype
```

例子如下。

```
mkstup f1166a0
```

(2) 固件 ROM 程序的升级

```
reprom chiptype
```

例子如下。

```
reprom f1166a0
```

(3) getchar 函数的升级

```
regetc chiptype
```

例子如下。

```
repgetc f1166a0
```

(4) putchar 函数的升级

```
repputc chiptype
```

例子如下。

```
repputc f1166a0
```

(5) putchar 函数 (支持 SM78K0R) 的升级

```
repputcs chiptype
```

例子如下。

```
repputcs f1166a0
```

(6) setjmp/longjmp 函数的升级 (恢复 / 保存处理)

```
repselo chiptype
```

例子如下。

```
repselo f1166a0
```

(7) setjmp/longjmp 函数的升级 (无恢复 / 保存处理)

```
repselon chiptype
```

例子如下。

```
repselon f1166a0
```

(8) 中断向量表的升级

```
repvect chiptype
```

例子如下。

```
repvect f1166a0
```

(9) 乘数使用库的升级

```
repmul.bat chiptype
```

UPD78F1235_64 的升级例子显示如下。

```
repmul.bat f123564
```

升级如下。

src\cc78k0r\lib\cl0rxm.lib
cl0rxme.lib
cl0rxl.lib
cl0rxle.lib

(10) 乘数 / 除数使用库的升级

```
repmuldiv.bat chiptype
```

UPD78F1235_64 髓的升级例子显示如下。

```
repmuldiv.bat f123564
```

升级如下。

src\cc78k0r\lib\cl0rdm.lib
cl0rdme.lib
cl0rdl.lib
cl0rdle.lib

第 7 章 启动例程

本章介绍启动程序。

7.1 功能概述

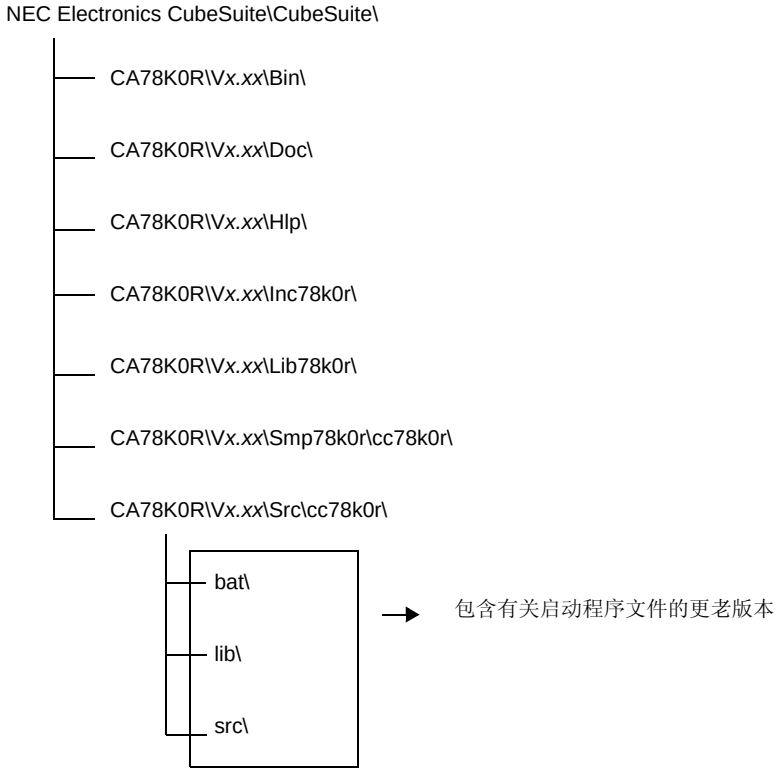
为了执行 C 语言程序，需要有一个程序来激活系统和用户程序 （main 函数） ROM 化过程。这个程序叫做启动程序。

为了执行用户程序，必须为程序创建启动程序。 CA78K0R 提供标准启动程序目标文件，其中包含程序执行前必须的处理和启动例程的源文件 （汇编源代码），用户可以修改启动例程的源文件来满足具体的系统需求。将启动例程的目标文件链接到用户程序，就可以创建一个可执行的程序。即使用户没有描述预处理执行过程，也同样可以成功创建。

本章介绍启动程序使用的内容，使用方法和修改方法。

7.2 文件结构

有关启动程序的文件都存储在 C 编译包的文件夹 Src\cc78k0r 中。



在 Src\cc78k0r 下的文件夹内容显示如下。

7.2.1 "bat" 文件夹内容

在该文件夹中的批处理文件不能在 IDE 中使用。
只有必须修改源程序时例如启动程序，才使用这些批处理文件。

表 7-1. "bat" 文件夹内容

批处理文件名称	说明
mkstup.bat	启动程序的汇编批处理文件
reprom.bat	用于更新 rom.asm 的批处理文件[注 1]
repgetc.bat	用于更新 getchar.asm 的批处理文件
repputc.bat	用于更新 putchar.asm 的批处理文件
repputcs.bat	用于更新 _putchar.asm 的批处理文件
repselo.bat	用于更新 setjmp.asm 和 longjmp.asm 的批处理文件 (保存编译器保留区域) [注 2]
repselon.bat	用于更新 setjmp.asm,longjmp.asm 的批处理文件 (不保存编译器保留区域)) [注 2]
repvect.bat	用于更新 vect*.asm 的批处理文件
repmul.bat	用于更新 乘法库的批处理文件
repmuldiv.bat	用于更新 乘法 / 除法库的批处理文件

注 **1.** ROM 化程序在库中 , 所以由该批处理文件来更新库。
2. 保存编译器保留区域的 setjmp 和 longjmp 函数 （KREGxx, 等准备好 saddr 区域），以及 setjmp 和 longjmp 函数不保存创建的编译器保留区域 （值保存寄存器）。

7.2.2 "lib" 文件夹内容

lib 文件夹包含从启动程序源文件和库中汇编而来的目标文件。这些目标文件能链接任何目标设备程序 78K0R。如果不需要特别修改代码，则链接默认的目标文件。当执行由 CA78K0R 提供的批处理文件 mkstup.bat 时 , 覆盖目标文件。

表 7-2. "lib" 文件夹内容

文件名			文件规则
普通	引导区域	Flash 区域	
cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0rxl.lib cl0rdl.lib	cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0rxl.lib cl0rdl.lib	cl0rme.lib cl0rle.lib cl0rmfe.lib cl0rlfe.lib cl0rxme.lib cl0rdme.lib cl0rxle.lib cl0rdle.lib	库 (运行时间和标准库)[注 1]
s0rm.rel s0rml.rel s0rl.rel s0rll.rel	s0rmb.rel s0rmlb.rel s0rlb.rel s0rllb.rel	s0rme.rel s0rmle.rel s0rle.rel s0rlle.rel	启动程序的目标文件[注 2]

注 1.　如下为根据规则给库的命名。

```
lib78k0r\cl0r<mul><model><float><flash>.lib
```

< *乘法* >
　None ： 标准库
　x　　： 使用乘法
　d　　： 使用乘法 / 除法

< *模式* >
　m　　： 小型或中型
　l　　： 大型
< *浮点* >
　None ： 标准库和运行时间库 (不使用浮点库)
　f　　： 浮点库
<flash>
　None ： 正常 / 启动区域
　e　　： flash 存储区域

2.　如下为根据规则给库的命名。

```
lib78k0r\s0r<model><lib><flash>.rel
```

< *模式* >
　m　　： 中型 (也能用于设定小型)
　l　　： 大型
<lib>
　None ： 当不使用标准库函数时
　l　　： 当不使用标准库函数时
<flash>
　None ： 正常
　b　　： 启动区域
　e　　： flash 存储区域

3.　78K0R C 编译库与下列乘法器 , 乘法 / 除法设备相兼容。
但是，如果在计算过程中发生中断，为了它们不出错，从中断点以后部分的计算结果无效。
有关库函数和中断禁止次数的内容，参阅 "6.15　库中最大中断禁用时间列表 "。
以下为特殊的功能寄存器。

功能	保留字节	地址	容量
对数据 A 做乘法	MULA	FFFF0H	16 位
对输入数据 B 做乘法	MULB	FFFF2H	16 位
相乘的结果	MULOH, MULOL	FFFF4H, FFF6H	16 位 x 2

- 寄存器配置

< 乘数 A >　　< 乘数 B >　　< 乘积 >
MULA (bits 15 to 0) x MULB (bits 15 to 0) = MULOH (upper) (bits 15 to 0), MULOL (lower) (bits 15 to 0)

7.2.3 "src" 文件夹内容

src 文件夹包含启动程序的汇编程序源文件， ROM 程序，错误处理程序，和标准库函数 （部分）。如果应用系统要求源程序必须修改，可以对这个程序修改，并使用 BAT 文件夹中的一个批处理文件进行汇编，由此创建链接所需的目标文件。

表 7-3. "src" 文件夹内容

启动程序源文件名称	说明
cstart.asm 注	启动程序源文件 (使用标准库时)
cstartn.asm 注	启动程序源文件 (不使用标准库时)
rom.asm	ROM 化程序的源文件
_putchar.asm	_putchar 函数
putchar.asm	putchar 函数
getchar.asm	getchar 函数
longjmp.asm	longjmp 函数
setjmp.asm	setjmp 函数
vectxx.asm	每个中断的向量源 （xx: 向量地址）
def.inc	根据类型设置库
macro.inc	各典型模式的宏定义
vect.inc	flash 存储区域分支表的起始地址
library.inc	明确选择分配给启动区域库
imul.asm, lmul.asm	乘法函数，乘法 / 除法库
csdiv.asm, cudiv.asm, csrem.asm, curem.asm, isdiv.asm, iudiv.asm, isrem.asm, iurem.asm, lsdiv.asm, ludiv.asm, lsrem.asm, lurem.asm, divuw.asm, div.asm, ldiv.asm	乘法 / 除法库函数

注 文件名中 "n" 的启动程序不包含标准库处理。只有在未使用标准库时，另外， cstarte*.asm 是引导区域的启动例程， cstarte*.asm 是 flash 区域启动程序。

7.3 批处理文件说明

本节介绍批处理文件

7.3.1 创建启动程序的批处理文件

在 bat 文件夹中的 mkstup.bat 用于创建启动程序的目标文件。

在 CA78K0R 汇编程序中对必须用 mkstup.bat。因此，如果没有指定路径，在运行批处理文件前设定它。

以下介绍如何使用这个文件。

- 在包含 mkstup.bat 的 Src\cc78k0r\bat 文件夹中执行以下命令行。

```
mkstup device-type注
```

注 参阅 目标设备的用户手册或 " 设备文件操作注意事项 "。

使用例子描述如下。

- 将要创建的启动程序使用的目标设备是 uPD78F1166_A0。

```
mkstup f1166a0
```

批处理文件 mkstup.bat 存储新的启动程序，为了在与 bat 文件夹同级的 lib 文件夹中覆盖启动程序的目标文件。需要链接目标文件的启动程序将输出到每个文件夹中。

在 lib 中创建目标文件名显示如下。

lib —— s0rm.rel
s0rmb.rel
s0rme.rel
s0rml.rel
s0rmlb.rel
s0rmle.rel
s0rl.rel
s0rlb.rel
s0rle.rel
s0rll.rel
s0rllb.rel
s0rlle.rel

7.4 启动程序

本章介绍启动程序。

7.4.1 启动程序概述

启动程序的作用是为了执行用户编写的 C 源程序而作的准备工作。通过链接到用户程序，可以创建载入模块文件，已达到完成目标。

(1) 功能

存储器初始化，包含在系统中 ROM 化和 C 源程序的进行开始和终止处理。

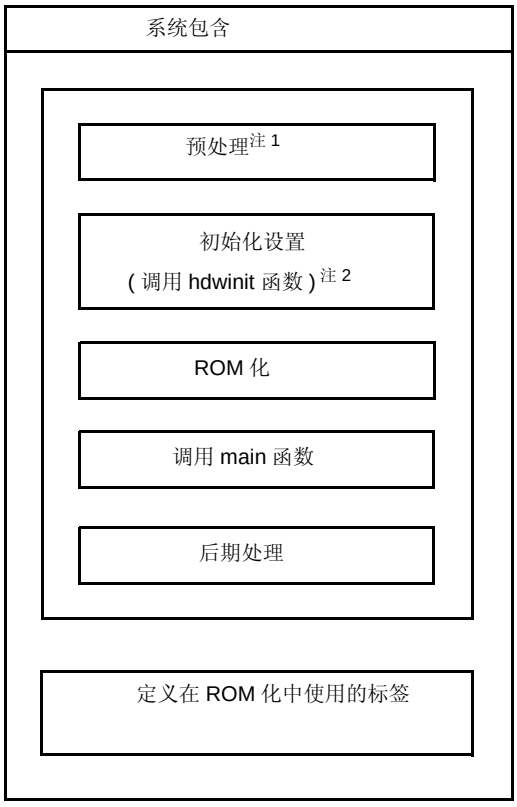
- ROM 化

  在 C 源程序中定义的外部变量，静态变量，和 sreg 变量的初始值存放于 ROM 中。然而 ROM 中的变量值不能重写，只能在存放在 ROM 中保持不变。因此，定义到 ROM 中的初始值必须复制到 RAM 中运行。这个过程叫作 ROM 化。当程序写入到 ROM 时，可由微控制器来调用执行。

(2) 配置

下图显示与启动程序有关的程序和配置。

图 7-1. 与启动程序有关的程序及它们的配置



注 1. 如果使用标准库，首先进行库有关的处理。启动程序源文件中名称后面没有追加 "n" 的文件在处理时与标准库有关。文件末加 "n" 的文件不需要处理。

2. hdwinit 函数是当用户外围设备的函数 (sfr) 初始化时创建的函数。通过创建 hdwinit 函数，能加速初始化设置的时间 （初始化设置可以在 main 函数中进行）。如果用户不创建 hdwinit 函数，则不做任何处理就直接返回。

cstart.asm 和 cstartn.asm 的内容几乎相同。
下表显示 cstart.asm 和 cstartn.asm 的区别。

启动程序类型	使用库处理
cstart.asm	是
cstartn.asm	No

(3) 启动程序的使用

下表显示由 CA78K0R 提供的源文件的目标文件名称列表。

文件类型	源文件	目标文件
启动程序	cstart*.asm 注 1, 2	s0r*.rel 注 2, 3, 4
ROM 化文件	rom.asm	包含在库中

注 1. *: 如果不使用标准库，则需要添加 "n"。如果使用标准库，则不需要添加 "n"。

2. *: "b" 表示引导区域使用的启动程序 ,"e" 表示 flash 区域使用的启动程序。

3. *: 如果在标准库使用固定区域，则添加 "l"。

4. *: 如果设定了小型或中型，则添加 "m" 。如果设定大型，则添加 "l" 。
即使在使用小型或中型时，如果分配变量在 far 区域中，则使用添加 "l" 的启动程序。

备注　rom.asm 定义标识，用来指示 ROM 化过程中数据复制的结束地址。
rom.asm 生成目标文件包含在库中。

7.4.2　启动程序预处理

下面将对样例程序 （cstart.asm) 预处理进行说明。

备注　调用 cstart 时需要在前面添加 _@。

```
        NAME    @cstart

$INCLUDE ( def.inc )                                                    ; (1)
$INCLUDE ( macro.inc )
                                                                        ; (2)
BRKSW     EQU   1     ; brk, sbrk, calloc, free, malloc, realloc function use
EXITSW    EQU   1     ; exit, atexit function use
RANDSW    EQU   1     ; rand, srand  function use
DIVSW     EQU   1     ; div           function use
LDIVSW    EQU   1     ; ldiv          function use
FLOATSW   EQU   1     ; floating point variables use
STRTOKSW  EQU   1     ; strtok        function use

        PUBLIC  _@cstart, _@cend                                        ; (3)

$_IF ( BRKSW )
        PUBLIC  _@BRKADR, _@MEMTOP, _@MEMBTM
          :
$ENDIF
        EXTRN   _main, _@STBEG, _hdwinit, _@MAA                         ; (4)
$_IF ( EXITSW )
        EXTRN   _exit
$ENDIF
                                                                        ; (5)
        EXTRN   _?R_INIT, _?RLINIT, _?R_INIS, _?DATA, _?DATAL, _?DATS
@@DATA  DSEG    BASEP ; near                                            ; (6)

$_IF ( EXITSW )
_@FNCTBL :      DS    4 * 32
_@FNCENT :      DS    2
  :
_@MEMTOP : DS       32
_@MEMBTM :
$ENDIF
```

(1) 包含 include 文件

def.inc　　->　根据库类型设置。
macro.inc　->　定义宏的典型模板。

(2) 库切换

如果没有使用注释中标准库，如果把 EQU 定义为 0，会保留未使用的库处理所需空间。设置默认为全部使用（如果启动程序中无需库处理，则不进行该处理）。

(3) 符号定义

定义使用标准库所需的符号。

(4) 堆栈解析所需的符号外部引用声明

用于堆栈解析的公共符号 (_@STBEG) 是外部引用声明。
_@STBEG 的值是堆栈区的最终地址 +1。
在链接器中为堆栈解析设定符号生成选项 （-s), 则自动会产生 _@STBEG。因此，在链接时总要设定 -s 选项。在这种情况下，指定堆栈中使用区域的名称。如果省略了区域名称，使用 RAM 区域，但是创建一个链接指令文件可以将堆栈链接到任何地方 。有关存储映射的内容 , 参阅 目标设备用户手册。
链接指令文件的例子显示如下。链接指令文件是由用户使用普通编辑器创建的文本文件 (有关方法说明的详细内容，参阅 "7.6　编码举例 ").

示例　　在链接中指定 -sSTACK 时
创建 lk78k0r.dr (链接指令文件)。由于 ROM 和 RAM 的分配都是通过默认引用目标设备的存储映射操作进行默认操作的，因此不需要设置 ROM 和 RAM 分配，除非必须要改变。
有关链接指令 , 参阅在 Smp78k0r\CA78K0R 文件夹中的 lk78k0r.dr。

```
                首地址  大小
                    |       |
memory  SDR     : ( 0xFFE20h, 0000098h )
memory  STACK   : ( 0xxxxxxh, 0xxxxxxh ) <- Specify the first address and size here,
                                            然后用 -d 来设定 lk78k0r.dr
                                             链接器选项。 78k0r
                                             ( 例子 : -dlk78k0r.dr)
merge @@INIS    : = SDR
merge @@DATS    : = SDR
merge @@BITS    : = SDR
```

(5) ROM 化处理标签的外部引用声明

ROM 化处理所需的标签在后处理部分定义。

(6) 为标准库保留区域

对标准库所需的区域进行保留。

7.4.3 启动程序初始化设置

下面将对样本程序 (cstart.asm) 的初始化设置做说明。

```
@@VECT00        CSEG    AT      0                                       ; (1)
        DW      _@cstart

@@LCODE CSEG    BASE
_@cstart :
        SEL     RB0                                                     ; (3)
        MOV     A, #_@MAA                                               ; (2)
        MOV1    CY, A.0
        MOV1    MAA, CY
        MOVW    SP, #LOWW _@STBEG       ; SP <-stack begin address      ; (4)
        CALL    !!_hdwinit                                              ; (5)
          :
$_IF ( BRKSW OR EXITSW OR RANDSW OR FLOATSW )
        CLRW    AX
$ENDIF
          :
```

(1) 复位向量设置

定义重置向量表的段如下。设置启动程序的起始地址。

```
@@VECT00        CSEG    AT      0000H
                DW      _@cstart
```

(2) 镜像区域设置

设置镜像区域。

有关镜像区域的内容，参考目标设备用户手册。

(3) 设置寄存器组

将寄存器组 RB0 设置为当前工作寄存器组。

(4) 堆栈指针 (SP) 设置

将 _@STBEG 保存在堆栈指针内。

在链接器中为堆栈解析设定符号生成选项 （-s), 则自动会产生 _@STBEG。

(5) 硬件初始化函数调用

在用户需要初始化外围设备 （SFR) 功能时，创建 hdwinit 函数。通过创建这个函数，初始化的设置可以与用户目标匹配。

如果用户不创建 hdwinit 函数，则不做任何处理就直接返回。

(6) ROM 化处理

下面将介绍 cstart.asm 中的 ROM 化处理。

```
; 复制有初始值的外部变量
$_IF ( _ESCOPY )
        MOV     ES, #HIGHW _@R_INIT
$ENDIF
        MOVW    HL, #LOWW _@R_INIT
        MOVW    DE, #LOWW _@INIT
        BR      $LINIT2
LINIT1 :
$_IF ( _ESCOPY )
        MOV     A, ES : [HL]
$ELSE
        MOV     A, [HL]
$ENDIF
        MOV     [ DE ] , A
        INCW    HL
        INCW    DE
LINIT2 :
        MOVW    AX, HL
        CMPW    AX, #LOWW _?R_INIT
        BNZ     $LINIT1
```

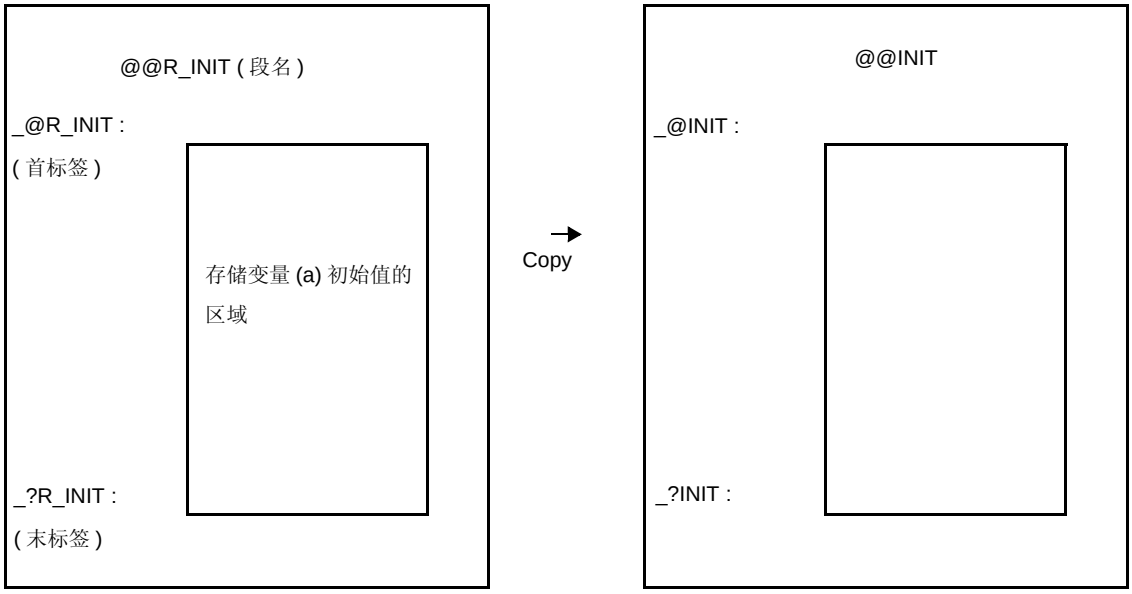
在 ROM 化处理中，复制外部变量的初始值和存储在 ROM 中的 sreg 变量到 RAM 中。在以下例子中显示处理变量有 4 个类型 (a) 到 (d)

```
char    c = 1 ;                 (a) 有初始值的外部变量
int     i ;                     (b) 没有初始值的外部变量注
__sreg  int     si = 0 ;        (c) 有初始值的 sreg 变量
__sreg  char    sc ;            (d) 没有初始值的 sreg 变量注

void main ( void ) {
          :
}
```

注 没有初始值的外部变量和没有初始值的 sreg 变量无需复制，相应的 RAM 直接清零。

- 下图显示带初始值外部变量 (a) 的 ROM 化处理。
  使用 C 编译器放置变量 (a) 初始值在 ROM 的 @@R_INIT 段内 78K0R。
  ROM 化处理复制这个数值到 ROM 内的 @@INIT 段上 (对变量 (c) 进行相同处理)。

图 7-2. 有初始值外部变量的 ROM 化处理



- 使用 _@R_INIT 和 _?R_INIT 在 @@R_INIT 段内定义首标签和末标签。使用 _@INIT and _?INIT 在 @@INIT 段内定义首标签和末标签。
- 不复制变量 (b) 和 (d)，但在预设段中， RAM 直接清零。

下表显示放置变量 (a) 到 (c)ROM 区域的段名，以及各段中初始值的首标签和末标签。

变量类型	段	首标签	末标签
带初始值 （a) 的外部变量 (在分配到 near 区域)	@@R_INIT	_@R_INIT	_?R_INIT
带初始值 （a) 的外部变量 (在分配到 near 区域时)	@@RLINIT	_@RLINIT	_?RLINIT
带初始值 （c) 的 sreg 变量	@@R_INIS	_@R_INIS	_?R_INIS

下表显示放置变量 (a) 到 (d)RAM 区域的段名，以及各段中初始值的首标签和末标签。

变量类型	段	首标签	末标签
带初始值 （a) 的外部变量 (在分配到 near 区域)	@@INIT	_@INIT	_?INIT
带初始值 （a) 的外部变量 (在分配到 near 区域时)	@@INITL	_@INITL	_?INITL
不带初始值 （b) 的外部变量 (在分配到 near 区域时)	@@DATA	_@DATA	_?DATA
不带初始值 （b) 的外部变量 (在分配到 far 区域时)	@@DATAL	_@DATAL	_?DATAL
带初始值 （c) 的 sreg 变量	@@INIS	_@INIS	_?INIS
不带初始值 （d) 的 sreg 变量	@@DATS	_@DATS	_?DATS

7.4.4　启动 main 函数和后处理

下面将介绍在样本程序 (cstart.asm) 中 main 函数的开始和后处理。

```
            CALL    !!_main          ; main ( );             ; (1)
$_IF ( EXITSW )
            CLRW    AX
            CALL    !!_exit          ; exit ( 0 );           ; (2)
$ENDIF
            BR      $$
;
_@cend :
                                                            ; (3)
@@R_INIT    CSEG    UNIT64KP
_@R_INIT :
@@RLINIT    CSEG    UNIT64KP
_@RLINIT :
@@R_INIS        CSEG    UNIT64KP
_@R_INIS :
@@INIT      DSEG    BASEP
_@INIT :
@@INITL     DSEG    UNIT64KP
_@INITL :
@@DATA          DSEG    BASEP
_@DATA :
@@DATAL     DSEG    UNIT64KP
_@DATAL :
@@INIS          DSEG    SADDRP
_@INIS :
@@DATS          DSEG    SADDRP
_@DATS :
@@CALT      CSEG    CALLT0
@@CNST      CSEG    MIRRORP
@@CNSTL     CSEG    PAGE64KP
@@BITS      BSEG
;
            END
```

(1) 启动 main 函数

调用 main 函数

(2) 启动 exit 函数

如果需要，调用 exit 函数。

(3) 对 ROM 化处理使用的段和标签进行定义

有关在 ROM 化处理中定义各变量 (1) 到 (4) 的已用段和标签 (参阅 "(6)　ROM 化处理 ")。段显示存储各变量初始值的区域。在各段中标签显示首地址。

下面将介绍 ROM 化处理文件 rom.asm。 rom.asm 可重定位的目标文件在库中。

```
        NAME    @rom
;
        PUBLIC  _?R_INIT, _?RLINIT, _?R_INIS
        PUBLIC  _?INIT, _?INITL, _?DATA, _?DATAL, _?INIS, _?DATS
;
@@R_INIT        CSEG    UNIT64KP                        ; (4)
_?R_INIT :
@@RLINIT    CSEG    UNIT64KP
_?RLINIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT      DSEG    BASEP
_?INIT :
@@INITL     DSEG    UNIT64KP
_?INITL :
@@DATA          DSEG    BASEP
_?DATA :
@@DATAL     DSEG    UNIT64KP
_?DATAL :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS :
;
        END
```

(4) 定义 ROM 化处理中使用的标签

定义了在 ROM 化处理中 (1) 到 (4) 所有变量使用的标签 (参阅 "(6) ROM 化处理 ")。这些标签显示存储每个变量初始值段的末地址。

如果在属于这些库的目标模块文件间存在多个用户库和相互引用 , 不能改变 CA78K0R 终止模块的模块名称 "@rom" 和 "@rome"。

如果改变终端模块名称，不可以在结尾链接。

7.5 在 Flash 区域启动模块中的 ROM 化处理

flash 启动模块与常规启动模块区别如下。

表 7-4. 初始化数据的 ROM 区域段

变量类型	段	首标签	末标签
带初始值 （a) 的外部变量 (在分配到 near 区域时)	@ER_INIT CSEG UNIT64KP	E@R_INIT	E?R_INIT
带初始值 （a) 的外部变量 (在分配到 far 区域时)	@ERLINIT CSEG UNIT64KP	E@RLINIT	E?RLINIT
带初始值 （c) 的 sreg 变量	@ER_INIS CSEG UNIT64KP	E@R_INIS	E?R_INIS

表 7-5. 复制目标的 RAM 区域段

变量类型	段	首标签	末标签
带初始值 （a) 的外部变量 (在分配到 near 区域时)	@EINIT DSEG BASEP	E@INIT	E?INIT
带初始值 （a) 的外部变量 (在分配到 far 区域时)	@EINITL DSEG UNIT64KP	E@INITL	E?INITL
不带初始值 （b) 的外部变量 (在分配到 near 区域时)	@EDATA DSEG BASEP	E@DATA	E?DATA
不带初始值 （b) 的外部变量 (在分配到 far 区域时)	@EDATAL DSEG UNIT64KP	E@DATAL	E?DATAL
带初始值 （c) 的 sreg 变量	@EINIS DSEG SADDRP	E@INIS	E?INIS
不带初始值 （d) 的 sreg 变量	@EDATS DSEG SADDRP	E@DATS	E?DATS

- 在启动模块中，在 ROM 区域和 RAM 区域中添加以下标签在各段开头。
  E@R_INIT, E@R_INIS, E@INIT, E@DATA, E@INIS, E@DATS, E@INITL, E@DATAL
  此外，如果设定大模式或在 far 区域分配变量，添加以下标签。
  E@RLINIT, E@INITL, E@DATAL
- 在该末端模块中，在 ROM 区域和 RAM 区域中添加以下标签在各段的末尾。
  E?R_INIT, E?R_INIS, E?INIT, E?DATA, E?INIS, E?DATS, E?RLINIT, E?INITL, E?DATAL
- 启动模块从 ROM 区域各段的首标签地址中复制内容到末尾标签地址 -1 的位置上 , 以及从 RAM 区域各段首标签地址的区域。
- 从 E@DATA 到 E?DATA 嵌入零，以及从 E@DATS 到 E?DATS 嵌入零。
- 此外，如果设定大模式或在 far 区域分配变量，则从 E@DATAL 到 E?DATAL 嵌入零。

7.6 编码举例

可以修改由 CA78K0R 提供的启动程序以实际使用的目标系统相匹配。在本节介绍有关修改这些文件的关键事项。

7.6.1 修改启动程序时

下面将对修改启动程序源文件要注意的关键要点做说明。修改后 , 使用在 Src\cc78k0r\bat 文件夹中的批处理文件 mkstup.bat 汇编修改过的源文件 (cstart*.asm) (*: 含字符和数字的符号)。

(1) 在库函数中使用的符号

如果不使用在下表中列出的库函数，则能删除在启动程序中与各函数对应的符号。然而，由于在启动程序中使用 exit 函数，则不能删除 _@FNCTBL 和 _@FNCENT （如果删除 exit 函数，则能删除这些符号）通过库切换能删除未使用的库函数中对应符号。

库函数名称	使用的符号
brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR

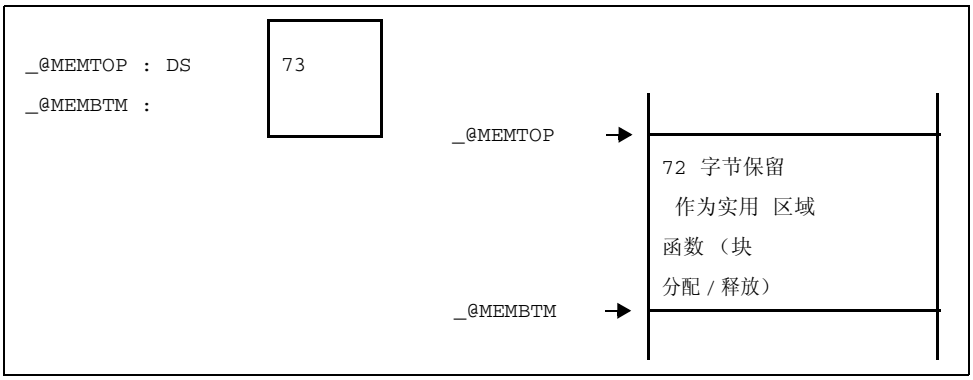
库函数名称	使用的符号
exit	_@FNCTBL _@FNCENT
rand srand	_@SEED
div	_@DIVR
ldiv	_@LDIVR
strtok	_@TOKPTR
atof strtod 数学函数 浮点运行时间库	_errno

(2) 实用函数的使用区域（块分配 / 释放）

如果由用户定义实用函数使用区域的大小，这个设置显示在以下例子中。

示例　　如果为使用实用函数保留 72 个字节（块分配 / 释放），需对启动程序的初始设置做以下修改。

图 7-3. 启动程序初始设置举例



添加一个字节在准备的区域大小上并在启动程序中设置值。在上面例子中，在启动程序中保留 73 个字节，但实际最多可以为实用函数保留 72 个字节。

如果指定的区域过大，无法存储在 RAM 中，在链接时可能产生错误。

在这种情况下，增加已设定的大小显示如下，或通过修正链接指令文件来避免。有关对链接指令文件的修正，参阅 "5.3.2　在使用编译器时 "。

示例　　增加指定区域的大小

7.6.2　在使用 RTOS 时

RX78K0R 和 78K0R C 编译器提供初始化程序的样本程序 (汇编程序格式)。当在编译时使用 RX78K0R 和 78K0R C 编译器 , 因此两个工具的初始化程序必须修改。

第 8 章 ROM 化

ROM 化是执行系统时，放置初始值 (比如那些已初始化的外部变量) 在 ROM 中并复制它们到 RAM 中的过程。
CA78K0R 给启动程序提供内置程序 ROM 化处理，在启动时 ROM 化处理省去写程序的麻烦。
有关启动程序信息的内容 , 参阅 "7.4 启动程序 "。
进行程序 ROM 化的方法如下。
在 ROM 化启动程序、目标模块文件和链接库期间。启动程序初始化目标程序。

(1) s0r*.rel

启动程序 （兼容 ROM 化）
包含初始化数据的复制程序，并标识初始数据的起始位置。在起始地址上添加标号 “_@cstart”（符号）。

(2) cl0r*.lib

库包含 CA78K0Rr
C 编译器中的库文件包含以下库类型。

- 运行时间库

  添加 “@@” 作为运行时间库名称的的起始符号。为特殊库 cprep 和 cdisp, 然而 , 添加 "_@" 到符号的起始位置上。

- 标准库

  添加 “_” 作为标准库名的起始符号。

(3) *.lib

用户创建的库
添加 "__" 到符号起始位置。

注意事项 CA78K0R 提供多种启动程序和库。有关启动程序的内容 , 参阅 " 第 7 章 启动例程 "。有关库相关内容，参见 "7.2.2 "lib" 文件夹内容 "。

第 9 章 编译程序和汇编程序的引用

本章介绍如何链接汇编语言编写的程序。

如果从 C 源程序中调用的函数是由其他语言编写的，则链接器需要链接两个目标模块。本章介绍从 C 语言编写的程序中调用其他语言编写的程序的步骤和其他语言编写的程序中调用 C 语言编写的程序的步骤。

按照下列顺序介绍使用 CA78K0R 与其他语言的接口 :

- 访问参数和自动变量
- 寄存器 HL 作为访问存储在堆栈中的参数与自动变量的基址指针。存储返回值
- 从 C 语言中调用汇编语言程序
- 从汇编语言中调用 C 语言程序
- 引用在 C 语言中定义的变量
- 从 C 语言中引用汇编语言中定义的变量
- 在 C 语言函数和汇编语言函数中调用的注意事项

9.1 访问参数和自动变量

有关参数和自动变量分配的详细内容 , 参阅 "3.4.2 普通函数调用接口 "。

9.2 寄存器 HL 作为访问存储在堆栈中的参数与自动变量的基址指针。存储返回值

参见 "3.4.1 返回值 "。

9.3 从 C 语言中调用汇编语言程序

本节显示默认 t 步骤的例子。

从 C 语言中调用汇编语言程序的描述如下。

- C 语言函数调用步骤
- 汇编语言程序的数据保存和调用返回

9.3.1 C 语言函数调用步骤

以下是 C 语言程序调用汇编语言程序的例子。

```
extern  int     FUNC ( int, long );     /* Function prototype */

void main ( void ) {
        int     i, j ;
        long    l ;

        l = 0x54321 ;
        i = 1 ;
        j = FUNC ( i, l );              /* Function call */
}
```

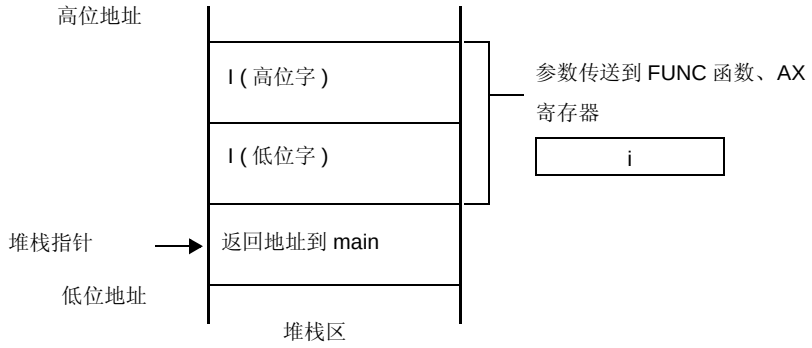
在这个程序例子中，执行程序的程序接口和控制流如下。

(a) 在寄存器中存放的第一个参数从 main 函数传送到 FUNC 函数中，且以相同方式传送在堆栈上的第二个以及随后的参数。

(b) 通过使用 CALL 指令传送控制给 FUNC 函数。

下一个图表显示在以上程序例子中控制移动到 FUNC 函数之后的堆栈。

图 9-1. 在调用函数后的立即堆栈



9.3.2　汇编语言程序的数据保存和调用返回

以下是 main 函数调用 FUNC 函数的处理步骤。

(1) 保存基址指针 , 存放寄存器变量的 saddr 区域。

(2) 复制堆栈指针 （SP) 到基址指针 （HL)。

(3) 执行 FUNC 函数描述的处理过程。

(4) 设置返回值。

(5) 恢复所保存的寄存器值。

(6) 返回 main 函数

汇编语言程序的例子说明如下。

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2

@@DATA          DSEG    BASEP
_DT1 :  DS      ( 2 )
_DT2 :  DS      ( 4 )

@@CODE CSEG
_FUNC :
        PUSH    HL              ; 保存基址指针     (1)
        PUSH    AX
        MOVW    HL, SP          ; 复制堆栈指针     (2)
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; 传递第 1 个参数 ( i )
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]    ; arg2
        MOVW    !_DT1, AX       ; 传递第 2 个参数 ( i )
        MOVW    BC, #0AH        ; 设置返回值       (4)
        POP     AX
        POP     HL              ; 恢复基址指针     (5)
        RET                     ;                        (6)
        END
```

(1) 保存基址指针和工作寄存器。

在用 C 源程序中描述的函数名称前加上标签前缀 "_"。基址指针和工作寄存器按照 C 源程序内定义的函数名来进行保存。

在设定标签后，对 HL 寄存器 （基址指针）保存。

在 C 编译器处理程序的情况下，调用其它函数时并不保存寄存器变量的 saddr 区域。因此，如果改变被调用函数的这些寄存器的值，务必事先确认保存该值。但是，如果在函数调用方法 y 有使用寄存器变量 , 则无需对其保存。

(2) 将基址指针中 （HL) 复制到堆栈指针 (SP)

由于函数中的 "PUSH, POP"，改变堆栈指针 （SP) 的值。因此，将堆栈指针复制到寄存器 "HL" 并作为参数的基址指针使用。

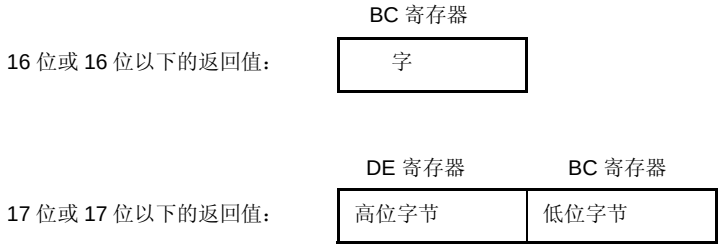
(3) FUNC 函数的基本处理过程

在处理步骤 （1）和 （2）执行后，执行被调用函数的基本处理过程。

(4) 设置返回值

如果有返回值，则保存在 "BC" 和 "DE" 寄存器中。如果没有返回值，则无需存储。

图 9-2. 设置返回值

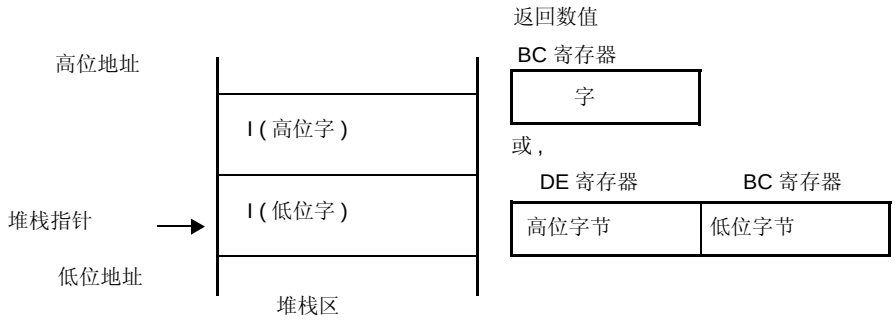


(5) 恢复寄存器值

恢复保存的基址指针和工作寄存器。

(6) 返回到 main 函数

图 9-3. 返回到 Main 函数



9.4　从汇编语言中调用 C 语言程序

本节介绍从汇编语言程序中调用 C 语言编写的函数的步骤。

9.4.1　从汇编语言程序中调用 C 语言函数

从汇编语言程序中调用 C 语言编写的函数的步骤如下。

(a) 保存 C 工作寄存器 （AX,BC, 和 DE)。

(b) 将参数保存在堆栈。

(c) 调用 C 语言函数。

(d) 根据参数的所占字节数对堆栈指针 （SP) 值。

(e) 引用 C 语言函数的返回值 （在 BC 或 DE 和 BC)。

- 下面是汇编语言程序的例子。

```
$PROCESSOR ( F1166A0 )

        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2

@@CODE CSEG
_FUNC2 :
        movw    ax, #20H        ; set 2nd argument ( j )
        push    ax              ;
        movw    ax, #21H        ; set 1st argument ( i )
        call    !_CSUB          ; call "CSUB ( i,  j )"
        pop     ax              ;
        ret
        END
```

(1) 保存工作寄存器 （AX,BC, 和 DE)

AX,BC 和 DE 三组寄存器用于 C 语言。在调用返回时不能恢复这些寄存器的值。因此，如果需要寄存器中的值，在调用方法上保存它们。

在参数传递前后，要保存或恢复寄存器的值。

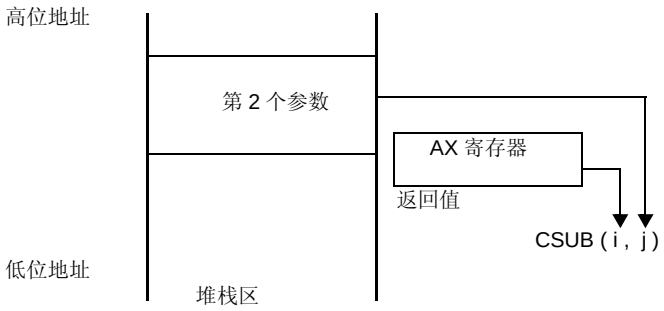
当在 C 语言中使用了 HL 寄存器时，该寄存器始终要在 C 语言函数方进行保存。

(2) 参数入栈

任何参数均放置在堆栈上。

下图显示参数的传递。

图 9-4. 参数传递



(3) 调用 C 语言函数。

用 CALL 指令调用 C 语言函数。如果 C 语言函数为 "callt" 函数，则 callt 指令执行函数调用，并如果它是 "callf" 函数，则 callf 指令执行函数调用。

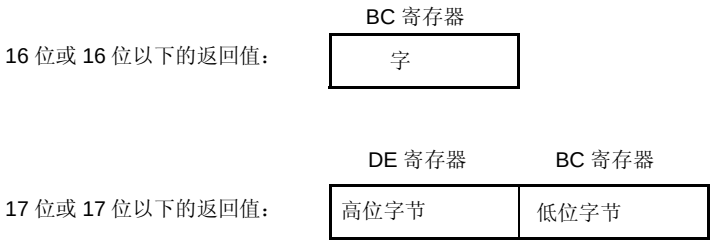
(4) 恢复堆栈指针 (SP)

根据参数所占字节数来恢复堆栈指针。

(5) 引用返回值 (BC 和 DE)

从 C 语言中返回值的返回如下。

图 9-5. 引用返回值



9.5　引用在 C 语言中定义的变量

如果在汇编语言程序中引用在 C 语言程序中定义的外部变量，则要使用 extern 进行外部声明。
要引用汇编语言程序中定义的变量，在变量名称前要添加下划线 "_"。
C 语言程序例程显示如下。

```
extern  void subf ( void );

char    c = 0 ;
int     i = 0 ;

void main ( void ) {
        subf ( );
}
```

在汇编程序中的例程如下。

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

9.6 从 C 语言中引用汇编语言中定义的变量

从 C 语言中引用汇编语言中定义的变量按如下方式进行。
C 语言程序例程显示如下。

```
extern  char    c ;
extern int       i ;

void subf ( void ) {
        c = ' A ' ;
        i = 4 ;
}
```

在 78K0R 汇编程序的例程如下所示。

```
        NAME ASMSUB

        PUBLIC  _i
        PUBLIC  _c

ABC DSEG        BASEP
_i : DW         0
_c : DB         0
        END
```

9.7 在 C 语言函数和汇编语言函数中调用的注意事项

- "_"(下划线)

78K0R 该 C 编译器对外部定义及输出目标模块中的引用名称前添加下划线 "_" (ASCII 码为 "5FH")。
在下面 C 程序例子中， "j = FUNC(i, l);" 作为 " 对外部名称 _FUNC 的引用 "。

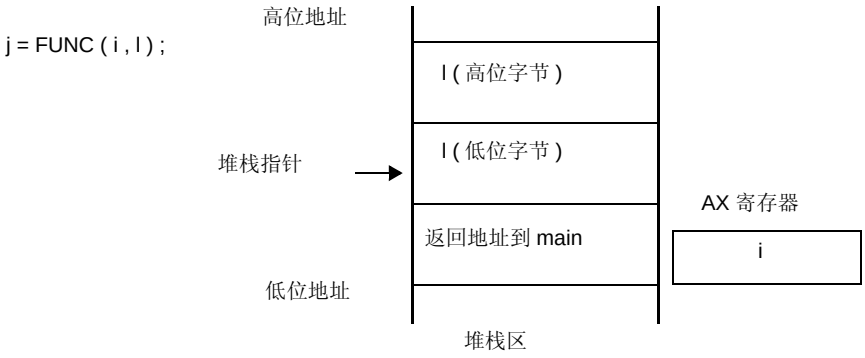
```
extern  int     FUNC ( int, long );     /* Function prototype */

void main ( void ) {
        int     i, j ;
        long    l ;

        l = 0x54321 ;
        i = 1 ;
        j = FUNC ( i, l );              /* Function call */
}
```

在汇编程序中用 "_FUNC" 来命名程序名称 78K0R。

- 参数在堆栈上的位置

  参数在堆栈上存放位置是按照从后缀参数到前缀参数，由高地址向低地址按方向可从高位地址到低位地址方向顺序存放的。

图 9-6.　参数在堆栈上的位置



第 10 章 注意事项

本章节说明了编写代码时的一些注意事项。

(1) 汉字代码分类

当使用的源程序包含 Windows 中的 EUC 代码时，可把环境变量 LANG78K 设置到 EUC 或指定 -ze 选项。

(2) Include 文件

带有 include 文件的函数 （除声明以外）不能在 C 源代码中扩展。

如果在 include 文件中定义，可能会在程序调试时出现诸如定义行不能正确显示之类的不希望出现的现象。

(3) 输出汇编器源程序

当 C 源代码中出现诸如 #asm 块或 __asm 语句的汇编语言说明符时，装载模块文件创建如下顺序：编译、汇编和链接。

假设存在汇编语言的说明符，当由 78K0R C 编译器首先输出汇编器源程序编译而不直接输出目标文件时，需遵守以下几点注意事项。

- 如果可以在 #asm 块 （#asm 和 #endasm 之间的区域）或在 __asm 语句中定义符号，使用等于或小于 8 个字符的符号并以 ?L@ （例如：?L@01、 ?L@sym 等）开头。但是，不要通过 PUBLIC 声明）在外部定义这些符号)。此外，段不能定义在 #asm 块或 __asm 语句内。如果未使用以 ?L@ 字符序列为首的符号，在汇编中将输出严重错误 （F2114）。
- 当使用变量集在 C 源程序的 #asm 块中定义 , 假如在 C 源程序中其它部分没有涉及那么 EXTRN 不产生并且发生连接错误。因此 , 当 C 没有涉及时，在 #asm 块内完成 EXTRN 。
- 当以 #pragma 片段命令修改段名时，就象源文件名的首名称那样不能指定。错误 (F2106) 在汇编期间输出。

(4) 建立链接指令文件

当由 C 编译器 78K0R 链接目标生成的时，如果使用目标器件 ROM 或 RAM 存储器之外的区域或您希望指定任何地址存放代码或数据时，创建一个链接指令文件并在链接时指定链接器选项 -d。

查阅 “第 5 章 链接指令说明 ? 或包含 C 编译器的 lk.dr （位于 smp 文件夹下）文件，以获取建立链接指令文件的方法。

示例 在 C 源代码文件中，分配初始值为空的外部变量 （非 sreg 变量）至外部存储器。

(a) 在 C 源代码的头文件中，修改用于初始值为空的外部变量的片段名。

```
#pragma section @@DATAL EXTDATA
    :
```

注意事项 为了初始化或 ROM 化已修改的段，也可通过修改启动程序实现该操作。

(b) 创建 lk78k0r.dr 链接指令文件。

```
memory  EXTRAM  : ( 040000H, 1000H )
merge   EXTDATA : = EXTRAM
```

创建链接指令文件时，请注意以下几点。

- 当链接时，如果使用堆栈拆分符号来指定选项 -s，那么推荐用链接指令文件指定存储器保证堆栈区并且明确定义有保证的堆栈区。若排除区域名，则 RAM 存储器 (除 SFR 存储器外) 可以用于堆栈区。

示例 当链接指令文件加到 lk78k0r.dr 。

```
memory  EXTRAM  : ( 040000H, 1000H )
memory  STK     : ( 0FB000H, 100H )
merge   EXTDATA : = EXTRAM
```

命令行如下。

```
C>lk78k0r s0rml.rel prime.rel -bcl0rm.lib -sSTK -dlk78k0r.dr
```

- 当链接到已定义存储器区域时，可能输出如下链接错误。

```
RA78K0R error E3206 : Segment 'xxx' can't allocate to memory-ignored.
```

这表示，由于指定存储器区域空间不够，不能存放指定段。
处理方法大体上分为下列三个步骤。

<1> 查询未被存放段的大小 （查看 .map 文件）。
但是，根据错误指出的段类型，查询段大小的方法有如下区别。

- 对于编译时自动生成的段
  从链接产生的 map 文件中查询段的大小
- 对于用户创建的段
  查询未列于汇编列表文件 （.prn）上的段的大小。

<2> 基于上述找到的段的大小，用指令文件增加用于存放该段的存储器容量。

<3> 指令文件和链接的指定选项 -d。

(5) 当使用 va_start 宏

由于取决不同功能的第一个参数偏移，定义在 stdarg.h 中的 va_start 宏无法得到保证。
遵照如下使用宏。

- 当指定第一个参数时，使用 va_starttop 宏。

(6) 关于启动程序和库

- 使用如 (cc78k0r.exe) 执行文件版本那样提供相同的启动程序和库的执行版本。
- 在浮点兼容的 sprintf、 vprintf 和 vsprintf 中 , 低精度的 "%f", "%e", "%E", "%g" 和 "%G" 指定转换结果是向下舍入的。 . 另外 , 即使指定 "%g" 和 "%G" 转换， "%f" 发生的转换结果高于上面的精度。
  在浮点兼容的 sscanf 和 scanf 中 , 如果指定 "%f", "%e", "%E", "%g" 和 "%G" 没有有效字符读出，那么转换结果是 +0, 并且如果 "±" 是仅识别字符那么转换结果是 ± 0。

(7) 当完成 ROM 化

ROM 化是定位初始值的过程，如当系统执行时，用 ROM 中的初始值作为外部变量复制到 RAM 中。 .78K0RC 编译器由默认产生码实行 ROM 化。所以当链接时需要链接包含 ROM 化过程的启动程序。
对于小模式和中等模式，远程存储器的 ROM 化过程不包含在启动程序中。当使用 __far 对远程存储器已经定位的变量时，使用大模式启动程序。
78K0R 提供了以下启动程序，所有这些程序包含了 ROM 化过程。
如果使用闪存自重写模式，请参见 "(3) 启动程序的使用 "。

当没有使用 C 标准存储库时	s0rm.rel, s0rl.rel
当使用 C 标准存储库时	s0rml.rel, s0rll.rel

下列为一个有用的示例。
-s 选项为堆栈符号 （_@STBEG 和 _@STEND）自动生成的选项。

```
C>lk78k0r s0rl.rel sample.rel -s -bcl0rxm.lib -bcl0rm.lib -osample.lmf
```

sample.rel	用户编写的目标模块文件
s0rl.rel	启动程序
cl0rxm.lib	乘法器库
cl0rdm.lib	乘法器 / 除法器库
cl0rm.lib	运行库，标准库

注意事项 1. 确保先链接启动程序。
2. 当创建用户生成的库时，需与 CA78K0R 提供的库分开，并且在链接时先于 CA78K0R 库定义。
3. 不要向 CA78K0R 库添加用户函数
4. 当使用浮点数库 （cl0r*f.lib）时，也需要链接常规库 （cl0r*.lib）。

示例 使用兼容浮点数的 sprintf、 sscanf、 printf、 scanf、 vprintf 和 vsprintf。

```
-bmylib.lib -bcl0rmf.lib -bcl0rm.lib
```

示例 使用不兼容浮点数的 sprintf、 sscanf、 printf、 scanf、 vprintf 和 vsprintf。

```
-bmylib.lib -bcl0rm.lib -bcl0rmf.lib
```

(8) 原型声明

在原型声明函数中，若未指定函数类型，会导致 （E0301 和 E0701）错误产生。

```
f ( void );     /* E0301 : Syntax error */
                /* E0701 : External definition syntax */
```

这种情况下，需指定函数类型。

```
int     f ( void );
```

(9) 错误信息输出

除了函数，如果在命令行的开始出现关键字的拼写错误的话，错误行的显示位置可能偏移或可能输出不正确的错误信息。

```
extren int      i ;             /* extern is misspelled.An error does not occur here*/
/* comment */
void f ( void );
[EOF]                           /* Error such as E0712 */
```

(10) 在预处理器命令中输入注释

在函数宏命令行中，如果在预处理器命令的开头或中间插入注释，将产生 （E0803、 E0814、 E0821 等）错误。

```
/* com1 */      #pragma         sfr                     /* E0803 */
/* com2 */      #define         ONE     1               /* E0803 */
#define         /* com3 */      TWO     2               /* E0814 */
#ifdef          /* com4 */      ANSI_C                  /* E0814 */

/* com5 */      #endif
#define         SUB ( p1, /* com6 */ p2 ) p2 = p1       /* E0821 */
```

这种情况下，在预处理器命令之后插入注释。

```
#pragma         sfr                                     /* com1 */
#define         ONE     1                               /* com2 */
#define         TWO     2                               /* com3 */
#ifdef          ANSI_C                                  /* com4 */

#endif                                                  /* com5 */
#define         SUB ( p1, p2 ) p2 = p1                  /* com6 */
```

(11) 用于 structure 的标签， union 和 enum

在原型声明函数中，如果 （structure，union 和 enum）标签在使用前定义，如果遇到下面条件就发生警告。

(a) 如果在参数声明中用标签来对 structure 和 union 定义指针，当函数调用时发生 (W0510) 警告。

```
void func ( int, struct st );

struct  st {
        char    memb1 ;
        char    memb2 ;
} st[ ] = {
        { 1, ' a ' }, { 2, ' b ' }
};

void caller ( void ) {
                        /* W0510 Pointer mismatch */
        func ( sizeof ( st ) / sizeof ( st[0] ), st );
}
```

(b) 如果标签用于返回数值声明和参数陈述指定 structure， union 和 enum 类型，将发生 (E0737) 错误。

```
                        /* E0737 Undeclared structure/union/enum tag */
void func1 ( int, struct st );
                        /* E0737 Undeclared structure/union/enum tag */
struct  st      func2 ( int );
struct  st {
        char    memb1 ;
        char    memb2 ;
};
```

在这种情况中首先定义 structure, union 和 enum 标签。

(12)初始函数中的 arrays, structures 和 unions

使用其它静态变量地址，常数和字符串在函数内的 array, structure 和 union 不能完成初始化。

```
void f ( void );

void f ( void ) {
        char    *p, *p1, *p2 ;
        char    *ca[3] = { p, p1, p2 };    /* Error( E0750 ) */
}
```

在这种情况中，输入赋值语句和子程序来替代初始化。

```
void f ( void );

void f ( void ) {
        char    *ca[3] ;
        char    *p, *p1, *p2 ;
        ca[0] = p ;  ca[1] = p1 ; ca[2] = p2 ;
}
```

(13)外部 callt 函数

如果引用初始函数表中的外部 callt 函数地址并且在相同的模块中调用函数，汇编列表无效和在汇编期间发生出错。

```
callt   extern  void funca ( void );
callt   extern  void funcb ( void );
callt   extern  void funcc ( void );

static  void ( * const func[] ) ( ) = {
        funca, funcb, funcc
};

void func2 ( void ) {
        funcc ( );
        funcb ( );
        funca ( );
}
```

在这种情况中从函数调用模块中分出函数表。

(14) Structure-returning 功能

当函数返回 structure 本身，如果在数值返回的过程中发生中断并且中断处理过程中调用相同的函数，中断处理完成后的返回值是无效的。

```
struct  str {
        char    c ;
        int     i ;
        long    l ;
} st ;

struct  str     func ( ) {
        /* Interrupt occurrence */
          :
}

void main ( ) {
        st = func ( );  /* Interrupt occurrence */
}
```

在处理上述引用期间，如果在中断目的地调用 func 函数，可能会死机。

(15) Union 初始化

初始化中，保持 structure, union 和 array 成员，如果初始化列表指定途径嵌套，一个错误 (E0750) 发生。

```
struct  Ss {
        int     d1, d2 ;
};
union   Au {
        struct  Ss t1 ;
} u = { { 1, 2 } };    /* E0750  Initializer syntax */
```

这种情况中，不要使用嵌套指定 union 初始化列表。

```
struct  Ss {
        int     d1, d2 ;
};
union   Au {
        struct  Ss t1 ;
} u = { 1, 2 };
```

(16) 存储器初始化指令

如果存储器初始化指令用数据段 (DSEG)DB,DW 和 DG 输入，将输出目标码，但是在目标码转换时警告 (W4301) 将发生。这是由于代码存在在 ROM 区以外的地址 (编码区)。
如果在这个状态中调用 ROM 码 (存取操作调用和发送)，将发生错误。

(17) 存储器指令

每个设备的默认存储器区名是不能擦除的。

对于默认存储器的设置存储器大小不能是 0。

因而，某些段可以指定为默认区，但当改变区名时要小心。

见 每个设备的用户手册所标出的存储器区名。

(18) 段名称

当输入段名时，不要与源文件初始名相同。关于在汇编时发生的错误 F2106。

(19) EQU 定义 SFR 名

SFR 名能在操作 EQU 指令中被指定，但是 SFR 名在 saddr 区域之外指定，一个汇编错误将发生。

(20) 启动地址指定

用 #pragma 来指定启动地址的位置，所指定的位置始终是偶地址。

详细信息请联系：

中国区

MCU 技术支持热线：
电话：+86-400-700-0606 (普通话)
服务时间：9:00-12:00，13:00-17:00 （不含法定节假日）

网址：
http://www.cn.necel.com/ （中文）
http://www.necel.com/ （英文）

[北京]

日电电子（中国）有限公司
中国北京市海淀区知春路 27 号
量子芯座 7，8，9，15 层
电话：（+86）10-8235-1155
传真：（+86）10-8235-7679

[上海]

日电电子（中国）有限公司上海分公司
中国上海市浦东新区银城中路 200 号
中银大厦 2409-2412 和 2509-2510 室
电话：（+86）21-5888-5400
传真：（+86）21-5888-5230

上海恩益禧电子国际贸易有限公司
中国上海市浦东新区银城中路 200 号
中银大厦 2511-2512 室
电话：（+86）21-5888-5400
传真：（+86）21-5888-5230

[深圳]

日电电子（中国）有限公司深圳分公司
深圳市福田区益田路卓越时代广场大厦 39 楼
3901，3902，3909 室
电话：（+86）755-8282-9800
传真：（+86）755-8282-9899

[香港]

香港日电电子有限公司
香港九龙旺角太子道西 193 号新世纪广场
第 2 座 16 楼 1601-1613 室
电话：（+852）2886-9318
传真：（+852）2886-9022
2886-9044

[成都]

日电电子（中国）有限公司成都分公司
成都市二环路南三段 15 号天华大厦 7 楼 703 室
电话：(+86)28-8512-5224
传真：(+86)28-8512-5334

[长春]

日电电子（中国）有限公司长春分公司
吉林省长春市朝阳区
西安大路 727 号中银大厦 A 座 1609 室
电话：(+86)431-8859-7533 / 8859-8533
传真：(+86)431-8680-2944

[大连]

日电电子（中国）有限公司长春分公司
大连市中山路 88 号天安国际大厦 2701 室
电话：(+86)411-8230-8815 / 8230-8825
传真：(+86)411-8230-8835