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

CC78K0S Ver. 2.00

C 编译器

语言篇

目标设备

78K0S 微控制器

文档编号. U17415CA2V0UM00 (第 2 版)
发行日期 2009 年 1 月

日本印刷

[备忘录]

HP9000 系列 700 and HP-UX 是惠普公司的注册商标。
SPARCstation 是 SPARC 国际（有限）公司的注册商标。
PC/AT 是 IBM（国际商用机器）公司的注册商标。

- **本文档所刊登的内容有效期截至 2009 年 1 月。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所开发或制造，或为日本电气电子株式会社（定义如上）开发或制造的产品。

[备忘录]

前言

CC78K0S C 编译器（下文简称为 C 编译器）开发的基础，是**美国信息系统国家标准草案**中的**第 2 章 环境**和**第 3 章 语言 — C 语言编程**（1988 年 12 月 7 日）。因此，使用本 C 编译器可以对符合 ANSI 标准的 C 语言源程序进行编译，可以开发 78K0S 系列应用产品。

CC78K0S C 编译器语言篇（本手册）是为了让那些使用本 C 编译器进行软件开发的用户能够正确了解本 C 编译器的基本功能及语言规范而编写的。

本手册不介绍如何操作本 C 编译器。因此，在您掌握了本手册的内容后，请阅读 **CC78K0S C 编译器 操作篇 (U17416E)**。

关于 **78K0S** 系列的体系结构，敬请参阅 **78K0S** 系列各个产品的用户手册。

[目标设备]

可以使用本 C 编译器开发 78K0 系列微控制器的软件。

请注意，开发时需要安装与目标设备对应的设备文件。

[读者]

尽管本手册适用于那些已经阅读过微控制器用户手册并且具有软件开发经验的读者。但是，关于 C 编译器和 C 语言的知识并不是一定需要的，本手册中的内容假设读者熟悉软件术语。

[组织结构]

这本手册的结构组织如下描述。

第 1 章 概述

概括介绍该 C 编译器的一般功能、性能指标及特色。

第 2 章 C 语言的结构

介绍 C 源程序模块文件的构成要素。

第 3 章 数据类型与存储类的声明

介绍 C 中使用的数据类型及存储类，以及如何声明数据对象或函数的类型及存储类。

第 4 章 类型转换

介绍本 C 编译器自动执行的数据类型转换。

第 5 章 运算符与表达式

介绍 C 中使用的运算符和表达式，以及运算符的优先级。

第 6 章 C 语言的控制结构

介绍 C 的程序控制结构及在 C 中执行的语句。

第 7 章 结构体与共用体

关于结构体与共用体的概念，以及如何引用结构体与共用体成员。

第 8 章 外部定义

介绍外部定义的类型，以及如何使用外部定义。

第 9 章 预处理指令

详细介绍预处理指令的类型，以及如何使用预处理指令。

第 10 章 库函数

详细介绍 C 库函数的类型，以及如何使用各个库函数。

第 11 章 扩展函数

介绍该 C 编译器的扩展函数，以便用户最大限度地发挥目标设备的功能。

第 12 章 汇编程序与 C 程序的引用和兼容性

介绍将 C 源程序与汇编源程序连接的方法。

第 13 章 编译器的高效使用

概括介绍如何更有效地使用本 C 编译器。

附录

附录中包含有 **saddr** 区域标签列表、区段名称列表、运行时刻库列表、库堆栈占用列表、库响应中断的最长时间列表，及索引以方便用户的快速查找。

[如何阅读这本手册]

- 对于不熟悉 C 编译器或 C 语言的读者：
  从**第 1 章**开始阅读，因为本手册涵盖了从 C 的程序控制结构到本 C 编译器的扩展函数内容。在**第 1 章**中，使用一个 C 源程序示例来介绍本手册中的引用部分。
- 对于熟悉 C 编译器或 C 语言的读者：

本 C 编译器的语言规范符合 **ANSI 标准 C**。因此，您可以从**第 11 章**开始，该章介绍了本 C 编译器特有的扩展函数。在阅读**第 11 章**时，如有必要，还可以参考 78K/0 系列中的目标设备附带的用户手册。

[相关文档]

下面的表格显示了这本手册的相关文档(如用户手册)。在出版物中出现的相关资料可能会包括初稿版本。但是，并未对初稿版本作特殊标注。

开发工具的相关文档（用户手册）

文档名称		文档编号
CC78K0S Ver. 2.00 C编译器	操作篇	U17416E
	语言篇	本手册
RA78K0S Ver. 2.00汇编程序包	操作篇	U17391E
	语言篇	U17390E
	结构化汇编语言	U17389E
SM+ 系统仿真器	操作篇	U18601E
	用户开放接口	U18212E
SM78K 系列 Ver. 2.52 系列仿真器	操作篇	U16768E
PM+ Ver. 6.30 项目管理器		U18416E
ID78K0S-NS Ver. 2.52 集成调试器	操作篇	U16584E
ID78K0S-QB Ver. 3.00 集成调试器	操作篇	U17287E

注意 上述列出的相关资料如有变动恕不另行通知，请务必使用最新版本的设计文件。

[参考文献]

美国信息系统国家标准草案 — （C 语言编程）（1988 年 12 月 7 日）

[术语]

RTOS = **78K/0 系列实时操作系统 RX78K0**

[规则]

本手册中使用了以下符号及缩写。

符号	**含义**
…	相同格式的数据的连续（重复）
“ ”	括在双引号中的字符必须原样输入。
‘ ’	括在单引号中的字符必须原样输入。
⋮	本部分程序描述被省略
/	定界符
\	反斜线
[ ]	方括号中的参数可以被省略。

[备注]

目录

插图列表

表格列表

第 1 章　概述

本章介绍 CC78K0S 在系统开发中的作用 , 并且提供其功能概要。

CC78K0S 系列 C 编译器是一款语言处理程序，不论编写的 C 语言源程序符合 78K/0S 系列规范或是符合 ANSI-C 规范，78K0S 系列 C 编译器都会将 C 语言转换为机器语言。通过 CC78K0S 系列 C 编译器 , 能获得 78K0S 系列目标文件或汇编源文件。

1.1　C 语言与汇编语言

为了让微控制器按照用户的安排来完成工作，程序和数据是必不可少的。程序和数据必须由用户来编写，并存储在微控制器的存储器区。微处理器能够处理的程序和数据，只不过是称为机器语言的二进制数组合。

汇编语言是一种符号语言，其特征是符号 （助记符）语句与机器语言指令一一对应。正是由于这种一一对应，汇编语言才能够为计算机提供详细的指令 （例如，为了提升 I/O 处理速度）。但是，这意味着使用者必须对计算机的每一步操作都给出具体指令。正是由于这种原因，程序的逻辑结构比较复杂，一般都比较晦涩难懂，而且使用者在编写代码时也容易出错。

所以，人们开发了高级语言来代替这种汇编语言。 C 语言是高级语言的其中一种，它使得使用者在编程时无需考虑计算机的体系结构。

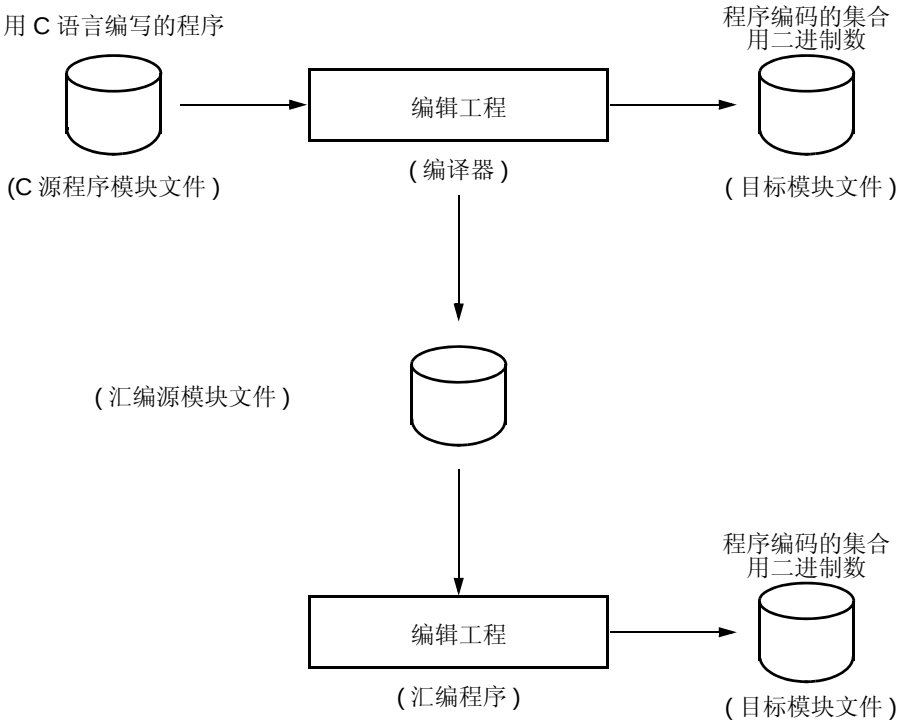
与汇编语言程序相比，可以认为 C 语言编写开发的程序逻辑结构更加易于理解。

C 语言具有丰富的调用函数，可用来开发程序。换句话说，使用者可以使用这些函数来编写程序。

C 语言的特点是便于使用者理解。但是， C 语言编写的程序无法被微处理器直接理解。所以，要使计算机理解 C 语言程序，需要另外一个软件程序将 C 语言语句翻译成对应的机器语言指令。把 C 语言翻译成机器语言的程序称为 C 编译器。

C 编译器将 C 源程序模块作为输入，并产生目标模块或汇编语言源程序模块作为输出文件。因此，如果使用者希望能够指定计算机的程序执行具体过程，可以对 C 源程序生成的汇编语言源程序进行修改。 C 编译器的转换流程在图 1-1 中所示。

图 1-1 编译流程


用 C 语言编写的程序
程序编码的集合
用二进制数
编辑工程
(C 源程序模块文件)
(编译器)
(目标模块文件)
(汇编源模块文件)
程序编码的集合
用二进制数
编辑工程
(汇编程序)
(目标模块文件)

1.2 使用 C 编译器的程序开发过程

使用 C 编译器进行产品 （程序）开发，需要链接器来组织编译器生成的目标模块文件，需要库管理程序来创建库文件，需要调试器来检查并纠正每个 C 源程序中的 bug （错误或失误）。

1.2.1 软件需求

C 编译器有关的软件要求如下。

- 编辑器 : 用于创建源程序模块文件
- RA78K0S 汇编程序包

  结构化汇编程序的预处理器 : 为通过汇编语言得到结构化编程

  汇编器 : 用于将汇编语言转换为机器语言

  链接器 : 用于连接目标模块文件

  以确定为重定位代码段分配的地址

  目标转换器 : 用于转换生成 HEX 格式的目标模块文件

  库管理程序 : 用于创建库文件

  列表转换器 : 用于输出绝对汇编列表文件的

  PM+ : 集成开发环境平台
- 调试器 (为 78K0S) : 用于调试 C 源模块文件

1.2.2 产品开发程序

使用 C 编译器进行程序开发的步骤如下。

(1) 将程序划分成若干个功能块。

(2) 创建每个功能块对应的 C 源程序模块。

(3) 对每个 C 源程序模块进行转换。

(4) 将常用的模块注册到库中。

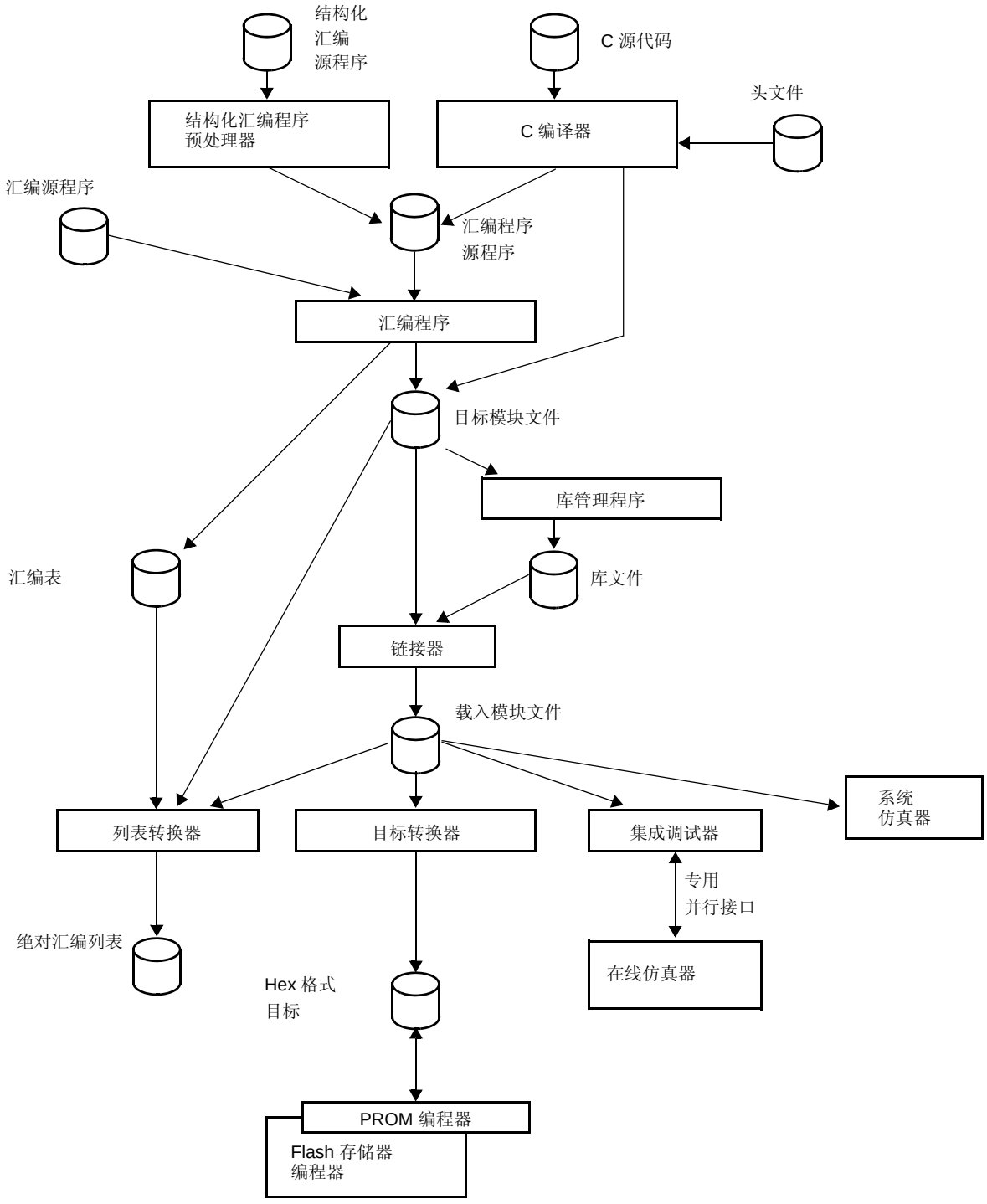
(5) 链接目标模块文件。

(6) 对每个模块进行调试。

(7) 将目标模块转换成 HEX 格式的目标文件。

如上所述，该 C 编译器对 C 源程序模块进行翻译 （编译），生成目标模块文件或汇编语言源程序模块文件。对生成的汇编语言源程序模块文件进行手工优化，并将其嵌入到 C 源程序中，可以产生效率更高的目标模块。这种处理方法对于必须执行高速处理，或模块必须非常紧凑的情况很有用。

图 1-2 使用 CC78K0S 的程序开发过程

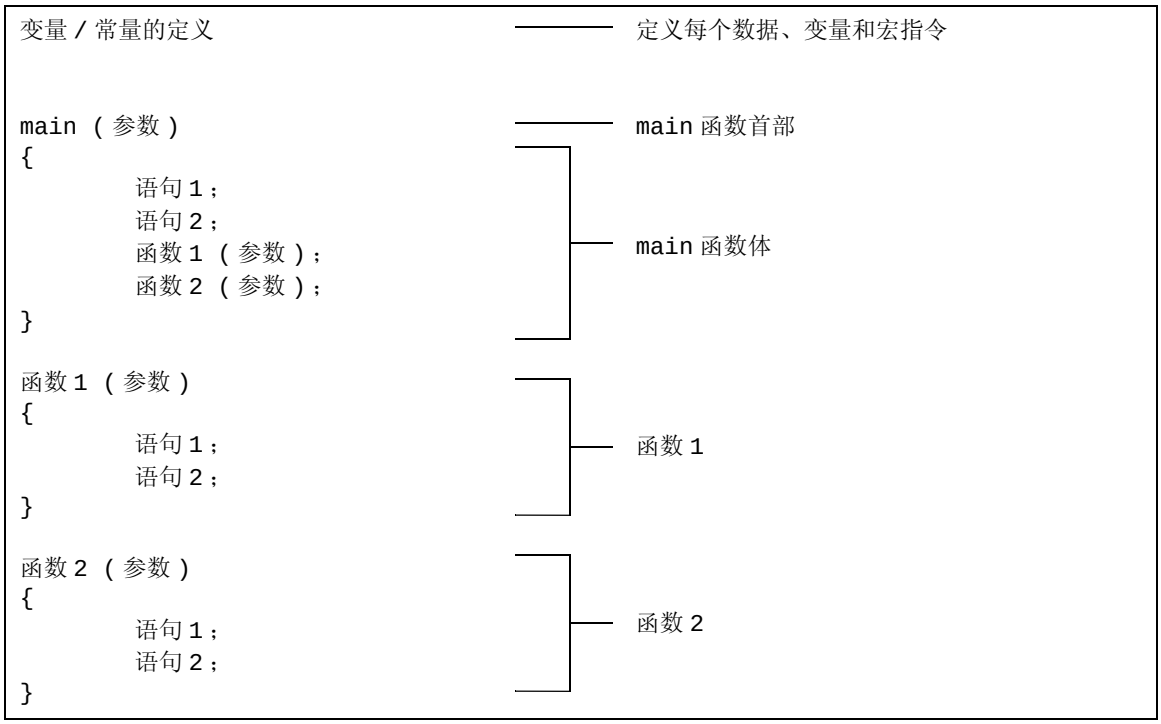


1.3 C 源程序的基本结构

1.3.1 程序格式

C 程序是函数的集合。这些函数必须创建，因为每一个都有独立的特殊用途或者特征动作。所有 C 语言程序都必须有一个 main 函数， main 函数是 C 语言中的入口主程序，并且是程序开始执行时被调用的第一个函数。

每个函数由两部分组成，一部分是函数首部用于定义函数名称和参数，另一部分是函数体包括声明和语句。 C 程序的格式如下所示。



实际的 C 源程序示例。

```
#define TRUE  1
#define FALSE   0
#define SIZE    200

void    printf ( char* , int ) ;
void    putchar ( char ) ;

char    mark [ SIZE + 1 ] ;
main ( )
{
    int     i , prime , k , count ;

    count = 0 ;

    for ( i = 0 ; i <= SIZE ; i ++ )
        mark [ i ] = TRUE ;

    for ( i = 0 ; i <= SIZE ; i++ ) {
        if ( mark [ i ] ) {
            prime = i + i + 3 ;
            printf ( " %6d " , prime ) ;

                    count++ ;
            if ( ( count%8 ) == 0 ) putchar ( ' \n ' ) ;
                for ( k = i + prime ; k <= SIZE ; k += prime )
                    mark [ k ] = FALSE ;
        }

    printf ( " \n%d primes found." , count ) ;
}

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

    j = i ;
    ss = s ;
}

void    putchar ( char c )
{
    char    d ;
    d = c ;
```

```
#define xxx xxx /* 预处理指令 */
                /* ( 宏定义 ) (6) */

xxx xxxx ( xxx , xxx )
/* 函数原型声明符 (7) */

char xxx        /* 类型声明符 (1) */
                /* 外部定义 (5) */
xx [ xx ]       /* 运算符 (2) */

int xxx     /* 类型声明符 (1) */

xx = xx     /* 运算符 (2) */

for ( xx ; xx ; xx ) xxx ;
                /* 控制结构 (3) */

xxx = xxx + xxx + xxx /* 运算符 (2) */
xxx ( xxx ) ;  /* 运算符 (2) */

if ( xxx ) xxx ;
                /* 控制结构 (3) */

xxx ( xxx ) ;
                /* 运算符 (2) */
```

(1) 类型与存储类型的声明

标识符的数据类型与存储类声明了对应的数据目标。详情请参阅 “第 3 章 数据类型与存储类型的声明”。

(2) 运算符与表达式

它们是指示编译器运算的语句，指示编译器运行算术运算、逻辑运算或赋值运算等。详情请参阅 “第 5 章 表达式和运算符”。

(3) 控制结构

这是语句用来规定程序流程。 C 语言有多个指令用于控制结构，如条件控制、迭代和分支跳转。详情请参阅 “第 6 章 C 语言的控制结构”。

(4) 结构体或共用体

声明一个结构体或共用体。结构体是包括多个不同类型子目标或成员的数据目标。当两个或多个变量共用相同的内存时，可以定义为一个共用体。详情请参阅 “第 7 章 结构体和共用体”。

(5) 外部定义

声明一个函数或外部目标。在根据特别目标或独有的行为来区别 C 语言程序时，函数是一个元素。 C 程序是这些函数的集合。详情请参阅 “第 8 章 外部定义”。

(6) 预处理指令

这是编译器专有的的指令。当源程序中出现对应参数时， #define 指令通知编译器将与第一个操作数相同的参数替换为第二个操作数。详情请参阅 “第 9 章 预处理器指令 （编译器指令)”。

(7) 函数原型声明

声明一个函数的返回值和参数类型。

1.4 C 编译程序的最佳运行特性

在进行程序开发之前，请牢记以下几点 (限制值或最小保证值) 表 1-1。

表 1-1 C 编译器的性能指标最大值

项目	限制值 / 最低保证值
复合语句、循环语句或条件控制语句的嵌套层数	45 层
条件转移的嵌套层数	255 层
在一个声明语句中，算术类型、结构体类型、仅用于共用体类型或不完全类型的指针、数组和函数 （或这些项的任意组合）声明符的数量	12 个
每个表达式中的括号嵌套层数	32 层
用作宏名称的字符数量	256 字符
用作内部或外部符号名称的字符数量	249 字符
每个源程序模块文件的符号数量	1,024 符号[注 1]
一个块中具有块作用域的符号的数量 （块的嵌套层数）	255 符号[注 1]
每个源程序模块文件的宏的数量	10,000 个宏[注 2]
每个函数定义或函数调用中的参数数量	39 个参数
每个宏定义或宏调用中的参数数量	31 个参数
每个源代码逻辑行的字符数量	2048 字符
连接后一个字符串内的字符数量	509 字符
一个数据目标的大小	65,535 字节
#include 指令的嵌套层数	8 层
每个 switch 语句中 case 标签数量	257 个标签
每个转换单元的源代码行数	大约 30,000 行
无需创建临时文件就能完成翻译的源代码行数	大约 300 行
函数调用的嵌套层数	40 层
每个函数中的标签数量	33 个标签
每个目标模块的代码、数据及栈段的总大小	65,535 字节
每个结构体或共用体的成员数量	256 个成员
每个枚举类型中 enum 常量的数量	255 个常量
一个结构体或共用体中包含结构体或共用体的嵌套层数	15 层
初始化时元素的嵌套层数	15 层
每个源程序模块文件中定义的函数数量	1,000

表 1-1 C 编译器的性能指标最大值

项目	限制值 / 最低保证值
一个完整的声明符中，括号内的声明符嵌套层数	591
宏的嵌套	200
-i 选项指定包含文件的路径数量	64

注 1. 当符号可以使用现有的内存空间进行处理而不需要使用任何临时文件时，以上的各项数值为最大值。当由于内存空间不足而使用临时文件时，必须根据文件大小对该值进行更改。

注 2. 该值包括 C 编译器保留的宏定义。

1.5 C 编译器的特色

CC78K0S 支持用于指导 CPU 代码生成的扩展函数，这些扩展函数是 ANSI （美国国家标准协会） C 标准不支持的。C 编译器的扩展函数使得 78K0S 系列的特殊功能寄存器 （SFR）能在 C 语言中进行描述，从而缩短了目标代码，并改善了程序执行速度。

下面概括介绍这些用于缩短目标代码并改善执行速度的扩展函数 :

- 函数可以使用 callt 表区域进行调用。 : callt /__callt 函数
- 变量可以分配到寄存器中。 : 寄存器变量
- 变量可以分配到 saddr 区域中。 : sreg/__sreg
- 能使用特殊功能寄存器名称。 : sfr 区域
- 可以创建不输出堆栈帧信息的函数。:

  noauto 函数 , norec/__leaf 函数
- 在 C 源程序中可以进行汇编语言程序的描述 :

  ASM 语句
- 能按位访问 saddr 或 sfr 区域。:

  bit 类型变量 , boolean/__boolean 类型变量
- 可以用 unsigned char 型指定位段 : 位段声明
- 乘法代码可以使用内联展开直接输出。 : 乘法函数
- 除法代码可以使用内联展开直接输出。 : 除法函数
- 移位代码可以使用内联展开直接输出 : 移位函数
- 可以访问内存空间中的特定地址。 : 绝对地址函数
- 特定的数据和指令可以直接嵌入到代码区域中 : 插入数据函数
- 使用的堆栈在被调用函数方进行校正。 : __pascal 函数
- memcpy 和 memset 直接内联展开并输出。 : 存储器操控函数

下面概括介绍该 CC78K0S 的扩展函数。关于每个扩展函数的详细情况，请参阅 “第 11 章 扩展函数”。

(1) callt 函数 (callt / __callt)

函数可以使用 callt 表区域进行调用。每个待调用的函数 （该函数称为 callt 函数）地址存储在 callt 表中，供以后调用。这样能使代码比常规调用指令要短且对缩减目标代码有帮助。

(2) 寄存器变量 (register)

使用寄存器存储说明符进行声明的变量被分配到寄存器或 saddr 区域。分配到寄存器或 saddr 区域的变量，其相关指令比那些分配到内存的变量使用的指令在代码长度上要更短。这样有助于缩短目标代码和改善程序执行速度。

(3) 如何使用 saddr 区域 (sreg / __sreg)

使用关键字 sreg 声明的变量可以分配到 saddr 区域。 sreg 变量的相关指令比分配到内存的那些变量的指令在代码长度上要短。这有助于缩短目标代码和改进程序执行速度。还可以根据选项将对应类型的变量分配到 saddr 区域。

(4) 如何使用 sfr 区域 (sfr)

通过声明使用 sfr 名称，对 sfr 区域的操作可以基于直接在 C 源代码文件进行描述。

(5) noauto 函数 (noauto)

声明为 noauto 的函数不输出代码的预处理和后处理（堆栈帧信息）过程。通过调用 noauto 函数可以用寄存器传递参数。这样有助于缩短目标代码和改进程序执行速度。该函数对参数 / 自动变量加以限制。关于细节，请参阅 “11.5 (5) noauto 函数 (noauto)”。

(6) norec 函数 (norec)

声明为 norec/__leaf 的函数不输出代码的预处理和后处理（堆栈帧信息）过程。通过调用 norec/__leaf 函数，参数将尽可能经过寄存器进行传递。 norec/__leaf 函数内使用的自动变量被分配给寄存器或 saddr 区域。这有助于缩短目标代码和改进程序执行速度。该函数对参数 / 自动变量加以限制，并且不允许调用其他函数。关于细节，请参阅 “11.5 (6) norec 函数 (norec)”。

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

产生占用 1 位存储区的变量。使用位型变量或 boolean/__boolean 型变量，可以按位访问 saddr 区域。boolean/__boolean 型变量与位型变量的功能和用法相同。

(8) ASM 语句 (#asm #endasm / __asm)

用户编写的汇编源程序可以嵌入到 CC78K0S 输出的汇编源文件中。

(9) 中断函数 (#pragma vect / #pragma interrupt)

预处理器指令输出一个向量表，并输出与中断对应的目标代码。该指令允许在 C 源代码级别上对中断函数进行编程。

(10) 中断函数修饰符 (__interrupt)

该修饰符允许设置一个向量表，并允许定义在另一个文件中描述中断函数。

(11) 中断函数 (#pragma DI, #pragma EI)

将中断禁止指令和中断使能指令嵌入到目标代码中。

(12) CPU 控制指令 (#pragma HALT / STOP / NOP)

Each of 以下指令都嵌入到目标 中 :

halt 指令

stop 指令

nop 指令

(13) 绝对地址访问函数 (#pragma access)

访问普通存储空间的代码通过直接内联展开进行创建，无需借助于函数调用，并创建一个目标文件。

(14) 位段声明

将位段指定为无符号字符型，可以节省内存，缩短目标代码，并提高执行速度。

(15) 改变编译器输出区名称的函数 (#pragma section ...)

通过更改编译器输出区名称，被改名的段就可以脱离连接器进行独立地分配。

(16) 二进制常量 （二进制常量 0bxxx）

可以在 C 源代码中描述二进制常量。

(17) 模块名更改函数 (#pragma name)

可以在 C 源代码中自由地更改目标模块名称。

(18) 循环移位函数 (#pragma rot)

将表达式的值循环移位的代码可以在目标文件中用内联展开直接输出。

(19) 乘法函数 (#pragma mul)

将计算乘法表达式的值的代码用内联展开直接输出。该函数可以缩短目标代码，并改进执行速度。

(20) 除法函数 (#pragma div)

将计算除法表达式的值的代码用内联展开直接输出。该函数可以缩短目标代码，并改进执行速度。

(21) BCD 运算函数 (#pragma bcd)

该函数将目标操作值的 BCD 调整操作代码使用直接内联展开直接输出到目标文件。 BCD 操作是指将十进制数的每位数字转换为 4 位二进制数加以存储。

(22) 数据插入函数 (#pragma opc)

常量数据被插入到指定地址中。可以不用汇编语言就将特殊数据和指令嵌入到代码区。

(23) 静态模式

在编译过程中指定 -sm 选项，可以缩短目标代码，改进执行速度，实现高速中断处理，并节省内存空间。

(24) 类型更改 (-ZI)

通过指定 -ZI 选项和 -ZL 选项， int/short 型将被视为 char 型， long 型将被视为 int 型。

(25) Pascal 函数 (__pascal)

用于在函数调用时放置参数的堆栈调整工作在被调用函数方执行，而不是在函数调用方执行。当此类函数调用大量出现时，能够缩短目标代码。

(26) 函数调用接口的自动 pascal 函数化 (-zr)

在编译过程中指定 -ZR 选项，除了 norec/__interrupt/variable 可变的函数之外，其他函数全都会增加 __pascal 属性。

(27) 参数 / 返回值的 int 扩展限制方法 (-ZB)

编译过程中指定 -ZB 选项，可以缩短目标代码，并提高执行速度。

(28) 数组偏移量计算的简便方法 (-qw2 / -qw4)

编译过程中指定 -qw2 和 -qw4 选项，可以简化偏移量计算代码，缩短目标代码，并提高执行速度。

(29) 寄存器直接引用函数 (#pragma realregister)

在源程序中编写调用该函数的代码，调用方式和函数调用相同，或者在模块中使用 #pragma realregister 指令声明都可以使用寄存器直接引用函数，就可以通过 C 规范轻松实现对寄存器的访问。

(30) 内存控制函数 (#pragma inline)

使用 #pragma 内联指令，可以使用内联展开 （而非函数调用）来输出标准库函数 memcpy 和 memset，从而生成一个目标文件。该函数可以改进执行速度。

(31) 绝对地址分配规范 (__directmap)

通过在模块中声明 __directmap，可以为任意地址分配一个或多个变量，在该模块定义待分配到绝对地址的变量。

(32) 静态模式展开规范 (-zm)

编译过程中指定 -ZM 选项，可以放松对现有静态模式的限制，从而改进描述性能。

(33) 临时变量 (__temp)

编译过程中指定 -SM 和 -ZM 选项，并将参数和自动变量声明为 _ _temp ，则可以保留一个存储区用来存储参数和自动变量。

此外，如果包含参数和自动变量的区域被明确定义的话，且对那些在函数调用前后无须保证值匹配的变量加 __temp 声明，则可以保留内存。

(34) 支持序言 / 尾声的库 (-zd)

编译过程中指定 -ZD 选项，可以用库代替序言 / 尾声代码，从而缩短目标代码。

第 2 章　C 语言的结构

本章介绍 C 源程序模块文件的构成要素。

一个 C 源程序模块文件由以下标记 （字符序列中可以辨别出来的单元）构成。

关键字	标识符	常量
字符串	运算符	分隔符
头文件名	预处理的编号	注释

下面举例介绍在 C 程序中使用的标记。

```
#include    " expand.h "
extern      void    testb ( void ) ;        extern          /* 关键字 */
extern      void    chgb ( void ) ;
extern      bit     data1 ;                 data1 , data2   /* 标识符 */
extern          bit     data2 ;

void    main ( )                            void            /* 关键字 */
{
        data1 = 1 ;                         1               /* 常量 */
        data2 = 0 ;                         0               /* 常量 */

        while ( data1 ) {                   while           /* 关键字 */
                data1 = data2 ;             { }             /* 分隔符 */
                testb ( ) ;                 =               /* 运算符 */
        }
        if ( data1 && data2 ) {             if              /* 关键字 */
                chgb ( ) ;                  &&              /* 运算符

        }                                   ( )             /* 运算符

}

void    lprintf ( char *s , int i )         lprintf         /* 标识符 */
{                                           char , int      /* 关键字 */
        int     j ;                         s , I           /* 标识符 */
        char    *ss ;                       *               /* 运算符 */
        j = i ;
        ss = s ;
}
            :
```

2.1 字符集

2.1.1 字符集

C 程序中使用的字符集包括用于描述源文件的源字符集，和在执行环境下进行解释的执行字符集。

执行字符集中每个字符的值由 JIS 代码表示。

以下字符可以同时在源字符集和执行字符集中使用：

表 2-1 可以在字符集中使用的字符列表

26 个大写字母 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 26 个小写字母 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 十进制数字 0 1 2 3 4 5 6 7 8 9 29 个图形字符 ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~ 以及用于显示空格、横向制表、垂直制表、换页等的非打印控制字符。（请参阅下文中的转义字符序列。）

备注　在字符常量、字符串文字、注释语句中，可能会使用上述字符以外的字符。

2.1.2 转义字符序列

转义字符可以用来表示控制字符，比如提示警告或换页等。每个转义字符都是由 \ 符号和一个字母字符组成。

由转义字符表示的非图形字符如下表所示。

表 2-2 转义字符列表

转义字符序列	含义	字符代码
\a	警示	07H
\b	退格	08H
\f	换页	0CH
\n	换行	0AH
\r	回车	0DH
\t	水平制表符	09H
\v	垂直制表符	0BH

2.1.3 三字符序列

当源文件中包含下表左列中的三个字符的序列（称为“三字符序列”）时，这些列出的 3 字符将被转换为下表右列中相应的单个字符。

当设定编译器选项 -za 时，字符序列可用（该选项禁用不使用 ANSI 规范编译的函数，且允许部分符合 ANSI 规范的函数）。

表 2-3 三字符序列列表

三字符序列	含义
??=	#
??(	[
??/	\
??)	]
??í	^
??<	{
??!	|
??>	}
??-	~

2.2 关键字

2.2.1 ANSI-C 关键字

以下标记在本 C 编译程序 C 编译器中用作关键字，因此不可用作标号标签或变量名。

表 2-4 ANSI-C 关键字列表

auto	break	case	char	const	continue	default
do	double	else	enum	extern	for	float
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

2.2.2 为 CC78K0S 增加的关键字

CC78K0S 中，下列标记被作为关键字填加用来使一些扩展函数生效。这些标记不可用作标号标签或变量名，ANSI C 对这些标记不兼容 （但是当包含大写字母时，此标记将不被视为关键字）。

不是以 “_” 开头的关键字可以通过选中 ANSI-C 规范使能选项 (-za) 来设置为无效。

callf, __callf, rtos_interrupt, 和 interrupt_brk 被 CC78K0 兼容为关键字。

表 2-5 为 CC78K0S 增加的关键字

__callt / callt :	callt 函数声明
__callf / callf :	callf 函数声明
__sreg / sreg :	sreg 变量声明
noauto :	noauto 函数声明
__leaf / norec :	norec 函数声明
bit :	bit 型变量声明
__boolean / boolean :	boolean 型变量声明
__interrupt :	硬件中断函数
__interrupt :	软件中断函数
__asm :	汇编语句
__rtos_interrupt :	RTOS 中断处理程序
__pascal :	Pascal 函数
__flash :	固件 ROM 函数
__directmap :	绝对地址分配规范
__temp :	临时变量
__mxcall :	__mxcall 函数[注]

注　　用于与 MX 链接的保留关键字。用户不能使用该关键字。

2.3 标识符

标识符是用以下变量的名称 :

表 2-6 标识符列表

函数
目标
结构体标记、共用体标记或枚举型标记
构体成员、共用体成员或枚举型成员
typedef 名称
标签名称
宏名称
宏参数

每个标识符都可以由大写字母、小写字母、数值字符和下划线组成。以下字符可以被用作标识符 :

标识符的最大长度没有限制。然而 ,CC78K0S 中 , 目前只能识别 249 个字符。

表 2-7 标识符可用使用的数字和字符

_ _ (下划线)
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
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
0 1 2 3 4 5 6 7 8 9

所有的标识符都必须以非数值字符开始 （即，以字母或下划线开始），且不得与任何关键字重名。

2.3.1 标识符的作用域

标识符的有效作用域取决于标识符声明的位置。标识符的作用域分为以下四种类型：

- 函数作用域
- 文件作用域
- 块作用域
- 函数原型作用域

```
extern __boolean data1 , data2 ;  data1 , data2 /* 文件范围 */
void    testb ( int x )           x             /* 函数原型作用域 */
void   main ( void )
{
       int     cot ;              cot           /* 块作用域 */
       data1 = 1 ;
       data2 = 0 ;

       while ( data1 ) {
                data1 = data2 ;
       j1 :                       j1            /* 函数作用域 */
                testb ( cot ) ;
}
}

void    testb ( int x )           x             /* 块作用域 */
{
     :
}
```

(1) 函数作用域

函数作用域指的是函数内的全部范围。作用域与函数作用域相同的标识符可以从该函数内的任何地方进行引用。

具有函数作用域的标识符只有标签名称。

(2) 文件作用域

文件作用域指翻译 （编译）单元内的全部范围。在块的外部或参数列表之外声明的标识符的有效范围为文件作用域。作用域与程序作用域相同的标识符可以在程序内的任何地方引用。

(3) 块作用域

块作用域指一个块的范围 （由一对花括号括起来的声明和语句序列，开始于左括号，结束于右括号）。

在块或参数列表内声明的标识符均具有块作用域。具有块作用域的标识符的有效范围是从定义位置直到包含标识符声明最内层的一对括号闭合为止。

(4) 函数原型作用域

函数原型作用域是指一个已声明函数从开始到结束的范围。在函数原型内的参数列表中声明的所有标识符均都具有函数原型作用域。具有函数原型作用域的标识符在指定的函数内均有效。

2.3.2 标识符的连接

标识符的连接是指标识符可以作为相同的对象或函数来引用，要求是在不同作用域声明一次以上的同一标识符，或在同一作用域声明一次以上的同一标识符通过相互连接，多个标识符可以被视为同一个标识符。可以使用以下三种不同的方法将标识符连接起来：外部连接、内部连接和无连接。

(1) 外部连接

外部连接是指标识符在多个翻译（编译）单元内被连接并构成了整个程序和库的集合。

下面给出了具有外部连接的标识符示例：

- 已经声明却未指定存储类型的函数标识符
- 已经声明为 extern 却未指定存储类型对象或函数的标识符
- 具有文件作用域却未指定存储类型的对象标识符

(2) 内部连接

内部连接是指标识符将在一个翻译（编译）单元内进行连接。

下面给出了具有内部连接的标识符示例：

- 具有文件作用域且指定存储类型为 static 的对象或函数的标识符

(3) 无连接

与其他标识符没有任何连接的标识符，本身就是是一个固有整体。

没有连接的标识符示例如下：

- 不引用数据对象或函数的标识符
- 被声明为函数参数的标识符
- 在块内没有指定存储类型 extern 的对象的标识符

2.3.3 标识符名字空间

所有的标识符可以归类为以下“名字空间”：

- 标签名：　　由一个标签声明来辨别。
- 结构体、共用体或者枚举类型标签名字：　　由关键字 struct, union 或者 enum 来辨别
- 结构体或者共用体成员名：　　由点运算符 (.) 或箭头运算符 (->) 辨别。
- 普通标识符 (除了上述以外)：　　声明为普通标识符或枚举型常量。

2.3.4 对象的存储区间

每个对象都有一个决定其生存期（它在内存中存在的时间）的“存储区间”。存储区间分为以下两类 : 静态存储区间和自动存储区间

(1) 静态存储区间

在执行具有静态时间存储的目标程序前，为存储的对象和值保留一个区域，并初始化。在整个程序的执行过程中都存在并保留最后存放的值。

具有静态存储区间的对象如下所示。

- 具有外部连接的对象
- 具有内部连接的对象
- 由存储类别标识符 static 声明的对象

(2) 自动存储区间

对于具有自动存储时间的对象，当这些对象进入他们被声明的程序块时，将为其保留一个区域。

如果规定了初始化，当这些对象从进入程序块的开头时就会被初始化。在这种情况下，如果有对象进入程序块的方式是通过跳转到程序块内的一个标签标号而进入程序块时，则该对象不进行初始化。

对于具有自动存储区间的对象，当声明的程序块执行完毕时，存储区域将无法保留。

具有自动存储区间的对象如下所示 :

- 没有连接的对象
- 在没有存储类别标识符 static 的块内声明的对象。

2.4 数据类型

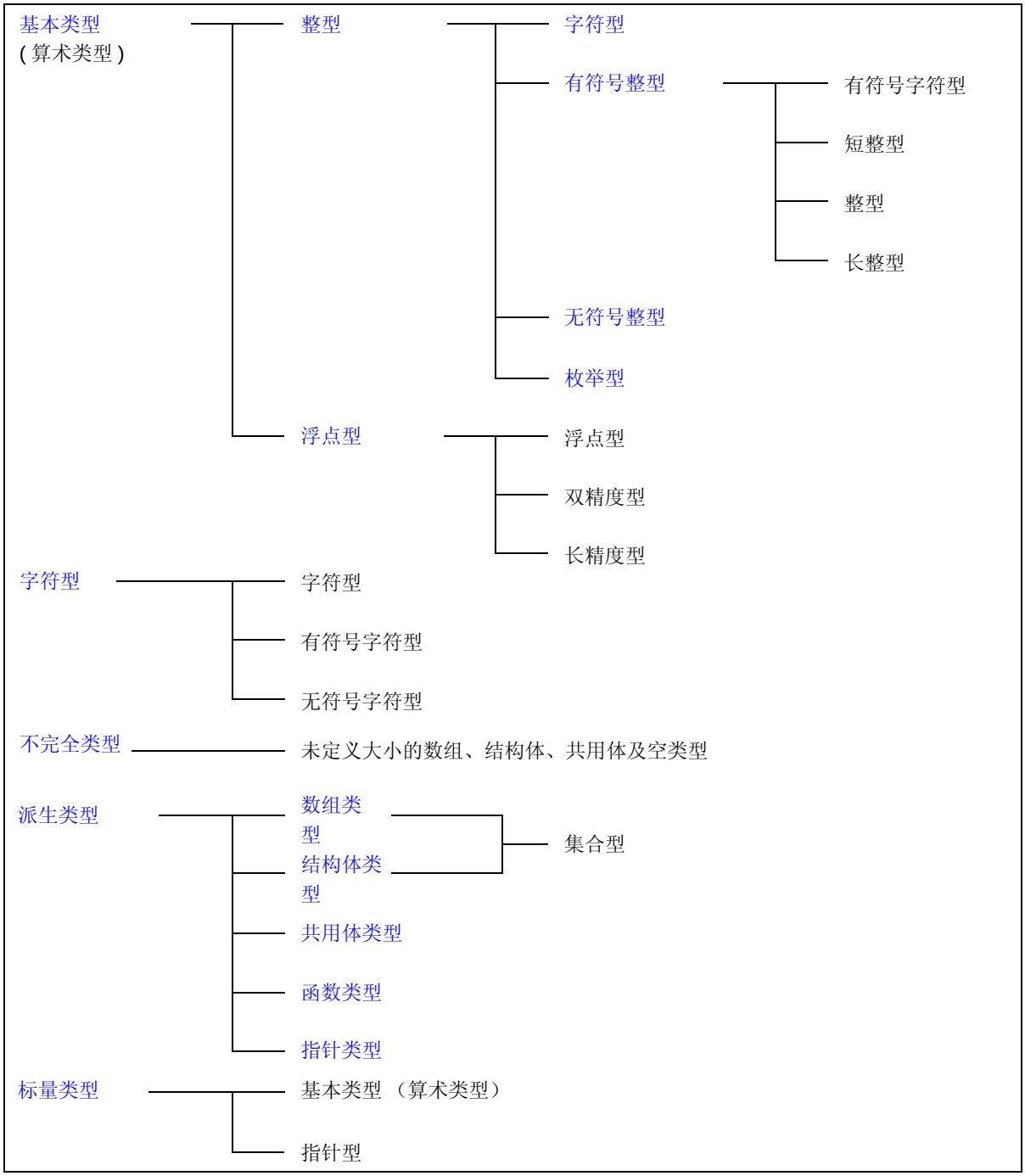
类型决定了存储在每一个目标内的值的含义。

数据类型根据被声明的变量分为以下 3 大类。

- 对象类型 ： 表示具有大小信息的对象的类型
- 函数类型 : 指示函数的类型
- 不完全类型 : 表示一个不具有大小信息的对象的类型

数据类型的分类可用图 2-1 表示。

图 2-1 数据类型分类



2.4.1 基本类型

基本数据类型也称为“算术类型”。算术类型由整型和浮点型组成。

(1) 整型

整型数据类型又分为 4 类。每一类的值都是由二进制数 0 和 1 表示。

- 字符型
- 有符号整型
- 无符号整型
- 枚举型

(a) 字符型

字符型具有足够的长度，来存储基本执行字符集中的任何字符。存储在字符型对象中的字符的值将成为正值。非字符数据将作为无符号整数处理。但是，如果在此情况下发生溢出，则溢出的部分将被忽略。

(b) 有符号整型

有符号整型被分为以下四种类型：

- 有符号字符型
- 短整型
- 整型
- 长整型

声明为有符号字符型的对象，其存储区与无修饰符的字符型的存储区大小相同。

无修饰符的 int 型对象具有与执行环境的 CPU 体系结构相适应的大小。有符号整型数据具有其对应的无符号整型数据相同大小的存储区。有符号整型数据中的正数都属于无符号整型数据。

(c) 无符号整型

无符号整型数据是用关键字 unsigned 定义的。任何涉及无符号整型数据的计算都不会发生溢出。原因在于，如果涉及无符号整型数据参与的计算的结果是一个不能用一个整型表示的值，则该值将被除，除数就是无符号整型可表示的最大值加 1，并用相除的结果中余数来代替原值。

(d) 枚举型

枚举就是一个集合，或是已知的整型常量列表。枚举型由一组或多组枚举类型数据组成。

(2) 浮点型

浮点型细分为三类。

- 浮点型
- 双精度型
- 长精度型

在 CC78K0S 中，将双精度型、长精度型及浮点型都作为 ANSI/IEEE 754-198 中规定的单精度规格标准化数的浮点表达式，标准化的具体内容由 ANSI/IEEE 754-198 规定。因此，浮点型、双精度型、长精度型的值具有相同的范围。

表 2-8 基本数据类型列表

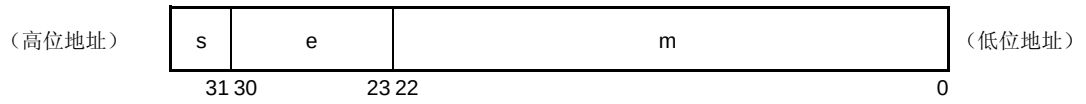
类型	取值范围
（有符号）字符型	-128 ～ +127
无符号字符型	0 ～ 255
（有符号）短整型	-32768 ～ +32767
无符号短整型	0 ～ 65535
（有符号）整型	-32768 ～ +32767
无符号整型	0 ～ 65535
（有符号）长整型	-2147483648 ～ +2147483647
无符号长整型	0 ～ 4294967295
浮点型	1.17549435E-38F ～ 3.40282347E+38F
双精度型	1.17549435E-38F ～ 3.40282347E+38F
长精度型	1.17549435E-38F ～ 3.40282347E+38F

- 关键字 signed 可以省略。然而，对于字符型，系统将根据编译时的情况判断它为有符号字符型还是无符号字符型。
- 短整型数据和整型数据将被当作具有相同取值范围的不同类型数据来处理。
- 无符号短整型数据和无符号整型数据将被当作具有相同取值范围的不同类型数据来处理。
- 浮点型、双精度型和长精度型数据将作为具有相同取值范围的不同类型数据来处理。
- float, double, 和 long double 类型的值范围都是绝对 的。

(a) 浮点数 （浮点型）规范

- 格式

浮点数的格式如下所示。



这种格式的数值如下。

```
          ( 符号的值 )                                                ( 数值的值 )
( -1 )                                        *  （尾数值） *2
```

s : 符号 （1 位）

0 表示正数， 1 表示负数。

e : 指数 （8 位）

以底数 2 的指数表示为一个 1 字节的整数 （在负数情况下，用 2 的补码表示），并在增加偏移量 7FH 后使用。对应关系如下表 2-9 所示。

表 2-9 指数关系

指数 （十六进制）	指数的值
FE : 81 80 7F 7E : 01	127 : 2 1 0 -1 : -126

m : 尾数 （23 位）

尾数以一个绝对值形式表示，其第 22 位至 0 位相当于二进制数的第 1 位至 23 位。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，在这个格式中忽略表示。

- 0 的表示

当指数为 0，尾数为 0 时，±0 表示如下。

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

- 无穷大的表示

  当指数为 FFH，尾数为 0 时 , ± ∞表达如下。

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

- 非标准化的值

  当指数为 0，且尾数 ″≈ ∈♠ 0 时，非规格化的值表示如下。

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

  备注　　这里的尾数是一个小于 1 的值，因此尾数的 22 至 0 位表示为十进制的第 1 至 23 位。

- 非数值 （NaN）表达式

  当指数为 FFH，尾数≠ 0 时，无论符号是什么，均表示为 NaN。

- 运算结果舍入

  数值被舍入至最接近的偶数。如果运算结果不能用上述的浮点格式表示，则舍入至最接近的可以表示的数。

  如果舍入前的值有两个不同的值可以表示，则舍入至一个偶数 （二进制最低位为 0 的数）。

- 运算异常

  有 5 种运算异常，如下表所示。

表 2-10 运算异常列表

异常	返回值
下溢	非标准化数
不准确	±0
溢出	± ∞
除零	± ∞
无法运算	非数值 （NaN）

当发生异常时，调用 matherr 函数会导致出现警告。

2.4.2 字符型

字符数据类型包括以下三种。:

- 字符型
- 有符号字符型
- 无符号字符型

2.4.3 不完全类型

不完全数据类型包括以下 4 种类型 :

- 未指定目标大小的数组
- 结构体
- 共用体
- 空类型

2.4.4 派生类型

派生类型分为以下 5 种 :

- 数组类型
- 结构体类型
- 共用体类型
- 函数类型
- 指针型

(1) 数组类型

数组类型连续分配一组称为元素类型的成员对象。成员对象均具有相同大小的存储区。数组类型指定数组的元素类型及数组的元素数量。不能创建不完全类型的数组。

(2) 结构体类型

结构体类型连续分配大小可以不同的成员对象。需要给成员对象一个名称来指定各个成员对象。

聚合类型又细分为两类 :

数组类型和结构体类型。聚合类型数据是一组依次提取的成员对象的集合。

(3) 共用体类型

共用体类型是一组在共用内存的成员对象。这些成员对象在大小和名称上不同，并可以单独指定。

(4) 函数类型

函数类型表示一个具有指定的返回值的函数。函数类型数据指定了返回值的类型、参数个数以及参数类型。如果返回值的类型是 T，该函数被称为是一个返回 T 的函数。

(5) 指针类型

指针类型是从一个被称为被引用类型的函数型对象类型中创建的，也可以由及一个不完全类型中创建。指针类型表示一个对象。对象指示的值用来指向被引用类型的实体。

根据被引用的类型 T 创建的指针型数据被称为指向 T 的指针。

2.4.5 标量类型

算术类型 （基本类型）及指针类型总称为标量类型。标量类型包括以下数据类型：

- 字符型
- 有符号整型
- 无符号整型
- 枚举型
- 浮点型
- 指针型

2.4.6 兼容类型

如果两个类型相同，则说它们是兼容的或者具有兼容性。例如，如果在不同的翻译 （编译）单元中声明的两个结构体、共用体或枚举型具有相同的成员数量，相同的成员名称及兼容的成员类型，则它们拥有兼容的类型。在此情况下，两个结构体或共用体的单个成员必须具有相同的顺序，两个枚举型的单个成员 （枚举的常量）必须具有相同的值。

与同一对象或函数有关的所有声明必须具有相容兼容的类型。

2.4.7 复合类型

复合类型是从两个兼容类型中创建的。复合类型的规则如下。

- 如果两个种类型中的任一个是具有已知类型大小的数组，则复合类型就是一个具有相同大小的数组。
- 如果这两个类型中仅有一个是具带有一个参数类型列表 （用原型声明）的函数类型，则复合类型是一个具有参数类型列表的函数原型。
- 如果两个类型都具有一个参数类型列表 （即具有原型的函数），则复合类型就是具有如下原型：包括从这两个原型中提取的所有信息。

< 复合原型示例 >

假设以下两个声明具有文件作用域：

```
int f ( int ( * ) ( ) , double ( * ) [ 3 ] ) ;
int f ( int ( * ) ( char * ) , double ( * ) [ ] ) ;
```

在此情况下，函数的复合类型成为：

```
int f ( int ( * ) ( char * ) , double ( * ) [ 3 ] ) ;
```

2.5 常量

常量是一个在程序执行过程中其值不会发生变化的量，该值必须预先设置。每个常量的类型根据为该常量指定的格式和值来决定。常量的类型有如下四种 :

- 浮点型常量
- 整型常量
- 枚举型常量
- 字符型常量

2.5.1 浮点型常量

浮点型常量由有效的数字部分、指数部分和浮点后缀组成。

- 有效数字部分 : 整数部分、小数点和小数部分
- 指数部分 : e 或 E，有符号指数
- 浮点后缀 : f/F (浮点型)
  l/L (长精度型)
  如果省略 (双精度)

指数部分的带符号指数和浮点后缀可以省略。

无论整数部分还是小数部分都必须被包括在有效数字内。而且，无论是小数点还是指数部分都必须被包括 （例如：1.23F, 2e3）。

2.5.2 整型常量

整型常量以一个数字开始，不包括小数点和指数部分。可以在整型常量后添加一个无符号后缀，以表明该整型常量是无符号的。可以在整型常量后添加一个长整型后缀，以表明该整型常量是长整型的。

共有以下三种整型常量。

- 十进制常量 : 以一个非 0 数字开始的十进制数

  十进制数 = 1 2 3 4 5 6 7 8 9

- 八进制常量 : 整型前缀 0 + 八进制数

  八进制数 = 0 1 2 3 4 5 6 7

- 十六进制常量 : 整型前缀 0x 或 0X + 十六进制数字

  十六进制数 = 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

```
无符号后缀
        u U
长整型 后缀
        l L
```

(1) 十进制常量

十进制常量是一个以 10 为底数 （基数）的整数，该数必须以非 0 的数值开始，其后跟随从 0 至 9 的任何数字数 （例如：56U）。

(2) 八进制常量

八进制常量是一个以 8 为底数 （基数）的整数，该数必须以 0 开始，其后跟随的数字可以是从 0 至 7 的任何数字 （例如：034U）。

(3) 十六进制常量

十六进制常量是一个以 16 为底数 （基数）的整数，该数必须以 0x 或 0X 开始，其后跟随的数字可以是从 0 至 9 的任何数，或 a 至 f 或 A 至 F 表示从 10 至 15 的任何数 （例如：0xF3）。

这个类型的整型常量被视为首选的 “可表示类型”，如下所示。

CC78K0S 中 , 根据编译程序的条件 （选项），无下标的常量的类型可以更改为字符型或无符号字符型。

表 2-11 整型常量 和 可表示的类型

整型常量	可表示的类型
无后缀十进制数	整型 , 长整型 , 无符号长整型
无后缀八进制数、十六进制数	整型 , 无符号整型 , 长整型 , 无符号长整型
后缀 u 或 U	无符号整型 , 无符号长整型
后缀 l 或 L	长整型 , 无符号长整型
后缀 u 或 U， l 或 L	无符号长整型

2.5.3 枚举型常量

枚举常量用于表示一个枚举型变量的一个元素，即枚举型变量的值只能是特定值，此值由标识符给定。

枚举型 （enum）可以是以下列出的三种类型中的任何一种，可以表示所有的枚举常量。枚举常量由标识符表示。

- 有符号字符型
- 无符号字符型
- 有符号整型

它的描述方法是 “enum 枚举类型 { 枚举常量列表 }”。

< 示例 >

```
enum    months { January = 1 , February , March , April , May } ;
```

当使用 = 指定整数时，枚举变量具有整数值，且其后的枚举变量值为上述整数值顺序 +1。在上述示例中，枚举变量的值分别为 1， 2， 3， 4，5。当上例中没有 “=1” 时，每个常量的值为 0， 1， 2， 3， 4， 5

2.5.4 字符型常量

字符常量是括在一对单引号中的一单个字符或多个字符串，如 ‘X’ 或 ‘ab’。

字符常量不包括单引号 （’）、反斜线 （\）和换行符 （\n）。要表示这些字符，需要使用转义字符序列。有以下三种转义字符序列。

- Simple escape sequence : \í \" \?\\ \a \b \f \n \r \t \v
- 八进制转义字符序列 : \ 八进制数 [八进制数 八进制数]
  (例 : \012, \0 注 1)
- 十六进制转义字符序列 : \x 十六进制数
  (例 : \xFF 注 2)

注 1. 空字符

注 2. CC78K0S 中， \xFF 表示 -1。不过，如果增加了将 char 视为 unsigned char 的条件 （选项），它表示的值就是 +255 。

2.6 字符串

字符串是在一对双引号“XXX”中的零个或更多个字符（例如：“xyz”}

单引号（’）可以由单引号本身来表示，或者由转义字符序列\’表示，而双引号（”）则由 字符转义序列\”来表示。

数组元素可以是字符型的字符串文字，并由给出定的标记进行初始化（例如：char array [ ] = “abc”;）。

2.7 运算符

运算符如下所示。

表 2-12 运算符列表

```
[ ]  ( )  .  ->
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=
^  |  &&  ||
?  :
=  *=  /=  %=  +=  -=  <<=  >>=
&=  ^=  |=
,  #  ##
```

[ ]、( ) 和 ?: 运算符必须成对使用。

一个表达式可以在方括号 “[]” 中，圆括号 “（）” 中，和 “？” 与 “：” 中被描述。

和 ## 运算符只能用于预处理指令中定义宏的时候。(关于细节，请参阅 “第 5 章 表达式和运算符”。)

2.8 分隔符

分隔符是一个具有独立的句法或意义的符号。不过，但是它不会产生值。

在 C 语言中可以使用的分隔符如下所示。

```
[ ]  ( )  { }  *  , :  =  ;  ...  #
```

一个表达式可以在方括号“[]”中，圆括号“（）”中，或者花括号“{}”中声明或者定义。这些分隔符也必须如上述所示的那样成对使用。分隔符“#”只能用于预处理指令。

2.9 头文件名

头文件名表示一个外部源文件的名称。该名只能在预处理指令 “#include”中使用。

下面是一个头文件名的 #include 指令的示例。关于各个 #include 指令的细节，请参阅 “9.2 源文件包含指令”。

```
#include        < header name >
#include        " header name "
```

2.10 注释

注释是指包括在 C 源程序模块中只提供作为参考信息的语句。它以 “/*” 开头，以 “*/” 结尾。也可以使用 -ZP 选项将 “//” 后至换行符之间的部分所有标识作为注释语句处理。

< 示例 >

```
/* 注释语句 */
// 注释语句
```

第 3 章　数据类型与存储类型的声明

本章介绍如何声明 C 中使用的数据 （变量）或函数，以及每个数据或函数的作用域。声明是对一个标识符或一组标识符的解释或属性进行的说明。为标识符命名的对象或函数保留一个适当的存储区的声明被称为 “定义”。

下面是一个声明的示例。

表 3-1 类型声明和存储类型声明示例

```
#define TRUE    1
#define FALSE   0
#define SIZE    200
void    main ( void )
{
    auto    int     i , prime , k ;  /* 声明自动变量 */
    for ( i = 0 ; i <= SIZE ; i++ )
            mark [ i ] = TRUE ;
                :
}
```

声明由存储类声明符、类型声明符、初始化声明符等组成。存储类声明符及类型声明符将会指定由声明符所定义指定的实体的连接、存储生存期及类型。初始化声明符列表会列出所有的声明符，声明符之间由逗号分开。每个声明符都可以有附加的类型信息或初始化符，或者二者兼有。

如果一个对象的标识符声明它无没有连接，则该对象的类型必须是恰当的 （对象具有大小相关的信息），位置处于在声明符或初始化声明符 （如果有的话）尾部的对象的类型必须是完整的 （具有与大小相关信息的对象）。

3.1 存储类声明符

存储类声明符指定一个对象的存储类别。它说明对象的作用域和对象具有的值的存储位置单元。在一个声明中，只能说明一个存储类声明符。

共有以下 5 个存储类声明符可供使用。

表 3-2 存储类型声明符

声明符类型	含义
typedef	typedef 声明符为指定的类型声明一个替代名。 关于 typedef 声明符的详细情况，请参考 “3.6 typedef 声明”
extern	extern 声明符说明 （告诉编译器）在紧随该声明符之后的这一个变量是在其他程序中声明的 （即，一个外部变量）。
static	static 声明符说明对象具有静态存储区间。 对于具有静态存储区间的对象，在程序执行前就会为其保留一个存储区被保留下来，且待存储的值只初始化一次。对象存在于程序的整个执行过程中，并保留其最后存储的值。
auto	auto 声明符说明对象具有自动存储区间。 对于具有自动存储生区间的对象，当这些对象进入一个其声明语句所在的程序块时，将为其保留一个存储区。从开始顶部进入这个包含声明语句的程序块时，如果有指定的话，对象将进行初始化。如果对象是通过跳转到程序块内的一个标签的方式而进入程序块时，该对象将不被初始化。 当执行完毕声明语句所在的程序块时，为具有自动存储区间的对象所保留的存储区将无法得到保证。
register	register 声明符将一个对象指定分配给 CPU 的寄存器。 通过 CC78K0S, 它将被分配给到 CPU 的寄存器或 saddr 区域存储区。参阅 “第 11 章 扩展函数” 获取详细信息。

3.2 类型声明符

类型声明符指定 （或表示）一个对象的类型。可用下列类型声明符：

- void 型
- 字符型
- 短整型
- 整型
- 长整型
- 浮点型
- 双精度型
- 长精度型
- 有符号型
- 无符号型
- 结构体声明符与共用体声明符
- 枚举声明符
- typedef 名称

CC78K0S 中 , 增加了以下类型声明符。

- bit / boolean / __boolean

下面是对每个类型声明符的含义以及在 CC78K0S 中可以表示的极限值 （圆括号中的值）加以的说明。对于浮点运算，由于 CC78K0S 只支持 IEEE Std 754-1985 标准的单精度，因此 double 和 long double 数据被认为具有与 float 数据相同的格式。

用斜线分开的声明符具有相同大小。

表 3-3 类型声明符

声明符类型	含义	限制
void	空值集合	-
字符型	可以存储的基本字符集数量	-
有符号字符型	有符号整型	-128 ～ +127
无符号字符型	无符号整型	0 ～ 255
短整型 / 有符号短整型 / 短整型 / 有符号短整型	有符号整型	-32768 ～ +32767
无符号短整型 / 无符号短整型	无符号整型	0 ～ 65535
整型 / 有符号型 / 有符号整型	有符号整型	-32768 ～ +32767
无符号型 / 无符号整型	无符号整型	0 ～ 65535
长整型 / 有符号长整型 / 长整型 / 有符号长整型	有符号整型	-2147483648 ～ +2147483647
无符号长整型 / 无符号长整型	无符号整型	0 ～ 4294967295
浮点型	单精度浮点数	1.17549435E - 38F to 3.40282347E + 38F 注
双精度型	双精度浮点数	1.17549435E - 38F to 3.40282347E + 38F 注
长精度型	扩展精度浮点数	1.17549435E - 38F to 3.40282347E + 38F 注
结构体 / 共用体声明符	成员对象集合	-
枚举声明符	整型常量集合	-
typedef	指定类型的替代名	-
bit / boolean / __boolean	用 1 位表示的整数	0 ～ 1

注　绝对值范围

3.2.1 结构体声明符与共用体声明符

结构体声明符和共用体声明符均说明一组已命名的成员 （对象）的集合。这些成员对象的类型可以互不相同。

(1) 结构体声明符

结构体声明符将一组两个或多个不同类型的变量声明为一个对象。每个类型的对象称为一个成员，并可以为其赋予一个名称。对于成员，按照它们的声明顺序保留连续的存储区。

不过，由于 78K0S 系列具有如下限制：字数据不能从奇地址中读取，也无法将字数据写入奇地址。因此默认情况下代码大小要进行优先级更高排定，可以并插入对齐数据以确保 2 字节或多字节的成员被分配给到偶地址。由于对齐数据的原因，在成员之间可能会产生间隙。

可以指定 -RC 选项来禁止插入对齐数据，以便使得结构更加紧凑。在这种此情况下，尽管减小了数据的大小数量，但是，分配给到奇地址的 2 字节或多字节的成员通过使用单字节读 / 写代码来实现读 / 写，这样会就增加了代码的大小。

结构体的声明如下。但是，声明并不会分配内存地址，因为它没有结构体变量列表。关于结构体变量声明的细节，请参阅 " 第 7 章 结构体和共用体 "。

```
struct  标识符 { 成员声明列表 } ;
```

< 结构体声明示例 >

```
struct  tnode  {
        int     count ;
        struct  tnode   *left , *right ;
} ;
```

(2) 共用体声明符

共用体声明符将一组两个或多个不同类型的变量声明为一个对象。每个类型的对象称为一个成员，并可以为其赋予一个名称。共用体的成员在存储区上是互相重叠的，即它们共用相同的存储空间。

共用体的声明如下。但是，声明不会分配内存地址，因为它没有共用体变量列表。关于共用体变量声明的细节，请参阅 " 第 7 章 结构体和共用体 "。

```
union    标识符 { 成员声明列表 } ;
```

< 共用体声明示例 >

```
union   u_tag  {
        int     var1 ;
        long    var2 ;
} ;
```

每个成员对象可以是任意类型，不完全类型和函数类型除之外。成员可以用规定的位数来声明。具有指定的位数的成员称为位段。

CC78K0S 中 , 增加了与位段声明有关的扩展函数关于细节，请参阅 “11.5 (14) 位段声明”。

(3) 位段

位段是一个整型区域，由指定数量的位组成。可以在为位段中指定 int 型、unsigned int 型及 signed int 型的数据[注 1]。无修饰符的 int 域段的最高有效位或 signed int 域的最高有效位将被认定判为符号位[注 2]。

如果存在两个或多个位段，只要在指定的这个内存单元中有足够的空间，则第二个及后续的位段将被压缩到为相邻的位置。通过放置具有零宽度的未命名位段，则下一个位段将不被压缩到相同内存单元内的空间中。

未命名的位段没有声明符，仅有声明冒号和宽度。

单目 & 运算符 & （取地址）不能应用到位段对象。

注 1. CC78K0S 中 , 还可以指定字符型、无符号字符型及有符号字符型。它们均被视为无符号型，因为 CC78K0S 不支持有符号型位段

注 2. 在 CC78K0S 中 , 位段分配的方向能通过编译器 -rb 选项来改变 (有关更详细的内容 , 参阅 “第 11 章 扩展函数”)。

< 位段示例 >

```
struct  data {
        unsigned int    a : 2 ;
        unsigned int    b : 3 ;
        unsigned int    c : 1 ;
} no1 ;
```

3.2.2 枚举声明符

枚举型声明符声明按顺序放置的一系列对象列表。用 enum 声明符声明的对象将被声明为整型常量。

枚举型声明符的声明如下所示。

```
enum    标识符 { 成员列表 }
```

枚举符列表进行声明的对象。并为列表中的所有对象按照它们的声明顺序定义一个值，方法是：为第一个对象赋予 0 值，前一个对象的值加 1 赋予第二个及后续的对象的值等于前一个对象的值加 1。还可以使用“=”来指定一个常量的值。

在下面的例子中，“hue”被假设为枚举的标记名，“col”为具有该（enum）类型的一个对象，“cp”为指向该类型对象的指针。在该声明中，枚举的值变为 “{0,1,20,21}”

```
enum    hue  {
        chartreuse,
        burgundy,
        claret = 20 ,
        winedark
} ;

enum    hue     col , *cp ;
void main ( void ) {
        col = claret ;
        cp = &col ;
        /* ... */  ( *cp != burgundy ) /* ... */
            :
}
```

3.2.3 标记

标记是为结构体、共用体或枚举型指定的一个名称。标记具有一个已声明的数据类型，可以使用标记声明相同类型的对象。

以下声明中的标识符是一个标记名。

```
structure/union 标识符  { 成员声明列表 }
或
enum  标识符 { 枚举符列表 }
```

标记包含结构体 / 共用体或枚举的内容，其中已经定义了成员变量。 在后续的声明中，结构体、共用体或枚举型的结构变得与标记的列表结构相同。在具有相同作用域内的后续声明中，花括号中的列表必须省略。下面的类型声明符未定义其内容，因此，结构体或共用体具有不完全类型。

```
结构体 / 共用体标识符
```

仅当对象大小无需指定时，才可以使用标记指定该类型声明符的类型。原因在于，在相同作用域内定义标记的内容时，类型说明将变得不完整。

在下面的例子中，“tnode”标记指定了一个结构体，其中包含一个整数指针及两个同类型的对象。

```
struct  tnode{
        int     count ;
        struct  tnode   *left , *right ;
} ;
```

下例将“s”声明为一个具有由标记（tnode）指定的类型的对象，将“sp”声明为指向此类对象的指针，这个对象由标记来说明其类型。通过这种声明，表达式“sp->left”表示一个指向“sp”所指的对象左侧的“struct tnode”指针；“s.right->count”表示“count”，它是“s”右侧“struct tnode”的一个成员。

```
typedef struct  tnode   TNODE ;
struct  tnode  {
        int     count ;
        struct  tnode   *left , *right ;
} ;

TNODE   s , *sp ;
void main ( void ) {
        sp -> left = sp -> right ;
        s.right -> count = 2 ;
}
```

3.3 类型修饰符

有两个类型修饰符可供使用： const 和 volatile. 这些修饰符只影响左侧的值。

使用一个具有非 const 型修饰符的左侧值不能更改使用 const 型修饰符定义的一个对象。使用具有非 volatile 型修饰符的左侧值不能引用使用 volatile 型修饰符定义的对象。

一个用具有 volatile 修饰符定义类型的对象可能会被编译程序编译器无法不易识别的方法更改，或者具有其他不易注意的副作用。因此，引用该对象的表达式必须对该对象严格求解，必须符合根据规定用 C 语言编写的程序如何抽象调整执行的顺序规则对引用该对象的表达式进行严格求解。此外，在每个序列点，最后存储在对象中的值必须与那些由程序确定决定的值一致，除非出现由于上述的编译程序编译器无法不能识别的因素引起的更改。

如果使用类型修饰符指定了一个数组类型，则修饰符适用于数组元素，而非数组本身。

在声明函数类型时不可能包含类型修饰符。然而 , callt, __callt, callf, __callf, noauto, norec, __leaf, __interrupt, __interrupt_brk, __rtos_interrupt, __pascal, 它们是 "2.2 关键字 " 中提及的 CC78K0S 所独有的类型修饰符 , 能作为类型修饰符包含在内。

sreg, __sreg, __directmap, and __temp 也是类型修饰符。

在下面的例子中，“real_time_clock”可以被硬件更改，但是指定、加和减不被允许。

```
extern  const   volatile        int     real_time_clock ;
```

一个使用类型修饰符更改聚合类型数据的示例如下。

```
const   struct  s  { int mem ; } cs = { 1 } ;
struct  s       ncs ;                               /* 对象 ncs 是可变的 */
typedef int     A [ 2 ] [ 3 ] ;
const   A a = { { 4 , 5 , 6 } , { 7 , 8 , 9 } } ; /* array of const int array */
int     *pi ;
const   int     *pci ;

ncs = cs ;              /* 正确 */
cs = ncs ;
  /*  违反对左值的 volatile 限制，左值它具有可调整更改的赋值运算符  */
pi = &ncs.mem ;         /* 正确 */
pi = &cs.mem ;  /* 违反赋值操作 = 的类型限制 */
pci = &cs.mem ;         /* 正确 */
pi = a [ 0 ] ;          /* 不正确 : a [ 0 ] 具有  “ const int * ” 类型 */
```

3.4 声明符

声明符用于声明一个标识符。这里主要讨论指针声明符、数组声明符和函数声明符。声明符可以决定标识符的作用域，也可以决定函数或对象的存储生存期和类型。

下面对各种声明符依次进行介绍。

3.4.1 指针声明符

指针说明符说明表示待声明说明的标识符是一个指针。指针指向 （指示）存储一个值的存储位置单元。指针声明如下所示。

```
*   类型修饰符列表     标识符
```

通过该声明，标识符成为指向 T1 的指针。

以下两个声明分别表示一个指向常量值的变量指针，一个指向一个变量值的常量指针。

```
const   int     *ptr_to_constant ;
int     *const  constant_ptr ;
```

第一个声明表示指针 “ptr_to_constant”所指的常量 “const int”的值不能更改；但是指针 “ptr_to_constant”本身可以更改以指向另一个 “const int”型常量。同样，第二个声明表示指针 “constant_ptr”所指的变量 “int”的值可以更改；但是指针 “constant_ptr”本身则永远指向相同的位置。

通过添加给指向整型数据的指针类型的定义，可以使得常量指针 “constant_ptr”的声明有所不同。

下例将 “constant_ptr”声明为一个对象，该对象是具有指向 const 修饰符并指向 int 型的指针类型。

```
typedef int     *int_ptr ;
const   int_ptr constant_ptr ;
```

3.4.2 数组声明符

数组声明符向编译器声明的标识符是一个数组类型的对象。

数组声明的方式如下所示。

```
type     标识符        [ 常量表达式 ]
```

通过该声明，标识符成为具有类型声明的数组。常量表达式的值指定数组的元素个数。常量表达式必须是一个其值大于 0 的整型常量表达式。在声明数组时，如果未指定一个常量表达式，则数组将变为不完全类型。

在下面的例子中，声明了由 11 个元素组成的 char 型数组 “a[ ]”，还声明了由 17 个元素组成的 char 型指针数组 “ap[ ]”。

```
char    a [ 11 ] , *ap [ 17 ] ;
```

在以下两个声明的例子中，第一个声明中的 “x” 指定一个指向 int 型数据的指针，第二个声明中的 “y” 指定一个 int 型的数组，该数组没有指定大小，需要在程序中的其他地方声明。

```
extern  int     *x ;
extern  int     y [ ] ;
```

3.4.3 函数声明符 （包括原型声明）

函数声明符声明引用的函数返回值和参数的类型，以及待引用的函数的参数值的类型。

函数声明如下所示。

```
类型      标识符        （参数列表或标识符列表）
```

通过该声明，标识符变为一个函数，该函数具有参数类型列表指定的参数，函数并返回值的类型就是在标识符之前声明的类型。函数的参数由参数标识符列表来指定。通过这些列表，就指定了一个表示参数及其类型的标识符。在头文件 “stdarg.h” 中定义的宏将括省略号 （, ...）中描述的列表转换为参数。对于没有参数说明的函数，参数列表将变为 “void”。

3.5 类型名称

类型名称就是数据类型的名称，用于说明函数或对象的大小。从句法上讲，它是一个去掉标识符的函数或对象声明。

下面给出了类型名的示例。

- int : 指定 int 型。
- int * : 指定一个指向 int 型变量的指针。
- int *[3] : 指定一个数组，该数组有三个指向 int 型变量的指针。
- int (*) [3] : 指定一个指针，该指针指向有 3 个 int 型变量的数组。
- int *( ) : 指定一个函数，该函数返回一个指向 int 型变量的指针，该函数没有参数说明。
- int (*) (void) : 指定一个指向函数的指针，该函数返回一个 int 型的值，该函数没有参数说明。
- int (*const [ ]) (unsigned int, ...): 指定一组指向函数的指针，函数的变量名是大小未定的常量数组，这些数组中的函数数组具有一个 unsigned int 型的参数，还有和一个指向各个函数的常量指针，函数会返回一个 int 型值。

3.6 typedef 声明

typedef 关键字定义一个标识符，该标识符与指定的类型具有相同的使用方法。被定义的标识符成为 typedef 名称。

typedef 名称的语法如下所示。

```
typedef 类型     标识符 ;
```

在下面的例子中，“distance”是一个 int 型，“metricp”的类型是一个指向一个函数的指针，该函数返回一个没有参数说明的 int 型值，“z”的类型是一个指定的结构体，“zp”是一个指向该结构体的指针。

```
typedef int     MILES , KLICKSP ( ) ;
typedef struct  { long re , im ; }      complex ;
 /* ... */
MILES   distance ;
extern  KLICKSP *metricp ;
complex z , *zp ;
```

在下面的例子中，指定用 typedef 名 t 代表有符号整型，用 typedef 名 plain 代表整型，并声明了一个具有三个位段成员的结构体。位段成员如下所示。

- 位段成员名称为 t，值为 0 ~ 15 的位段成员
- 位段成员没有名称，const 限定的值为 -16 ~ +15 （如果访问）的位段成员
- 位段成员名称为 r，值为 -16 ~ +15 的位段成员

```
typedef signed  int     t ;
typedef int     plain ;
struct  tag {
        unsigned        t : 4 ;
        const           t : 5 ;
        plain           r : 5 ;
} ;
```

在本例中，这两个位段声明的差别在于，第一个位段声明有 unsigned 作为类型声明符 （因此， t 成为结构体成员的名称），第二个位段声明，在另一方面，有 const 作为类型声明符 （修饰符 t 可以被称为 typedef 名）。在这个声明之后 , 如果在有效范围内发现以下描述 , 作为 " 具有一个参数且返回带符号整型的函数 " 来声明函数 f, 且声明参数为 " 具有 1 个参数且返回带符号整型函数的指针类型 "。标识符 t 被声明为长整型。

```
t       f ( t ( t ) ) ;
long    t ;
```

typedef 名有助于程序的阅读。例如，signal 函数的以下三个声明效果都一样，指定函数与和未使用 typedef 的第一个种函数声明方法相同的类型。

```
typedef void    fv ( int ) ;
typedef void    ( *pfv ) ( int ) ;

void    ( *signal ( int , void ( * ) ( int ) ) ) ( int ) ;
fv      *signal ( int , fv * ) ;
pfv     signal ( int , pfv ) ;
```

3.7 初始化

初始化指的是预先为一个对象设置一个值。初始化符完成对象的初始化。

初始化的执行如下所示。

```
对象 = { 初始化符列表 }
```

初始化符列表必须包含待初始化的各个对象需要使用的初始化符。

对于具有静态存储区间的对象和以及具有聚合类型或共用体类型的对象来说，其初始化符中的所有表达式或初始化符列表都必须使用常量表达式。

声明其作用域为块作用域，但具有外部或内部连接的标识符不能初始化。

3.7.1 具有静态存储区间对象的初始化

如果未对具有静态存储区间的算术型对象进行初始化处理，则对象的值将隐性含地被初始化为 0。

同样地，具有静态存储区间的指针型对象将被默认初始化为一个空指针常量。

< 示例 >

```
unsigned int    gval1 ;         /* 被初始化为 0 */
static int      gval2 ;         /* 被初始化为 0 */
void    func ( void )  {
        static  char    aval ;  /* 被初始化为 0*/
}
```

3.7.2 具有自动存储区间的对象的初始化

如果未进行初始化处理，则具有自动存储区间的对象的值将变得不确定，并且无法得到保证。

< 示例 >

```
void    func ( void )  {
        char    aval ;  /* 在这里未定义 */
            :
        aval = 1 ;      /* 初始化为 1 */
}
```

3.7.3 字符数组的初始化

字符数组可以用字符串文字进行初始化（包含在“”中的字符串）。同样地，包含一系列字符串形式的文字的字符串可以用来对一个数组的单个成员或元素进行初始化。

在下面的例子中，定义了无类型修饰符的数组对象“s”和“t”，并使用字符串文字来初始化每个数组的元素。

```
char s [ ] = " abc " , t [ 3 ] = " abc " ;
```

下面的例子与上面的数组初始化示例作用相同。

```
char    s [ ] = { ' a ' , ' b ' , ' c ' , ' \0 ' } ,
        t [ ] = { ' a ' , ' b ' , ' c ' } ;
```

下面的例子定义 p 为“指向 char 型变量的指针”，且成员用字符串文字进行初始化，以便所以长度指示一个“字符数组”型对象。

```
char    *p = " abc " ;
```

3.7.4 聚合或共用体型对象的初始化

- 聚合型

  聚合类型的对象用初始化符列表来进行初始化，这些初始化符按照下标顺序或成员的描述顺序升序进行描述。待指定的初始化符列表必须用花括号括起来。

  如果列表中的初始化符的数量少于聚合成员的数量，则未被初始化符覆盖到的成员将隐式地被初始化，默认被当作是具有静态存储区间的对象。

  对于一个大小未知的数组，元素的数量由初始化符的数量控制，且数组将不再是一个完全的类型。

- 共用体类型

  共用体型对象用括号中的第一个成员作为的初始化符来进行共用体型对象的初始化处理过程。

在下面的例子中，具有大小未知的数组 “x”将变为一个一维数组，该数组初始化后有三个元素。

```
int     x [ ] = { 1 , 3 , 5 } ;
```

下面的例子显示了一个完整的定义，其中初始化符被括在花括号中。 "{1, 3, 5}" 初始化数组对象 "y [0]" 的第一个行中的 "y [0] [0]"， "y [0] [1]" 和 "y [0] [2]"。同样，在第二行中，数组对象 “y [1]” 和 “y [2]” 的元素被初始化。 "y [3]" 的初始值是 0，因为它未被指定它的值。

```
char    y [ 4 ] [ 3 ] = {
        { 1 , 3 , 5 } ,
        { 2 , 4 , 6 } ,
        { 3 , 5 , 7 } ,
} ;
```

下面的例子产生同上面的例子相同的结果。

```
char    z [ 4 ] [ 3 ] = {
        1 , 3 , 5 , 2 , 4 , 6 , 3 , 5 , 7
} ;
```

在下面的例子中，“z” 中第一行的元素被初始化为指定的值，其余的元素被初始化为 0。

```
char    z [ 4 ] [ 3 ] = {
        { 1 } , { 2 } , { 3 } , { 4 }
} ;
```

在下一个面的示例中，一个三维数组被初始化。

q[0][0][0]初始化为 1，q[1][0][0]初始化为 2，q[1][0][1]初始化为 3。q[2][0][0]，q[2][0][1]和 q[2][1][0]分别被初始化为 4， 5， 6。其余的元素均被初始化为 0。

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        { 1 } ,
        { 2 , 3 } ,
        { 4 , 5 , 6 }
} ;
```

下面的例子如上面的三维数组初始化产生相同的结果。

```
short q [ 4 ] [ 3 ] [ 2 ] = {
        1 , 0 , 0 , 0 , 0 , 0 ,
        2 , 3 , 0 , 0 , 0 , 0 ,
        4 , 5 , 6
} ;
```

下面的例子使用花括号显示了上面初始化的完整的定义。

```
short   q [ 4 ] [ 3 ] [ 2 ] = {
        {
                { 1 } ,
        } ,
        {
                { 2 , 3 } ,
        } ,
        {
                { 4 , 5 , 6 } ,
        }
} ;
```

第 4 章　类型转换

如果在一个表达式中参与运算的两个运算数的类型不同，则编译器将自动执行类型转换运算。这种类型转换与使用类型转换运算符得到的结果类似。这种自动类型转换叫做隐式类型转换。本章将详细介绍这种隐式类型转换。

类型转换运算包括通常见的算术转换、涉及截断 / 舍入的转换以及涉及符号变化的转换。表 4-1 给出类型转换的列表。

表 4-1 类型间转换列表

转换前		转换后										
		(有符号) 字符	无符号字符型	(有符号) 短整型	无符号短整型	(有符号) 整型	无符号整型	(有符号) 长整型	无符号长整型	浮点型	双精度型	长精度型
(有符号) 字符	+	\	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
	-	\	NG	OK	NG	OK	NG	OK	NG	OK	OK	OK
无符号字符型		Δ	\	OK	OK	OK	OK	OK	OK	OK	OK	OK
(有符号) 短整型	+			\	OK	\	OK	OK	OK	OK	OK	OK
	-			\	NG	\	NG	OK	NG	OK	OK	OK
无符号短整型				Δ	\	Δ	\	OK	OK	OK	OK	OK
(有符号) 整型	+			\	OK	\	OK	OK	OK	OK	OK	OK
	-			\	NG	\	NG	OK	NG	OK	OK	OK
无符号整型				Δ	\	Δ	\	OK	OK	OK	OK	OK
(有符号) 长整型	+							\	OK	OK	OK	OK
	-							\	NG	OK	OK	OK
无符号长整型								Δ	\	OK	OK	OK
浮点型										\	OK	OK
双精度型											\	\
长精度型											\	\

备注 1. 关键字 signed 可以省略。但是，对于 char 型数据，根据编译时的条件 （选项），它将被视为 signed char 或 unsigned char 型。

备注 2. 约定

OK : 能正常执行类型转换。

\ : 不能执行类型转换。

NG : 不会产生正确的值。（数据类型将会被视为一个无符号 int 型数据。）

Δ : 数据类型不会以位映像图方式转换。但是，如果一个正数 不足以表示它，也不会产生一个正确的值。（被视为一个无符号 整数）

Blank : （见参考）

4.1 算术运算数

(1) 字符与整数 （一般整型提升）

如果 char、 short int 与 int 型位段的数据类型 （无论是有符号的还是无符号的）或者枚举型对象的数据类型的值的范围在 int 型可以表示的范围内，则这些数据类型将被转换为 int 型，前提是他们的值的范围在 int 型可以表示的范围内。如果不在其范围内，则它们将被转换为无符号整型。这种隐式的转换称为 “一般整型提升”。其他的算术类型都不会受此一般整型提升的影响，也不会进行转换。

一般整型提升将保留原始数据类型的值及符号。

在 CC78K0S 中，无类型修饰符的 char 型数据通常被将当作为 signed char 型数据处理。它还可以使用选项来作为 unsigned char 型数据处理。

(2) 有符号整数与无符号整数

当一个整型数据被转换为另一种数据时，如果其值可以用整型转换后的类型来表示，则该值将不会更改。

当一个有符号整数被转换为一个具有相同或更大长度的无符号整数时，其值将不改变，除非有符号整数的值是负数。如果有符号整数的值是负的，且无符号整数的长度大于有符号整数的长度，则有符号整数将被扩展，符号位的扩展将保证有符号整数具有与无符号整数相同的长度，然后为其加上与无符号整数可以表示的最大值加 1 相等的值，这样才完成有符号整数向无符号值的转换。

当一个整型值被转换为一个具有较小长度的无符号整型值时，转换结果是一个非负余数，且这个余数是其该整型值除以极限值得到的，极限值等于被转换后无符号整数所能表示的最大值加 1。当一个整型值被转换为一个具有较小长度的有符号整型值时，或者当一个无符号整数被转换为一个具有相同长度的有符号整数时，如果转换后的值无法表示出来，则溢出的值将被忽略。关于转换方式，请参阅 表 4-1.

从有符号整型向无符号整型之间的转换运算在下面表 4-2 列出。

表 4-2 从有符号整型向无符号整型之间的转换

		无符号	
		取值范围更小	取值范围更大
有符号	+	/	OK
	-	/	+

OK： 能正常执行类型转换。

+： 数据将被转换为一个正数。

/： 转换结果是该整数值取模得到的余数，原整数值除以 （被转换的类型所能表示的最大值加 1）。

(3) 常见算术类型转换

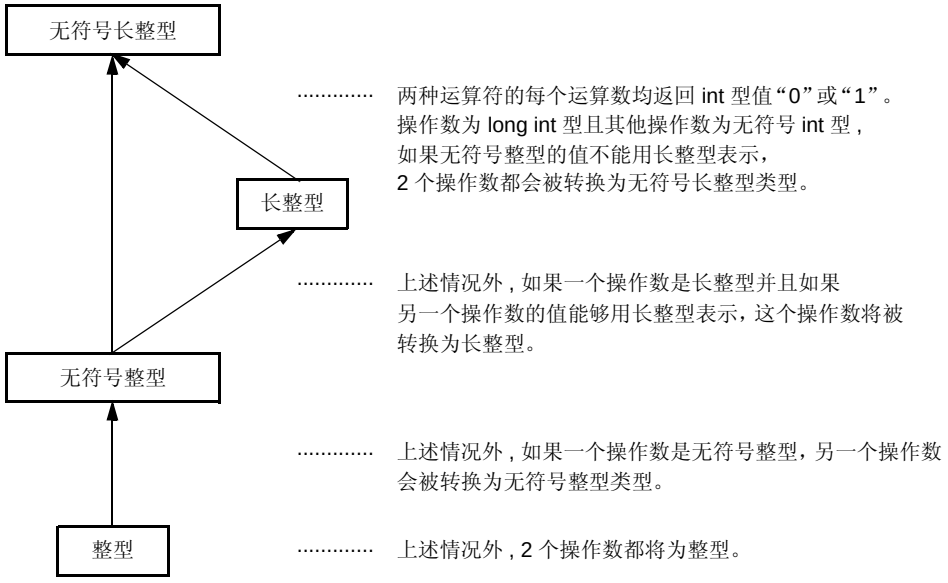
对算术型数据进行运算得到类型可以有各种值。

运算结果的类型转换方法介绍如下。

- 如果运算数中有任一个是长精度型，则另一个将被转换为长精度型。
- 如果运算数中有任一个是双精度型，则另一个将被转换为双精度型。
- 如果运算数中有任一个是浮点型，则另一个将被转换为浮点型。

对于其他情况，将根据以下规则对两个运算数进行通用整数扩展。图 4-1 给出了这些规则。

图 4-1 常见算术类型转换



在 CC78K0S 中，可以使用编译条件 （优化选项）禁止向 int 型的转换 （详细情况，请参阅 CC78K0S 编译器操作篇）。

4.2 其他运算数

(1) 左侧值和函数定位符

左值是指指定一个对象的表达式 （是除有对象类型或 void 型以外的不完全类型）。

不包括非数组类型、不完全类型或 const 修饰符类型的左值，以及没有加 const 修饰符类型成员的结构体或共用体都是 “可更改的左值”。

不包括数组类型的左值将被转换为一个存储在待指定的对象中的值，除非该值是 sizeof 运算符、单目 & 运算符、 ++ 运算符或 - - 运算符的一个运算数，或一个运算符的左运算数或赋值运算符的运算数。通过转换，它将不再是一个左值。

具有不完全类型而非数组类型的左值的行为将得不到保证。

具有除字符数组以外的 “数组”类型的左值将被转换为一个具有 “指向 ... 的指针”类型的表达式。这种表达式将不再是一个左值。

函数定位符是一个具有函数类型的表达式。除了 sizeof 运算符或单目 & 运算符的运算数之外，具有 “返回 ... 的函数型”的函数定位符将被转换为一个 “指向返回的函数的指针型”的表达式。

(2) void

void 表达式 （即具有 void 类 型的表达式）的值 （不存在）不能以任何方式使用。无论隐式还是显式的排除 void 的转换均都不能应用于该表达式。如果另一种类型的表达式出现在需要 void 表达式的地方，则表达式或声明符的值将被假定为不存在。

(3) 指针

空指针可以被转换为指向任何不完全类型的指针或对象类型的指针。反过来，一个指向任何不完全类型或对象类型的指针也可以被转换为空类型指针。在任意一种情况下，结果值都必须等于原始指针的值。

一个值为 0 且被转换为 void * 型的整型常量表达式被称为 “空指针常量”如果空指针常量用另一种指针来替代，或赋值，或与空指针常量之比较，则空指针常量将被转换为该指针类型。

第 5 章　表达式和运算符

本章介绍 C 语言中使用的运算符和表达式。

C 语言支持具有大量用于算术、逻辑和其他运算的运算符。其丰富的运算符集合还包括那些用于位运算和地址运算的运算符。

表达式是一个运算符和一个或多个运算数组成的字符串，或者说是其组合。运算符定义对运算数执行的操作，比如计算一个值，操作对象或调用函数的指令，同时产生副作用或这些操作的组合。

下面给出了运算符示例。

```
#define TRUE  1
#define FALSE 0
#define SIZE  200

void  lprintf ( char * , int ) ;
void    putchar ( char c ) ;
char  mark [ SIZE + 1 ] ; ——————————————— +       /* 算术运算符 */

void main ( void ) {
      int     i , prime , k , count ;
      count = 0 ; ————————————————————————— =       /* 算术运算符 */
      for ( i = 0 ; i <= SIZE ; i++ ) ———— ++      /* 后缀运算符 */
            mark [ i ] = TRUE ; ————————— <=      /* 比较运算符 */

      for ( i = 0 ; i <= SIZE ; i++ ) {
            if ( mark [ i ] ) {      ———————
                prime = i + i + 3 ;         +       /* 算术运算符 */
                lprintf ( " %d " , prime ) ;
                count++ ;                  ____ ++      /* 后缀运算符 */
                if ( ( count%8 ) == 0 )     ==      /* 比较运算符 */
                    putchar ( ' \n ' ) ;                    ____
                for ( k = i + prime ; k <= SIZE ; k += prime )    +=
                                                    /* 赋值运算符 */
                    mark [ k ] = FALSE ;
                }
        }

      lprintf ( " Total %d\n " , count ) ;
loop1 : ;
      goto    loop1 ;
}

lprintf ( char *s , int i ) {
      int     j ;
      char    *ss ;
      j = i ;
      ss = s ;
}

void  putchar ( char c )  {
      char    d ;
      d = c ;
```

表 5-1 给出了 C 中使用的运算符的计算优先级。

表 5-1 评价操作符的优先级

表达式类型	运算符	链接时[注]	优先
后缀表达式	[ ] ( ) . -> ++ --	-->	最高
单目表达式	++ -- & * + - ~ ! sizeof	<--	
类型转换表达式	(type)	<--	
乘法表达式	* / %	-->	
加法表达式	+ -	-->	
按位移位表达式	<< >>	-->	
关系表达式	< > <= >=	-->	
相等表达式	== !=	-->	
按位与表达式	&	-->	
按位异或表达式	^	-->	
按位或表达式	|	-->	
逻辑与表达式	&&	-->	
逻辑或表达式	||	-->	
条件表达式	? :	<--	
赋值表达式	= *= /= %= += -= <<= >>= &= ^= |=	<--	
逗号表达式	,	-->	最低

注 同一行内的操作运算符具有相同的优先级。 结合方向列中的箭头（<-- 或 -->）表示，当一个表达式包含两个或多个具有相同优先级的运算符时，则运算按箭头 "-->"（从左向右）或 "<--"（从右向左）指示的方向进行组合然后运算。

5.1 基本表达式

基本表达式包括以下几种：

- 声明对象或函数的标识符
  （标识符基本表达式）
- 常量 （常量基本表达式）
- 字符串文字 （常量基本表达式）
- 括在圆括号中的表达式
  （括号表达式）

如果声明了对象，则成为基本表达式的标识符是表达式的左侧值；如果声明了函数，则成为基本表达式的标识符是函数定位符。正如在“2.5 常量”中介绍的那样，常量的数据类型取决于为该常量指定的值。文字字符串 (s) 成为左侧值，具有数据类型在 "2.6 字符串 " 介绍。

5.2 后缀运算符

后缀运算符是出现在对象或函数的后面的运算符。

后缀运算符的类型如下所述。

- 下标运算符
- 函数调用运算符
- 结构体与共用体成员 (. ->)
- 后缀自增 / 自减运算符 (++ --)

5.2.1 下标运算符

语法

```
后缀表达式 [ 下标表达式 ]
```

功能

- 下标说明符 [ ] 指定或引用一个数组对象的某个元素。对数组或表达式 “E1 [E2]”的评估求解就是把它当成 “(*(E1+(E2)))”来进行的。换句话说， E1 的值是指向数组的第一个元素的指针， E2 （假设它是整数）则指示 E1 （从 0 开始计数）的第 E2 个元素。对于多维数组，下标运算符的数量必须与维数相等。
  在下面的例子中，x 是一个 3*5 的 int 型数组。换句话说，x 是一个具有三个成员的数组，每个成员由五个 int 型元素组成。

```
int     x       [ 3 ] [ 5 ] ;
```

  可以通过连接下标运算符来指定一个多维数组。假设 E 是一个由 i*j*...*k 组成的 n 维数组 （其中 n≥2），则可以使用 n 个下标运算符来指定该数组。在这种情况下， E 成为一个指向由 j*...*k 组成的 （n-1）维数组的指针。

注

- 后缀表达式必须有一个 “... 指针指向对象” } 数组的下标表达式必须使用整型数据来指定。表达式的结果将变成 “.....”

5.2.2 函数调用运算符

语法

```
后缀表达式 （参数表达式列表）；
```

功能

- 后缀运算符 “（）”用于调用一个函数。待调用的函数用后缀表达式来指定，传递给函数的参数在括号 （）中指定。
- 与该函数相关的描述包括函数原型声明、函数定义 （函数体）及函数调用。函数原型声明指定函数的返回的值、函数的参数类型及存储类型。
- 如果在函数调用中没有引用函数原型声明，则各个参数将使用一个通用整数来扩展。这个称为 “默认的实参数扩展”。执行函数原型声明可以避免了默认实参扩展，并能检测类型错误、参数数量是否匹配及返回值的类型。
- 如果调用一个的既无存储类说明又无数据类型说明的函数，如 “标识符 （）；”，则将被解释为调用一个具有外部对象的函数，并返回一个没有参数信息的 int 型值。换句话说，将隐含地进行以下的声明。

```
extern  int     identifier ( ) ;
```

[函数调用示例]

```
int     func ( char , int ) ;           /* 函数原型声明 */
char    a ;
int     b , ret ;
void main ( void ) {
        ret = func ( a , b ) ;          /* 函数调用 */
}
int     func ( char c , int i )  {      /* 函数定义 */
            :
        return  i ;
}
```

注

- 可以使用该运算符，调用的函数返回值必须是非数组型。后缀表达式必须有一个指向该函数的指针类型。
- 在包括原型的函数调用时，调用的参数的类型必须是能够兼容定义中给对应的参数的类型。参数的数量也必须一致。

5.2.3 结构体与共用体成员 (. ->)

(1) . （点）运算符

语法

后缀表达式 . 标识符

功能

- "."(点) 运算符 (也成为成员运算符) 指定结构体或联合体的单个成员。后缀表达式是指定的结构体或共用体的名称，标识符是成员的名称。

(2) -> （箭头）运算符

语法

后缀表达式 -> 标识符

功能

- “->”（箭头）运算符 （也称为间接成员运算符）指定一个结构体或共用体的某个单个成员。后缀表达式是指向特指定的结构体或共用体的指针的名称，标识符是成员的名称。

< “.”，“->”运算符举例 >

```
#include <stdlib.h>

union {
        struct  {
                int     type ;
        } n ;
        struct  {
                int     type ;
                int     intnode ;
        } ni ;
        struct  {
                int     type ;
                struct  {
                        long    longnode ;
                } *nl_p ;
        } nl ;
} u ;

void    func ( void )  {
        u.nl.type = 1 ;
        u.nl.nl_p -> longnode = -31415L ;
        /* ... */
        if ( u.n.type == 1 )
                u.nl.nl_p -> longnode = labs ( u.nl.nl_p -> longnode ) ;
}
```

5.2.4 后缀自增 / 自减运算符 (++ --)

(1) 后缀 ++ （自增）运算符

语法

```
后缀表达式 ++
```

功能

- 后缀自加运算符将对象的值增加 1。这种自加运算在执行时会根据对象的数据类型而自动调整考虑到了对象的数据类型。

(2) 后缀 -- （自减）运算符

语法

```
后缀表达式 - -
```

功能

- 后缀自减运算符将对象的值减去 1。这种自减运算在执行时会根据对象的数据类型而自动调整考虑到了对象的数据类型。

注

- 后缀自增或自减运算符的运算数必须是一个可更改的左值 （说明的或未说隐含明的）。

5.3 单目运算符

单目运算符执行会对一个对象或参数（即运算数）的进行运算。支持 有以下的单目运算符有以下几种可供使用：

- 前缀自增 / 自减运算符 (++ --)
- 地址和重定向运算符 (& *)
- 单目算术运算符 （+ ñ ~ !）
- sizeof 运算符

以下介绍各一元运算符。

5.3.1 前缀自增 / 自减运算符 (++ --)

(1) 前缀 ++ （自增）运算符

语法

```
++ 前缀表达式
```

功能

- 前缀自增运算符会将对象的值增加 1。 前缀自增运算符的表达式 “++E”将产生的效果与以下表达式相同的结果。

```
E = E + 1
    或者
E += 1
```

(2) 前缀 -- （自减）运算符

语法

```
-- 前缀表达式
```

功能

- 前缀自减运算符会将对象的值减去 1。 前缀自减运算符的表达式 “--E”将产生的效果与以下表达式相同的结果。

```
E = E - 1
   或者
E -= 1
```

5.3.2 地址和重定向运算符 (& *)

(1) 单目 & 运算符

语法

```
& 运算数
```

功能

- 单目 & (address) 运算符返回一个指定的对象的指针 （即，其后所描述的变量的地址）。

(2) 单目 * 运算符

语法

```
* 运算数
```

功能

- 单目 * (取值符号) 运算符返回指特定的指针所指示的值 （即，取其后描述的变量的实际值，并将该值用作内存中信息的地址）。

注

- 单目 & 运算符的运算数必须是左值，该左值所引用的对象不支持未使用寄存器存储类说明符进行声明的对象。函数定位符及位段域均不能用作该单目运算符的运算数。

  单目 * 运算符的运算数必须有一个指针类型。

5.3.3 单目算术运算符 （+ ñ ~ !）

功能

- + （单目加）运算符对其运算数执行正整型提升。
- 袜û ゾ考酰╥忿惴　云溢忿闶　葱懈赫　吞嵘　£
- The ~ (tilde) operator is a bitwise one 担 complement operator which inverts all the bits in a byte of its operand.
- !NOT 或逻辑非运算符返回 "0" 如果其操作数为 "0" and "1" 如果它不为 "0"。换句话说，运算符改变每个 "0" 为 "1" 以及 "1" 为 "0"。

语法

```
+ 运算数
- 运算数
~ 运算数
! 运算数
```

5.3.4 sizeof 运算符

语法

```
sizeof 单目表达式
sizeof （类型名）
```

功能

- sizeof 运算符以字节为单位返回一个指定的对象所占的大小。返回值由对象的数据类型控制，但不计算对象本身的值和此返回值无关。
- 执行了 sizeof 运算的 unsigned char 或 signed char 对象（包括其限定的类型）返回的值是 1。对于一个数组型对象，返回值是数组中字节的字节总数。对于一个结构体或共用体对象，结果返回值是对象占有的字节的总数，其中包括填充邻接对象之间的适当对齐边界所需要的字节。
- sizeof 运算结果的类型是整型，其名称是 size_t。该名称在 < stddef.h > 头文件中有定义。sizeof 运算符主要用于分配内存单元，以及与 I/O 系统之间传输数据。

示例

- 下面的示例通过用一个元素的大小除数组中的总字节数来求得一个数组中包含的元素数。结果是 5。

```
int     num ;
char    array [ ] = { 0 , 1 , 2 , 3 , 4 } ;

void    func ( void )  {
        num = sizeof array / sizeof array [ 0 ] ;
}
```

注

- 如果表达式具有函数类型或不完全类型，并且左值及引用位段域对象的左值的表达式，则该表达式不能用作该操作数符的运算数。

5.4 Cast 运算符

类型转换运算符是一个特殊的运算符，它强制将一个数据从某种类型转换为另一个种数据类型。类型转换运算符主要用于转换指针类型。

- Cast 运算符 (类型名)

5.4.1 Cast 运算符 (类型名)

语法

```
( 类型名 )       表达式
```

功能

- 类型转换运算符将另一个对象 （或另一个表达式的结果）的数据类型转换为括号 （）中指定的指定类型。

示例

```
void    func ( void )  {
        int     val ;
        float   f ;

        f = 3.14F ;
        val = ( int ) f ;               /* 通过转换 val 变为 3 */
        val = * ( int * ) 0x10000 ;     /* 传递常量 */
}
```

5.5 算术运算符

- 乘性乘法运算符 (* / %)
- 加法运算符 (+ -)

算数运算符分为乘除运算符和加减运算符。乘性乘法运算符可以求两个运算数的乘积、商及余数。加性加法运算符求两个运算数的和与差。

表 5-2 除法的符号 / 余数除法运算结果

a / b		b	
		+	-
a	+	+	-
	-	-	+

a % b		b	
		+	-
a	+	+	+
	-	-	-

备注 a, b 表示各操作数。

除法是对两个通过正常的算术转换对两个去掉符号 （如果有）的整数进行的，必要时，结果会尽量向 0 的方向进行靠拢截取。同样，使用 2 个带符号整数进行余数或取模运算，如果没有，通过常用算数转换来移除。表 5-2 分别显示有关 2 个操作数除法和余数除法的计算结果。

5.5.1 乘性乘法运算符 (* / %)

(1) * 运算符

语法

```
E1 * E2
```

功能

- * 运算符对两个运算数执行正通常的乘法操作，并返回乘积。

(2) / 运算符

语法

```
E1 / E2
```

功能

- / 运算符对两个运算数执行通常的除法，并返回商。

(3) % 运算符

语法

```
E1 % E2
```

功能

- % 运算符对两个运算数执行求余 （或取模除）运算，并返回结果中的余数。

5.5.2 加法运算符 (+ -)

(1) + 运算符

语法

```
E1 + E2
```

功能

- + 运算符对两个运算数执行加法运算，并返回两个数的和。

(2) - 运算符

语法

```
E1 - E2
```

功能

- ñ 运算符对两个运算数执行减法运算，并返回两个数的差 （第一个运算数减去第二个运算数）。

5.6 按位移动运算符

移位运算符将第一个 (左侧) 运算数按运算符指定的方向 (向左或向右) 移动第二个运算数指定位数 , 移动多少次由第二个运算数决定。移动运算符的类型如下所述。

- 移位运算符 (<< >>)

Table 5-3 移动运算符

a << b		b 注
a	+	0
	-	0

a >> b		b 注
a	+	0
	-	-1

注 该表说明了当右侧运算操作数大于左侧运算数中的位数，或当移位运算时发生溢出时的情况。如果右侧运算数时一个负数，则其值将被当作一个无符号正数处理。

备注 a，b 表示各操作数。

下面详细介绍各个移位运算符。 E1 和 E2 表示运算数或表达式。

5.6.1 移位运算符 (<< >>)

(1) 左移 (<<) 运算符

功能

- 二进制 << (左移) 运算符将左侧运算数向左移动，右侧运算数指定移动的位数，并将空出的位置填 0。如果在 “E1<<E2”中左侧运算数 E1 是有一个无符号类型，则结果的值就是将变为 E1 乘以 2 的 E2 次幂得到的值。

语法

```
E1 << E2
```

(2) 右移 (>>) 运算符

功能

- 二进制 >> （右移）运算符将左侧运算数向右移动，右侧运算数指定移动的位数。
- 如果 “E1”是无符号型的，则空出的位将被填充 0 （逻辑移位）。
- 如果 “E1”是有符号型的，则将符号位填充到移动后空出的位中。
- 如果在 “E1>>E2”中左侧运算数 “E1”是无符号型，或 “E1”是有符号型的且是一个非负值，则结果的值就是将变为 “E1”除以 2 的 “E2”次幂得到的值。

语法

```
E1 >> E2
```

5.7 比较运算符

有两种运算符可以表示两个运算数之间的关系：“关系运算符”和 “等性运算符等式运算符”。

比较运算符表示两个运算数之间的值的关系，如大于、小于。等性运算符等式运算符表示两个运算数是否相等。

比较运算符和等性运算符等式运算符如下所示。

- 比较运算符 (< > <= >=)
- 等式运算符 (== !=)

比较运算符所比较的两个指针之间的值的关系大小，由指针指示的对象在地址空间中的相对位置来决定。

在 CC78K0S 中，如果指定的关系是真，则关系运算符和等性运算符等式运算符产生一个？攡；如果为假，则产生一个 “0” 。得到的结果是 int 型的。

5.7.1 比较运算符 (< > <= >=)

(1) < （小于）运算符

语法

```
E1 < E2
```

功能

- 如果左操作数小于右操作数， （小于）操作数返回 1 ； 否则，返回 0。 <

(2) > （大于）运算符

语法

```
E1 > E2
```

功能

- 如果左操作数大于右操作数， （大于）操作数返回 1 ； 否则，返回 0。 >

(3) <= （小于或等于）运算符

语法

```
E1 <= E2
```

功能

- 如果左操作数小于或等于右操作数， （小于或等于）操作数返回 1 ； 否则，返回 0。 <

(4) >= （大于或等于）运算符

语法

```
E1 >= E2
```

功能

- 如果左操作数大于或等于右操作数， （大于或等于）操作数返回 1 ； 否则，返回 0。 >

5.7.2 等式运算符 (== !=)

(1) == （等于）运算符

功能

- 如果两个运算数相等，== （等于）运算符返回 1 ； 否则，返回 0。

语法

```
E1 == E2
```

(2) != （不等于）运算符

功能

- 如果两个运算数不相等，！ =（不等于）运算符返回 1；否则，返回 0。

语法

```
E1 != E2
```

5.8 按位逻辑运算符

按位逻辑运算符以位为单位，对对象的值会被执行指定的逻辑运算。按位逻辑表达式包括按位与 （&）、按位异或 （^）及按位或 （ | ）。

每个逻辑运算由下面的运算符来说明。

- 按位与运算符 (&)
- 按位异或运算符 (^)
- 按位或运算符 (|)

5.8.1 按位与运算符 (&)

语法

```
E1 & E2
```

功能

- & 运算符是一个按位与运算符，它返回一个整数值，该整数值在两个运算数均为 “1” 的位置上的值为 “1”，在其他位置上的值均为 “0”。
- 按位与运算符必须使用 “& 运算符” 来说明。

表 5-4 按位与运算符

		左运算数各位的值	
		1	0
右运算数各位的值	1	1	0
	0	0	0

5.8.2 按位异或运算符 (^)

语法

```
E1 ^ E2
```

功能

- ^（脱字符插入记号）运算符是一个按位异或运算符，它返回一个整数值，该整数值在两个运算数仅有一个为“1 的位置上的值为“1”，在两个运算数均为“1”或均为“0”的位置上的值为得到 0。

表 5-5 按位异或运算符

		左运算数各位的值	
		1	0
右运算数各位的值	1	0	1
	0	1	0

5.8.3 按位或运算符 (|)

语法

```
E1 | E2
```

功能

- | （运算符是一个按位或运算符，它返回一个整数值，该整数值在两个运算数至少有一个为 “1”的位置上的值为 “1”，在两个运算数均为 “0”的位置上的值为得到 0。

表 5-6 按位或运算符

		左运算数各位的值	
		1	0
右运算数各位的值	1	1	1
	0	1	0

5.9 逻辑运算符

逻辑运算符执行逻辑或及逻辑与运算。逻辑或运算用一个逻辑或运算符 ?|| 来说明，逻辑与运算用一个逻辑与运算“&&”来说。各个运算符介绍如下。

- 逻辑与运算符 (&&)
- 逻辑或运算符 (||)

两种运算符的每个运算数均返回 int 型值 “0”或 “1”。

5.9.1 逻辑与运算符 (&&)

语法

```
E1 && E2
```

功能

- && 运算符在 2 个操作数上进行逻辑与运算，且如果两个操作数有非零值则返回 “1”。否则将返回 “0”。返回结果的类型是 int 型。

表 5-7 逻辑与运算符

		左侧运算数左操作数的值	
		Zero	非 0
右侧运算数右操作数的值	Zero	0	0
	非 0	0	1

注

- 该运算符始终从左向右对运算数进行求解。如果左侧运算数左操作数的值为 0，则右侧的运算数将不再进行计算。

5.9.2 逻辑或运算符 (||)

语法

```
E1 || E2
```

功能

- || 运算符在 2 个操作数上进行逻辑或运算，且如果两个操作数为零时则返回 “0”。否则将返回 “1”。返回结果的类型是 int 型。

表 5-8 逻辑或运算符

		左运算数各位的值	
		Zero	非 0
右运算数各位的值	Zero	0	1
	非 0	1	1

注

- 该运算符始终从左向右对运算数进行求解。如果左侧运算数左操作数的值为非 0，则右侧的运算数将不再进行计算。

5.10 条件运算符

条件运算符根据第一个运算数的值判断下一步待将要执行的操作。条件运算符用 “?” 和 “:” 来进行判断。 条件运算符的类型如下所述：

- 条件运算符 (? :)

5.10.1 条件运算符 (? :)

语法

```
第一个运算数 ? 第二个运算数 : 第三个运算数
```

功能

- 如果第一个运算数的值非 0，则计算冒号前的第二个运算数。如果第一个运算数的值为 0，则计算冒号后的第三个运算数。整个条件表达式的值为第二个或第三个运算数的值。

示例

```
#define TRUE    1
#define FALSE   0
char    flag ;
int     ret ;
int     func ( ) {
        ret = flag ?TRUE : FALSE ;
        return  ret ;
}
```

注

- 如果第二个及第三个运算数类型均是算术类型，则执行常规的算术类型转换，使它们成为通用类型。转换结果的类型是通用类型。如果两种运算数类型均是结构体型或共用体型，则结果变为这两种类型。如果两种运算数类型均 void 型，则结果变为 void 型。

5.11 赋值运算符

赋值运算符包括将右操作数存储到左操作数中的简单赋值运算符，也包括将两个右运算数的运算结果存储到左操作数中的复合赋值运算符。

赋值运算符如下所示。

- 简单赋值运算符 (=)
- 复合赋值运算符 (*= /= %= += -= <<= >>= &= ^= |=)

5.11.1 简单赋值运算符 (=)

语法

```
E1 = E2
```

功能

- = （简单赋值）运算符将右操作数 （表达式）转换为左操作数 (Lvalue) 的类型，然后将其值存储到左侧对象中。

  在下面的例子中，从函数返回的 int 型值将通过简单赋值表达式的类型转换被转换为 char 型，结果中的溢出将被截断。当值被转换回 int 型后，将该值与 “-1”进行比较。如果变量 “c”在声明时未用加修饰符声明的，则不会认为 c 是 unsigned char 型，变量的结果将不会变为负值，且它与 “－ 1”的比较将永远不会产生相等的结果。在此情况下，变量 “c”必须用 int 型进行声明，以确保完全的可移植性。

```
int     f ( void ) ;

char    c ;
/* ... */   ( ( c = f ( ) ) == -1 )  /* ... */
```

5.11.2 复合赋值运算符 (*= /= %= += -= <<= >>= &= ^= |=)

语法

```
E1 *= E2
E1 /= E2
E1 %= E2
E1 += E2
E1 -= E2
E1 <<= E2
E1 >>= E2
E1 &= E2
E1 ^= E2
E1 |= E2
```

功能

- 复合赋值运算符对两个运算数执行指定的运算，并将结果保存到左操作数中。保存到左操作数中的值的类型将被转换为和左操作数的类型一致。
- 复合赋值表达式 “E1 op = E2”（其中 op 表示一个适当的二目运算符）等效于简单赋值表达式 “E1 = E1 op (E2)”，只是左侧的运算数 （E1）只计算了一次。 下面的复合赋值表达式将产生与右侧各自的简单赋值表达式相同的结果。

```
a *= b ;        a = a * b ;
a /= b ;        a = a / b ;
a %= b ;        a = a % b ;
a += b ;        a = a + b ;
a -= b ;        a = a - b ;
a <<= b ;       a = a << b ;
a >>= b ;       a = a >> b ;
a &= b ;        a = a & b ;
a ^= b ;        a = a ^ b ;
a |= b ;        a = a | b ;
```

5.12 逗号运算符

逗号运算符的类型如下所述。

- 逗号运算符 (,)

5.12.1 逗号运算符 (,)

语法

```
E1 , E2
```

功能

- 逗号运算符将左侧运算数作为 void 型来计算 （即忽略其值），然后计算右侧运算数。逗号表达式的结果的类型和值就是右侧运算数的类型和值。
- 如果逗号有其他含义 （如在函数参数列表或变量初始化列表中），则必须将逗号表达式括在括号中。换句话说，在本章中介绍的逗号运算符不会出现在这样的列表中。
- 在下面的例子中，逗号运算符求解函数 “f ()” 的第二个参数的值。第二个参数的值变为 5。

```
int     a , c , t ;
void    main ( void )  {
        f ( a , ( t = 3 , t + 2 ) , c ) ;
}
```

5.13 常量表达式

常量表达式包括一般的整型常量表达式、算术常量表达式、地址常量表达式和初始化常量表达式。这些常量表达式大多数可以在编译器翻译时进行计算，而不是执行时才计算。

以下的运算符不能用于常量表达式中，除非当它们出现在 sizeof 表达式中：

- 赋值运算符
- 自增运算符
- 自减运算符
- 函数调用运算符
- 逗号运算符

(1) 一般整型常量表达式

一般整型常量表达式的类型一般都是整型。可以使用以下的运算数：

- 整型常量
- 枚举型常量
- 字符型常量
- sizeof 表达式
- 浮点型常量

(2) 算术常量表达式

算术常量表达式为整型。可以使用以下的运算数：

- 整型常量
- 枚举型常量
- 字符型常量
- sizeof 表达式
- 浮点型常量

(3) 地址常量表达式

地址常量表达式是一个指针，指向具有静态存储生存期的对象，或指向一个函数定位符。这样的表达式必须使用一个单目 & 运算符显式地进行创建，或者使用一个数组型或函数型的表达式进行隐式地创建。可以使用以下的运算数。不过，这些运算符均都不能用来于访问一个对象的值。

- 数组下标运算符 “[ ]”
- "." （点）运算符
- "-> “（箭头）运算符
- “& ”地址运算符
- “*”间接运算符
- 指针类型转换

第 6 章　C 语言的控制结构

本章描述 C 语言的程序控制结构，以及要在 C 语言源程序中执行的语句。

一般说来，不管过程有多复杂，都可以分为三种基本控制结构。这三种控制结构是：顺序、选择和迭代。分支用于强制改变程序的流程。

(1) 顺序处理

根据在程序中的叙述顺序自顶部至底部逐条执行的。

(2) 条件控制 （选择）处理

根据程序执行的状态，选择并执行下一条可执行语句。选择条件在控制语句中指定。控制语句在两个备选分支语句中决定谁将获准执行，或多通道 （两个或以上）备选语句中也同样由控制语句决定执行哪一条语句。

(3) 循环 （迭代）处理

相同的处理过程被执行两次或更多次。当状态条件满足控制语句时，可执行语句会重复执行多次。

(4) 分支处理

当前程序流被强制中断，控制转移到指定标签处。程序从指定标签标号的下一条语句处开始执行。

在 C 语言当中可使用如下六类语句。

- 带标签语句 :　根据 switch 语句的值和或 goto 语句的目的地产生跳往一个分支
- 复合语句或块 :　把要处理的两个 （或更多）语句合为一个单元
- 表达式语句和空语句 :　由表达式和分号组成的语句
- 条件控制语句 :　根据表达式的值从几个备选语句中选择选出一个符合条件的
- 循环语句 :　反复执行调用循环体中的语句，直到控制表达式的值等于 0。
- 分支语句 :　无条件跳转到另外的目的地

如下所示是这些语句的一个说明示例。

[说明示例]

```
#define SIZE    10
#define TRUE    1
#define FALSE   0

extern  void    putchar ( char ) ;
extern  void    lprintf ( char * , int ) ;

charmark [ SIZE + 1 ] ;
void    main ( void )  {
        int     i , prime , k , count ;

    count = 0 ;
    for ( i = 0 ; i <= SIZE ; i++ )          /* 循环语句 */
        mark [ i ] = TRUE ;
    for ( i = 0 ; i <= SIZE ; i++ )  {       /* 循环语句 */
        if ( mark [ i ] )  {                 /* if ：条件语句 */
            prime = i + i + 3 ;
            lprintf ( " %d " , prime ) ;
            if ( ( count%8 ) == 0 )          /* if : 条件语句 */
                putchar ( ' \n ' ) ;
            for ( k = i + prime ; k <= SIZE ; k += prime )
                mark [ k ] = FALSE ;
        }
    }
    lprintf ( " Total %d\n " , count ) ;

loop1: ;                                     /* loop1 : 带标签语句 */
    goto    loop1 ;                          /* goto : 分支语句 */
}
```

6.1 带标签语句

带标签语句指定 switc 带标号带标签语句变成 switch 语句中包含的执行语句的标签。 goto 语句会从正常处理流程无条件跳转到相关标签。

带标签语句的语法如下所述。

- case 标签
- default 标签

6.1.1 case 标签

语法

```
case 常量表达式 : 语句
```

功能

- case 标签只能在 switch 语句的内部使用，用来枚举 switch 语句的控制表达式可能值。

例 1

```
int     f ( void ) , i ;
void    main ( void )  {
        /* ... */
        switch ( f ( ) ) {
                case 1 :
                        i = i + 4 ;
                        break ;
                case 2 :
                        i = i + 3 ;
                        break ;
                case 3 :
                        i = i + 2 ;
        }
        /* ... */
}
```

说明

- 在例 1 中，如果 f( ) 的返回值为 1，那么就会选择第一个 case 分支 （语句）并且执行表达式 “i=i+4”同样的，如果 f( ) 的返回值为 2 或 3，那么就会分别选择第二或第三条 case 语句。上面例子中的各 break 语句可以用来退出 switch 语句。

  本例中，在包含两个 （或更多）选项时使用了 case 标签。

例 2

```
int     i ;
void main ( void ) {
        /* ... */
        i = 2 ;
        switch ( i ) {
               case 1 :
                       i = i + 4 ;
               case 2 :
                       i = i + 3 ;
               case 3 :
                       i = i + 2 ;
        }
        /* ... */
}
```

说明

- 在例 2 中，由于 i 的值等于 2，所以在第二条 case 语句处开始执行。因为 case 语句中没有包含 break 语句，所以继续执行第三条语句。因此，如果 case 语句中的常量表达式和控制表达式匹配，那么随后的程序会顺序执行。 break 语句用来从 switch 语句退出。

6.1.2 default 标签

语法

```
default: 语句
```

功能

- default 标签是仅用于在 switch 语句内部使用的特殊情况标签。在控制表达式的值和所有 case 常量都不匹配时， default 标签指定 C 语言源程序要执行的处理内容。

示例

```
int     f ( void ) , i ;

switch ( f ( ) ) {
        case 1 :
                i = i + 4 ;
                break ;
        case 2 :
                i = i + 3 ;
                break ;
        case 3 :
                i = i + 2 ;
        default :
                i = 1 ;
}
```

说明

- 在上面的例子中，如果 f( ) 的返回值为 1、 2 或 3，那么就会选择相应的 case 分支（语句），执行 case 标签之后的表达式。上面例子中的各 break 语句用来退出 switch 语句。如果 f( ) 的返回值不属于 1 到 3，那么就会执行 default 标签之后的表达式。在这种情况下， i 的值变为 1。

6.2 复合语句或块

复合语句包含两个或更多的语句群，语句群都位于花括号内，在语法上作为一个执行单元。换句话说，只要在花括号中写入零个或更多声明，那么在程序需要执行单个语句时，花括号内的这些语句可以在需要单个居于出现的位置作为一个复合语句进行处理。

6.3 表达式语句和空语句

表达式语句由一个表达语句和分号。空语句只包含分号，在需要有但不需要任何具体内容的情况下，或者空循环中用作标签。

下面是表达式语句和空语句的说明示例。

在下面的例子中，作为以表达式语句形式被调用的函数仅仅用来演示对应的过程效果，其返回值的值可以用 cast 表达式去除。

```
int     p ( int ) ;
void    main ( void )  {
        /* ... */
        ( void ) p ( 0 ) ;
}
```

空语句可以当作循环语句的循环体使用，如下所示。

```
char    *s ;
void main ( void ) {
        /* ... */
        while ( *s++ != ' 0 ' ) ;
        /* ... */
}
```

此外，它还可以用来在结束复合语句结束处的花括号 “}”前放置标签，如下所示。

```
void    func ( void )  {
        /* ... */
        while ( loop1 )  {
                /* ... */
                while ( loop2 )  {
                        /* ... */
                        if ( want_out )
                                goto    end_loop1 ;
                        /* ... */
                }
end_loop1 : ;
        }
}
```

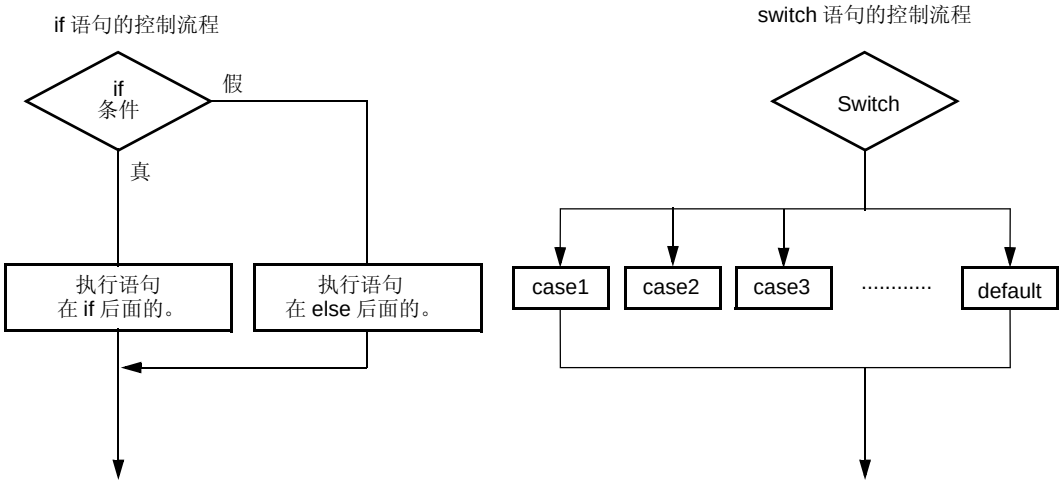
6.4 条件控制语句

条件控制（选择）语句包括 if 和 switch 语句。根据括号中的控制表达式的值， if 或 switch 语句允许程序能够在几组语句中选择其中一组执行。条件控制语句的类型如下所述。

- if 和 if ... else 语句
- switch 语句

if 和 switch 语句的控制流程在如下图 6-1 中被描述。

图 6-1 条件控制语句的控制流程



6.4.1 if 和 if ... else 语句

语法

```
if （表达式）语句
if （表达式）语句 -1 else 语句 -2
```

功能

- 如果控制表达式的值非零，那么 if 语句会执行紧随在控制表达式后面的括号内的语句。
- 如果控制表达式的值非零，那么 if ...else 语句就执行紧随在控制表达式后面的语句 -1 ；如果控制表达式的值为零，就执行 else 后面的语句 -2。

示例

```
unsigned char   uc ;
void    func ( void )  {
        if ( uc < 10 ) {
                /* 111 */
        } else {
                /* 222 */
        }
}
```

说明

- 在上面的例子中，根据 if 语句中的控制表达式，如果 uc 的值小于 10，那么就会执行 "{/*111*/}" 块中的内容，如果其值大于 10，就会执行 “{/* 222 */}”块中的内容。

注

- 当 if 语句 /if...else 语句后面的处理内容没有被放括在 “{ }”中时，只会把 if 语句 /if...else 语句后面的第一行作为执行体。

6.4.2 switch 语句

语法

```
switch （表达式）语句
```

功能

- switch 语句具有多路分支结构，根据括号中的控制表达式的值把控制权交给 switch 语句体中多个具有 case 标签的其中一组语句，要求这组语句的 case 标签值等于控制表达式的值。如果不存在控制表达式值的对应 case 标签，那么就会 default 执行标签后面的语句。如果所有 case 标签都不符合控制表达式的值，也不存在 default 标号 default 标签，那么就不会执行任何语句。

示例

```
extern  void    func ( void ) ;
unsigned char   mode ;
void main ( void ) {
        switch ( mode ) {
                case 2 :
                        mode = 8 ;
                        break ;
                case 4 :
                        mode = 2 ;
                        break ;
                case 8 :
                        func();
        }
}
```

注

- switch 语句中的各个 case 标签不能设置为相同的值。 switch 语句中只能有一个 default 标签。

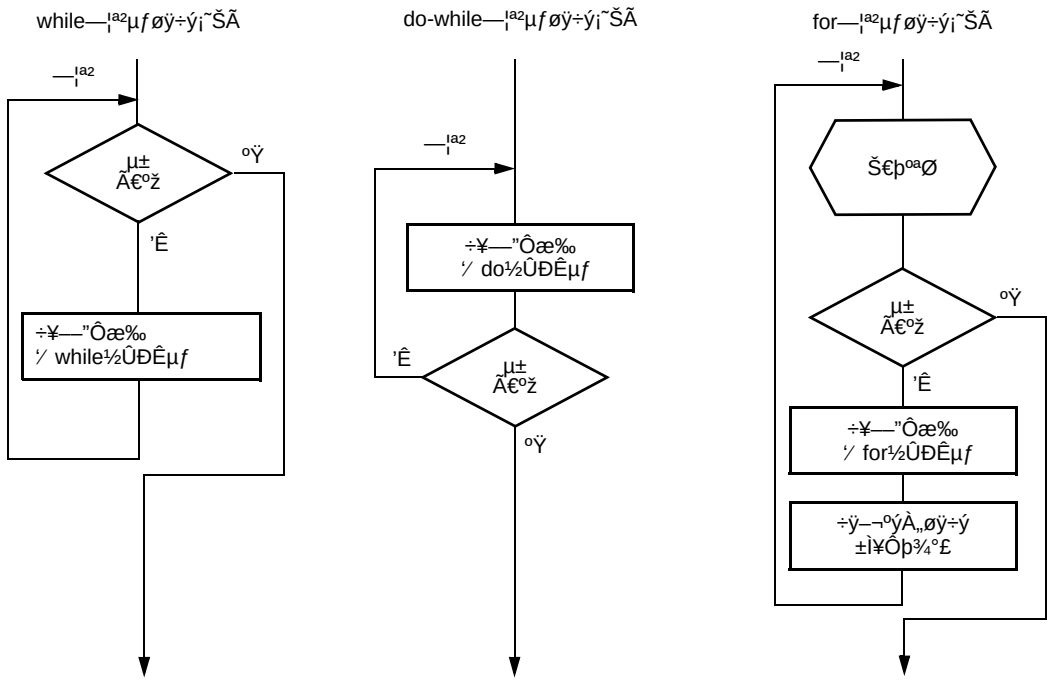
6.5 循环语句

只要括号中的控制表达式的值为真（非零），循环（迭代）语句就会反复执行循环体中的一组语句。C 语言中有包含下列三种类型的迭代语句：

- while 语句
- do 语句
- for 语句

各类循环语句的控制流程在下面的图 6-2 中说明。

图 6-2 循环语句的控制流程



6.5.1 while 语句

语法

```
while（表达式）语句
```

功能

- 只要括号中的控制表达式的值为真（非零），while 语句就会反复多次执行一条或更多语句（while 循环体）。while 语句在执行其循环体之前先计算控制表达式的值。

示例

```
int     i , x ;
void main ( void ) {
        i = 1 , x = 0 ;

        while ( i < 11 )  {
                x += i ;
                i++ ;
        }
}
```

说明

- 上面的例子得到从 1 到 10 的整数之和，并赋给 x。花括号中的两个语句是这个 while 循环的循环体。如果 i 的值变为 11，控制表达式“i<11”将返回 0。因此 , 只要 i 的值小于 11（在 1 和 10 之间）循环体将会被反复执行。
- “while(1) {statement}”被用作不断执行的循环语句。

6.5.2 do 语句

语法

```
do 语句 while（表达式）;
```

功能

- do 语句先执行循环体，然后检查括号中的控制表达式，确定其值是否为真（非零）。do 语句在执行其循环体之后才计算控制表达式的值。

示例

```
int     i , x ;
void main ( void ) {
        i = 1 , x = 0 ;

                do  {
                        x += i ;
                        i++ ;
                } while ( i < 11 ) ;
}
```

说明

- 上面的例子得到从 1 到 10 的整数之和，并赋给 x。花括号中的两个语句是这个 do ... while 循环的循环体。如果 i 的值变为 11，控制表达式“i<11”将返回 0。因此，只要 i 的值小于 11（在 1 和 10 之间）就会重复执行循环体。由于 do 语句在执行之后才计算控制表达式的值，所以循环体至少会执行一次或更多次。

6.5.3 for 语句

语法

```
for （第一表达式； 第二表达式； 第三表达式）语句
```

功能

- 只要控制表达式的值非零 （真）， for 语句就会将执行 for 循环体内的语句执行若干次，执行的次数也是由 for 语句指定的。括号中用分号隔开的三个表达式中，第一个表达式是初始化语句，初始化对一某个用作计数器的变量进行初始化，此表达式只在循环开始时执行一次，； 第二个是用来检查计数值的控制表达式的计数值，； 第三个是在每次循环末尾执行的步进语句，执行之后会对变量进行重新计算。

示例

```
int     i , x = 0 ;

for ( i = 1 ; i < 11 ; ++i )
        x += i ;
```

说明

- 上面的例子得到从 1 到 10 的整数之和，并赋给 x。“ x+=i"”是这个 for 循环的循环体，如果 i 的值变为 11，控制表达式 “i<11”将返回 0。因此 , 只要 i 的值小于 11 （在 1 和 10 之间）循环体将会被反复执行。

注

- 当 for 语句后面的处理内容没有被括放在 “{ }”中时，只会把 for 语句后面的第一行当作为 for 语句的循环体执行。
- for 语句的第一和第三表达式可以省略。当省略第二表达式被省略时，它会被由一个非 0 常量替代。“for (; ;) statement”类型被用来无限执行循环体。

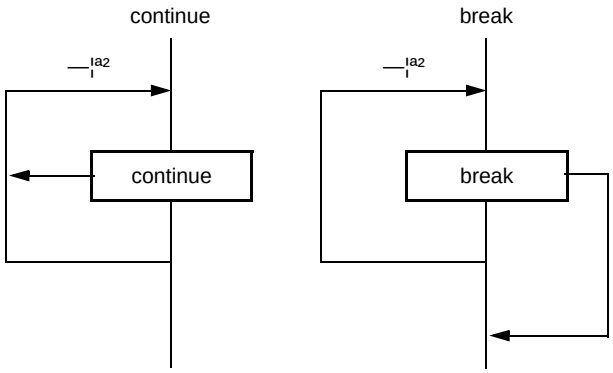
6.6 分支语句

分支语句用来从当前控制流流程中退出，并把控制权转移到程序的其它地方。分支语句共有包含下列四个种表达语句：

- goto 语句
- continue 语句
- break 语句
- return 语句

各类分支语句的控制流程如图 6-3 所示。

图 6-3 分支语句的控制流程



6.6.1 goto 语句

语法

```
goto 标识符 ;
```

功能

- goto 语句从当前函数中无条件跳转 goto 语句指定的标签名称处。

示例

```
do  {
        /* ... */
        goto    point ;
        /* ... */
} while ( /* ... */ ) ;
        /* ... */
point : ;
```

说明

- 在上面的例子中，当控制传程序执行到 goto 语句时， C 语言无条件地跳出对当前 do ... while 循环的处理，并把控制转到 point 后面的语句。

注

- 在 goto 语句指定的标签名称 （分支目的地）必须在包含该 goto 语句的当前函数中被指定过。换句话说，goto 只能跳转到当前函数中某个分支位置，而不能从一个函数跳转到另一个函数。

6.6.2 continue 语句

功能

- continue 语句用在循环语句的循环体中。continue 将控制转到循环体的末尾，从而结束一轮循环。当 continue 语句包含在不止一层的循环体中时，它会跳转到包含它的最小循环体的末尾。

语法

```
continue ;
```

示例

```
while ( /* ... */ ) {
        /* ... */
        continue ;
        /* ... */
contin : ;
}
```

说明

- 在上面的例子中，当 C 语言对 while 循环的执行到达 continue 语句时， C 语言会无条件跳转到 “contin”标签 “contin”标签指明了跳转目的地，是可以省略的。用 “goto contin”；也可以实现同样的跳转操作。

注

- continue 语句只能用在循环的循环体中。

6.6.3 break 语句

语法

```
break ;
```

功能

- break 语句可能出现在循环体内或 switch 语句主体内，会导致控制转移到循环体或 switch 语句之后面的语句位置处。

示例

```
int     i ;
unsigned char   count , flag ;

void main ( void ) {
        /* ... */
        for ( i = 0 ; i < 20 ; i++ )  {
                switch ( count )  {
                        case 10 :
                                flag = 1 ;
                                break ;          /* 退出 switch 语句 */
                        default :
                                func();
                }
                if ( flag )
                        break ;                  /* 退出 for 语句 */
        }
}
```

说明

- 在上面的例子中，使用 break 语句的目的是使得在 switch 语句体中无需进行冗余的额外计算。如果在计算 switch 语句时发现相应的 case 标签，那么 break 语句内。

注

- break 语句只能作用于循环主体或 switch 语句或在循环中或 switch 主体。

6.6.4 return 语句

语法

```
return 表达式 ;
```

功能

- return 语句退出包含 return 的当前函数并将控制转到对此函数发起调用 return 的函数处位置，然后调用并返回 return 语句表达式的值作为函数调用表达式的值。
- 一个函数中可以使用出现两个或更多个 return 语句。
- 在函数末尾使用结束花括号 "}" 与执行不带无表达式的纯 return 语句具有相同的结果。

示例

```
int     f ( int ) ;

void    main ( void )  {
        /* ... */
        int     i = 0 , y = 0 ;
        y = f ( i ) ;
        /* ... */
}

int     f ( int  i ) {
        int     x = 0 ;
        /* ... */
        return ( x ) ;
}
```

说明

- 在上面的例子中，当控制传到 return 语句时，函数 f() 将一个值返回给 main 函数。因为变量 “x” 的值作为返回值返回，所以赋值运算符会导致变量 “y” 的值被变量 “x” 的值替换。

注

- 对于 void 类型函数来说，表示返回值的表达式不能在 return 语句中使用。

第 7 章　结构体和共用体

结构体或共用体都是不同类型成员对象的集合，这些对象以群组方式存在于在处于一个给定名称的组集合中。结构体的成员对象连续分配到存储空间中，而共用体的成员对象共享同一块存储空间。

7.1　结构体

如前所述，结构体是连续分配到存储空间中的成员对象的集合。

(1) 结构体和结构体变量的声明

结构体声明列表及结构体变量均要用关键字 “struct”进行声明。任何可以被称为叫标记名的名称都可以放在结构体声明列表中。

在声明之后，就可以使用该标记名对同一个结构体的结构体变量进行声明这种结构体的结构体变量。

[结构体的声明]

```
struct  标记名 { 结构体声明列表 } 变量名 ;
```

在下面的例子中，在第一个 struct 声明当中，指定了标记名为 “data”的 int 类型数组，包括 “code”和、char 类型数组 name、 addr、 tel 四项成员，且声明 no1 为拥有这种结构的结构体变量。在第二个声明中，声明了结构体变量 no2、 no3、 no4 和 no5，与 no1 具有相同的结构。

[示例]

```
struct  data {
        int     code ;
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
struct  data    no2 , no3 , no4 , no5 ;
```

(2) 结构体声明列表

结构体声明表结构体声明列表指定要声明的结构体类型的内在结构。结构体声明表结构体声明列表中的每个单独单个元素都是是可以被调用的成员，以按照声明的次序给其中各个成员分配存储区。在下面的 [结构体声明表结构体声明列表举例] 中，按照变量 a、数组 b、二维数组 c 的次序分配存储区。

成员的类型不能指定为不完备完整类型 （未知长度的数组）或函数类型均不能指定为各成员的类型。因此，结构体本身不能被包含在结构体声明表结构体声明列表中。

各成员的对象类型可以是除上述两种类型之外的任何类型。还可以用位段方式来指定用来对各成员的位数进行指定的位字段。

如果变量取二进制值 “0” 或 “1”，则位字段位段所需的最少位数指定为 1。通过这种对位字段位段所需最少位数的指定，可在一个整型区域中存储两个 （或更多）成员。

[结构体声明列表举例]

```
int     a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

[位字段域声明举例]

```
struct  bf_tag {
        unsigned int    a : 2 ;
        unsigned int    b : 3 ;
        unsigned int    c : 1 ;
} bit_field ;
```

位段

(3) 数组和指针

结构体变量也可以声明为数组，也可以或用指针进行引用。

[结构体数组]

结构体数组的声明方式与其它对象相同。

```
struct  data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
struct  data    no [ 5 ] ;
```

[结构体指针]

指向结构体的指针会具有指针所指示的结构体的特性。换句话说，如果前移结构体指针，就会将结构体的整体长度加到指向下一个结构体的指针中，这样才能指向下一个结构体。

在下面的例子中，“dt_p”为指向 “struct data”类型值的指针。此时，如果前移指针 “dt_p”前移（增加），则指针与 “&no[1]”的值相同。

```
struct  data    no [ 5 ] ;
struct  data    *dt_p = no ;
```

(4) 如何引用结构体成员

结构体成员可用两种方式引用：一种是利用结构体变量，一种是利用指向变量的指针。

[用结构体变量进行引用]

"." 在利用结构体变量引用结构体成员时，用到 （点）运算符。

```
struct  data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { " NAME " , " ADDR " , " TEL " } , *data_ptr = no ;

void main ( ) {
        char    c ;
        c = no [ 0 ] . name [ 1 ] ;
}
```

[用指向共用体变量的指针进行引用]

在利用指向变量的指针引用结构体成员时，用到 "->" （箭头）操作符。

```
struct  data {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { " NAME " , " ADDR " , " TEL " } , *data_ptr = no ;

void main ( ) {
        char    c ;
        data_ptr -> tel [ 3 ] = ' E ' ;
}
```

7.2 共用体

如前所述，共用体是共享同一个存储空间 （或在存储空间中重叠）的成员的集合。

(1) 共用体和共用体变量的声明

共用体声明列表及共用体变量用关键字 “union”进行声明。任何可以被称为叫标记名的名称都可以放在共用体声明列表中。之后就可以使用该标记名对同一个共用体的共用体变量进行声明。

[共用体的声明]

```
union 标记名 { 共用体声明列表 } 变量名 ;
```

在下面的例子中，在第一个 union 声明当中，指定了标记名 “data”的 char 类型数组，包括 “name”、“addr”、“tel”三项成员，并将 “no1”声明为同类共用体变量。在第二个 union 声明中，共用体变量 “no2、 no3、 no4、 no5”所属的共用体与 “no1”声明的共用体相同。

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no1 ;
union   data    no2 , no3 , no4 , no5 ;
```

(2) 共用体结构体声明列表

共用体声明列表指定要声明的共用体类型的内在结构。在联合体声明列表上的各元素称为成员。声明分配到相同区域的成员。在下面的 [共用体声明列表举例] 中，给 “c”分配的存储区是成员中最大的。其它成员不会为其它成员分配新存储区，而是使用同一块区域。

不完备完整类型 （未知长度的数组）或函数类型均不能被指定为共用体声明列表中各成员的类型。

各成员的对象类型可以是除上述两种类型之外的任何类型。

[共用体声明表共用体声明列表]

```
int             a ;
char    b [ 7 ] ;
char    c [ 5 ] [ 10 ] ;
```

(3) 共用体数组和指针

共用体变量也可以声明为数组，或也可以用指针 （与结构体数组和指针大致相同）进行引用。

[共用体数组]

共用体数组声明方法与其它对象相同。

```
union   data  {
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} ;
union   data    no [ 5 ] ;
```

[共用体指针]

指向共用体的指针会具有该指针指示的共用体的特性。换句话说，若前移共用体指针，则会在指向下一个共用体的指针上加上共用体的长度，这样才能够指向下一个共用体。

在下面的例子中，“dt_p”为指向 “union data”类型值的指针。

```
union   data    no [ 5 ] ;
union   data    *dt_p = no ;
```

(4) 如何引用共用体成员

共用体成员（或共用体元素）可用两种方式进行引用：一种是利用共用体变量，一种是利用指向变量的指针。

[利用共用体变量进行引用]

"." 在利用共用体变量引用共用体成员时，用到（点）操作符。

```
union data{
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} no [ 5 ] = { " NAME " , " ADDR " , " TEL " } ;

void main ( void ) {
        no [ 0 ] .addr [ 10 ] = ' B ' ;
            :
}
```

[用指向共用体变量的指针进行引用]

在利用指向变量的指针引用共用体成员时，用到 "->"（箭头）操作符。

```
union data{
        char    name [ 12 ] ;
        char    addr [ 50 ] ;
        char    tel [ 12 ] ;
} *data_ptr ;

void main ( void ) {
        data_ptr -> name [ 1 ] = ' N ' ;
            :
}
```

第 8 章　外部定义

在程序中，外部声明列表位于预处理语句之后。这些声明引用作为 “外部声明”，因为它们出现的函数以外且有效的文件范围内。

用标识符给外部对象命名的声明，或者保证函数存储类型的声明都称为外部定义。如果有外部连接声明的标识符用外部连接声明，又在表达式中被使用 （除 sizeof 运算符的运算对象而外），那么在整个程序中的某处必定存在该标识符的唯一外部定义。

外部定义的语法如下所示。

```
#define TRUE  1
#define FALSE 0
#define SIZE  200
void    printf ( char * , int ) ;
void    putchar ( char c ) ;

char    mark [ SIZE + 1 ] ;     /* 外部对象声明 */

main ( )
{
        int     i , prime , k , count ;

        count = 0 ;

        for ( i = 0 ; i <= SIZE ; i++ )
                mark [ i ] = TRUE ;
        for ( i = 0 ; i <= SIZE ; i++ ) {
                if ( mark [ i ] ) {
                        prime = i + i + 3 ;
                        printf ( " %d " , prime ) ;
                        count++ ;
                        if ( ( count%8 ) == 0 ) putchar ( ' \n ' ) ;
                        for ( k = i + prime ; k <= SIZE ; k += prime )
                                mark [ k ] = FALSE ;
                }
        }
        printf ( " Total %d\n " , count ) ;

loop1 :
        goto    loop1 ;
}
```

8.1 函数定义

函数定义是以函数声明开头的外部定义。如果在声明中省略了存储类说明符，则默认假定定义为 “extern”。外部函数定义意味着定义的函数可能会被从其它文件被引用。例如，在包含两个 （或更多）个文件的程序中，如果在某个文件中引用另一个文件中的函数，则该函数必须是外部定义的。

外部函数的存储类说明符为 extern 或 static。若函数声明为 extern，则该函数可以从另一个文件引用。若函数声明为 static，则该函数不能从另一个文件来引用。

在下面的例子中，存储类说明符为 "extern" , 类型说明符为 "int"。这两个都是缺省值，因此通过设定这样可以省略。函数声明符 "max(int a, int b)" 和函数的主体为 "{ 返回 a > b ? a : b ;)"。

[函数定义举例]

```
extern  int     max ( int a , int b )
{
        return a > b ? a : b ;
}
```

因为此函数定义在函数声明中指定了参数类型，所以编译程序器对参数类型会进行强制转换。该这种类型转换可以用参数的标识符列表的形式进行描述。如下所示为类型转换该标识符列表的示例。

```
extern  int     max ( a , b )
int     a , b ;
{
       return a > b ? a : b ;
}
```

当调用函数的参数时，可以传递函数的地址。通过在表达式中使用函数名，能产生指向函数的指针。

```
int     f ( void ) ;
void    main ( ) {
    :
        g ( f ) ;
}
```

在上面的例子中，函数 g 被指向函数 f 的指针传递给函数 f。函数 g 必须只能以下列两种方式之一进行定义：

```
void    g ( int ( *funcp ) ( void ) )
{
        ( *funcp ) ( ) ;        /* or funcp ( ) ; */
}
```

或者

```
void    g ( int func ( void ) )
{
        func ( ) ;              /* or ( *func ) ( ) ; */
}
```

8.2 外部对象定义

外部对象定义引用对象标识符的声明，该声明具有文件作用域或有对应的初始化程序。如果对象标识符的声明有文件作用域而无对应初始化程序，且无存储类说明，或存储类为 static，则该对象定义被认为是临时的，因为该声明的文件作用域的初始化程序为 0。

如下所示为外部对象定义的示例。

[外部对象定义示例]

- int i1 = 1 ; : 用外部连接定义
- static int i2 = 2 ; : 用内部连接定义
- extern int i3 = 3 ; : 用外部连接定义
- int i4 ; : 用外部连接进行临时定义
- static int i5 ; : 用内部连接进行临时定义
- int i1 ; : 引用先前声明的有效临时定义
- int i2 ; : 链接规则非法
- int i3 ; : 引用先前声明的有效临时定义
- int i4 ; : 引用先前声明的有效临时定义
- int i5 ; : 链接规则非法
- extern int i1 ; : 引用以前有外部链接的声明
- extern int i2 ; : 引用以前有内部链接的声明
- extern int i3 ; : 引用以前有外部链接的声明
- extern int i4 ; : 引用以前有外部链接的声明
- extern int i5 ; : 引用以前有内部链接的声明

第 9 章　预处理器指令（编译器指令）

预处理指令是位于 # 预处理记号和换行符之间的一串预处理记号。

在预处理记号字符串之间的空白字符只能是空格和横向制表符。

预处理器指令指定源文件编译前进行的处理工作。预处理器指令包括的操作有如：根据条件处理或跳过部分源文件、从其它源文件获取附加代码、将原来的源代码替换为其它文本（与宏扩展相同）。下面详细介绍各个预处理器指令。

9.1　条件编译指令

条件编译根据常量表达式的值跳过部分源文件。如果由条件编译指令指定的常量表达式的值为 0，那么指令后面的语句不会进行转换（编译）。 sizeof 操作符、 cast 操作符或枚举类型常量都不能在任何条件编译指令的常量表达式中不能使用 sizeof 操作符、 cast 操作符或枚举类型常量。

条件编译指令的类型如下所述：

- #if 指令
- #elif 指令
- #ifdef 指令
- #ifndef 指令
- #else 指令
- #endif 指令

在预处理器指令中可以使用下列单目表达式（称为定义表达式）：

```
defined  标识符
或
defined  ( 标识符 )
```

若已经用 #define 预处理器指令定义了标识符，则单目表达式返回 1，若标识符没有定义或其定义已被取消，则返回 0。

[示例]

- 在本例中，因为 SYM 已定义，所以单目表达式返回 1，并对编译 #if 和 #endif 之间的内容进行编译 （关于 #if 到 #endif 的说明，请参见后文的说明）。

```
#define SYM     0

#if defined     SYM
    :
#endif
```

9.1.1 #if 指令

语法

```
#if 常量表达式 换行 " 组 "
```

功能

- #if 指令在常量表达式的值为 0 的情况下，使 C 语言编译器在翻译阶段中跳过 （丢弃）一段源代码。

示例

```
#if FLAG == 0
    :
#endif
```

说明

- 在上面的例子中，通过计算常量表达式 “FLAG == 0” \xmlclose 是否成立，以此来确定是否在翻译阶段中使用 #if 和 #endif 之间的一组若干语句 （即源代码）。如果 "FLAG" 的值为非零，则将取消在 #if 和 #endif 之间的源代码。如果值为零，则会对 #if 和 #endif 之间的源代码进行翻译。

9.1.2 #elif 指令

语法

```
#elif 常量表达式 换行 " 组 "
```

功能

- #elif 指令通常跟在 #if 指令后。如果 #if 指令的常量表达式的值为 0，则会计算 #elif 指令的常量表达式。若 #elif 指令的常量表达式为 0，则 C 编译器在翻译阶段将跳过 （丢弃） #elif 和 #endif. 之间的语句 （一段源代码）。

示例

```
#if FLAG == 0
    :
#elif FLAG != 0
    :
#endif
```

说明

- 在上面的例子中，计算常量表达式 “FLAG==0”或 “FLAG!=0”，以此来确定在翻译阶段内是否用到 #if 后面的一组语句及 #elif 后面的另一组语句。如果 “FLAG”的值为零，则会对 #if 和 #elif 之间的源代码进行翻译。如果值为非零，则会对 #elif 和 #endif 之间的源代码进行翻译。

9.1.3 #ifdef 指令

语法

```
#undef 标识符       换行        " 组 "
```

功能

- #ifdef 指令等于 #if defined (标识符)
- 如果已用 #define 指令定义了标识符，则会翻译 #ifdef 和 #endif 之间的语句。如果从未定义该标识符或其定义已取消，则在翻译阶段内会跳过 #ifdef 和 #endif 之间的源代码。

示例

```
#define ON
#ifdef  ON
    :
#endif
```

说明

- 在上面的例子中，已用 #define 指令定义了标识符 “ON” } 因此，会翻译 #ifdef 和 #endif 之间的源代码。若从未定义标识符 “ON”，则 #ifdef 和 #endif 之间的源代码会被丢弃。

9.1.4 #ifndef 指令

语法

```
#ifndef  标识符  换行 " 组 "
```

功能

- #ifndef 指令等于 #if !defined (标识符)。如果从未用 #define 指令定义该标识符，则不会翻译 #ifndef 和 #endif 之间的源代码。

示例

```
#define ON
#ifndef ON
    :
#endif
```

说明

- 在上面的例子中，已用 #define 指令定义了标识符 “ON” } 因此，不会编译 #ifndef 和 #endif 之间的程序。若从未定义标识符 “ON”，则会编译 #ifndef 和 #endif 之间的程序。

9.1.5 #else 指令

语法

```
#else   换行        " 组 "
```

功能

- 在前面的条件包含指令的标识符为非零的情况下， #else 指令使 C 编译器在翻译阶段内时翻译 #else 后面的一段源代码。 #if、 #elif、 #ifdef 或 #ifndef 指令可能位于 #else 指令之前。

示例

```
#define ON
#ifdef  ON
    :
#else
    :
#endif
```

说明

- 在上面的例子中，已用 #define 指令定义了标识符 “ON” } 因此，会翻译 #ifndef 和 #endif 之间的源代码。如果从来没有定义过标识符 "ON"，则翻译在 #else 和 #endif 中的源代码。

9.1.6 #endif 指令

语法

```
#endif  换行
```

功能

- #endif 指令指示 #ifdef 块的结尾。

示例

```
#define ON
#ifdef  ON
    :
#endif
```

说明

- 在上面的例子中， #endif 指示 #ifdef 块 （#ifdef 指令的有效范围）的结尾。

9.2 源文件包含指令

预处理指令 #include 搜索指定的头文件，并用该头文件的全部内容来替换 #include 指令语句。 #include 指令在包含其它源文件时可能会采用下列三种形式之一：

- #include < > 指令
- #include " " 指令
- #include 预处理记号字符串指令

#include 指令可以出现在由 #include 得到的源程序中。但是， CC78K0S 中对 #include 指令的嵌套是有限制的。关于限制的详细内容，请参阅表 1-1。

备注　　预处理记号字符串： #define 指令定义的字符串

9.2.1 #include < > 指令

语法

```
#include        < 文件名 >    换行
```

功能

- 若指令形格式为 #include <>，则 C 编译器会使用编译器选项 -i 在指定目录中搜索， INC78K0S 环境变量指定的目录和在注册库表中注册的目录 \NECTools32\78k0s 目录中搜索尖括号中指定的头文件，并用指定文件的全部内容来替换 #include 指令。

示例

```
#include        < stdio.h >
```

说明

- 在以上例子中 , C 编译器查找 INC78K0S 环境变量设定文件夹和注册在文件 "stdio.h" 中的文件夹 \NECTools32\inc78k0s ， 且用指定文件 "stdio.h" 的全部内容替换指令行 "#include < stdio.h >" 。

  备注　　上面所述的文件夹可能会因安装方法的不同而有异。

9.2.2 #include " " 指令

语法

```
#include          " 文件名 "       换行
```

功能

- 如果指令格式为 #include “ ”, 当前工作文件夹是对在双括号中设定的头文件的首次查找。如果没有发现，则使用 -i 编译器选项查找指定文件夹 , 由 INC78K0S 环境变量指定的文件夹 , 以及注册在注册表中的文件夹 \NECTools32\inc78k0s 。然后，编译器用搜索到的指定文件的全部内容来替换将 #include 指令行替换为搜索到的指定文件的全部内容。

示例

```
#include        " myprog.h "
```

说明

- 在上面的例子中， C 编译器在当前工作目录、 INC78K0S 环境变量指定的目录、注册库中注册的目录 \NECTools32\INC78k0s 中搜索双引号中指定的文件 myprog.h，用指定文件 myprog.h 的全部内容来替换指令行 #include “myprog.h”。将指令行 #include “myprog.h”替换为指定文件 myprog.h 的全部内容。

  备注　　上面所述的文件夹可能会因安装方法的不同而有异。

9.2.3 #include 预处理记号字符串指令

语法

```
#include          预处理记号字符串          换行
```

功能

- 若指令形式为 “#include 预处理记号字符串”，则要搜索的头文件由宏替换指定，用指定文件的全部内容替换 #include 指令行用指定文件的全部内容替换。

示例

```
#define          INCFILE " myprog.h "
#include         INCFILE
```

说明

- 在用 “#include 预处理记号字符串”的形式包含其它源文件时，必须用宏替换将指定的 “预处理记号字符串”替换为 < 文件名 > 或 “文件名”。若用 < 文件名 > 替换记号字符串，则 C 编译器在 -i 编译器选项 -i 指定的目录、 INC78K0S 环境变量指定的目录、注册库中注册的目录 \NECTools32\INC78k0s 中搜索指定文件。若用 “文件名”替换记号字符串，则首先搜索当前工作目录。如果没有发现指定文件，则使用 **-i** 编译器选项查找指定文件夹 , 由 INC78K0S 环境变量指定的文件夹 , 以及注册在注册表中的文件夹 \NECTools32\inc78k0s 。

  备注　　上面所述的文件夹可能会因安装方法的不同而有异。

9.3 宏替换指令

宏替换指令 #define 和 #undef 用于将 “替换列表”替换为 “标识符”指定的字符串，且替换到 #define 分别给定的标识符的范围结尾处。 #define 指令有两种形式：对象格式和函数格式：

- 对象格式

  #define 指令

- 函数格式

  #define ( ) 指令

(1) 实际参数替换

函数调用时的实际参数替换，必须是在标识了函数形式宏调用的参数之后执行的。如果替换表替换列表中的参数没有前缀 # 或 ## 预处理记号，或如果 ## 预处理记号之后未跟在任何此类参数之后，那么列表中的所有宏在替换为对应宏参数前都会进行扩展。

(2) # 操作符

预处理记号将对应宏参数替换为 char 字符串处理记号。换句话说，如果替换表替换列表中的参数前缀有该预处理记号，则对应宏参数将被翻译为字符或字符串。

(3) ## 操作符

预处理记号将 ## 符两侧的两个记号连接成一个记号。连接将在下一个宏扩展前进行，## 预处理记号将在连接之后被删除。由此连接产生的记号如果有在遇到宏名时就会进行宏扩展。

[## 操作示例]

上面的宏替换指令将进行如下的扩展：

```
printf ( " x " " 1 " " = %d , x " " 2 " " = %s " , x1 , x2 ) ;
```

连接后的 char 字符串像这样。

```
printf ( " x1 = %d , x2 = %s " , x1 , x2 ) ;
```

```
#include        < stdio.h >
#define debug ( s , t ) printf ( " x " #s " = %d , x " #t " = %s " , x##s
, x##t ) ;

void    main ( ) {
        int     x1 , x2 ;
        debug ( 1 , 2 ) ;
}
```

(4) 重扫描和再替换

将再次扫描由列表中宏参数的替换而产生的预处理记号字符串将会被再次扫描，以及源文件中所有剩余的预处理记号一起被扫描。当前替换的宏名 （不包括源文件中剩余的预处理记号），即使在扫描替换表替换列表扫描中时再次发现当前替换的宏名 （不包括源文件中剩余的预处理记号），也不会进行替换。

(5) 宏定义的作用域

宏定义 （#define 指令）一直会持续进行宏替换，直至遇到对应的 #undef 指令为止。

9.3.1 #define 指令

语法

```
#define 标识符        替换表替换列表              换行
```

功能

- #define 指令会用最简单的形式将指定标识符 （明显的）替换为给定的替换列表 （任何字符队列不包含换行），无论何时使用此指令定义后在源代码中出现相同标识符。

示例

```
#define PAI     3.1415
```

说明

- 在上面的例子中，标识符 “PAI”只要在源代码中指令定义位置之后出现，都会被替换为 “3.1415”替换。

9.3.2 #define ( ) 指令

语法

```
#define 标识符  ( " 标识符列表 " )  替换列表      换行
```

功能

- 函数形式的 #define 指令有其格式：

  #define 名称 (名称 , ..., 名称) 替换列表

  将使用函数格式指定的标识符替换为给定的替换列表 （任何字符队列不包含换行）。在第一个名称和开口括号 "(" 间不允许有空格。这个名称的列表 （标识符列表）可能为空。因为此指令以这个形式定义宏，将宏调用替换为括号内的宏参数。

示例

```
#define F ( n )  ( n * n )
void    main ( ) {
        int     i ;
        i = F ( 2 ) ;
}
```

说明

- 在上面的例子中， #define 指令将用 “(2*2)”替换 “F(2)”，因此 i 的值将为 4。为安全起见，一定要将替换列表放在括号当中，因为此函数形式的宏与函数定义不同，仅仅用来替换字符序列。

9.3.3 #undef 指令

语法

```
#undef 标识符        换行
```

功能

- #undef 指令未定义给定的标识符。换句话而言，这条指令终止由对应 #define 指令设置的标识符作用域。

示例

```
#define F ( n )  ( n * n )
    :
#undef  F
```

说明

- 在上面的例子中， #undef 指令将使先前由 “#define F(n) (n*n)”指定的标识符 “F”无效。

9.4 行控制指令

行控制预处理器指令 “#line”将 C 编译器要在翻译中用到的行号替换为该指令指定的数字。如果随同数字附带一起还给出了字符串，则该指令还会将 C 编译器的源文件名称替换为指定字符串。

(1) 更改行号

要想更改行号，应进行如下的指定。无法指定 0 和大于 32767 的数都无法指定。

```
#line    数字字符串  换行
```

[示例]

```
#line   10
```

(2) 更改行号和文件名

要想更改行号和文件名，应进行如下指定。

```
#line   数字字符串   " 字符串 "     换行
```

[示例]

```
#line   10      " file1.c "
```

(3) 用预处理程序记号字符串进行更改

除上述指定外，还可以进行如下的指定。在这种情况下，指定的预处理器记号字符串在所有的替换之后的结果必须属于上述两例中的其中一种。

```
#line   预处理记号字符串        换行
```

[示例]

```
#define LINE_NUM        100
#line   LINE_NUM
```

9.5 #error 预处理指令

#error 预处理指令是输出包括指定预处理记号在内的信息且使编译过程不完全终止的指令。 此预处理器用来终止编译。

此预处理器指令的指定说明如下。

```
#error   " 预处理记号字符串 "   换行
```

[示例]

- 在本例中使用了指示 CC78K0S 设备系列号的宏名称 __K0S__。 若设备为 78K0S 系列，则编译 #if 和 #else 之间的程序。 在其它情况下，编译 #else 和 #endif 之间的程序，但 #error 指令会终止编译并输出错误报错信息 " 不适用 78K0S" 。

```
#if     __K0S__
    :
#else
#error   " 不适用 78K0S "
    :
#endif
```

9.6 #pragma 指令

#pragma 指令是使用编译器定义方法来指示编译器进行操作的指令。在 CC78K0S 中 , 多条 #pragma 指令生成 78K0S 序列代码 (关于 #pragma 指令的详细内容 , 参考 " 第 11 章 扩展函数 ").

[示例]

- 在本例中， #pragma NOP 指令使该语句描述语句能在 C 源程序中直接输出 NOP 指令语句。

```
#pragma NOP
```

9.7 空指令

仅包含 # 字符及空白的源程序行称为空指令。空指令在预处理时会被直接丢弃。换句话说，这些指令对编译器没有影响。空指令的语法如下所示。

```
#   换行
```

9.8 编译程序定义的宏名称

CC78K0S 中定义了下列 宏名宏名称。

__LINE__ 当前源程序行的行号 （十进制常量）

__FILE__ 源文件名 （string literal 字符文字）

__DATE__ 源文件编译日期 （字符串字面量文字，形如 "Mmm dd yyyy")

__TIME__ 源文件编译时间 （字符串文字字面量，形如 "hh:mm:ss")

__STDC__ 十进制常量 “1”，指示表示与 ANSI 注规范相符兼容

注 ANSI 是美国国家标准协会的缩略词

绝对不能用 #define 或 #undef 预处理器指令定义这些宏名或者将其 #define 或 #undef 预处理指令一定不能用于这些宏名和定义为的标识符。 所有编译器定义的宏名都以下划线开头，跟着是一个大写字符或第二又一个下划线。

除上述宏名之外，还会根据开发中所用的设备提供用来于 （根据所用产品开发中用到的设备）指示设备序列名的宏名和指示具体设备名称名的宏名。 要想针对输出目标设备的输出目标码，必须用通过选项在编译时刻指定宏名，或由 C 源程序中的处理器类型指定宏名。

- 表示设备序列名的宏名

```
__K0S__
```

- 表示设备名名称的宏名

  "__" 在设备类型名称前被添加，名称后添加了 ‘_’ 。

  英语字符为大写。

  < 示例 >

```
__9216_    __9216Y_
```

  注 设备类型 Device type 名称与 -C 选项指定的类型名称相同。 关于设备类型名称名，请参见与设备文件相关的参考资料。

CC78K0S 具有指示存储模型的宏名。

当指定静态模型时，定义如下

```
#define__STATIC_MODEL__   1
```

通过在命令行中加入下列内容可指定编译的设备类型

"-c 设备类型名 "

< 示例 >

```
cc78k0s  -c9216Y  prime.c
```

不需要在对编译的时指定设备类型，可以的指定不需要在 C 源程序的开头进行指定。

"#pragma PC (设备类型)"

< 示例 >

```
#pragma PC      ( 9216Y )
    :
```

但是，下列内容可以在 “#pragma PC (设备类型)”之前进行描述。

- 注释语句
- 不会产生变量的定义 / 调用或函数的预处理器指令。

第 10 章　库函数

在 C 语言中，没有专门的指令用于和与外部来源 （外围器件和设备）之间进行传输 （输入输出）数据的传出传入的指令。这是因为 C 语言的设计者希望其这种函数的个数尽可能保持最少。但是，对于实际系统开发来说， I/O 操作是必须的。因此， CC78K0S 中必定会含有进行 I/O 操作的库函数。

CC78K0S 中含有的库函数如 I/O 函数、字符 / 存储器操作函数、程序控制函数、和数学函数等。本章介绍该在 CC78K0S 中所提供的库函数。

10.1　函数间的接口

要想使用库函数，必须进行调用。对库函数的调用是需要通过调用指令来完成的。函数的参数和返回值分别由栈和寄存器进行传递。然而 , 第一个参数也通过寄存器传递。此外，在静态模式中，所有的参数都会被寄存器传递。

10.1.1　参数

把参数放入堆栈中的操作，或从堆栈中移除参数的操作都是由调用器方 （发起调用的函数端）进行的。被调用者方 （被别的函数调用的函数端）只引用参数值。但是，当参数由寄存器传递时，被调用者调用方会直接引用寄存器，如果必要的话，还会将参数值复制到另一个寄存器中。另外，当指定了函数调用接口为自动 pascal 函数选项 -ZR 时，若如果参数是通过在堆栈上进行传递的，那么从堆栈中移除参数的操作是由被调用方端完成的。

如果参数通过堆栈传递，那么参数在堆栈中的存放是自底至顶逐个降序进行的。

可以放入堆栈的最小数据单位是 16 位。大于 16 位的数据类型从其最高有效位 （MSB）开始以 16 位为单位逐个放入堆栈中。 8 位类型的数据在入栈时扩展为 16 位类型的数据。

在静态模式时，所有参数都通过寄存器传递。

最多可传递 3 个参数，总共可传递 6 字节。不支持对浮点、双精度及结构结构体参数的传递。

首个参数传递的列表如下所示。在正常模式中，第二个及之其后的参数通过堆栈进行传递。

标准库的函数接口 （参数传递及返回值存储）与普通函数相同。

表 10-1 第一参数传递列表 （正常模式）

第一参数的类型	传递方法
1 字节、 2 字节整型	AX
3 个字节的整数	AX, BC
4 个字节的整数	AX, BC
浮点数 （float 类型）	AX, BC
浮点数 （double 类型）	AX, BC
其它数据	通过堆栈传递

备注： 在上述类型中， 1 到 4 字节整型中包含括了结构体和共用体。

表 10-2 参数传递列表 （正常模式）

参数的类型	第一参数	第二参数	第三参数
1 个字节的整数	A	B	H
2 个字节的整数	AX	BC	HL

备注 如果参数共有 4 字节，那么部分参数会分配给 AX 和 BC，其余参数分配给 HL 或 H。

1 至 4 字节整型不包括结构体和共用体。

10.1.2 返回值

函数的返回值按从寄存器 BC 到寄存器 DE 的顺序从它的最低有效位 （LSB）开始以 16 位库为单位进行存储，顺序是从寄存器 BC 到寄存器 DE。在当返回结构体时，结构体的第一首地址储存到在寄存器 BC 中。在当返回指针时，结构体的第一首地址储存到在寄存器 BC 中。

如下所示为返回值存储的列表如下所示。，返回值的存储方法与普通函数相同。

(1) 普通模式

表 10-3 存储返回值列表 (Normal 模式)

返回值的类型	存储方法
1 位	CY
1 字节、 2 字节整型	BC
4 个字节的整数	BC (低阶端），， DE (高端阶）
浮点数 （float 类型）	BC (低阶端），， DE (高端阶）
浮点数 （double 类型）	BC (低阶端），， DE (高端阶）
结构	复制结构体，返回到函数专用区域，把地址存储到 BC 中。
指针型	BC

(2) 静态模式

表 10-4 存储返回值列表 (静态模式)

返回值的类型	存储方法
1 位	CY
1 个字节的整数	A
2 个字节的整数	AX
4 个字节的整数	AX (低阶端), BC (高阶端)
指针型	AX

10.1.3 保存个别单独库 （Individual Libraries）所用的寄存器

使用 HL （在正常模式时）和 DE （在静态模式时）的库，把使用的寄存器保存到堆栈中。

各库使用 saddr 区域保存使用堆栈保存的 saddr 区域。堆栈区由被各个库当作工作区使用。

(1) 未指定 -ZR 选项

传递参数和返回值的步骤如下所示。

< 函数调用 >

```
" long func ( int a , long b , char *c ) ; "
```

(a) （由调用器调用方）把参数放入栈中

参数 "c" 和 "b" 的高 16 位，参数 "b" 的低 16 位按命名的顺序依次放置在堆栈内。通过 AX 寄存器传递 a。

(b) （由调用器调用方）通过 call 指令调用 func

返回地址放置在堆栈中，且接着为参数 "b" 的低 16 位和控制传送到函数 func 中。

(c) （由被调用者调用方）在函数中保存在函数中要使用的寄存器

如果要使用寄存器 HL，那么 HL 原来的值就被放压在入堆栈中。

(d) （由被调用者调用方）把由寄存器传递的第一个参数放入堆栈中。

(e) （由被调用者调用方）处理 func 并把返回值存储在寄存器中。

在 BC 中存储返回值 "long" 的低 16 位，在 DE 中存储返回值的高 16 位。

(f) （由被调用者调用方）恢复存储的第一个参数

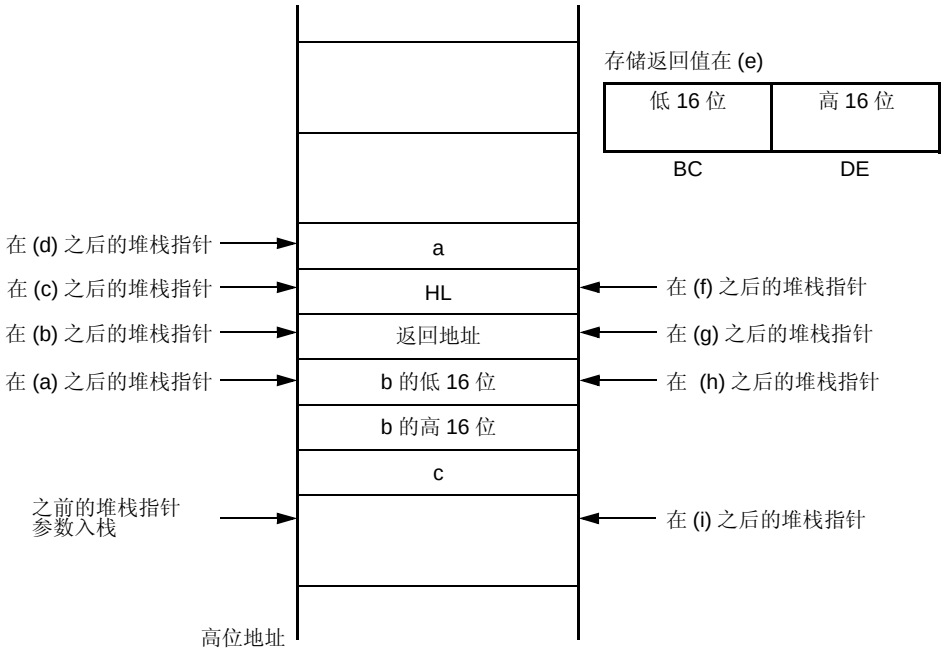
(g) （由被调用者调用方）恢复保存的寄存器

(h) （由被调用者调用方）通过 ret 指令把控制权返回给调用器调用方

(i) （由调用器调用方）从堆栈中移除参数

参数的字节数 （以 2 字节为单位）添加到堆栈指针中。在图 10-1 的例子当中，添加了 6。

图 10-1 函数被调用时的栈区 （未指定 -ZR）



(2) 指定了 -X 选项。

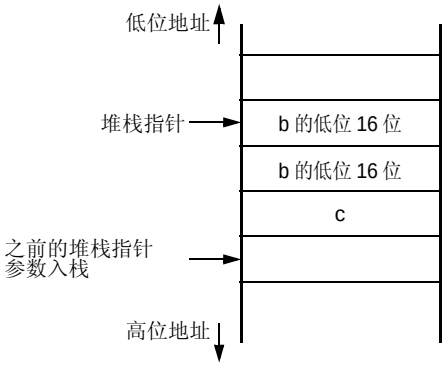
下面的例子展示了在指定了 -ZR 选项时传递参数和返回值的步骤。

< 函数调用 >

```
" long func ( int a , long b , char *c ) ; "
```

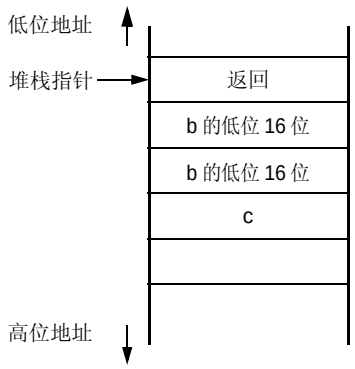
(a) （由调用器调用方）把参数放入栈中

参数 "c" 和 "b" 的高 16 位，参数 "b" 的低 16 位按命名的顺序依次放置在堆栈内。通过 AX 寄存器传递 a。

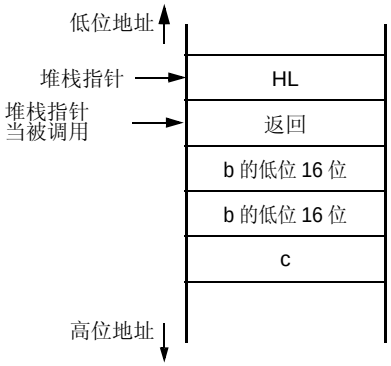


(b) （由调用器调用方）通过 call 指令调用 func

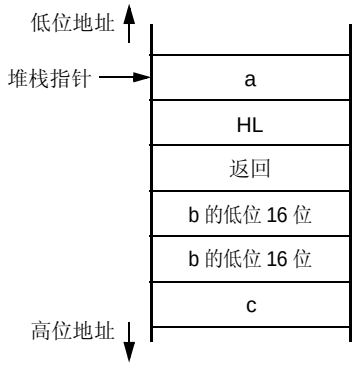
当堆栈处于如下所示的状态时，控制权转移到函数 func。



(c) （由被调用者调用方）保存使用的寄存器



(d) 由寄存器调用的第一参数放入栈中。

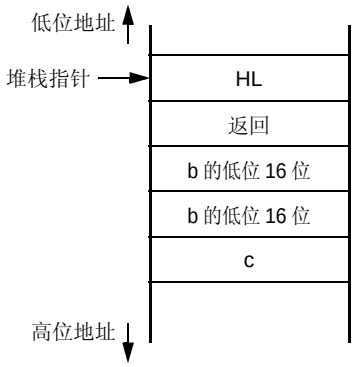


(e) （由被调用者调用方）进行对函数 func 的处理，把返回值存储在寄存器中。

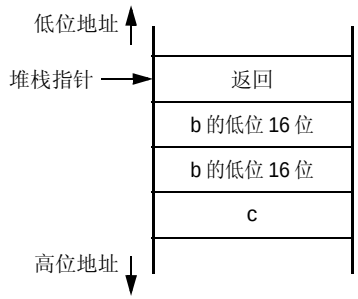
在 BC 中存储返回值的低 16 位，在 DE 中存储高 16 位。

BC	DE
返回值的低 16 位	返回值的低 16 位

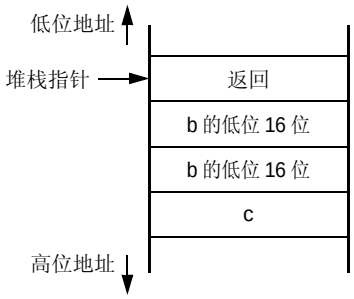
(f) （由被调用者调用方）恢复第一个放置的参数



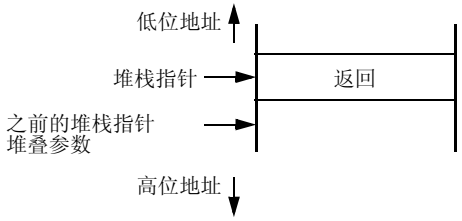
(g) （由被调用者调用方）恢复保存的寄存器



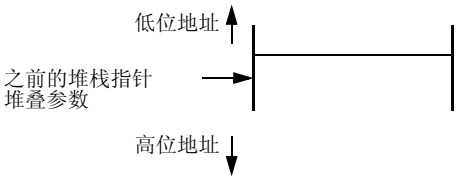
(h) 存储返回值在寄存器中，移动堆栈指针指向位置的值，并压入该位置的参数到堆栈中，从堆栈中移除参数 （在被调用方）。



(i) （由调用方）恢复存储在寄存器中的返回地址。



(j) （由调用方）通过 ret 指令将控制权返回给调用方的函数



10.2 头部

CC78K0S 具有 13 个头部 （或头文件）。每个头文件对标准库函数、数据类型名和宏名宏名称进行定义或声明。CC78K0S 的头文件如下所示。

ctype.h	setjmp.h	stdarg.h (仅正常模式)	stdio.h
stdlib.h	string.h	error.h	errno.h
limits.h	stddef.h	math.h (仅正常模式)	float.h
assert.h (仅正常模式)			

备注　支持的函数随存储模型式 （正常模式和静态模式）的不同而有所差异不同。而且，在普通模式操作中工作的函数的操作会随 -ZI 和 -ZL 选项而有所不同。对于因 -zi 和 -zl 选项的存在而无法正常工作的函数，会输出报警信息 “未进行原型声明”。

(1) ctype.h

此头文件用来定义字符函数和字符串函数。在此标准头文件中定义了下列库函数。

但是，在指定了编译程序编译选项 -ZA （此选项禁止使用不符合 ANSI 规范的函数，允许使用符合 ANSI 规范的一部分函数）时， _toupper 和 _tolower 将不会无被定义。，取而代之定义了 tolow 和 toup 作为替代。当未指定 -ZA 时， tolow 和 toup 不做定义。声明函数随选项和指定模式的不同而有所差异。

表 10-5 ctype.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
isalnum	OK	OK	OK	OK	OK	NG	OK	NG
isalpha	OK	OK	OK	OK	OK	NG	OK	NG
iscntrl	OK	OK	OK	OK	OK	NG	OK	NG
isdigit	OK	OK	OK	OK	OK	NG	OK	NG
isgraph	OK	OK	OK	OK	OK	NG	OK	NG
islower	OK	OK	OK	OK	OK	NG	OK	NG
isprint	OK	OK	OK	OK	OK	NG	OK	NG
ispunct	OK	OK	OK	OK	OK	NG	OK	NG
isspace	OK	OK	OK	OK	OK	NG	OK	NG
isupper	OK	OK	OK	OK	OK	NG	OK	NG
isxdigit	OK	OK	OK	OK	OK	NG	OK	NG
tolower	OK	OK	OK	OK	OK	NG	OK	NG
toupper	OK	OK	OK	OK	OK	NG	OK	NG
isascii	OK	OK	OK	OK	OK	NG	OK	NG
toascii	OK	OK	OK	OK	OK	NG	OK	NG
_tolower	OK	OK	OK	OK	OK	NG	OK	NG
_toupper	OK	OK	OK	OK	OK	NG	OK	NG
tolow	OK	OK	OK	OK	OK	NG	OK	NG
toup	OK	OK	OK	OK	OK	NG	OK	NG

OK： 支持

NG： 不支持

(2) setjmp.h

此头文件用来定义程序控制函数。在此头文件中定义了下列函数。声明的函数随选项和指定模式的不同而有所差异声明的函数随选项和规范模型的不同而不同。

表 10-6 setjmp.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
setjmp	OK	OK	OK	OK	OK	NG	OK	NG
longjmp	OK	OK	OK	OK	OK	NG	OK	NG

OK： 支持

NG： 不支持

在头文件 setjmp.h 中定义了下列对象：

[对 int 数组类型 jmp_buf 的声明]

- 普通模式

```
typedef int     jmp_buf [ 11 ]
```

- 静态模式

```
typedef int     jmp_buf [ 3 ]
```

(3) stdarg.h (仅正常模式)

此头文件用来定义特殊函数。在此头文件中定义了下列 3 个函数：

表 10-7 stdarg.h 的内容

函数	对 -zi 或 -zl 的设定			
	普通模式			
	None	ZI	ZL	ZI ZL
va_arg	OK	OK	OK	OK
va_start	Δ	Δ	Δ	Δ
va_starttop	Δ	Δ	Δ	Δ
va_end	OK	OK	OK	OK

OK： 支持

Δ： 操作可以有保证，但有某些存在限制

在头文件 stdarg.h 中声明了下列对象：

[把 char 指针类型 “va_list”声明为指针类型 va_list 给 char]

```
typedef char    *va_list ;
```

(4) stdio.h

此头文件用来定义 I/O 函数。在此头文件中定义了下面的函数。

声明函数随选项和指定模式的不同而有所差异。

表 10-8 stdio.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
sprintf	OK	-	OK	-	NG	NG	NG	NG
sscanf	OK	-	OK	-	NG	NG	NG	NG
printf	OK	-	OK	-	NG	NG	NG	NG
scanf	OK	-	OK	-	NG	NG	NG	NG
vprintf	OK	-	OK	-	NG	NG	NG	NG
vsprintf	OK	-	OK	-	NG	NG	NG	NG
getchar	OK	OK	OK	OK	OK	NG	OK	NG
gets	OK	OK	OK	OK	OK	OK	OK	OK
putchar	OK	OK	OK	OK	OK	NG	OK	NG
puts	OK	OK	OK	OK	OK	NG	OK	NG

OK： 支持

NG： 不支持

-： 操作无法保证

声明了下列宏名宏名称。

```
#define  EOF     ( -1 )
#define NULL    ( void * ) 0
```

(5) stdlib.h

此头文件用来定义字符函数和字符串函数、存储器函数、程序控制函数、数学函数及特殊函数。在此标准头文件中定义了下列库函数：

但是，在指定了编译程序编译选项 -ZA （此选项禁止使用不符合 ANSI 规范的函数，允许使用符合 ANSI 规范的一部分函数）时，brk、sbrk、itoa、ltoa、ultoa 无定义。取而代之定义 strbrk、strsbrk、stritoa、strltoa 和 strultoa 作为替代。当没有设定 -za 时，则这些函数不作定义。

表 10-9 stdlib.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
atoi	OK	-	OK	-	OK	NG	OK	NG
atol	OK	OK	-	-	NG	NG	NG	NG
strtol	OK	OK	-	-	NG	NG	NG	NG
strtoul	OK	OK	-	-	NG	NG	NG	NG
calloc	OK	OK	OK	OK	OK	NG	OK	NG
free	OK	OK	OK	OK	OK	NG	OK	NG
malloc	OK	OK	OK	OK	OK	NG	OK	NG
realloc	OK	OK	OK	OK	OK	NG	OK	NG
abort	OK	OK	OK	OK	OK	OK	OK	OK
atexit	OK	OK	OK	OK	OK	NG	OK	NG
exit	OK	OK	OK	OK	OK	NG	OK	NG
abs	OK	OK	OK	OK	OK	NG	OK	NG
div	OK	NG	OK	NG	NG	NG	NG	NG
labs	OK	OK	-	-	NG	NG	NG	NG
ldiv	OK	OK	NG	NG	NG	NG	NG	NG
brk	OK	OK	OK	OK	OK	NG	OK	NG
sbrk	OK	OK	OK	OK	OK	NG	OK	NG
atof	OK	OK	OK	OK	NG	NG	NG	NG
strtod	OK	OK	OK	OK	NG	NG	NG	NG
itoa	OK	OK	OK	OK	OK	NG	OK	NG
ltoa	OK	OK	NG	NG	NG	NG	NG	NG
ultoa	OK	OK	NG	NG	NG	NG	NG	NG
rand	OK	-	OK	-	NG	NG	NG	NG
srand	OK	OK	OK	OK	NG	NG	NG	NG
bsearch	OK	OK	OK	OK	NG	NG	NG	NG

表 10-9 stdlib.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
qsort	OK	OK	OK	OK	NG	NG	NG	NG
strbrk	OK	OK	OK	OK	OK	NG	OK	NG
strsbrk	OK	OK	OK	OK	OK	NG	OK	NG
stritoa	OK	OK	OK	OK	OK	NG	OK	NG
strltoa	OK	OK	NG	NG	NG	NG	NG	NG
strultoa	OK	OK	NG	NG	NG	NG	NG	NG

OK： 支持

NG： 不支持

-： 操作无法保证

在头文件 stdlib.h 中定义了下列对象：

[对结构体类型 div_t 进行的声明，其具有 int 类型成员 “quot”和 “rem”为 int 类型 （静态模式除外）]

```
typedef struct {
        int     quot ;
        int     rem ;
} div_t ;
```

[对结构体类型 ldiv_t 的进行声明，其具有 long int 类型成员 "quot" 和 "rem" （在静态模式和正常模式中指定了 -zl 时除外）]

```
typedef struct {
        long int        quot ;
        long int        rem ;
} ldiv_t ;
```

[对宏名宏名称 “RAND_MAX”的定义]

```
#define RAND_MAX        32767
```

[对宏名宏名称的定义]

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

(6) string.h

此头文件用来定义字符函数和字符串函数、存储器函数及特殊函数。在此头文件中定义了下列函数。定义的函数随选项和指定模式规范模型的不同而有所差异不同。

表 10-10 string.h 的内容

函数	对 -zi 或 -zl 的设定							
	普通模式				静态模式			
	None	ZI	ZL	ZI ZL	None	ZI	ZL	ZI ZL
memcpy	OK	OK	OK	OK	OK	NG	OK	NG
memmove	OK	OK	OK	OK	OK	NG	OK	NG
strcpy	OK	OK	OK	OK	OK	OK	OK	OK
strncpy	OK	OK	OK	OK	OK	NG	OK	NG
strcat	OK	OK	OK	OK	OK	OK	OK	OK
strncat	OK	OK	OK	OK	OK	NG	OK	NG
memcmp	OK	-	OK	-	OK	NG	OK	NG
strcmp	OK	-	OK	-	OK	NG	OK	NG
strncmp	OK	-	OK	-	OK	NG	OK	NG
memchr	OK	OK	OK	OK	OK	NG	OK	NG
strchr	OK	OK	OK	OK	OK	NG	OK	NG
strcspn	OK	-	OK	-	OK	NG	OK	NG
strpbrk	OK	OK	OK	OK	OK	OK	OK	OK
strrchr	OK	OK	OK	OK	OK	NG	OK	NG
strspn	OK	-	OK	-	OK	NG	OK	NG
strstr	OK	OK	OK	OK	OK	OK	OK	OK
strtok	OK	OK	OK	OK	OK	OK	OK	OK
memset	OK	OK	OK	OK	OK	NG	OK	NG
strerror	OK	OK	OK	OK	OK	NG	OK	NG
strlen	OK	-	OK	-	OK	NG	OK	NG
strcoll	OK	-	OK	-	OK	NG	OK	NG
strxfrm	OK	-	OK	-	OK	NG	OK	NG

OK： 支持

NG： 不支持

-： 操作无法保证

(7) error.h

error.h 包含 errno.h。

(8) errno.h

在此头文件中定义了下列对象 :

[宏名称 "EDOM", "ERANGE", 和 "ENOMEM" 的定义]

```
#define EDOM    1
#define ERANGE  2
#define ENOMEM  3
```

[对 volatile int 类型外部变量 errno 的声明]

```
extern volatile int errno ;
```

(9) limits.h

在此头文件中定义了下列宏名宏名称 :

```
#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
```

但是，当指定了 -qu 选项 （它把未指定修饰符 char 当作 unsigned char）时，通过由编译程序编译器声明的宏 _ _CHAR_UNSIGNED_ _，可以对 CHAR_MAX 和 CHAR_MIN 以下列方式进行声明。

```
#define CHAR_MAX        ( 255U )
#define CHAR_MIN        ( 0 )
```

当编译选项中指定了 -ZI 选项 （int 和 short 类型被当作 char 类型， unsigned int 和 unsigned short 类型被当作 unsigned char 类型）作为编译程序选项时，通过由编译程序编译器声明的宏 _ _FROM_INT_TO_CHAR_ _，可以对 INT_MAX、 INT_MIN、 SHRT_MAX、 SHRT_MIN、 SINT_MAX、 SINT_MIN、 SSHRT_MAX、 SSHRT_MIN、 UINT_MAX 和 USHRT_MAX 进行如下的声明。

```
#define INT_MAX         CHAR_MAX
#define INT_MIN         CHAR_MIN
#define SHRT_MAX        CHAR_MAX
#define SHRT_MIN        CHAR_MIN
#define SINT_MAXS       CHAR_MAX
#define SINT_MINS       CHAR_MIN
#define SSHRT_MAXS      CHAR_MAX
#define SSHRT_MINS      CHAR_MIN
#define UINT_MAX        UCHAR_MAX
#define USHRT_MAX       UCHAR_MIN
```

当编译选项中指定了 -zl 选项 （long 类型当作 int 类型， unsigned long 类型当作 unsigned int 类型）为编译程序选项时，通过由编译程序编译器声明的宏 _ _FROM_LONG_TO_INT_ _，可以对 LONG_MAX、 LONG_MIN 和 ULONG_MAX 进行如下的声明。

编译器声明的 __FROM_LONG_TO_INT__ 。

```
#define LONG_MAX        ( +32767 )
#define LONG_MIN        ( -32768 )
#define ULONG_MAX       ( 65535U )
```

(10) stddef.h

在此头文件中声明和定义了下列对象：

[对将 int 类型声明为 “ptrdiff_t” 的声明]

```
typedef int     ptrdiff_t ;
```

[对将 unsigned int 类型声明为 “size_t”的声明]

```
typedef unsigned int    size_t ;
```

[对宏名宏名称 “NULL”的定义]

```
#define NULL    ( void * ) 0 ;
```

[对宏名宏名称 “offsetof” 的定义]

```
#define offsetof ( type , member )  ( ( size_t ) & ( ( ( type* ) 0 ) -> member ) )
```

注 offsetof (类型，成员说明符)

Offsetof 扩展为普通整型常量表达式，此表达式包含有 size_t 类型，其值是以字节为单位的偏移值，从结构体 （由这个类型指定）开始处向结构体成员 （由成员指定符来进行定义）进行扩展顺序查找。

当声明了 static type t; 时，成员说明符必须使得 “该表达式 & （t. 成员说明符）” 的计算结果为地址常量。当指定说明的成员为位段字段时，操作无法保证。

(11) math.h (仅正常模式)

math.h 定义了下列函数。

表 10-11 math.h 的内容

函数	对 -zi 或 -zl 的设定			
	普通模式			
	None	ZI	ZL	ZI ZL
acos	OK	OK	OK	OK
asin	OK	OK	OK	OK
atan	OK	OK	OK	OK
atan2	OK	OK	OK	OK
cos	OK	OK	OK	OK
sin	OK	OK	OK	OK
tan	OK	OK	OK	OK
cosh	OK	OK	OK	OK
sinh	OK	OK	OK	OK
tanh	OK	OK	OK	OK
exp	OK	OK	OK	OK
frexp	OK	OK	OK	OK
ldexp	OK	OK	OK	OK
log	OK	OK	OK	OK
log10	OK	OK	OK	OK
modf	OK	OK	OK	OK
pow	OK	OK	OK	OK
sqrt	OK	OK	OK	OK
ceil	OK	OK	OK	OK
fabs	OK	OK	OK	OK

表 10-11 math.h 的内容

函数	对 -zi 或 -zl 的设定			
	普通模式			
	None	ZI	ZL	ZI ZL
floor	OK	OK	OK	OK
fmod	OK	OK	OK	OK
matherr	OK	NG	OK	NG
acosf	OK	OK	OK	OK
asinf	OK	OK	OK	OK
atanf	OK	OK	OK	OK
atan2f	OK	OK	OK	OK
cosf	OK	OK	OK	OK
sinf	OK	OK	OK	OK
tanf	OK	OK	OK	OK
coshf	OK	OK	OK	OK
sinhf	OK	OK	OK	OK
tanhf	OK	OK	OK	OK
expf	OK	OK	OK	OK
frexpf	OK	OK	OK	OK
ldexpf	OK	OK	OK	OK
logf	OK	OK	OK	OK
log10f	OK	OK	OK	OK
modff	OK	OK	OK	OK
powf	OK	OK	OK	OK
sqrtf	OK	OK	OK	OK
ceilf	OK	OK	OK	OK
fabsf	OK	OK	OK	OK
floorf	OK	OK	OK	OK
fmodf	OK	OK	OK	OK

OK： 支持

NG： 不支持

下列对象的定义。

[对宏名宏名称 “HUGE_VAL” 的定义]

```
#define HUGE_VAL  DBL_MAX
```

(12) float.h

float.h 定义了下列对象。

当 double 型的大小为 32 位，通过宏来排序定义的宏。

__DOUBLE_IS_32BITS__ declared by the compiler.

```
#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 */
```

(13) assert.h (仅正常模式)

表 10-12 assert.h 的内容

函数	对 -zi 或 -zl 的设定			
	普通模式			
	None	ZI	ZL	ZI ZL
__assertfail	OK	OK	OK	OK

OK： 支持

assert.h 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 */
```

但是，如果 assert.h 头文件引用另一个宏 NDEBUG （此宏名称不未在 assert.h 头文件中定义），如果当 assert.h 被加载到进入源文件时，NDEBUG 被定义为宏，那么 assert.h 头文件就会简单地声明 assert 宏为：

```
#define assert ( p )    ( ( void ) 0 )
```

10.3 可重入性（仅适用于正常模式）

重入是一种状态，是指由一个程序调用的函数能够继续被另一个程序继续调用。

CC78K0S 的标准库不使用静态区，从而具有可重入性。因此，函数所使用的存储区内的数据不会因为来自另一程序的调用而被破坏。

但是，在 (1) 至 (3) 中所示的函数是不能可重入的。

(1) 不能重入的函数

setjmp, longjmp, atexit, exit

(2) 下列函数所使用的区域在启动例程中得到被特意保留保证的区域

div, ldiv, brk, sbrk, rand, srand, strtok

(3) 处理浮点数的函数

sprintf, sscanf, printf, scanf, vprintf, vsprintf[Note], atof, strtod, 所有数学函数

注 在 sprintf、 sscanf、 printf、 scanf、 vprintf 和 vsprintf 中，不支持浮点数的都是可重入的。

10.4 标准库函数

本节按照下列函数分类方式说明本 CC78K0S 的标准库函数。即使在指定了 -ZF 参数时也支持所有的标准库函数。

表 10-13 标准库函数列表

函数类型	函数
字符和字符串函数	is-
	toupper, tolower
	toascii
	_toupper/toup, _tolower/tolow
程序控制函数	setjmp, longjmp
特殊函数	va_start (仅正常模式), va_starttop (仅正常模式), va_arg (仅正常模式), va_end (仅正常模式)
I/O 函数	sprintf （仅正常模式）
	sscanf （仅正常模式）
	printf (仅正常模式)
	scanf （仅正常模式）
	vprintf (仅正常模式)
	vsprintf (仅正常模式)
	getchar
	gets
	putchar
	puts
应用函数	atoi, atol
	strtol, strtoul
	calloc
	free
	malloc
	realloc
	abort
	atexit, exit
	abs, labs
	div (仅普通模式), ldiv (仅普通模式)
	brk, sbrk
	atof, strtod

表 10-13 标准库函数列表

函数类型	函数
应用函数	itoa, ltoa (仅普通模式), ultoa (仅普通模式)
	rand, srand
	bsearch (仅正常模式)
	qsort (仅正常模式)
	strbrk
	strsbrk
	stritoa, strltoa (normal model only), strultoa (normal model only)
字符串 / 存储函数	memcpy, memmove
	strcpy, strncpy
	strcat, strncat
	memcmp
	strcmp, strncmp
	memchr
	strchr, strrchr
	strspn, strcspn
	strpbrk
	strstr
	strtok
	memset
	strerror
	strlen
	strcoll
	strxfrm
数学函数	acos (仅正常模式)
	asin (仅正常模式)
	atan (仅正常模式)
	atan2 (仅正常模式)
	cos (仅正常模式)
	sin (仅正常模式)
	tan (仅正常模式)
	cosh (仅正常模式)
	sinh (仅正常模式)

表 10-13 标准库函数列表

函数类型	函数
数学函数	tanh (仅正常模式)
	exp (仅正常模式)
	frexp (仅正常模式)
	ldexp (仅正常模式)
	log (仅正常模式)
	log10 (仅正常模式)
	modf (仅正常模式)
	pow (仅正常模式)
	sqrt (仅正常模式)
	ceil (仅正常模式)
	fabs (仅正常模式)
	floor (仅正常模式)
	fmod (仅正常模式)
	matherr (仅正常模式)
	acosf (仅正常模式）
	asinf (仅正常模式)
	atanf (仅正常模式)
	atan2f(仅正常模式)
	cosf (仅正常模式)
	sinf(仅正常模式)
	tanf (仅正常模式)
	coshf (仅正常模式)
	sinhf(仅正常模式)
	tanhf (仅正常模式)
	expf(仅正常模式)
	frexpf (仅正常模式)
	ldexpf(仅正常模式)
	logf (仅正常模式)
	log10f (仅正常模式)
	modff(仅正常模式)
	powf (normal model only)
	sqrtf (仅正常模式)

表 10-13 标准库函数列表

函数类型	函数
数学函数	ceilf (仅正常模式)
	fabsf(仅正常模式)
	floorf (仅正常模式)
	fmodf (仅正常模式)
诊断函数	__assertfail(仅正常模式)

10.4.1 字符和字符串函数

(1) is-

功能

- is- 判断字符的类型。

头文件

- ctype.h，用于所有字符函数

函数原型

- int is- (int c) ;

函数	参数	返回值
is-	c : 进行判断的字符	如果字符 c 包含在字符范围内 : 1 如果字符 c 不包含在字符范围内 : 0

说明

函数	字符范围
isalpha	字母字符 A 至 Z 或 a 至 z
isupper	大写字母 A 至 Z
islower	小写字母 a 至 z
isdigit	数字字符 0 至 9
isalnum	字母数字字符 0 至 9、 A 至 Z 或 a 至 z
isxdigit	十六进制数 0 至 9、 A 至 F 或 a 至 f
isspace	空白字符 （空格、制表符、回车、换行、垂直制表符、换页符）
ispunct	除空白字符之外的标点字符
isprint	可打印字符
isgraph	可印非空字符
iscntrl	控制字符
isascii	ASCII 字符集

(2) toupper, tolower

功能

- 字符函数 toupper 和 tolower 的作用都是将一种类型的字符转换成另一种类型。
- 若 c 为小写字母，则 toupper 函数会返回对应 c 的大写字母。
- 若 c 为大写字母，则 tolower 函数会返回对应 c 的小写字母。

头文件

- ctype.h

函数原型

- int toupper (int c) ;
- int tolower (int c) ;

函数	参数	返回值
toupper, tolower	c : 要进行判断的字符	如果 c 是可转换字符 : 相当于大写字母 如果不能转换 : 返回的字符 "c" 不变

说明

toupper

- toupper 函数检查参数是否为小写字母，如果是，则将该字母转换成对应的大写字母。

tolower

- 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。

(3) toascii

功能

- 字符函数 toascii 转换 "c" 到 ASCII 码。

头文件

- ctype.h

函数原型

- int toascii (int c) ;

函数	参数	返回值
toascii	c : 要进行判断的字符	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" 的位 （位 7 到 15）超出了 "c" (位 0 到 6) 到 "0" 的 ASCII 码范围，则返回转换的位值。

(4) _toupper/toup, _tolower/tolow

功能

- 字符函数 _toupper/toup 从 "c" 减去 "a" 且加上 "A" 的结果。
- 字符函数 _tolower/tolow 从 "c" 减去 "A" 且加上 "a" 的结果。

（_toupper 与 toup 完全相同， _tolower 与 tolow 完全相同）

备注　　a : 小写 ;A : 大写

头文件

- ctype.h

函数原型

- int _toupper/toup (int c) ;
- int _tolower/tolow (int c) ;

函数	参数	返回值
_toupper/toup	c : 要进行判断的字符	"c" - "a" 的减法结果加上 "A"，获取该值。
_tolower/tolow		"c" - "A" 的减法结果加上 "a" ，获取该值。

备注　　where a : 小写字母 ;A : 大写字母

说明

_toupper

- _toupper 函数与 toupper 类似，只是它不检查参数是否为小写字母。

_tolower

- _tolower 函数与 tolower 类似，只是它不检查参数是否为大写字母。

10.4.2 程序控制函数

(1) setjmp, longjmp

功能

- 程序控制函数 setjmp 在被调用时会保存环境信息 （程序的当前状态）。
- 程序控制函数 longjmp 恢复由 setjmp 保存的环境信息。

头文件

- setjmp. h

函数原型

- int setjmp (jmp_buf env) ;
- void longjmp (jmp_buf env , int val) ;

函数	参数	返回值
setjmp	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。
longjmp	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

setjmp

- setjmp, 当直接调用时 , 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

- longjmp 将保存过的环境恢复到 env （saddr 区域，作为 HL 寄存器或寄存器变量使用的 saddr 区和 SP）中。程序继续执行，就好像相对应的 setjmp 返回了 val 一样 （但是，如果 val 为 0，则返回 1）。

10.4.3 特殊函数

(1) va_start (仅正常模式), va_starttop (仅正常模式),
va_arg (仅正常模式), va_end (仅正常模式)

功能

- va_start 函数 （宏）用来启动变量参数列表。
- va_starttop 函数 （宏）用于设置参数变量号的处理。
- va_arg 函数 （宏）从变量参数列表获得参数的值。
- va_end 函数 （宏）指明已到达变量参数列表的末尾。

头文件

- stdarg.h

函数原型

- void va_start (va_list ap , parmN) ;
- void va_starttop (va_list ap , parmN) ;
- type va_arg (va_list ap , type) ;
- void va_end (va_list ap) ;

  { 在 stdarg.h 中使用 typedef 定义 va_list。 }

函数	参数	返回值
va_start, va_starttop	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
va_arg	ap : 处理参数列表的变量 类型 : 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
va_end	ap : 变量用于处理可变参数数量	None

说明

va_start

- 在 va_start 宏里 , 它的参数 ap 必须是 va_list 类型 (char* type) 对象。
- 指向 parmN 中下个参数的指针存储在 ap 中。
- parmN 是函数原型中指定的最后一个 （最右侧）参数的名称。
- 如果 parmN 具有 register 存储类特征，那么就无法保证此函数的正确操作。

va_starttop

- 不能为 va_start 函数设定首个参数，因为首个参数已经通过寄存器传递。
- 遵照如下使用宏。

  (i) 在设定首个参数时，使用 va_starttop 宏。

  (ii) 在设定第二个以及接下去的参数时，使用 va_start 宏。

va_arg

- 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 空指针 , 则 va_arg 返回 0 (类型输入)。

va_end

- va_end 宏在参数指针 ap 中设置一个空指针，告诉通知宏处理器变量参数列表中的所有参数都已经处理完毕。

10.4.4 I/O 函数

(1) sprintf （仅正常模式）

功能

- sprintf 函数根据格式将数据写入字符串 （数组）中。

头文件

- stdio.h

函数原型

- int sprintf (char *s , const char *format , ...) ;

函数	参数	返回值
sprintf	s : 指向字符串的指针写入在其输出区内 格式 : 指向字符串的指针表示格式命令 ... : 转换零个或更多个参数	在 s 中写入的字符的数量 （结尾的空字符不计数）

说明

- 如果实际的参数数量少于格式中所定义的数数量，那么适当的操作就无法保证。在尽管超出格式的情况下，任保留实参数，只统计多余的实参数并忽略它们。
- 根据由 format 指定的格式命令， sprintf 对 format 后面的零个 （或更多）参数进行转换，并将其写入 （复制到）字符串 s 中。
- 可能会使用零个 （或更多）格式命令。普通字符 （除了以 % 字符开头的格式命令之外）照原样输出到字符串 s 中。每个格式命令取得 format 之后的零个 （或更多）参数并将其输出到字符串 s 中。
- 各每个格式命令都以一个 % 字符开头，后面跟的内容可以是着：

  (i) 零个 （或更多）标志 （以后说明），这些标志可以修改格式命令的含义。

  (ii) 可选的十进制整数，指定最小字段宽度

  如果在转换后输出的宽度小于这个最小字段宽度，该说明符就会用零在其左侧进行填充。（如果在 % 后面有左对齐标记 “-”（负号）符，那么就会在输出宽度的右侧填充零。）默认用空格进行填充。若要用 0 对输出进行填充，则应在字段宽度说明符之前放一个 0。如果此数字或字符串大于最小字段宽度，那么仍然会完整打印出来，即使超出部分很少。

- 可选精度 （小数位数）指定说明 (. 整数)

  用 d、 i、 o、 u、 x 和 X 类型说明符指定最小位数。用 s 类型说明符，可以指定最大字符数 （最大字段宽度）。对 e、 E、 f 转换指定输出的小数点后的位数。对 g 和 G 转换指定最大有效位数。此精度说明指定必须采用 (. 整数) 的形式。若省略整数部分，则假定已指定为 0。由此精度指定说明指定产生的填充字符的数量优先于由字段宽度指定产生的填充字符。

- 可选的 h、 I 和 L 修饰符

  h 修饰符要求 sprintf 函数以短整型形或无符号短整型形类型来进行此修饰符后面的 d、i、o、u、x 或 X 类型的转换。 h 修饰符要求 sprintf 函数用短整型指针进行此修饰符后面的 n 类型转换，将指针转换为 short int 类型。

  I 修饰符要求 sprintf 函数以长整型 （long int） long int 或无符号长整型 （unsigned long int） unsigned long int 类型进行此修饰符后面的 d、i、o、u、x 或 X 类型的转换。h 修饰符要求 sprintf 函数用长整型指针进行此修饰符后面的 n 类型转换，将指针转换为 long int 类型。

  其它类型说明符，忽略 h、 I 或 L 修饰符。

- 对转换进行指定的字符 （后文说明）

  在对最小字段宽度或精度 （小数位数）的指定中， * 可以用 * 来代替整型字符串。在这种情况下，整型值将由 int 参数给出 （在参数转换前）。由此产生的负字段宽度都会被解释为 -（负号）标志之后加的正字段。忽略所有负精度。

下列标志用来修饰格式命令：

表 10-14 sprintf 标志

标志	内容
-	转换结果在字段内左对齐。
+	带符号转换的结果总是用 + 或 - 符号开头。
空间	若带符号转换的结果没有符号，则会在输出中加入空格前缀。若同时指定 + （加号）标志和空格标志，将忽略空格标志。
#	以 “赋值形式”对结果进行转换。 在 o 类型转换中，增加精度使第一位变成 0。在 x 或 X 类型的转换中，在非零结果中加入 0x 或 0X 的前缀。在 e、 E 和 f 类型转换中，所有输出值都强制插入一个小数点 （默认无 # 的情况下，只有在真正有不等于零的小数后面还有数值时才显示小数点）。 在 g 和 G 类型转换中，所有输出值都强制插入一个小数点，并且不允许截断后面的 0 （默认无 # 的情况下，只有在后面还有数值时才显示小数点。并且后面跟着的 0 会被截掉。在所有其它转换中，忽略 # 标志。

对输出转换说明的格式码如下所示：

表 10-15 sprintf 的格式代码

格式代码	内容
d	将 int 参数转换为带符号十进制格式。
i	将 int 参数转换为带符号十进制格式。
o	将 int 参数转换为无符号八进制格式。
u	将 int 参数转换为无符号十进制格式。
x	将 int 参数转换为无符号十六进制格式 （含有小写字母 abcdef）。
X	将 int 参数转换为无符号十六进制格式 （含有大写字母 ABCDEF）。

用 d、i、o、u、x 和 X 类型说明符指定结果的最小位数（最小字段宽度）。若输出小于最小字段宽度，则用零填充。如果若未指定精度，则假定默认指定为 1。若用 0 精度转换 0，则什么也看不到。

表 10-16 sprintf 的精度码

精度码	内容
f	用 [-] dddd.dddd 格式将 double 型参数作为带符号值进行转换。 dddd 为一个 （或更多）十进制数。小数点之前的数位由该数的绝对值决定，小数点之后的位数由所需的精度决定。When the precision is omitted, it is interpreted as 6。当省略精度时，默认为精度为 6。
e	将 double 型参数用 “[-] d.dddd e [符号] ddd” [-] d.dddd e [符号] ddd 格式将 double 参数作为带符号值进行转换。d 为一个十进制数， dddd 为一个 （或更多）十进制数。当省略精度时，默认精度为 6。
E	与 e 相同的格式，只是在指数之前添加的是 E 而不是 e。
g	根据指定的精度，在对 double 型参数进行转换时使用 f 或 e 格式中较短的格式。只有当数值的指数小于 -4 或大于由精度指定的数时才会使用 e 格式。 后面的 0 被截掉，并且只有当后面有一个 （或更多）数位小数时才会显示小数点。
G	与 g 相同的格式，只是在指数之前添加的是 E 而不是 e。
c	将 int 整型参数转换为无符号字符型 unsigned char 并将结果写为单个字符。
s	相关参数是一个指向一串字符串的指针，其中的字符会持续写入，直到至遇到终止空字符（但不包含在输出当中）为止。若指定了精度，就会在末尾截断超出最大字段宽度的字符。在未指定精度或精度大于该数组时，该数组必须包含一个空字符。
p	相关参数为一个指向 void 的指针，指针值以十六进制 4 位显示 （小于 4 位的指针值加 0 前缀）。若存在精度指定，将被忽略。
n	相关参数为整型指针，其中放置迄今已经写入字符串 “s” 中的字符的数量。不进行任何转换。
%	Prints a % sign. 不转换相关参数 （但对标志及最小字段宽度的说明指定是有效的）。

- 无效的转换说明符在的操作时无法保证。
- 当实际参数为共用体或结构体或指向它们的指针 （% s 转换中的字符类型数组或 % p 转换中的指针除外）时，操作也无法保证。
- 即使没有字段宽度或字段宽度较小，转换结果也不会被截断。换句话说，如果转换结果的字符数大于字段宽度，则字段会扩展到包含转换结果的宽度。

- 在 %f, %e, %E, %g, %G 转换中特殊的输出字符串的格式如下所示。

  non-numeric -> "(NaN)"

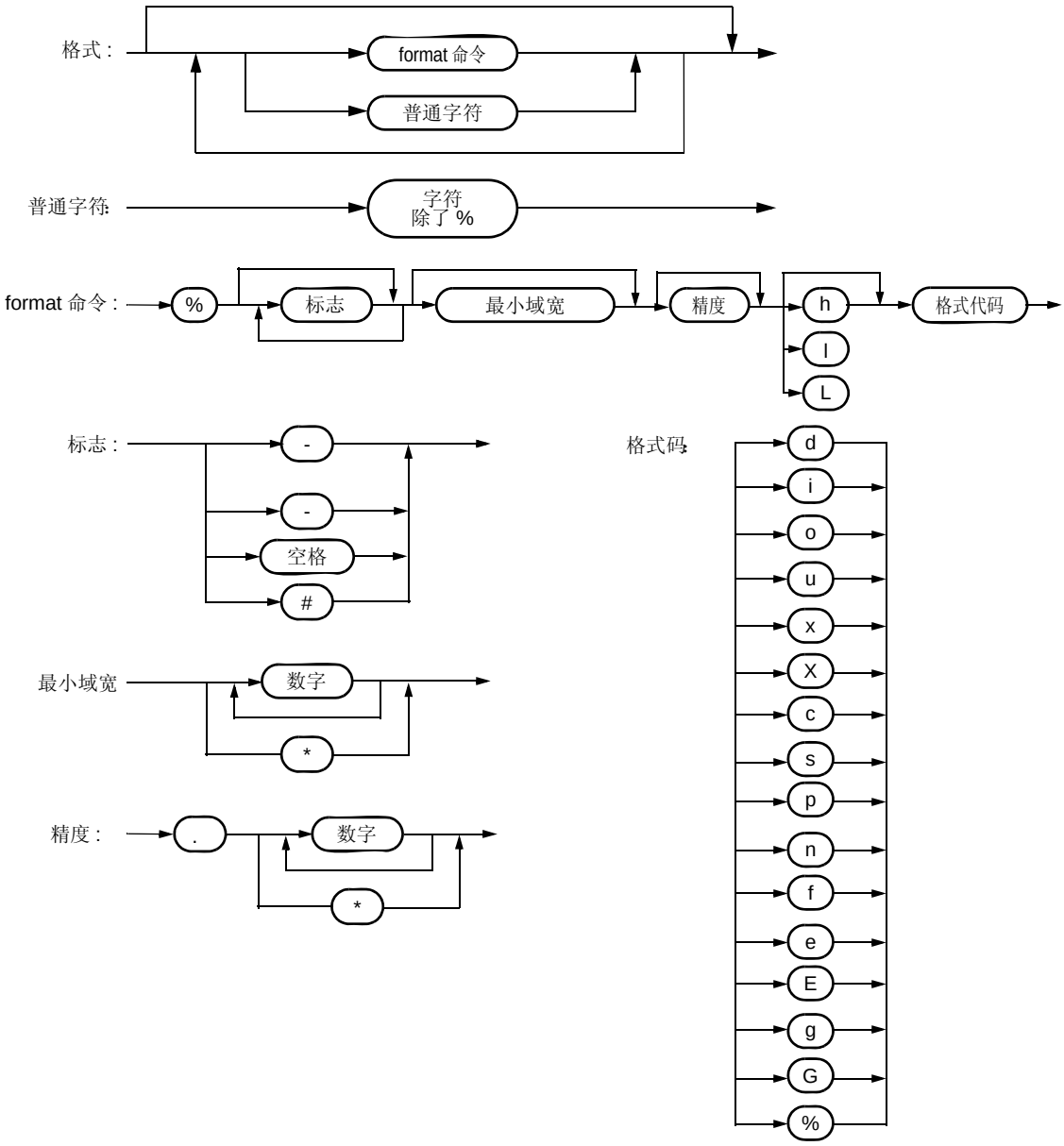
  \+ -> "(+INF)"

  \- ->"(-INF)"

  sprintf 在字符串 s 的末尾会自动写入一个空字符。（该字符包含在返回值计数中）

  format 命令的语法描述在图 10-2 内。

图 10-2 format 命令的语法



(2) sscanf （仅正常模式）

功能

- sscanf 函数根据格式从字符串 （数组）中读取数据。

头文件

- stdio.h

函数原型

- int sscanf (const char *s , const char *format , ...) ;

函数	参数	返回值
sscanf	s : 指向输入字符串的指针 格式 : 指向字符串的指针表示输入格式命令 ... : 保存指向目标的转换值，以及零个或更多参数	若字符串 s 为空 : -1 若字符串 s 不为空 : 已分配输入数据项的个数

说明

- sscanf 从 s 所指的字符串输入数据。指定允许输入的输入字符串格式所指的字符串。由 format 所指的字符串才可以被指定为允许进行输入的输入字符串。format 后面的零个 （或更多）参数用作指向一个对象的指针。format 指定如何从输入字符串进行数据的转换。
- 如果没有足够的参数的数量少于匹配由 format 所指向的格式命令，那么就无法保证编译程序编译器的正确运行。

  如果参数的数量多于由 format 所指向的格式命令，对于多余的参数来说，会进行表达式评估计算但不会有输出任何数据输出。
- 由 format 指向的控制字符串由零个 （或更多）格式命令组成，可分为下列三类：

  1 : 空白字符 （使 isspace 为真的一个或更多个字符）

  2 : 非空白字符 （% 除外）

  3 : 格式说明符

- 各格式说明符都以 % 字符开头，后面跟着下列内容：

  (i) 可选的 * 字符，会限制对向相应参数分配数据

  (ii) 可选的十进制整数，指定最大字段宽度

  (iii) 可选的 h、 l 或 L 修饰符，说明接收端的目标对象大小

  如果 h 在 d、 i、 o 或 x 格式说明符之前，那么参数就不是指向 int 的指针，而是指向 short int 的指针。

  若 l 在其中任何一个所有的格式说明符之前，那么参数就是指向 long int 的指针。

  类似地同样，若 h 在 u 格式说明符之前，则参数就是指向 unsigned short int 的指针。

  若 l 在 u 格式说明符之前，则参数就是指向 unsigned long int 的指针。

  若 l 在转换说明符 e、 E、 f、 g、 G 之前，则参数就是指向 double 的指针 （在没有 l 的情况下为指向 float 的指针）。若 L 在前面，则会忽略。

  备注　　转换说明符 : 标识相应转换类型的字符串 (以后会提到)

sscanf 依次执行在 “format”中的格式命令，若有格式命令失败则终止函数。

(1) 控制字符串中有空白字符时， sscanf 会读取任何数量 （包括零）的空白字符，直到第一个非空白字符 （此字符不读取）为止。如果未遇到非空白字符，则此空白字符命令失败。

(2) 非空白字符使得 sscanf 读取并去除丢弃正在匹配检测的字符。如果未发现指定字符，此命令失败。

(3) 格式命令为每个类型说明符定义一组输入流集合 （在后文中详述）。格式命令按照下列步骤执行：

- 跳过输入的空白字符 （由 isspace 指定），当类型说明符为 [、 c 或 n 时除外。
- 从字符串 “s”中读取输入数据项，当类型说明符为 n 时除外。输入数据项的被定义为为，类型说明符所说明的字符串的第一部分流的最长输入流 （但若有这样指定，则不能超过最大字段宽度）。紧随输入数据项之后的那个字符被认为尚未读取。若输入数据项的长度为 0，则格式命令执行失败。
- 输入数据项 （对应类型说明符 n 的输入字符的个数）转换为由类型说明符指定的类型 （类型说明符 % 除外）。若输入数据项不匹配和指定类型不匹配，则命令执行失败。除非由 * 对分配进行限制，转换结果都会存储在由第一参数 （该参数在 “format”之后，且尚未收到转换结果）所指向的对象中。

可用下列类型声明符：

表 10-17 sscanf 的转换说明符

转换说明符	内容
d	转换十进制整数 （可能带符号）。对相应参数必须是指向整数的指针。
I	转换整数 （可能带符号）。若数字前面有 0x 或 0X，则该数被当作十六进制整数。若数字前面有 0，则该数被当作八进制整数。其它数字被当作十进制整数。对相应参数必须是指向整数的指针。
o	转换八进制整数 （可能带符号）。对相应参数必须是指向整数的指针。
u	转换无符号十进制整数。 对相应参数必须是指向无符号整数的指针。
x	转换十六进制整数 （可能带符号） .
e, E, f, g, G	浮点数值包含可选的符号 （+ 或 - ）、一个 （或更多）包含小数点的连续十进制数、可选的指数 （e 或 E）及下列可选的带符号整数值。当转换结果溢出时，或当转换结果为 + ∞而出现下溢时，转换结果就会是一个非标准化数或± 0。对应参数为指向 float 的指针。
s	输入由非空白字符串组成的字符串。对应参数为指向整型的指针。可以在第一个十六进制整数处前放置 0x 或 0X。对应参数必须是指向数组的指针，该数组必须有足够的长度容纳该字符串外加并含有一个空的字符串结束终结符。字符串结束符会自动添加。
[	输入由期望字符群 （称为 scanset）组成的字符串。对应参数必须是一个指向一个数组的首元素字符的指针，该数组必须有足够的长度容纳该字符串并含有一个字符串结束符。字符串结束符会自动添加。格式命令从此字符处继续，直到右方括号 （1）为止。方括号中的字符串（称为扫描列表 scanlist）构成了 scanset，但当左方括号后紧跟的字符为抑扬音调符号 （）时除外。 当该字符为抑扬符音调符号时，在抑扬符音调符号和右方括号之间除 scanlist 之外的所有字符构成 scanset。但是，当 scanlist 以 [ ] 或 [^] 开头时，此时右方括号也被包含在 scanlist 中，而遇到的下一个右方括号会变成该 scanlist 的结束。 如果指定连字符 （ - ）的范围内左侧字符的 ASCII 码不小于右侧字符，则除 scanlist 最左端或最右端之外的连字符 （ - ）被认为是标点符号中的连字符。
c	输入字符串中的由字符数量 （由字段宽度指定）组成的字符串。（如果省略对字段宽度的指定，就假定为 1。对应参数必须是一个指向一个数组的首字符的指针，该数组必须有足够的长度容纳该字符串。不会添加字符串结束符空终结符。
p	读取无符号十六进制整数。对应参数必须是指向 void 指针的指针。
n	不从字符串 s 接收输入。对应参数必须是指向整数的指针。迄今为止由该函数从字符串 “s” 中读取且已经的字符的数量存储在到由该指针指向的对象当中的字符数量。 %n 格式命令不包含在返回值赋值计数中。
%	读取 % 符号。既不进行转换，也不进行赋值。

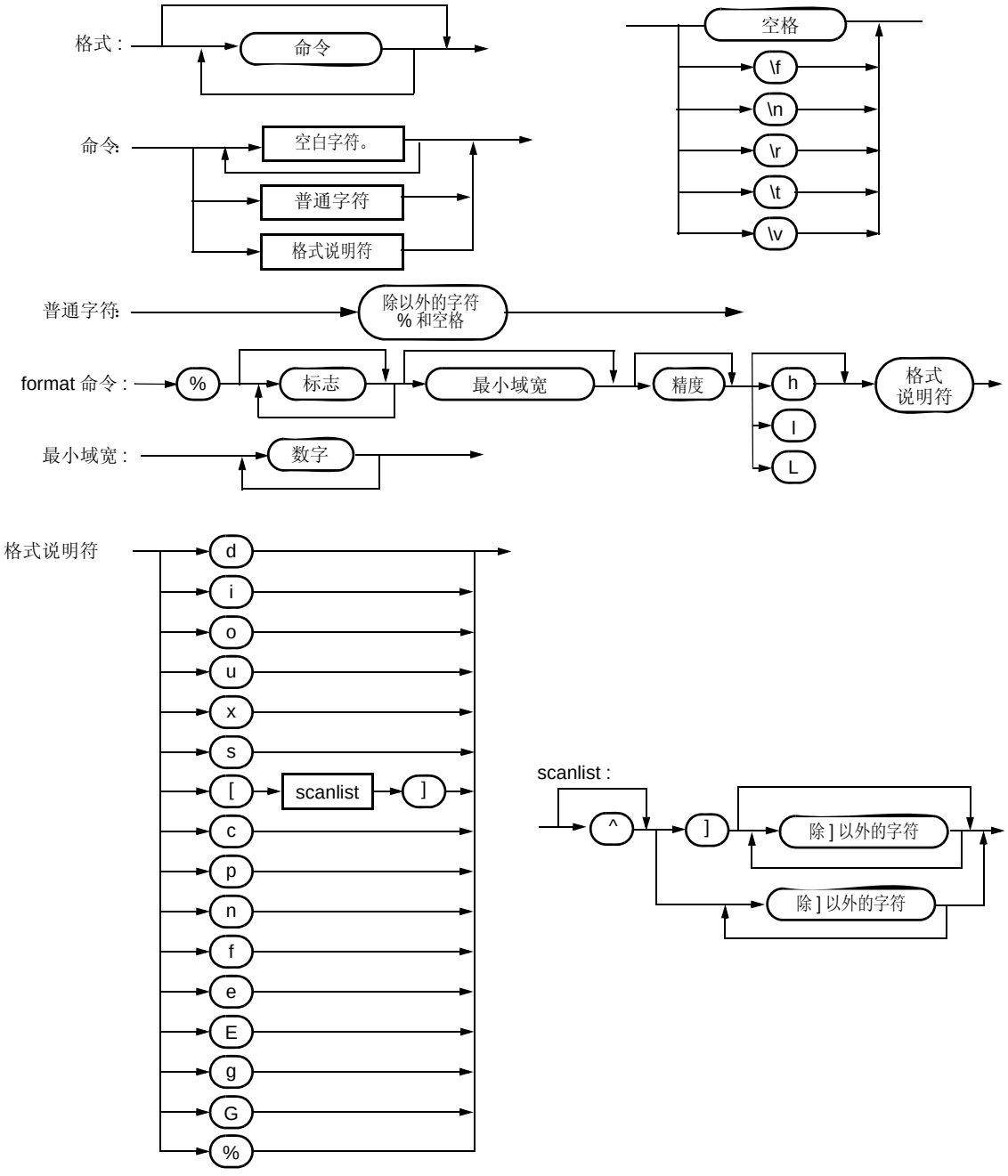
若格式说明符无效，则格式命令执行失败。

若输入流中出现空字符串结束符，则会 sscanf 终止。

若在整型转换（利用 d、i、o、u、x 或 p 格式说明符）中出现上溢，则会根据转换后数据类型的位数截断高位。

输入 format 命令的语法如下所示。

图 10-3 输入格式命令的语法



(3) printf (仅正常模式)

功能

- printf 根据格式输出数据到 SFR 中。

头文件

- stdio.h

函数原型

- int printf (const char *format , ...) ;

函数	参数	返回值
printf	格式 : 指向表示输出转换说明的字符串 ... : 转换零个或更多个参数	输出到 s 的字符的数量 （末尾空字符不计数）

说明

- 根据在格式中指定的输出转换说明，利用 putchar 函数转换并输出符合该格式之后的 （0 个或更多）参数。
- 输出转换说明为 0 个或更多指令。标准字符 （除以 % 开头的转换说明外）由 putchar 函数照原样输出。通过提取并转换后面的 （0 个或更多）参数利用 putchar 函数对转换说明进行输出。
- 各转换说明与 sprintf 函数的情况相同。

(4) scanf（仅正常模式）

功能

- scanf 按照格式从 SFR 中读取数据。

头文件

- stdio.h

函数原型

- int scanf (const char *format , ...) ;

函数	参数	返回值
scanf	格式 : 指向表示输入转换说明的字符串 ... : 指针指向用来分配转换值所赋的对象的指针 （0 个或更多）参数	当字符串 s 不为空时 : 分配输入条目的数量

说明

- 用 getchar 函数进行输入。指定由格式指示的字符串所许可的输入字符串。用格式之后的参数作为指向一个对象的指针。设定如何由输入字符串进行转换的格式。
- 当没有足够的参数供 format 使用时，无法保证正常操作。当参数个数过剩时，会计算对表达式评估表达式但不会有实际输入。
- format 由 0 个或更多指令组成。其指令如下。

  1 : 一个或更多空字符 （使 isspace 为真的字符）

  2 : 标准字符 （除 % 之外）

  3 : 转换指示

- 若转换末尾的输入字符与该指定的输入字符相冲突，则冲突的输入字符被向下舍去入。转换的各种指示与 sscanf 函数的相同。

(5) vprintf (仅正常模式)

功能

- vprintf 根据格式输出数据到 SFR 中。

头文件

- stdio.h

函数原型

- int vprintf (const char *format, va_list p) ;

函数	参数	返回值
vprintf	格式 : 指向表示输出转换说明的字符串 p : 指向参数列表的指针	输入出字符的个数 （末尾的空字符不计数）

说明

- 按照由格式规范指定中的输出转换说明，用 putchar 函数转换并输出参数列表中的指针所指示的参数。
- 各转换说明与 sprintf 函数的情况相同。

(6) vsprintf (仅正常模式)

功能

- vsprintf 按照格式将数据写入字符串中。

头文件

- stdio.h

函数原型

- int vsprintf (char *s , const char *format, va_list p) ;

函数	参数	返回值
vsprintf	s : 指向表示写入输出区的字符串 格式 : 指向表示输出转换说明的字符串 p : 指向参数列表的指针	输出到 s 的字符的数量 （末尾空字符不计数）

说明

- 按照由 format 指定的输出转换说明，从将参数列表的指针所指示的参数写入到 s 所指示的字符串中。
- 输出说明与 sprintf 函数的情况相同。

(7) getchar

功能

- getchar 从 SFR 中读取一个字符

头文件

- stdio.h

函数原型

- int getchar (void) ;

函数	参数	返回值
getchar	None	从 SFR 中读取的一个字符

说明

- 返回从 SFR 的 P0 （端口 0）处读取的值。
- 不进行与读取有关的错误校验。
- 要想改变读取在的 SFR 中的读取位置，必须改变的源重新注册到库中，或者由用户创建一个新的 getchar 函数。

(8) gets

功能

- gets 读取一个字符串。

头文件

- stdio.h

函数原型

- char *gets (char *s) ;

函数	参数	返回值
gets	s : 指向输入字符串的指针	正常 : s 如果在文件的结尾没有读取到字符 : 空指针

说明

- 用 getchar 函数读取字符串并存储在 s 指示的数组中。
- 当检测到文件末尾（getchar 函数返回 -1）或读到换行符时，结束对字符串的读取。读取的换行符会丢弃，在数组存储的最后一个字符末尾写入一个空的字符串结束符。
- 当返回值正常时，返回 s。
- 当检测到文件末尾且数组中未读取字符时，数组的内容保持不变，返回一个空指针。

(9) putchar

功能

- putchar 输出一个字符到 SFR 中。

头文件

- stdio.h

函数原型

- int putchar (int c) ;

函数	参数	返回值
putchar	c : 输出字符	输出字符

说明

- 把由 c 指定的字符写入到 SFR 符 P0 （端口 0）中 （转换为 unsigned char 类型）。
- 不进行与写入有关的错误校验。
- 要想改变在 SFR 中的写入位置，必须将改变的源重新注册到库中，或者由用户创建一个新的 putchar 函数要想改变写入的 SFR。

(10) puts

功能

- puts 输出一个字符串。

头文件

- stdio.h

函数原型

- int puts (const char *s) ;

函数	参数	返回值
puts	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

- 用 putchar 函数将写由 s 指示的字符串内容写入，在输出末尾添加一个换行符。
- 不在字符串末尾写空的字符串结束符字符。
- 当返回值正常时，返回 0，；当 putchar 函数返回 -1 时，返回 -1。

10.4.5 字符串 / 存储函数

(1) memcpy, memmove

功能

- 存储器函数 memcpy 将指定数量的字符从存储器的源区域拷贝到存储器的目的区域。
- 存储器函数 memmove 与 memcpy 相同，只是它允许源区域和目的区域之间的重叠。

头文件

- string.h

函数原型

- void *memcpy (void *s1, const void *s2 , size_t n) ;
- void *memmove (void *s1, const void *s2 , size_t n) ;

函数	参数	返回值
memcpy, memmove	s1 : 复制指向目标指针的数据 s2 : 复制指向包含数据的目标 n : 复制多个字符	s1 的值

说明

memcpy

- memcpy 函数从 s2 所指向的对象中将 n 个连续字节拷贝复制到 s1 指向的对象中。
- 若 s2<s1<s2+n （s1 和 s2 重叠），则 memcpy 的存储器复制拷贝操作无法得到保证 （因为拷贝复制从区域开头处开始按顺序进行）。

memmove

- memmove 函数也将 n 个连续字节从 s2 指向的对象中将 n 个连续字节拷贝复制到 s1 指向的对象中。
- 即使 s1 和 s2 重叠，该函数也能正确进行存储器复制拷贝。

(2) strcpy, strncpy

功能

- 字符串函数 strcpy 用来将一个字符串的内容复制到另一个字符串中。
- 字符串函数 strncpy 用来将不超过指定个数数量的字符从一个字符串复制到另一个字符串中。

头文件

- string.h

函数原型

- char *strcpy (char *s1 , const char *s2) ;
- char *strncpy (char *s1 , const char *s2 , size_t n) ;

函数	参数	返回值
strcpy	s1 : 指向复制目标数组的指针 s2 : 指向复制源数组的指针	s1 的值
strncpy	s1 : 指向复制目标数组的指针 s2 : 指向复制源数组的指针 n : 复制多个字符	s1 的值

说明

strcpy

- strcpy 函数将 s2 指向的字符串的内容复制到 s1 指向的数组中 （包括终止字符）。
- 若 s2 < s1 <（s2 + 要复制拷贝的字符长度数量），则无法保证 strcpy 的行为 （由于复制是从开头开始，而不是从指定的字符串按顺序开始进行）。

strncpy

- strncpy 函数将不超过 n 指定的字符从 s2 指向的字符串中将不超过 n 数量的字符复制到 s1 指向的数组中。
- 若 s2<s1<（s2 + 要复制的字符长度或 s2 + n ñ 1 的最小值 + n ñ 1），则无法保证 strncpy 的行为 （由于复制拷贝是从开头而不是从指定的字符串按顺序开始进行）。
- 如果 s2 指向的字符串长度小于指定的 n 指定的字符，那么就会在 s1 的末尾添加空字符直到复制了 n 个字符为止。若 s2 指向的字符串长度大于 n 个字符，则得到的 s1 指向的字符串就不会以空字符终止。

(3) strcat, strncat

功能

- 字符串函数 strcat 将一个字符串连接到另一个字符串。
- 字符串函数 strncat 将不超过指定数量的字符从一个字符串连接到另一个字符串。

头文件

- string.h

函数原型

- char *strcat (char *s1 , const char *s2) ;
- char *strncat (char *s1 , const char *s2 , size_t n) ;

函数	参数	返回值
strcat	s1 : 指向与另一字符串 （s2) 副本相关联的字符串 s2 : 指向与另一字符串 （s1) 副本相关联的字符串	s1 的值
strncat	s1 : 指向与另一字符串 （s2) 副本相关联的字符串 s2 : 指向与另一字符串 （s1) 副本相关联的字符串 n : 相关联的多个字符	s1 的值

说明

strcat

- strcat 函数将 s2 指向的字符串的副本复制到 （包括空终结符）连接到 s1 指向的字符串中。原先 s1 末尾的空终结符被 s2 的第一个字符覆盖改写。
- 当复制对象相互重叠时，操作无保证。

strncat

- strncat 函数将 s2 指向的字符串中不超过 n 个指定的字符 （不包括空终结符）连接到 s1 指向的字符串。原先 s1 末尾的空终结符被 s2 的第一个字符覆盖改写。
- 如果 s2 指向的字符串字符数小于 n 指定的值，则 strncat 函数在连接字符串时会自动添加包括空终结符。如果字符数大于 n 指定的值，则从顶部开始复制连接 n 个字符。
- 空终结符总会添加连接空终结符。
- 当复制对象相互重叠时，操作无保证。

(4) memcmp

功能

- 存储器函数 memcmp 就将给定的字符数对两个数据个数据对象进行比较，只关心位置靠前的给定字符数量。

头文件

- string.h

函数原型

- int memcmp (const void *s1 , const void *s2 , size_t n) ;

函数	参数	返回值
memcmp	s1, s2 : 比较指向 2 个数据目标的指针 n : 比较字符的数量	如果比较 s1 和 s2 两者的 n 个字符且发现相同 : 0 如果比较 s1 和 s2 两者的 n 个字符且发现不相同 : 转换不同数值的初始化不同的字符为 int 型 (s1 字母 - s2 字母)

说明

- memcmp 函数比较 s1 和 s2 两个目标的 n 个字符。
- memcmp 函数返回 0，在比较 s1 和 s2 两者的 n 个字符且发现相同。
- memcmp 函数返回数值差异 （s1 字母 -s2 字母）并转换初始化不同的字符为 int 型，如果比较 s1 和 s2 两者的 n 个字符并发现不同。

(5) strcmp, strncmp

功能

- 字符串函数 strcmp 对两个字符串进行比较。
- 字符串函数 strncmp 对来自两个字符串不超过指定个数数量的字符进行比较。

头文件

- string.h

函数原型

- char *strcmp (char *s1 , const char *s2) ;
- char *strncmp (char *s1 , const char *s2 , size_t n) ;

函数	参数	返回值
strcmp	s1 : 比较指向一个字符串的指针 s2 : 比较指向另一字符串的指针	若 s1 等于 s2 : 0 如果 s1 小于或大于 s2 : 转换不同数值的初始化不同的字符为 int 型 (s1 字母 - s2 字母)
strncmp	s1 : 比较指向一个字符串的指针 s2 : 比较指向另一字符串的指针 n : 比较字符的数量	如果比较 s1 和 s2 两者的 n 个字符且发现相同 : 0 如果比较 s1 和 s2 两者的 n 个字符且发现不相同 : 转换不同数值的初始化不同的字符为 int 型 (s1 字母 - s2 字母)

说明

strcmp

- 使用 strcmp 函数 比较 s1 和 s2 两者的字符串。
- 如果 s1 等于 s2, 则函数返回 0。 如果 s1 小于或大于 s2, the strcmp 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 型。

strncmp

- 使用 strncmp 函数比较 s1 和 s2 两个目标的 n 个字符。
- strncmp 函数返回 0，在比较 s1 和 s2 两者的 n 个字符且发现相同。strncmp 函数返回数值差异 （s1 字母 -s2 字母）并转换初始化不同的字符为 int 型，如果比较 s1 和 s2 两者的 n 个字符并发现不同。

(6) memchr

功能

- 存储器函数 memchr 将指定字符转换为 unsigned char，在对象给定长度中对它进行搜索，并返回指针指向在给定长度对象中首次出现的此该字符的指针。

头文件

- string.h

函数原型

- void *memchr (const void *s , int c , size_t n) ;

函数	参数	返回值
memchr	s : 指向存储体中查找目标的指针 c : 要查找的字符 n : 要查找的字节数	如果找到 c : 指向 c 的先后顺序。 如果 c 没有找到 : 空指针

说明

- memchr 函数首先将 c 指定的字符转换为无符号字符型 （unsigned char） unsigned char，然后返回一个指针，指向 s 指针所指的指向的对象中从开头起 n 个字节范围内首次出现的该字符。
- 若未找到该字符，函数返回空指针。

(7) strchr, strrchr

功能

- 字符串函数 strchr 返回一个指针，指向字符串中首次出现的指定字符。
- 字符串函数 strrchr 返回一个指针，指向字符串中最后出现的指定字符。

头文件

- string.h

函数原型

- char *strchr (const char *s , int c) ;
- char *strrchr (const char *s , int c) ;

函数	参数	返回值
strchr, strrchr	s : 指向要搜索的字符串的指针 c : 查找特定字符	如果 c 在 s 内找到 : 指针表示在字符串 s 中的 c 的先后顺序。 如果 c 没有在 s 内找到 : 空指针

说明

strchr

- strchr 函数在 s 指针所 指向的字符串中搜索 c 指定的字符，返回指向该字符串中首次出现的 c （转换为 char 类型）的指针。
- 空终结符被当作字符串的一部分。
- 如果在字符串中没有找到指定字符，则函数返回空指针。

strrchr

- strrchr 函数在 s 指针所指向的字符串中搜索 c 指定的字符，返回指向该字符串中最后出现的 c （转换为 char 类型）的指针。
- 空终结符被当作字符串的一部分。
- 如果在字符串中若未找到匹配项，则函数返回空指针。

(8) strspn, strcspn

功能

- 字符串函数 strspn 返回一个字符串的初始字符串的长度，该字符串仅由在另一个字符串中包含的字符仅由该子串组成。
- 字符串函数 strcspn 返回一个字符串的初始字符串的长度，在另一个字符串中包含的字符仅由该字符串组成该字符串仅由另一个字符串中未包含的字符组成。

头文件

- string.h

函数原型

- size_t strspn (const char *s1 , const char *s2) ;
- size_t strcspn (const char *s1 , const char *s2) ;

函数	参数	返回值
strspn	s1： 指向要搜索的字符串的指针 s2： 指向字符串与指定的字符相匹配	字符串 s1 中仅由字符串 s2 中包含的字符构成的字符串的长度
strcspn		字符串 s1 中仅由字符串 s2 中未包含的字符构成的字符串的长度

说明

strspn

- strspn 函数返回 s1 指针所指向的字符串内包含的字符串的长度，该字符串仅由 s2 指针所指向的字符串中包含的字符组成组成。换句话说，如果 s1 字符串和 s2 中指定的任何一个字符都不符合，该函数返回字符串 s1 中第一个字符的索引位置不匹配字符串 s2 中任意字符的字符的位标。
- s2 的空终结符不计入 s2 的长度不当作 s2 的一部分。

strcspn

- strcspn 函数返回 s1 指针所指向的字符串内包含 指向的字符串的长度，该字符串仅由 s2 指针所指向的字符串中未包含的字符组成。换句话说，如果 s1 字符串中的字符和 s2 中指定的任何一个字符相符的话，该函数返回字符串 s1 中第一个匹配字符串 s2 中任意字符的索引位置字符的位标。
- s2 的空终结符不计入 s2 的长度不当作 s2 的一部分。

(9) strpbrk

功能

- 字符串函数 strpbrk 返回一个指针，该指针指向要待搜索的字符串中匹配的第一个字符匹配指定字符串中任意字符的第一个字符。

头文件

- string.h

函数原型

- char *strpbrk (const char *s1 , const char *s2) ;

函数	参数	返回值
strpbrk	s1 : 指向要搜索的字符串的指针 s2 : 指向字符串与指定的字符相匹配	如果找到任何相匹配处 : 指向字符串 s1 中的第 1 个字符与字符串 s2 中的任何字符相匹配 如果没有找到任何相匹配处 : 空指针

说明

- strpbrk 函数返回一个指针，指向 s1 指针所指向的字符串中第一个和匹配 s2 指向的字符串中任意字符匹配的字符。
- 如果在字符串 s1 字符串中未发现字符串 s2 字符串中的任何字符，则函数返回空指针。

(10) strstr

功能

- strstr 字符串函数返回一个指针，指向在字符串中搜索到的首次出现的指定字符串。

头文件

- string.h

函数原型

- char *strstr (const char *s1 , const char *s2) ;

函数	参数	返回值
strstr	s1 : 指向要搜索的字符串的指针 s2 : 指向特定字符串的指针	如果 s2 在 s1 内找到 : 指向字符串 s1 首次出现在字符 s2 中的指针。 如果 s2 没有在 s1 内找到 : 空指针 如果 s2 是空指针 : s1 的值

说明

- strstr 函数返回一个指针，指向 s1 指针所指向的字符串中首次出现的 s2 指向的字符串 （除 s2 的空终结符之外）。
- 如果在字符串 s1 中未找到字符串 s2，则函数返回空指针。
- 若字符串 s2 为空字符串串，则函数返回 s1 的值。

(11) strtok

功能

- 字符串函数 strtok 返回的指针指向来自字符串中取出的记号的指针 （将该其字符串分解反汇编为字符串，该字符串中不包含非分界符字符的字符串）。

头文件

- string.h

函数原型

- char *strtok (char *s1 , const char *s2) ;

函数	参数	返回值
strtok	s1 : 获取指向标记的字符串或空指针 s2 : 指向包含分隔符标记的字符串	如果找到 : 指向标记的首个字符 若无记号返回 : 空指针

说明

- 记号为字符串，包含指定字符串中除分界符之外的字符。
- 如果 s1 为空指针，则在先前的 strtok 调用中保存下来的指针所指向的字符串将被拆分反汇编。但是，若保存的指针为空指针，则函数不进行任何操作，返回空指针。
- 若 s1 不是空指针，则 s1 指向的字符串会被反汇编拆分。
- strtok 函数在 s1 指针所 指向的字符串中搜索 s2 指向的字符串不包含的字符。如果未找到字符，函数会将保存的指针更改为空指针并返回该空指针。如果找到字符，该字符就会成为记号的第一个字符。
- 如果找到记号的第一个字符，那么函数就会在记号的第一个字符之后搜索字符串 s2 包含的任一字符。如果这些字符都未找到，那么函数就将保存的指针更改为空指针。如果找到任一字符，则该字符被空字符改写覆盖，且指向下个字符的指针将会被变成保存起来作为保存的指针。
- 函数返回的指针指向记号的第一个字符的指针。

(12) memset

功能

- 存储器函数 memset 用指定字符对存储器中的对象的指定字节数进行初始化。

头文件

- string.h

函数原型

- void *memset (void *s , int c , size_t n) ;

函数	参数	返回值
memset	s : 初始化指向存储体中的目标 c : 字符值分配给各字节 n : 要初始化的字节数	s 的值

说明

- memset 函数首先将 c 指定的字节转换为 unsigned char，然后将该字符的值从 s 指针所指向的对象的开始位置开头起连续赋给 n 个字节。

(13) strerror

功能

- 函数返回一个指针，所指向的位置处存储的字符串用来描述与给定错误号相关联的报错错误信息，并附带有给定的错误编号。

头文件

- string.h

函数原型

- char *strerror (int errnum) ;

函数	参数	返回值
strerror	errnum : 错误号	如果没有与错误编号相关的消息 : 指向描述错误消息的字符串指针 若没有与错误编号相关的信息 : 空指针

说明

- strerror 函数返回一个指针，指向与 errnum 值相关的下列字符串中的之一个。

  0 : "Error 0"

  1 (EDOM) : “参数太大”

  2 (ERANGE) : “结果太大”

  3 (ENOMEM) : “没有足够内存”

  其它情况下，函数返回空指针。

(14) strlen

功能

- 字符串函数 strlen 返回字符串的长度。

头文件

- string.h

函数原型

- size_t strlen (const char *s) ;

函数	参数	返回值
strlen	s : 指向字符串的指针	字符串 s 的长度

说明

- strlen 函数返回 s 指针所指向的以空字符结尾的字符串的长度。

(15) strcoll

功能

- strcoll 根据区域的特定信息对两个字符串进行比较。

头文件

- string.h

函数原型

- int strcoll (const char *s1 , const char *s2) ;

函数	参数	返回值
strcoll	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 : 转换两个不相同的数值中的首个不同字符为 int 型 （s1 字符 -s2 字符）

说明

- CC78K0S 不支持文化领域 （特殊字符）的特定操作。与 strcmp 操作相同。

(16) strxfrm

功能

- strxfrm 根据区域的特定信息对字符串进行转换。

头文件

- string.h

功能

- size_t strxfrm (char *s1 , const char *s2 , size_t n) ;

函数	参数	返回值
strxfrm	s1 : 指向比较字符串的指针 s2 : 指向比较字符串的指针 n : 字符串 s1 包含的最大字符数	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 或更多个，则未定义的 s1 表示数组的内容。

说明

- CC78K0S 不支持文化领域 （特殊字符）的特定操作。与下列函数操作相同。

```
strncpy (s1, s2, c) ;
return (strlen (s2)) ;
```

10.4.6 应用函数

(1) atoi, atol

功能

- 字符串函数 atoi 将十进制整数字符串的内容转换为 int 值。
- 字符串函数 atol 将十进制整数字符串的内容转换为 long int 值。

头文件

- stdlib.h

函数原型

- int atoi (const char *nptr) ;
- long int atol (const char *nptr) ;

函数	参数	返回值
atoi	nptr : 要转换的字符串	如果能适当转换 : int 数值 如果产生正溢出 : INT_MAX (32767) 如果产生负溢出 : INT_MIN (-32768) 若字符串无效 : 0
atol		如果能适当转换 : long int value 正溢出 : LONG_MAX (2147483647) 负溢出 : LONG_MIN (-2147483648) 若字符串无效 : 0

说明

atoi

- atoi 函数将 nptr 指针指向的字符串的第一部分转换为 int 值。
- atoi 函数跳过字符串开头的零个 （或更多个）空白字符 （使 isspace 为真），从跳过的空白字符之后的下一个字符起将字符串转换为整型 （直到字符串中出现非数位数字符号或空字符串结束字符为止）。如果在字符串中未发现要可以转换的数字符号位，那么函数就会返回 0。若出现上溢，在正上溢情况下函数返回 INT_MAX （32767），在负上溢情况下函数返回 INT_MIN （-32768）。

atol

- atol 函数将 nptr 指针指向的字符串的第一部分转换为 long int 值。
- atol 函数跳过字符串开头的零个（或更多个）空白字符（使 isspace 为真），从跳过的空白字符之后的下一个字符起将字符串转换为整型（直到字符串中出现非数位数字符号或空字符串结束字符为止）。如果在字符串中未发现要可以转换的数字符号位，那么函数就会返回 0。若出现上溢，在正上溢情况下函数返回 LONG_MAX（2147483647），在负上溢情况下函数返回 LONG_MIN（-2147483648）。

(2) strtol, strtoul

功能

- 字符串函数 strtol 将字符串转换为 long 整型。
- 字符串函数 strtoul 将字符串转换为 unsigned long 整型。

头文件

- stdlib.h

函数原型

- long int strtol (const char *nptr , char **endptr, int base) ;
- unsigned long int strtoul (const char *nptr, char **endptr, int base) ;

函数	参数	返回值
strtol	nptr : 要转换的字符串 endptr : char 指针地址 基数 : 放置在字符串中的数字基数	如果能适当转换 : long int value 正溢出 : LONG_MAX (2147483647) 负溢出 : LONG_MIN (-2147483648) 如果不能转换 : 0
strtoul		如果能适当转换 : 无符号长整型 当产生溢出时 : ULONG_MAX (4294967295U) 如果不能转换 : 0

说明

strtol

- strtol 函数将 nptr 指针指向的字符串拆分为下列三个部分 :

  (i) 空白字符串，可能为空 （由 isspace 来判定指定）

  (ii) 以由 base 值确定的基来表示的整型

  (iii) 无法识别的一个或多个 （或更多）字符组成的字符串 （包括空字符串结束符）

  strtol 函数将字符串的第 (ii) 部分转换为整型，返回该整型值。

- 若 base 为 0，表示 base 应由字符串中靠前的数位决定。0x 或 0X 在前表示是十六进制数；以 0 开头的数字 0 在前表示八进制数；其它情况下，该数被认为是十进制数。（在这种情况下，此数可能带符号）。
- 若 base 为 2 至 36，则从 a 至 z 或 A 至 Z 的字母集可以作为数字的部分 （这个数字可能带符号），这些基被 （可能为这些基中的一个数且可能带符号）用来代表 10 至 35。若 base 为 16，则会忽略在前的 0x 或 0X。
- 若 endptr 不是空指针，则指向字符串第 (iii) 部分的指针存储在 endptr 指向的对象中。

- 若正确的值引起了上溢，则根据符号，在正上溢情况下函数返回 LONG_MAX （2147483647），在负上溢情况下返回 LONG_MIN （-2147483648），并将 errno 设置给 ERANGE (ii)。
- 若字符串 (ii) 为空或字符串 (ii) 的首个非空白字符在给定的基下不适合当作不适于整型使用，则函数不进行转换并返回 0。在这种情况下， nptr 字符串的值存储在由 endptr 指向的对象中 （如果不是非空字符串的话）。这一点适用于 bases 0 及 2 至 36。

strtoul

- strtoul 函数将 nptr 指针指向的字符串拆分为下列三个部分 :

  (i) 空白字符串，可能为空 （由 isspace 来判定指定）

  (ii) 以由 base 值确定的基来表示的整型

  (iii) 无法识别的一个或多个 （或更多）字符组成的字符串 （包括空字符串结束符）

  strtoul 函数将字符串的第 (ii) 部分转换为无符号整型，返回该无符号整型值。

- 若 base 为 0，表示 base 应由字符串中靠前的数位决定。 0x 或 0X 在前表示是十六进制数；以 0 开头的数字 0 在前表示八进制数；其它情况下，该数被认为是十进制数。
- 若 base 为 2 至 36，则从 a 至 z 或 A 至 Z 的字母集可以作为数字的部分 （这个数字可能带符号），这些基被 （可能为这些基中的一个数且可能带符号）用来代表 10 至 35。若 base 为 16，则会忽略在前的 0x 或 0X。
- 若 endptr 不是空指针，则指向字符串第 (iii) 部分的指针存储在 endptr 指向的对象中。
- 若正确的值引起了上溢，则函数返回 ULONG_MAX （4294967295U）并将 errno 设置给 ERANGE (ii)。
- 若字符串 (ii) 为空或字符串 (ii) 的首个非空白字符在给定的基下不适合当作不适于整型使用，则函数不进行转换并返回 0。在这种情况下， nptr 字符串的值存储在由 endptr 指向的对象中 （如果不是非空字符串的话）。这一点适用于 bases 0 及 2 至 36。

(3) calloc

功能

- 存储器函数 calloc 分配为某个数组分配区域，并将该区初始化为 0。

头文件

- stdlib.h

函数原型

- void *calloc (size_t nmemb , size_t size) ;

函数	参数	返回值
calloc	nmemb : 数组中成员的数量 大小： 各成员的大小	如果已分配需要的大小 : 指向分配区域的起始位置 如果未分配需要的大小 : 空指针

说明

- calloc 函数给数组分配存储区，并将该区初始化为零，此数组由 n 个成员 （由 nmemb 指定）组成，各成员所占用的字节数由 size 指定。
- 若分配了所请求的大小，则返回指向所已分配区域开始处的指针。
- 若未能分配所请求的大小，则返回空指针。
- 存储器的分配从间中断值处开始，紧邻所分配空间的下一个地址将成为新的间断中断值。存储器函数 brk 的间断值设置见 10.4.6 应用函数 (11) brk, sbrk 存储器分配会从中断值开始，所分配区域的下一个地址将成为新的中断值。

(4) free

功能

- 存储器函数 free 会释放已分配的存储块。

头文件

- stdlib.h

函数原型

- void free (void *ptr) ;

函数	参数	返回值
free	ptr : 指针指向将要释放的存储块开头位置的指针	None

说明

- free 函数释放由 ptr 指向的分配空间（在中间断值前）。(在释放区域调用 malloc, calloc, 或 realloc，释放区域将给你更早释放的空间。)
- 若 ptr 不未指向已分配空间，则 free 不会有任何效果动作。（通过将 ptr 设置为新的中间断值，对已分配的空间进行释放。

(5) malloc

功能

- 存储器函数 malloc 分配存储块。

头文件

- stdlib.h

函数原型

- void *malloc (size_t size) ;

函数	参数	返回值
malloc	大小： 要分配的存储块的大小	如果已分配需要的大小： 指向分配区域的起始位置 如果未分配需要的大小： 空指针

说明

- malloc 函数会分配一个将存储块，其大小分配给由 size 指定的字节数决定，并返回一个指向所已分配区域首字节的指针。
- 若无法分配存储器，则函数返回空指针。
- 存储器的分配从间断值处开始，紧邻所分配空间的下一个地址将成为新的间断值。存储器函数 brk 的间断值设置见 10.4.6 应用函数 (11) brk, sbrk 存储器分配会从中断值开始，所分配区域的下一个地址将成为新的中断值。

(6) realloc

功能

- 存储器函数 realloc 重新分配存储块 （即，改变已分配存储区的大小）。

头文件

- stdlib.h

函数原型

- void *realloc (void *ptr , size_t size) ;

函数	参数	返回值
realloc	ptr : 指向以前分配块起始位置的指针 大小： 给这个块定义新的大小	如果重分配需要的大小： 指向重分配区域的起始位置 如果 ptr 是空指针： 指向分配空间的起始位置 若所请求的大小未得到重新分配，或 “ptr” 不是空指针： 空指针

说明

- realloc 函数将由 ptr 指向的分配空间 （在中断值前）的大小改变为由 size 指定的值。若 size 的值大于已分配空间的大小，那么在原长度之内的已分配空间的内容会保持不变。 realloc 函数仅对增加的空间进行分配。如果长度值小于已分配空间的大小，那么函数会释放已分配空间中减少的空间。
- 若 ptr 为空指针，则 realloc 函数会重新分配一块指定 size 的存储块 （与 malloc 相同）。
- 如果 ptr 不指向先前分配的存储块或如果无法分配存储块，那么该函数就无法不会执行，而是返回空指针。
- 重新分配的方式为，将 ptr 的地址加上 size 指定的字节数设置为新的中间断值。

(7) abort

功能

- 程序控制函数中止会立即造成程序的异常终止。

头文件

- stdlib.h

函数原型

- void abort (void) ;

函数	参数	返回值
abort	None	不返回其调用方。

说明

- 中止函数循环运行，则无法返回到其调用方。
- 用户必须创建中止处理例程。

(8) atexit, exit

功能

- atexit 对在正常终止中时调用的函数进行注册。
- exit 使程序终止。

头文件

- stdlib.h

函数原型

- int atexit (void (*func) (void)) ;
- void exit (int status) ;

函数	参数	返回值
atexit	func : 指向要注册的函数的指针	如果函数注册为完成函数 : 0 如果无法注册函数： 1
exit	status : 指示终止的状态值	exit 不能返回。

说明

atexit

- atexit 函数注册由 func 指向的收卷完整结束函数，使得通过调用 exit 或从 main 返回的正常程序终止，该函数可以在正常程序终止时无参数的情况下被调用，正常程序终止可以是调用 exit 或从 main 返回而引起的。
- 可建立最多 32 个收卷完整结束函数。若完成函数可以注册，则 atexit 返回 0。若因已注册满了 32 个完成函数而不能再注册更多完成函数，则函数返回 1。

exit

- exit 函数会立即造成程序的正常终止。
- 该函数调用收卷完整结束函数，调用顺序与用 atexit 注册时相反。
- exit 函数循环运行，永远无法返回其调用器调用方。
- 用户必须创建 exit 处理例程。

(9) abs, labs

功能

- 数学函数 abs 返回其 int 类型参数的绝对值。
- 数学函数 labs 返回其 long 类型参数的绝对值。

头文件

- stdlib.h

函数原型

- int abs (int j) ;
- long int labs (long int j) ;

函数	参数	返回值
abs	j : 获取符号整数的绝对值	如果 j 应列入 -32767 < j < 32767 范围 : j 的绝对值 如 j 为 -32768 : -32768 (0x8000)
labs		如果 j 应列入 <-2147483647 < j 2147483647 范围 : j 的绝对值 如果 j 的值为 -2147483648 : -2147483648 (0x80000000)

说明

abs

- abs 返回其 int 类型参数的绝对值。
- 若 j 为 -32768，函数返回 -32768。

labs

- labs 返回其 long 类型参数的绝对值。
- 若 j 的值为 -2147483648，则函数返回 -2147483648。

(10) div (仅普通模式), ldiv (仅普通模式)

功能

- 数学函数 div 进行用分子除以分母的整数除法。
- 数学函数 ldiv 进行用分子除以分母的长整数除法。

头文件

- stdlib.h

函数原型

- div_t div (int numer , int denom) ;
- ldiv_t ldiv (long int numer , long int denom) ;

函数	参数	返回值
div	numer : 除法的分子 分母 : 除法的分母	商返回到 div_t 类型成员的 quot 单元中，余数返回到其 rem 单元中。
ldiv		商返回到 ldiv_t 类型成员的 quot 单元中，余数返回到其 rem 单元中。

说明

div

- div 函数进行分子除以分母的整数除法。
- 商的绝对值定义为，不大于分子的绝对值除以分母的绝对值所得的最大整数。余数的符号总是与除法结果相同 （若分子和分母符号相同则为正；否则为负）。
- 余数为分子柏帜 ³* 商的值。
- 若分母为 0，则商为 0，余数为分子。
- 若分子为 -32768 且分母为 -1，则商为 -32768，余数为 0。

ldiv

- ldiv 函数进行分子除以分母的长整数除法。
- 商的绝对值定义为，不大于分子的绝对值除以分母的绝对值所得值的最大 long int 类型整数。余数的符号总是与除法结果相同 （若分子和分母符号相同则为正；否则为负）。
- 余数为分子柏帜 ³* 商的值。
- 若分母为 0，则商为 0，余数为分子。
- 若分子为 -2147483648 且分母为 -1，则商为 -2147483648，余数为 0。

(11) brk, sbrk

功能

- 存储器函数 brk 设置中断值。
- 存储器函数 sbrk 增加或减小设置的中断值。

头文件

- stdlib.h

函数原型

- int brk (char *endds) ;
- char *sbrk (int incr) ;

函数	参数	返回值
brk	endds : 用来设置释放存储块释放位置的中断值	如果正确设置中断值 : 0 如果无法改变中断值 : -1
sbrk	incr : 设置的中断值要增加 / 减小的值 （字节）	如果正常增加或减少 : 原中断值 如果原中断值无法增加或减小 : -1

说明

brk

- brk 函数将 endds 给出的值设置为中断值 （紧邻已分配存储块末尾地址的下一个地址）。
- 若 endds 在许可的地址范围之外，则函数不设置中断值并将 errno 设置给 ENOMEM (3)。

sbrk

- sbrk 函数用 incr 指定的字节数增加或减小设置的中断值。（增加还是减小，由 incr 的正负号决定。
- 若增加或减小后的中断值处于许可地址范围之外，则函数不改变原中断值，并将 errno 设置给 ENOMEM (3)。

(12) atof, strtod

功能

- 字符串函数 atof 将十进制整数字符串的内容转换为 double 值。
- 字符串函数 strtod 将字符串的内容转换为 double 值。

头文件

- stdlib.h

函数原型

- double atof (const char *nptr) ;
- double strtod (const char *nptr , char **endptr) ;

函数	参数	返回值
atof	nptr : 要转换的字符串	如果正确转换 : 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 若字符串无效 : 0
strtod	nptr : 要转换的字符串 endptr : 指向不可识别块的存储指针	如果能适当转换 : 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 若字符串无效 : 0

说明

atof

- atof 函数将指针 nptr 指向的字符串转换为 double 值。
- atof 函数从字符串开头起跳过零个 （或更多）个空白字符 （使 isspace 为真）并从跳过的空白字符之后的下一个字符起将该字符串转换为浮点数 （直到字符串中出现非数字符号或空字符串结束符）。
- 当正确转换返回一个浮点数。
- 如果在转换时出现上溢，则会返回带上溢值符号的 HUGE_VAL 且 ERANGE 设置给 errno。
- 如果由于下溢或上溢而删除了有效位，那么分别会返回非规格化数和 +0，且 ERANGE 设置给 errno。
- 如果转换无法进行，则返回 0。

strtod

- strtod 函数将 nptr 指针指向的字符串转换为 double 值。
- strtod 函数从字符串开头起跳过零个 （或更多）个空白字符 （使 isspace 为真）并从跳过的空白字符之后的下一个字符起将该字符串转换为浮点数 （直到字符串中出现非数字符号或空字符串结束符非数字或空字符为止）。
- 当正确转换返回一个浮点数。
- 如果在转换时出现上溢，则会返回带上溢值符号的 HUGE_VAL 且 ERANGE 设置给 errno。
- 如果由于下溢或上溢而删除了有效位，那么分别会返回非规格化数和 ±0，且 ERANGE 设置给 errno。此外，并且此这时候 endptr 存储的下下一个字符串的指针指向下一个字符串。
- 如果转换无法进行，则返回 0。

(13) itoa, ltoa (仅普通模式), ultoa (仅普通模式)

功能

- itoa 字符串函数将 int 整数转换为对应的字符串。
- 字符串函数 ltoa 将 long int 整数转换为对应的字符串。
- 字符串函数 ultoa 将 unsigned long 整数转换为对应的字符串。

头文件

- stdlib.h

函数原型

- char *itoa (int value , char *string , int radix) ;
- char *ltoa (long value , char *string , int radix) ;
- char *ultoa (unsigned long value , char *string , int radix) ;

函数	参数	返回值
itoa, ltoa, ultoa	value : 转换字符串为整数型 字符串 : 指向转换结果的指针 基数 : 输出字符串的基数	如果能适当转换 : 指向转换字符串的指针 如果不能适当转换 : 空指针

说明

itoa , ltoa , ultoa

- itoa、 ltoa 和 ultoa 函数均将由 value 指定的整数值转换为对应的字符串 （该字符串用空字符终止）并将结果存储在由 “string”指向的区域中。
- 输出字符串的由 radix 决定， radix 必须在 2 至 36 范围内。各函数均按照指定的 radix 进行转换，返回指向转换后字符串的指针。若指定的基数不在 2 至 36 范围内，则函数不进行转换，并返回空指针。

(14) rand, srand

功能

- 数学函数 rand 生成伪随机数序列。
- 数学函数 srand 对由 rand 生成的序列进行初始值 （seed）设置。

头文件

- stdlib.h

函数原型

- int rand (void) ;
- void srand (unsigned int seed) ;

函数	参数	返回值
rand	None	从 0 至 RAND_MAX 的伪随机整数
srand	seed : 伪随机数生成器的起始值	None

说明

rand

- 每次调用 rand 函数时，它都会返回 0 至 RAND_MAX 范围内的一个伪随机整数。

srand

- srand 函数为随机数序列设置初始值。seed 用来设置随机数级数计算过程 （调用 rand 时的返回值）的起点，也是调用 rand 时的返回值。如果使用了相同的 seed 值，那么在再次调用 srand 时会得到相同的伪随机数序列。
- 在使用 srand 设置 seed 之前调用 rand ，与在已调用 srand 之后 且 seed = 1 时再调用 rand 相同。（默认 seed 为 1。

(15) bsearch (仅正常模式)

功能

- bsearch 函数进行二分检索。

头文件

- stdlib.h

函数原型

- void *bsearch (const void *key , const void *base , size_t nmemb , size_t size ,
  int (*compare) (const void *, const void *)) ;

函数	参数	返回值
bsearch	key : 建立指向查询关键字的指针 基数 : 指向包含查找信息的排序数组指针 nmemb : 数组元素的个数 大小： 数组大小 比较： 指向函数的指针用于比较 2 个关键字	如果数组包含关键字 : 指向首个成员的指针与 "key" 匹配 若数组中不包含关键字 : 空指针

说明

- bsearch 函数对 base 指向的排序数组进行二分检索，返回的指针指向匹配 key 指向的关键字的首个成员。
  base 指向的数组必须由 nmemb 个数成员组成，各成员具有由 size 指定的长度，且必须按升序排列。
- compare 指向的函数取得两个参数 （第一参数为 key ， 第二参数为数组元素）进行比较，并返回：

  若第一参数小于第二参数，返回负值。

  若两个参数相等，返回 0。

  若第一参数大于第二参数，返回正整数。
- 当指定了 -ZR 选项时，被转递给 bsearch 函数的函数类型必须为是 pascal 函数。

(16) qsort (仅正常模式)

功能

- qsort 函数用 quicksort 算法对指定数组的成员进行排序。

头文件

- stdlib.h

函数原型

- void qsort (void *base , size_t nmemb , size_t size , int (*compare) (const void * , const void *)) ;

函数	参数	返回值
qsort	基数 : 指向排序的数组 nmemb : 数组中成员的数量 大小： 数组成员大小 比较： 指向函数的指针用于比较 2 个关键字	None

说明

- qsort 函数对 base 指向的数组的成员进行升序排序。由 base 指向的数组由 nmemb 个数成员组成，各成员的长度由 size 指定。
- compare 指向的函数取出得两个参数 （数组元素 1 和 2）进行比较，并返回情况如下：
- 返回数组元素 1 作为第一参数，数组元素 2 作为第二参数。

  若第一参数小于第二参数，返回负值。

  若两个参数相等，返回 0。

  若第一参数大于第二参数，返回正整数。
- 若两个数组元素相等，则会把靠近数组顶部的元素排在前面。
- 当指定了 -ZR 选项时，被转递给 qsort 函数的函数类型必须为 pascal 函数。

(17) strbrk

功能

- strbrk 设置中断值。

头文件

- stdlib.h

函数原型

- int strbrk (char *endds) ;

函数	参数	返回值
strbrk	endds : 设置中断值	正常 : 0 当无法改变中断值时 : -1

说明

- 将 endds 给出的值设置给为中断值 （位于要分配区域末尾地址后面的地址）。
- 当 endds 超过许可范围时，不会改变中断值。 ENOMEM(3) 设置给 errno ，返回 -1。

(18) strsbrk

功能

- strsbrk 增大 / 减小中断值。

头文件

- stdlib.h

函数原型

- char *strsbrk(int incr);

函数	参数	返回值
strsbrk	incr : 合计增加 / 减少中断值	正常 : 原中断值 当无法增加 / 减少中断值时 : -1

说明

- incr 字节增大 / 减小中断值 （根据 incr 的符号）。
- 若中断值在增大 / 减小后超出许可范围，则中断值不会改变。 ENOMEM(3) 设置给 errno ， 返回 -1。

(19) stritoa, strltoa (normal model only), strultoa (normal model only)

功能

- stritoa 将 int 转换为字符串。
- strltoa 将 long 转换为字符串。
- strultoa 将 unsigned long 转换为字符串。

头文件

- stdlib.h

函数原型

- char *stritoa (int value , char *string , int radix) ;
- char *strltoa (long value , char *string , int radix) ;
- char *strultoa (unsigned long value , char *string , int radix) ;

函数	参数	返回值
stritoa, strltoa, strultoa	value : 转换字符串 字符串 : 指向转换结果的指针 基数 : 设定基数	正常 : 指向转换字符串的指针 其它： 空指针

说明

stritoa、 strltoa、 strultoa

- 将指定的数值 value 转换为以空字符串结束符结尾的字符串，并将结果存储在由字符串 string 指定的区域中。进行转换时，使用指定的 radix，会返回的指针会指向转换后字符串的指针。
- radix 必须在 2 至 36 范围内。在其它情况下，不会进行转换，并返回空指针。

10.4.7 数学函数

(1) acos (仅正常模式)

功能

- acos 计算反余弦。

头文件

- math.h

函数原型

- double acos (double x) ;

函数	参数	返回值
acos	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 （从 0 到 π 范围内）。
- 当 x 非数值时，返回 NaN。
- 当出现 x< -1, 1 <x 的定义域错误时，返回 NaN ， 设置 EDOM 。

(2) asin (仅正常模式)

功能

- asin 计算反正弦。

头文件

- math.h

函数原型

- double asin (double x) ;

函数	参数	返回值
asin	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 (范围在 -π /2 和 +π /2 之间) of x.
- 当出现 x< -1, 1 <x 的数定义域错误时，返回 NaN， EDOM 设置给 errno。
- 当 x 非数值时，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。

(3) atan (仅正常模式)

功能

- atan 计算反正切。

头文件

- math.h

函数原型

- double atan (double x) ;

函数	参数	返回值
atan	x : 进行数字值运算	正常 : x 的 atan 当 x = NaN : NaN 当 x = -0 : -0

说明

- 计算 x 的 atan (在 -π /2 和 +π /2 之间) 的值。
- 当 x 非数值时，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。

(4) atan2 (仅正常模式)

功能

- atan2 计算 y/x 的反正切。

头文件

- math.h

函数原型

- double atan2 (double y , double x) ;

函数	参数	返回值
atan2	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 两个都为 0 或 不能表达 y/x 的值 , 或无论 x 或 y 为 NaN 以及 x 和 y 两者为 + ∞ : 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)。 如果 x 和 y 均为 0 或 y/x 的值为无法表达示的值，或当 x 和 y 均为无穷大时，返回 NaN ，且将 EDOM 设置给 errno。
- 如果 x 或 y 为非数值，则返回 NaN。
- 如果操作结果中产生下溢出，返回非正常数字。

(5) cos (仅正常模式)

功能

- cos 计算余弦值。

头文件

- math.h

函数原型

- double cos (double x) ;

函数	参数	返回值
cos	x : 进行数字值运算	正常 : x 的 cos 当 x = NaN, x = + ∞ : NaN

说明

- 计算 x 的余弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(6) sin (仅正常模式)

功能

- sin 计算正弦值。

头文件

- math.h

函数原型

- double sin (double x) ;

函数	参数	返回值
sin	x : 进行数字值运算	正常 : x 的 sin 当 x = NaN, x = + ∞ : NaN 当产生下溢出时： 非正常数字

说明

- 计算 x 的正弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(7) tan (仅正常模式)

功能

- tan 计算正切值。

头文件

- math.h

函数原型

- double tan (double x) ;

函数	参数	返回值
tan	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 的正切。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(8) cosh (仅正常模式)

功能

- cosh 计算双曲余弦。

头文件

- math.h

函数原型

- double cosh (double x) ;

函数	参数	返回值
cosh	x : 进行数字值运算	正常 : x 的 cosh 当发生上溢出时 , x = NaN, x = + ∞ : HUGE_VAL (带正号) x = NaN : NaN

说明

- 计算 x 的双曲余弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，返回正无穷大的值。
- 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。

(9) sinh (仅正常模式)

功能

- sinh 计算双曲正弦。

头文件

- math.h

函数原型

- double sinh (double x) ;

函数	参数	返回值
sinh	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 的双曲正弦。
- 若 x 为非数值，返回 NaN。
- 当 x 是 + ∞时 , + ∞被返回。
- 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。

(10) tanh (仅正常模式)

功能

- tanh 计算双曲正切。

头文件

- math.h

函数原型

- double tanh (double x) ;

函数	参数	返回值
tanh	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 的双曲正切。
- 若 x 为非数值，返回 NaN。
- 如果 x 是 + ∞ , +1 被返回。
- 如果运行结果中产生下溢出，返回 +0。

(11) exp (仅正常模式)

功能

- exp 计算指数函数。

头文件

- math.h

函数原型

- double exp (double x) ;

函数	参数	返回值
exp	x : 进行数字值运算	正常 : x 的指数函数 当 x = NaN : NaN 当 x = + ∞ : + ∞ 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 的指数函数。
- 若 x 为非数值，返回 NaN。
- 当 x 是 + ∞时 , + ∞被返回。
- 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，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。

(12) frexp (仅正常模式)

功能

- frexp 计算尾数和指数部分。

头文件

- math.h

函数原型

- double frexp (double x , int *exp) ;

函数	参数	返回值
frexp	x : 进行数字值运算 exp : 指针指向存储指数部分	正常 : x 的尾数 当 x = NaN, x = + ∞ : NaN 当 x = +0 : +0

说明

- 分隔浮点数 x 为尾数部分 m 和指数部分 n 例如：x = m * 2 ^ n 并返回尾数部分 m。
- 指数 n 存储在指针 exp 指示的位置。 但是， m 的绝对值大于等于 0.5 小于 1.0。
- 若 x 为非数值，返回 NaN 且 *exp 值为 0。
- 若 x 为无穷大，则返回 NaN，将 EDOM 设置给 errno ，且 *exp 的值为 0。
- 如果 x 为 +0, +0 被返回，并且 *exp 的值为 0.

(13) ldexp (仅正常模式)

功能

- ldexp 发现 x * 2 ^ exp。

头文件

- math.h

函数原型

- double ldexp (double x , int exp) ;

函数	参数	返回值
ldexp	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.
- 若 x 为非数值，返回 NaN。
- 当 x 是 + ∞时 , + ∞被返回。
- 如果 x 是 +0, +0 被返回。
- 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。

(14) log (仅正常模式)

功能

- log 计算自然对数。

头文件

- math.h

函数原型

- double log (double x) ;

函数	参数	返回值
log	x : 进行数字值运算	正常 : x 的自然对数 当 x < 0 : HUGE_VAL (带负号) 当 x 为非数字时 : NaN 当 x 为无穷时 : + ∞

说明

- 发现 x 的自然对数。
- 若 x 为非数值，返回 NaN。
- 如果 x 为 + ∞ , + ∞被返回。
- 当出现 x<0 的定义域错误时，返回带负号的 HUGE_VAL 且将 EDOM 设置给 errno。
- 若 x = 0，返回带负号的 HUGE_VAL 且将 ERANGE 设置给 errno。

(15) log10 (仅正常模式)

功能

- log10 计算以 10 为底的对数。

头文件

- math.h

函数原型

- double log10 (double x) ;

函数	参数	返回值
log10	x : 进行数字值运算	正常 : 发现 x 为 10 作为底的对数。 当 x < 0 : HUGE_VAL (带负号) 当 x 为非数字时 : NaN 当 x 为无穷时 : + ∞

说明

- 发现 x 为 10 作为底的对数。
- 若 x 为非数值，返回 NaN。
- 如果 x 为 + ∞ , + ∞被返回。
- 当出现 x<0 的定义域错误时，返回带负号的 HUGE_VAL 且将 EDOM 设置给 errno。
- 若 x = 0，返回带负号的 HUGE_VAL 且将 ERANGE 设置给 errno。

(16) modf (仅正常模式)

功能

- modf 计算分小数部分和整数部分。

头文件

- math.h

函数原型

- double modf (double x , double *iptr) ;

函数	参数	返回值
modf	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 是 +0 : +0

说明

- 将浮点数 x 分为分数部分和整数部分
- 返回与 x 符号相同的分数部分，将整数部分存储在指针 iptr 指示的位置。
- 若 x 为非数值，返回 NaN 并将其存储在 iptr 指针指示的位置。
- 若 x 为无穷大，返回 NaN 并将其存储在指针 iptr 指示的位置，并将 EDOM 设置给 errno。
- 如果 x = +0, 返回 指针 iptr 表示存储 +0 的位置。

(17) pow (仅正常模式)

功能

- pow 计算 x 的 y 次幂。

头文件

- math.h

函数原型

- double pow (double x , double y) ;

函数	参数	返回值
pow	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 和 y = + , x = 0 和 y < 0 : NaN 当产生下溢出时： 非正常数字 当产生溢出时 : HUGE_VAL (溢出值符号) 当由于下溢出产生有效数字丢失时： +0

说明

- 计算 x ^ y.
- 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。
- 当 x = NaN 或 y = NaN 时，返回 NaN。
- 无论 x = + 和 y = 0, x < 0 和 y integer, x < 0 和 y = + 还是 x = 0 和 y < 0, 返回 NaN 闭关设置 errno 为 EDOM。
- 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 。

(18) sqrt (仅正常模式)

功能

- sqrt 计算平方根。

头文件

- math.h

函数原型

- double sqrt (double x) ;

函数	参数	返回值
sqrt	x : 进行数字值运算	当 x > 0 : x 的平方根 当 x = +0 : +0 当 x < 0 : NaN

说明

- 计算 x 的平方根。
- 在 x <0 的域错误情况下，返回 0 且将 EDOM 设置给 errno。
- 若 x 为非数值，返回 NaN。
- 如果 x 是 +0, +0 被返回。

(19) ceil (仅正常模式)

功能

- ceil 计算不小于 x 的最小整数。

头文件

- math.h

函数原型

- double ceil (double x) ;

函数	参数	返回值
ceil	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 x = + ∞ : 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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 当不小于 x 的最小整数无法表达时，返回 x。

(20) fabs (仅正常模式)

功能

- fabs 返回浮点数 x 的绝对值。

头文件

- math.h

函数原型

- double fabs (double x) ;

函数	参数	返回值
fabs	x : 发现数字值的绝对值	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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。

(21) floor (仅正常模式)

功能

- floor 计算不大于 x 的最大整数。

头文件

- math.h

函数原型

- double floor (double x) ;

函数	参数	返回值
floor	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 x = + ∞ : 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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 若不大于 x 的最大整数无法表达，返回 x。

(22) fmod (仅正常模式)

功能

- fmod 计算 x/y 的余数。

头文件

- math.h

函数原型

- double fmod (double x , double y) ;

函数	参数	返回值
fmod	x : 进行数字值运算 y : 进行数字值运算	正常 : x/y 的余数 When x 为非数字 或 y 为非数字 , 当 y 为 +0, 当 x 为 + ∞ : NaN 当 x ∞ 且 y = + ∞ : x

说明

- 计算 x/y 的余数，表达为 x - i*y。
- 若 y 0，则返回值的符号与 x 相同，且其绝对值小于 y 的绝对值。
- 如果 y 为 + 0 或 x = + ∞ , 返回 NaN 并设置 errno 为 EDOM。
- 若 x 为非数值或 y 为非数值，返回 NaN。
- 若 y 为无穷大，则返回 x （除非 x 也为无穷大）。

(23) matherr (仅正常模式)

功能

- matherr 对浮点数操作库进行异常处理。

头文件

- math.h

函数原型

- void matherr (struct exception *x) ;

函数	参数	返回值

matherr	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。

  如下所示为算术异常类型和 errno 之间的关系。

类型	算数异常	值设置为 errno
1	下溢	ERANGE
2	丢失	ERANGE
3	溢出	ERANGE
4	除零	EDOM
5	不能操作	EDOM

  通过更改或创建 matherr，对原有错误进行处理。

(24) acosf (仅正常模式）

功能

- acosf 计算反余弦。

头文件

- math.h

函数原型

- float acosf (float x) ;

函数	参数	返回值
acosf	x : 进行数字值运算	当 -1 < x < 1 : x 的 acos 当 x < -1, 1 < x, x = : NaN

说明

- 计算 x 的 acos(在 0 和 π 之间) 值。
- 若 x 为非数值，返回 NaN。
- 当出现 x< -1, 1 <x 的数定义域错误时，返回 NaN，设置 errno 为 EDOM。

(25) asinf (仅正常模式)

功能

- asinf 计算反正弦。

头文件

- math.h

函数原型

- float asinf (float x) ;

函数	参数	返回值
asinf	x : 进行数字值运算	当 -1 < x < 1 : x 的 acos 当 x < -1, 1 < x, x = NaN : NaN x = -0 : -0 当产生下溢出时： 非正常数字

说明

- 计算 x 的 asin (在 -π /2 和 +π /2 之间) 的值。
- 若 x 为非数值，返回 NaN。
- 当出现 x< -1, 1 <x 的数定义域错误时，返回 NaN，设置 errno 为 EDOM。
- 若 x = -0，返回 -0。
- 如果运行结果中产生下溢出，返回非正常数字。

(26) atanf (仅正常模式)

功能

- atanf 计算反正切。

头文件

- math.h

函数原型

- float atanf (float x) ;

函数	参数	返回值
atanf	x : 进行数字值运算	正常 : x 的 atan 当 x = NaN : NaN 当 x = -0 : -0

说明

- 计算 x 的 atan (在 -π /2 和 +π /2 之间) 的值。
- 若 x 为非数值，返回 NaN。
- 若 x = -0，返回 -0。
- 如果运行结果中产生下溢出，返回非正常数字。

(27) atan2f(仅正常模式)

功能

- atan2f 计算 y/x 的反正切。

头文件

- math.h

函数原型

- float atan21 (float y , float x) ;

函数	参数	返回值
atan21	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 两个都为 0 或 不能表达 y/x 的值 , 或无论 x 或 y 为 NaN 以及 x 和 y 两者为 + ∞ : NaN 当产生下溢出时： 非正常数字

说明

- 计算 y/x 的反正切 （在 -π 到 +π 范围内）。当 x 和 y 均为 0 或 y/x 的值无法表达时，或当 x 和 y 均为无穷大时，返回 NaN 且将 EDOM 设置给 errno。
- 当 x 或 y 为非数值时，返回 NaN。
- 如果运行结果中产生下溢出，返回非正常数字。

(28) cosf (仅正常模式)

功能

- cosf 计算余弦。

头文件

- math.h

函数原型

- float cosf (float x) ;

函数	参数	返回值
cosf	x : 进行数字值运算	正常 : x 的 cos 当 x = NaN, x = + ∞ : NaN

说明

- 计算 x 的余弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(29) sinf(仅正常模式)

功能

- sinf 计算正弦。

头文件

- math.h

函数原型

- float sinf (float x) ;

函数	参数	返回值
sinf	x : 进行数字值运算	正常 : x 的 sin 当 x = NaN, x = + ∞ : NaN 当产生下溢出时： 非正常数字

说明

- 计算 x 的正弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(30) tanf (仅正常模式)

功能

- tanf 计算正切。

头文件

- math.h

函数原型

- float tanf (float x) ;

函数	参数	返回值
tanf	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 的正切。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 如果运行结果中产生下溢出，返回非正常数字。
- 若 x 的绝对值极大，则操作结果的值几乎无意义。

(31) coshf (仅正常模式)

功能

- coshf 计算双曲余弦。

头文件

- math.h

函数原型

- float coshf (float x) ;

函数	参数	返回值
coshf	x : 进行数字值运算	正常 : x 的 cosh 当发生上溢出时 , x = + ∞ : HUGE_VAL (带正号) x = NaN : NaN

说明

- 计算 x 的双曲余弦。
- 若 x 为非数值，返回 NaN。
- 若 x 为无穷大，返回正无穷大值。
- 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。

(32) sinhf(仅正常模式)

功能

- sinhf 计算双曲正弦。

头文件

- math.h

函数原型

- float sinhf (float x) ;

函数	参数	返回值
sinhf	x : 进行数字值运算	正常 : x 的 sinh 当产生溢出时 : HUGE_VAL (溢出值符号) x = NaN : NaN 当 x = + ∞ : + ∞ 当产生下溢出时： +0

说明

- 计算 x 的双曲正弦。
- 若 x 为非数值，返回 NaN。
- 当 x 是 + ∞时 , + ∞被返回。
- 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。

(33) tanhf (仅正常模式)

功能

- tanhf 计算双曲正切。

头文件

- math.h

函数原型

- float tanhf (float x) ;

函数	参数	返回值
tanhf	x : 进行数字值运算	正常 : x 的 tanh x = NaN : NaN 当 x = + ∞ : +1 当产生下溢出时： +0

说明

- 计算 x 的双曲正切。
- 若 x 为非数值，返回 NaN。
- 如果 x 是 + ∞ , +1 被返回。
- 如果运行结果中产生下溢出，返回 ±0。

(34) expf(仅正常模式)

功能

- expf 计算指数函数。

头文件

- math.h

函数原型

- float expf (float x) ;

函数	参数	返回值
expf	x : 进行数字值运算	正常 : x 的指数函数 当产生溢出时 : HUGE_VAL (带正号) x = NaN : NaN 当 x = + ∞ : + ∞ 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 的指数函数。
- 若 x 为非数值，返回 NaN。
- 当 x 是 + ∞时 , + ∞被返回。
- 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。

(35) frexpf (仅正常模式)

功能

- frexpf 计算尾数和指数部分。

头文件

- math.h

函数原型

- float frexpf (float x , int *exp) ;

函数	参数	返回值
frexpf	x : 进行数字值运算 exp : 指针指向存储指数部分	正常 : x 的尾数 当 x = NaN, x = + ∞ : NaN 当 x = +0 : +0

说明

- 分隔浮点数 x 为尾数部分 m 和指数部分 n 例如：x = m * 2 ^ n 并返回尾数部分 m。
- 指数 n 存储在指针 exp 指示的位置。 但是， m 的绝对值大于等于 0.5 小于 1.0。
- 若 x 为非数值，返回 NaN 且 *exp 值为 0。
- 若 x 为 + ∞，返回 NaN 且将 EDOM 设置给 errno， *exp 的值为 0。
- 如果 x 为 +0, +0 被返回，并且 *exp 的值为 0.

(36) ldexpf(仅正常模式)

功能

- ldexpf 发现 x * 2 ^ exp。

头文件

- math.h

函数原型

- float ldexpf (float x , int exp) ;

函数	参数	返回值
ldexpf	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.
- 若 x 为非数值，返回 NaN。 当 x 是 + ∞时 , + ∞被返回。如果 x 是 +0, +0 被返回。
- 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。

(37) logf (仅正常模式)

功能

- log 计算自然对数。

头文件

- math.h

函数原型

- float logf (float x) ;

函数	参数	返回值
logf	x : 进行数字值运算	正常 : x 的自然对数 当 x 为非数字时 : NaN 当 x 为无穷时 : + ∞ 当 x < 0 : HUGE_VAL (带负号)

说明

- 发现 x 的自然对数。
- 若 x 为非数值，返回 NaN。
- 如果 x 为 + ∞ , + ∞被返回。
- 当出现 x<0 的定义域错误时，返回带负号的 HUGE_VAL 且将 EDOM 设置给 errno。
- 若 x = 0，返回带负号的 HUGE_VAL 且将 ERANGE 设置给 errno。

(38) log10f (仅正常模式)

功能

- log10f 计算以 10 为底的对数。

头文件

- math.h

函数原型

- float log10f (float x) ;

函数	参数	返回值
log10f	x : 进行数字值运算	正常 : 发现 x 为 10 作为底的对数。 当 x 为非数字时 : NaN 当 x = + ∞ : + ∞ 当 x < 0 : HUGE_VAL (带负号)

说明

- 发现 x 为 10 作为底的对数。
- 若 x 为非数值，返回 NaN。
- 如果 x 为 + ∞ , 返回 + ∞。
- 当出现 x<0 的定义域错误时，返回带负号的 HUGE_VAL 且设置给 errno 为 EDOM。
- 若 x = 0，返回带负号的 HUGE_VAL 且将 ERANGE 设置给 errno。

(39) modff(仅正常模式)

功能

- modff 计算分小数部分和整数部分。

头文件

- math.h

函数原型

- float modff (float x , float *iptr) ;

函数	参数	返回值
modff	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 = +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。
- 若 x 为非数值，返回 NaN 并将其存储在 iptr 指针指示的位置。
- 若 x 为无穷大，返回 NaN 并将其存储在指针 iptr 指示的位置，并将 EDOM 设置给 errno。
- 如果 x = +0, 返回 +0 并通过指针 iptr 存储显示地址。

(40) powf (normal model only)

功能

- powf 计算 x 的 y 次幂。

头文件

- math.h

函数原型

- float powf (float x , float y) ;

函数	参数	返回值
powf	x : 进行数字值运算 y : 乘数	正常 : x ^ y Either when x = NaN 或 y = NaN x = + ∞ 和 y = 0 x < 0 和 y 整数 , x < 0 且 y = + ∞ x = 0 和 y < 0 : NaN 当产生下溢出时： 非正常数字 当产生溢出时 : HUGE_VAL (溢出值符号) 当由于下溢出产生有效数字丢失时： +0

说明

- 计算 x ^ y.
- 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。
- 当 x = NaN 或 y = NaN 时，返回 NaN。
- 无论 x = + ∞ 和 y = 0, x < 0 和 y integer, x < 0 和 y = + ∞ 还是 x = 0 和 y < 0, 返回 NaN 闭关设置 errno 为 EDOM。
- 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 。

(41) sqrtf (仅正常模式)

功能

- sqrtf 计算平方根。

头文件

- math.h

函数原型

- float sqrtf (float x) ;

函数	参数	返回值
sqrtf	x : 进行数字值运算	当 x > 0 : x 的平方根 当 x = +0 : +0 当 x < 0 : NaN

说明

- 计算 x 的平方根。
- 在 x <0 的域错误情况下，返回 0 且将 EDOM 设置给 errno。
- 若 x 为非数值，返回 NaN。
- 如果 x 是 +0, +0 被返回。

(42) ceilf (仅正常模式)

功能

- ceilf 计算不小于 x 的最小整数。

头文件

- math.h

函数原型

- float ceilf (float x) ;

函数	参数	返回值
ceilf	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 x = + ∞ : 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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 当不小于 x 的最小整数无法表达时，返回 x。

(43) fabsf(仅正常模式)

功能

- fabs 返回浮点数 x 的绝对值。

头文件

- math.h

函数原型

- float fabsf (float x) ;

函数	参数	返回值
fabsf	x : 发现数字值的绝对值	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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。

(44) floorf (仅正常模式)

功能

- floorf 计算不大于 x 的最大整数。

头文件

- math.h

函数原型

- float floorf (float x) ;

函数	参数	返回值
floorf	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。
- 若 x 为非数值，返回 NaN。
- 若 x 为 -0，返回 +0。
- 若 x 为无穷大，则返回 NaN 且将 EDOM 设置给 errno。
- 若不大于 x 的最大整数无法表达，返回 x。

(45) fmodf (仅正常模式)

功能

- fmodf 计算 x/y 的余数。

头文件

- math.h

函数原型

- float fmodf (float x , float y) ;

函数	参数	返回值
fmodf	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 ∞ 且 y = + ∞ : x

说明

- 计算 x/y 的余数，表达为 x ñ i*y。
- 若 y 0，则返回值的符号与 x 相同，且其绝对值小于 y 的绝对值。
- 如果 y 为 + 0 或 x = + ∞ , 返回 NaN 并设置 errno 为 EDOM。
- 若 x 为非数值或 y 为非数值，返回 NaN。
- 若 y 为无穷大，则返回 x （除非 x 也为无穷大）。

10.4.8 诊断函数

(1) __assertfail(仅正常模式)

功能

- __assertfail 支持 assert 宏。

头文件

- assert.h

函数原型

- int __assertfail (char*__msg , char*__cond , char*__file , int__line) ;

函数	参数	返回值
__assertfail	__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 : 源代码行号	未定义

说明

- __assertfail 函数从 assert 宏处接收信息 (参考 10.2 (13) assert.h (仅正常模式)), 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。

10.5 用于启动程序升级的批处理文件和库函数。

CC78K0S 具有用来更新部分标准库函数和启动例程的批处理文件。在 bat 文件夹中的批处理文件显示在以下表 10-18 中。

备注 在运行批处理文件更新库期间，而不是开发，使用 bat 文件夹的文件 d9026.78k。在开发系统时，必须有相关的设备文件 （单独销售）。

表 10-18 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	更新固件 ROM 终止例程 （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 默认将 SFR 的 P0 分配为输入端口。当需要更改此设定时，在 getchar.asm 中更改 PORT 的 EQU 的定义值并用本批处理文件对库进行更新更新库。
repputc.bat	升级 putchar 函数 默认将 SFR 的 P0 设置为输出端口。当需要更改此设定时，在 putchar.asm 中更改 PORT 的 EQU 的定义值并用本批处理文件对库进行更新。
repputcs.bat	更新 putchar 函数，使其支持 SM78K0S。 当需要用 SM78K0S 检查 putchar 函数的输出时，用本批处理文件对库进行更新。
repselo.bat	保存 / 恢复编译器的保留区 （_@KREGxx），且将其作为 setjmp/longjmp 函数保存 / 恢复处理 （默认为非保存 / 恢复）的一必须部分。当设定 -qr 选项时，使用这个批处理文件升级库。
repselon.bat	保存 / 恢复编译器的保留区 （_@KREGxx），不作为 setjmp/longjmp 函数保存 / 恢复处理 （默认为非保存 / 恢复）的一必须部分。当没有设定 -qr 选项时，使用这个批处理文件升级库。
repcmul.bat	更新运行时间库 @@csmul 和 @@cumul ，其包含在乘法库中。 默认时，允许使用在 SFR 中的设备做乘法操作 （MUL0), 且分配到地址 FF10H。 当为乘法使用在 SFR 中的设备 （MUL0)，其分配到除 FF10H 以外的地址上，指定设备，然后使用这个批处理文件更新库。
repimul.bat	在乘法库中添加或删除运行时间库 @@ismul 和 @@iumul。 @@ismul 和 @@iumul, 它们支持乘数，默认不包含。 在使用 @@ismul 和 @@iumul, 使用这个批处理文件添加它们到乘法库中。 在删除添加的 @@ismul 和 @@iumul 时，也使用这个批处理文件。
replmul.bat	在乘法库中添加或删除运行时间库 @@lsmul 和 @@lumul。 @@lsmul 和 @@lumul, 它们支持乘数，默认不包含。 在使用 @@lsmul 和 @@lumul, 使用这个批处理文件添加它们到乘法库中。 在删除添加的 @@lsmul 和 @@lumul 时，也使用这个批处理文件。

10.5.1 使用批处理文件

使用在子文件夹 bat 中的批处理文件。因为这些文件是用来激活汇编程序器和库管理程序的批处理文件，所以需要的环境必须能够能运行 RA78K0S 汇编程序包版本 1.50 或更高版本的环境。在使用批处理文件之前，使用环境变量 PATH 设置包含 RA78K0S 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。

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。这时需要下列参数。

产品类型 = 芯片类型 (目标芯片的分类)

9216 ... u PD789216, etc.

以下是描述如何使用每个批处理文件。

批处理文件：

(1) 启动程序

mkstup 芯片类型

< 示例 >

```
mkstup 9216
```

(2) 固件 ROM 例程的更新

reprom 芯片类型

< 示例 >

```
reprom 9216
```

(3) getchar 函数的更新

repgetc 芯片类型

< 示例 >

```
repgetc 9216
```

(4) putchar 函数的更新

repputc 芯片类型

< 示例 >

```
repputc 9216
```

(5) putchar 函数 (支持 SM78K0S) 的升级

repputcs 芯片类型

< 示例 >

```
repputcs 9216
```

(6) setjmp/longjmp 函数的升级 (恢复 / 保存处理)

repselo 芯片类型

< 示例 >

```
repselo 9216
```

(7) setjmp/longjmp 函数的升级 (无恢复 / 保存处理)

repselon 芯片类型

< 示例 >

```
repselon 9216
```

(8) 运行时间库支持乘数 (@@csmul, @@cumul) 更新

repcmul.bat 芯片类型

< 示例 >

```
repcmul.bat 9832
```

(9) 运行时间库支持乘数 (@@csmul, @@cumul) 添加 / 删除

repimul.bat 芯片类型 add/del

< 示例 >

```
repimul.bat 9832 add
```

(10) 运行时间库支持乘数 (@@lsmul, @@lumul) 添加 / 删除

replmul.bat 芯片类型 add/del

< 示例 >

```
replmul.bat 9832 add
```

第 11 章　扩展函数

本章介绍了该 CC78K0S 特有的扩展函数，这些扩展函数在 ANSI （美国国家标准学会） C 语言标准中未做说明。

此 C 编译器该 CC78K0S 的扩展函数用于产生生成代码，可以帮助用户有高效利用使用 78K0S 系列目标设备。并非所有扩展函数始终有效。 因此，建议用户根据使用目的来选择最有效的扩展函数。为了有效使用这些扩展函数，请结合本章参阅 “第 13 章 编译器的有效应用”。

通过使用 CC78K0S 的这些扩展函数创建的 C 源程序可以更好的利用 K0S 微控制器的特有功能。有关 C 源程序面向其他微控制器的可移植性，他们在 C 语言上是兼容的。因此，使用这些扩展函数开发的 C 源程序可以方便的移植到对于其他微控制器上，并且易于修改。

备注　　在本章介绍过程中，“RTOS”表示 78K/0 系列实时操作系统。

11.1　宏名称

CC78K0S 具有两种类型的宏名称：一类表示目标设备的系列名称，而另一类表示设备名称 （处理器类型）。这些宏名称根据编译时的选项指定，或者在 C 源程序中指定处理器类型，针对指定的目标设备来进行编译生成输出目标代码。如下例中 , 指定了 __K0S__ 和 __9216_ 。

如需有关宏名称的详细信息，请参阅 "9.8 编译程序定义的宏名称 ".

< 示例 >

```
编译选项
        > CC78K0S -c9216 prime.c ...

指定设备类型：
        #pragma pc ( 9216 )
```

11.2 关键字

以下的标记已经作为关键字添加到 CC78K0S 中来实现扩展函数。这些标记与 ANSI-C 关键字一样 , 不能用作标签或变量名称。所有关键字必须以小写字母表示形式出现。因为 C 编译器不会把对应的大写字母解释为的关键字。

以下展示了被添加到 CC78K0S 的关键字列表。对于这些关键字，可以通过指定严格意义的 ANSI-C 语言规范的选项 （-ZA）来禁用那些没有以 “_ _” 开始的关键字 （有关 ANSI-C 关键字信息，请参阅 "2.2 关键字 "）

表 11-1 添加关键字列表

关键字		使用
__callt	callt	callt/__callt 函数
__callf	callf	callf/__callf 函数
__sreg	sreg	sreg/__sreg 变量
	noauto	noauto 函数
__leaf	norec	norec/__leaf 函数
__boolean	boolean	boolean type/__boolean 类型变量
	bit	bit 型变量
__interrupt		硬件中断
__interrupt_brk		软件中断
__asm		ASM 语句
__rtos_interrupt		为 RTOS 分配的句柄
__pascal		Pascal 函数
__directmap		绝对地址分配
__temp		临时变量

注　　标记 callf, __callf, __interrupt_brk, 和 __rtos_interrupt 输出警告且忽略它们。

(1) 函数

关键字 callt, __callt, noauto, norec, __leaf, __interrupt, 和 __pascal 是属性修饰词 .

必须在函数声明之前描述这些关键字。每个属性修饰词的格式如下所示。

```
属性修饰词  通用说明符  函数名称 （参数类型列表 / 标识符列表）
```

< 示例 >

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

属性修饰词规范仅限于以下所列各项。(noauto 和 norec/__leaf 修饰符不能同时指定。) callt 和 __callt, callf 和 __callf, norec 和 __leaf 视作相同规范。但是，即使 -za 选项设定时，也要添加 “__” 修饰符。

- callt
- noauto
- norec
- callt noauto
- callt norec
- noauto callt
- norec callt
- __interrupt
- __pascal
- __pascal noauto
- __pascal callt
- noauto __pascal
- callt __pascal
- callt noauto __pascal

(2) 变量

- 适用于 sreg 或 __sreg 规范的规则对 C 语言中的寄存器同样有效 （详细信息，请参阅 11.5 (3) 如何使用 saddr 区域 (sreg / __sreg)）。
- 适用于 Char 或 int 型分类符的 bit、 boolean 或 _ _boolean 规范的规则对 C 语言中的 char 或 int 型说明符同样有效。

  然而，这些类型仅只能用来可指定对于函数外部定义的变量 （外部变量）。
- 适用于 _ __directmap 规范的规则对 C 语言中有关类型修饰词同样有效 （详细信息，请参阅 11.5 (31) 绝对地址分配规范 (__directmap)）。
- 适用于 __temp 规范的规则对 C 语言中有关类型修饰词同样有效 （详细信息，请参阅 11.5 (33) 临时变量 (__temp) ）。

11.3 存储器

存储模式由目标设备的存储空间决定。

(1) 存储模式

因为存储空间最大为 64 KB，该模式为由代码部和数据部组合的而成的 64 KB。

(2) 寄存器组

不存在寄存器组。

(3) 存储空间

CC78K0S 使用如下所示的存储空间。

(a) 正常模式 （默认）

图 11-1 存储空间的使用 (正常模式)

地址		用途	大小 (字节)
00	40 - 7FH	CALLT 表	64
FE	20 - D7H	sreg 变量，boolean 型变量	184
FE	D8 - E7H	寄存器变量[注 1]	16
FE	E8 - EFH	norec 函数的参数[注 2]	8
FE	F0 - F7H	norec 函数的自动变量[注 3]	8
FE	F8 - FFH	运行时刻库的参数[注 4]	8
FF	00 - FFH	sfr 变量	256

(b) 静态模式 （以 -SM16 规范）

图 11-2 存储空间的使用 (静态模式)

地址		用途	大小 (字节)
00	40 - 7FH	CALLT 表	64
FE	20 - EFH	sreg 变量，boolean 型变量	208
FE	F0 - FFH	共享区域[注 5]	16
FE	20 和 FFH 之间的连续区域	对于参数 , 自动变量 , 和工作区[注 6]	8
FF	00 - FFH	sfr 变量	256

注 1. 未用作于寄存器变量的区域用来存放于 sreg 变量和 boolean 型变量。

注 2. 如果未完全用于本区域的寄存器变量，则存放未用于 norec 函数自变量参数之后剩余的区域用来存放于 sreg 变量和 boolean 型变量。

注 3. 如果未完全用于寄存器变量和 norec 函数自变量参数，则存放 norec 函数参数之后剩余的区域用来存放 sreg 变量和 boolean 型变量。

注 4. 如果未完全用于寄存器变量和 norec 函数自变量参数 / 自变量自动变量，则存放未用于运行时库自变量的区域用于 sreg 变量和 boolean 型变量。

注 5. 由编译器使用的区域根据膧 M 选项指定的参数而会有所变化。这个区域不用作 sreg 和布尔型变量共享的区域。

注 6. 仅当指定静态模式扩展说明选项 （-ZM）时有效。

备注 如果未指定寄存器变量优化选项 （-QR）未指定，则注 1~ 注 3 中的区域始终用于来存放 sreg 变量和 boolean 型变量。

11.4 #pragma 指令

#pragma 指令是 ANSI 支持的预处理指令之一。 #pragma 指令取决于 #pragma 关键字之后的字符串，指示编译器使用由其自己的进行翻译。如果编译器不支持 #pragma 指令，则忽略 #pragma 指令并继续编译。如果通过该指令添加了某个关键字，如果 C 源代码程序中出现该新添的关键字，那么会输出错误。为了避免出现这种情况，应该删除 C 源程序代码中的该关键字或按用 #ifdef 指令对其分类处理。

CC78K0S 支持以下 #pragma 指令，借助以这些指令可以实现扩展功能。

#pragma 之后指定的关键字用大写或小写字母均可表示。

有关使用 #pragma 指令的扩展功能，请参阅 "11.5 如何使用扩展函数 "。

表 11-2 #pragma 指令列表

#pragma 指令	应用
#pragma sfr	在 C 语言中描述 SFR 名称 -> "11.5 (4) 如何使用 sfr 区域 (sfr)"
#pragma asm	在 C 源文件中插入的 ASM 标记 -> "11.5 (8) ASM 语句 (#asm #endasm / __asm)"
#pragma vect #pragma interrupt	C 语言中的中断处理 -> "11.5 (10) 中断函数 (#pragma vect / #pragma interrupt)"
#pragma di #pragma ei	描述 C 源程序中的 DI/EI 指令 -> "11.5 (11) 中断函数 (#pragma DI, #pragma EI)"
#pragma halt #pragma stop #pragma nop	C 语言中 CPU 控制指令的描述 -> "11.5 (12) CPU 控制指令 (#pragma HALT / STOP / NOP)"
#pragma access	使用绝对地址访问函数 -> "11.5 (13) 绝对地址访问函数 (#pragma access)"
#pragma section	改变编译器输出块名和指定块位置。 -> "11.5 (15) 改变编译器输出区名称的函数 (#pragma section ...)"
#pragma name	更改模块名称 -> "11.5 (17) 模块名更改函数 (#pragma name)"
#pragma rot	使用移位函数 -> "11.5 (18) 循环移位函数 (#pragma rot)"
#pragma mul	使用乘法函数 -> 11.5 (19) 乘法函数 (#pragma mul)
#pragma div	使用除法函数 -> 11.5 (20) 除法函数 (#pragma div)
#pragma bcd	使用 BCD 操作函数 -> 11.5 (21) BCD 运算函数 (#pragma bcd)
#pragma opc	使用插入数据函数 -> 11.5 (22) 数据插入函数 (#pragma opc)
#pragma realregister	使用寄存器直接引用函数 -> 11.5 (29) 寄存器直接引用函数 (#pragma realregister)
#pragma inline	扩展标准库 memcpy 和 memset 内联函数 -> 11.5 (30) 内存控制函数 (#pragma inline)

11.5 如何使用扩展函数

扩展函数类型如下所述。

(1) callt 函数 (callt / __callt)

(2) 寄存器变量 (register)

(3) 如何使用 saddr 区域 (sreg / __sreg)

(4) 如何使用 sfr 区域 (sfr)

(5) noauto 函数 (noauto)

(6) norec 函数 (norec)

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

(8) ASM 语句 (#asm #endasm / __asm)

(10) 中断函数 (#pragma vect / #pragma interrupt)

(9) 中断函数修饰符 (__interrupt)

(11) 中断函数 (#pragma DI, #pragma EI)

(12) CPU 控制指令 (#pragma HALT / STOP / NOP)

(13) 绝对地址访问函数 (#pragma access)

(14) 位段声明

(15) 改变编译器输出区名称的函数 (#pragma section ...)

(16) 二进制常量 （二进制常量 0bxxx）

(17) 模块名更改函数 (#pragma name)

(18) 循环移位函数 (#pragma rot)

(19) 乘法函数 (#pragma mul)

(20) 除法函数 (#pragma div)

(21) BCD 运算函数 (#pragma bcd)

(22) 数据插入函数 (#pragma opc)

(23) 静态模式

(24) 类型更改 (-ZI)

(25) Pascal 函数 (__pascal)

(26) 函数调用接口的自动 pascal 函数化 (-zr)

(27) 参数 / 返回值的 int 扩展限制方法 (-ZB)

(28) 数组偏移量计算的简便方法 (-qw2 / -qw4)

(29) 寄存器直接引用函数 (#pragma realregister)

(30) 内存控制函数 (#pragma inline)

(31) 绝对地址分配规范 (__directmap)

(32) 静态模式展开规范 (-zm)

(33) 临时变量 (__temp)

(34) 支持序言 / 尾声的库 (-zd)

本节按以下形式介绍了每一种扩展函数：

函数：	对可以通过扩展函数实现的功能进行概述。
效果：	介绍了扩展函数带来的影响效果。
使用：	介绍了如何使用扩展函数。
示例：	给出了扩展函数的应用示例。
限制：	介绍了有关扩展函数使用的限制。
说明：	介绍了对以上应用示例进行详细说明。
兼容性：	在使用CC78K0S编译C源程序时，介绍用其它C编译器开发的C源程序的兼容性。

(1) callt 函数 (callt / __callt)

功能

- callt 指令存储将会将被调用函数的地址存储在称作 callt 表的区域 [40H 至 7FH] 中，于是使用更短的代码来调用得该函数，比直接调用该函数使用的代码更小。
- 要调用由 callt(或 __callt)(称为 callt 函数) 声明的函数，在使用的函数名称之前需要加 ? 前缀。要调用该函数，请使用 callt 指令。
- 要调用的函数与普通函数并无不同之处。

效果

- 可以缩短目标代码。

用法

- 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
```

示例

```
__callt void func1 ( void );

__callt void func1 ( void ) {
                :
        /* 函数体 */
                :
}
```

限制

- 每一个有 callt/__callt 关键字声明的函数，其地址会在链接目标模块时分配在 callt 表中 。因此，在汇编源代码模块中使用这个 callt 表时，要创建的程序必须用符号说明为 “可重定位”型。
- 链接时会检查 callt 函数的数量。
- 指定 -ZA 选项时， __callt 函数可用，callt 函数不可用。
- callt 表的范围为 40H 至 7FH。
- 当 callt 表中的 callt 属性函数数量超出允许范围时，将出现编译错误。
- 指定 -QL 选项才能够使用 callt 表。因此，每个加载模块所允许的 callt 属性数和链接模块中允许的总数量如表 11-3 所示。
- 当指定使用支持序言 / 结尾的 （-ZD 选项）的库时，不能使用 -QL4 选项。同样，因为正常模式下序言 / 结尾库需要两个 callt 入口，且在静态模式下需要的 callt 入口高达十个，所以正常模式下可用的 callt 入口的数量会减少两个，和静态模式下可用的 callt 入口的数量会减少，减少数量不超过 10 个。

表 11-3 指定 -QL 选项时，可以使用的 callt 属性函数的数量

- 当 -qq 选项并未同时指定时

选项	-ql1	-ql2	-ql3	-ql4
普通模式	30	27	13	0
静态模式	30	29	15	2

- 当同时指定了 -qq 选项时

选项	-ql1	-ql2	-ql3	-ql4
普通模式	30	27	18	11
静态模式	30	29	20	13

- 其中不使用 -ql 选项的且默认选项如下所示的情况。

表 11-4 有关 callt 函数用法的限制

callt 函数	限定值
每个加载模块中允许的数量目	30 最大。
链接的模块中允许的总数量	30 最大。

例

```
(C 程序)
============ ca1.c ============          ============ ca2.c ============
__callt extern  int     tsub ( ) ;

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

```
（编译器的输出对象）
ca1 模块
        EXTRN   ?tsub                   ; 声明
        callt   [ ?tsub ]               ; 调用

ca2 模块
        PUBLIC  _tsub                   ; 声明
        PUBLIC  ?tsub                   ;
@@CALT  CSEG    CALLT0                  ; 分配至段
?tsub : DW      _tsub
@@CODE  CSEG
_tsub :                                 ; 函数定义
                :
                                        ; 函数体
                :
```

说明

- 函数 tsub() 被添加了 callt 属性，使得其可以储存在 callt 表中。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用关键字 callt/__callt，则无需修改 C 源程序。
- 要将函数改变成 callt 属性函数，请注意以上用法中所描述的程序。

< 从 CC78K0S 到其他 C 编译器 >

- 必须使用 #define。详情请参阅 “11.6 C 源代码的修改”。

(2) 寄存器变量 (register)

功能

- 将声明的变量（包括函数参数）分配到寄存器（HL）和 saddr 区 saddr 区域（_@KREG00 至 _@KREG15）。在声明寄存器的模块的预处理 / 后处理期间对 saddr 区域进行保存和恢复。
  在静态模式下，根据引用的次数来进行分配。因此，不能确定定义的寄存器变量被分配到哪个寄存器或 saddr 区域中的具体哪个位置。
- 关于寄存器分配的细节，请参阅 "11.7 函数调用接口 "。
- 在如下所示的编译条件时 , 寄存器变量会被分配到不同的区域（如需有关每个选项的信息 , 请参阅 CC78K0 C 编译器操作篇）。

  (i) 在正常模式情况下，寄存器变量按照声明的队列依次分配到寄存器 HL 或 saddr 区 [FED0H 至 FEDFH]。如果不存在堆栈帧，则寄存器变量分配到寄存器 HL。仅当指定了 -qr 选项时，寄存器变量才会被分配到 saddr 区域。

  (ii) 在静态模式情况下，寄存器变量根据引用次数分配到 -sm 选项参数保留的寄存器 DE 或 _@KREGxx。仅当指定 -zm2 选项时，寄存器变量分配到 _@KREGxx。关于 -zm2 选项的详细信息，请参阅 "(32) 静态模式展开规范 (-zm)".

效果

- 分配到寄存器或 saddr 区域的变量的操作指令的代码长度通常比较短，短于分配到存储器的变量的操作指令代码长度。这有助于缩短目标代码 , 且还提高程序运行速度。

用法

- 如下用 register 存储类型声明变量：

```
register        类型        变量 - 名称
```

示例

```
void    main ( void )  {
        register        unsigned char   c ;
                :
}
```

限制

- 如果寄存器变量的使用频率并不是很高，则目标代码可能会增加 （取决于源代码的大小和内容）。
- 寄存器变量声明可以指定为 char/int/short/long/float/double/long double 和 pointer data 型。

(正常模式）

- char 占用的空间只有使用其他类型 （整型）的一半。long/float/double/long double 使用整型两倍的区域。在 char 型变量之间存在字节界限，而在其他情况下，存在字界限。
- 在 int/short 和 pointers 情况下，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起分配到普通存储空间。
- 在函数无需堆栈帧的情况下，对于 int/short 和 pointers 来说，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起分配到普通存储空间。

(静态模式）

- char 型占用的空间只有其他类型 （整型）的一半。
- 在 int/short 和 pointers 情况下，每个函数最多可用 1 个变量。
- 从第 2 个变量起，寄存器变量被分配到普通存储空间。
- 对于 long/float/double/long double 型，不能声明为寄存器变量。

表 11-5 有关寄存器变量用法的限制

数据类型	可用数量 （每个函数）	
	普通模式	静态模式
int/short	最多 8 个变量。	最多 1 个变量。
Pointer	最多 8 个变量。 （如果函数不具有栈帧，则最多可用 9 个变量）	最多 1 个变量。

示例

< C 源代码 >

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

- 当未指定 -sm 选项时 （寄存器变量分配到寄存器 HL 和 saddr 区 saddr 区域的示例）

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

  < 编译器的输出目标 >

```
        EXTRN   _@KREG00        ;  对将要使用的 saddr 区域进行引用
_main :
        push    hl              ; 保存寄存器内容在函数的开头
        movw    ax , _@KREG14   ; 保存 saddr 的内容在函数的开头
        push    ax              ;

        movw    hl , #00H       ; 以下代码被输出函数的中间
        movw    ax, hl          ;
        incw    ax              ;
        movw    _@KREG14 , ax   ;
        xch     a , x           ;
        add     a , l           ;
        xch     a , x           ;
        addc    a , l           ;
        movw    hl, ax          ;
        call    !_func          ;

        pop     ax              ; 恢复 saddr 的内容在函数的结尾
        movw    _@KREG00 , ax   ;
        pop     ax              ; 恢复寄存器的内容在函数的结尾
        ret
```

- 当指定 -sm 选项时 （寄存器变量分配到寄存器 DE 的示例）

```
_main :
        push    de              ; 保存寄存器内容在函数的开头

        movw    de , #00H       ;
        movw    de , ax         ;
        incw    ax              ;
        movw    !?L0003 + 1 , a ;
        xch     a , x           ;
        mov     !?L0003 , a     ;
        add     a , e           ;
        xch     a , x           ;
        addc    a , d           ;
        mov     de , ax         ;
        call    !_func          ;
        pop     de              ; 恢复寄存器的内容在函数的结尾
        ret
```

说明

- 要使用寄存器变量，仅需要用 register 存储类型标志进行声明。
- 在库中用 PUBLIC 声明的模块中的标签诸如 _@KREG00, 会附加到该 C 编译器的输出文件中。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果另一种 C 编译器支持 register 声明，则无需修改 C 源程序。
- 要改变寄存器变量，添加变量的 register 声明到程序中。

< 从 CC78K0S 到其他 C 编译器 >

- 如果另一种编译器支持 register 声明，则无需修改 C 源程序。
- 可以使用多少寄存器变量以及将其分配到哪个区域取决于另一种 C 编译器的具体实现情况。

(3) 如何使用 saddr 区域 (sreg / __sreg)

(a) sreg 声明的使用方法

功能

- 用关键字 sreg 或 __sreg 声明的外部变量和函数内 static 变量（称作 sreg 变量）自动分配到 saddr 区域 [FE20H 到 FED7H]（正常模式）和 [FE20H 到 FEEFH]（静态模式）并且可以重定位。当这些变量超出以上区域范围时，出现编译错误。
- sreg 变量的处理方式与 C 源代码中普通变量的处理方式相同。
- 用 sreg 关键字声明的 char、 short、 int 和 long 型变量的每一位都会自动变为 boolean 型变量。
- 声明的 sreg 变量如果未赋初值，则自动将 0 作为初值赋给 sreg 变量。
- 汇编源代码中，声明的 sreg 变量可以引用的范围包括整个 saddr 区域 [FFE20H 至 FFF1FH]。该范围 [FEB8H 至 FEFFH]（正常模式）和 FED0H 至 FEFFH]（静态模式）由编译器使用，因此必须谨慎处理（请参阅图 11-1）。

效果

- saddr 区域的指令代码长度通常短于普通存储器的代码长度。这有助于缩短目标代码和改进程序执行速度。

用法

- 在模块中或函数内部用关键字 sreg 和 __sreg 对变量进行声明。函数内部只具有静态存储类型的变量可以成为 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 外部变量的模块内部声明以下变量，它们也可以在函数内部描述。

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

限制

- 如果函数被指定为 const 型，或为函数指定了 sreg/__sreg 类型，则输出一个警告消息并忽略 sreg 声明。
- char 类型占用空间是其它类型的一半， long/float/double/long double 类型占用空间是其它类型的两倍。
- 在 char 型变量之间存在字节界限，而在其他情况下，存在字界限。
- 当指定 -za 时，仅启用 __sreg 并同时禁用 sreg 标识。
- 在 int/short 和 pointers 情况下，每个加载模块最多可使用 92 个变量 （当使用 saddr 区域 [FE20H 到 FED7H] 时）。注意可使用的变量数量会减少 , 在使用 bit 型和 boolean 型变量，寄存器变量或 norec 和 noauto 函数 时可使用的变量数量会减少 （正常模式）。
- 在 int/short 和 pointers 情况下，每个加载模块最多可使用 104 个变量 （当使用 addr 区域 [FE20H 至 FEEFH] 时）。注意可使用的变量数量在使用 bit 型和 boolean 型变量和共享区域时会减少 （静态模式）。

下表说明每个加载模块最多可使用的 sreg 变量数量。

表 11-6 有关 sreg 变量用法的限制

数据类型	sreg 变量的可用数量 （每个加载模块）	
	使用 saddr 区域 [FE20H 至 FED7H] 时	使用 saddr 区域 [FE20H 至 FEEFH] 时
int/short、pointer	92 变量最多[注]	104 变量最多[注]

注 使用 bit 和 boolean 型变量时，可用的变量数量减少。

示例

< C 源代码 >

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

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

以下示例展示了由用户创建的 sreg 变量的定义代码。如果未在 C 源代码中作出 extern 声明，CC78K0S 会输出以下代码。在此情况下，将不输出 ORG 伪指令。

< 汇编源程序 >

```
                PUBLIC  _hsmm0  ; 声明
                PUBLIC  _hsmm1  ;
                PUBLIC  _hsptr  ;

@@DATS          DSEG    SADDRP  ; 分配到段
                ORG     0FE20H  ;
_hsmm0 :        DS      ( 2 )   ;
_hsmm1 :        DS      ( 2 )   ;
_hsptr :        DS      ( 2 )   ;
```

在函数中输出以下代码。

< 编译器的输出目标 >

```
movw    ax , _hsmm0
xch     a , x
sub     a , _hsmm1
xch     a , x
subc    a , _hsmm1 + 1
movw    _hsmm0 , ax
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果另一种编译器不使用关键字 sreg/__sreg，则不需要修改。

  要改为 sreg 变量，则根据以上所示方法进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 通过 #define 进行修改。关于详细信息，请参阅 "11.6 C 源代码的修改 "。因此 , sreg 变量作为普通变量处理。

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

功能

- 不管是否作了 sreg 声明，外部变量 / 外部静态变量 （除 const 型）自动分配到 saddr 区域。
- 取决于 n 值和 m 的设定，外部变量和外部静态变量的分配按如下来设定。

表 11-7 通过 -rd 选项分配到 saddr 区域的变量

n 值	分配到 saddr 区域的变量
1	char 和 unsigned char 型变量
2	当 n = 1 时的变量，另外加上 short，unsigned short，int，unsigned int，enum 和 pointer 型变量
4	当 n = 2 时的变量，另外加上 long， unsigned long， float， double 和 long double 型变量
m	结构体、共用体以及数组
忽略时	所有变量

- 不管以上规范如何，以关键字 sreg 声明的变量必定会分配到 saddr 区域。
- 以上规则还适用于 extern 声明引用的变量，则处理过程内容相同，效果和这些变量被分配到 saddr 区域一样。
- 通过此选项分配到 saddr 区域的变量，和 sreg 变量的处理方法相同。这些变量的函数和限制如在 (a) 中描所述。

规范方法

- 指定 -rd [n] [m] (n : 1, 2, or 4) 选项。

限制

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

(c) 内部静态变量的 saddr 自动分配选项的使用方法 (-rs)

功能

- 不管是否作出 sreg 声明 , 将内部静态变量 （除 const 型）自动分配到 saddr 区。
- 取决于 n 的值和 m 的设定，可以对内部静态变量的分配做如下设定。

表 11-8 通过 -rs 选项分配到 saddr 区域的变量

n 值	分配到 saddr 区域的变量
1	char 和 unsigned char 型变量
2	当 n = 1 时的变量，另外加上 short，unsigned short，int，unsigned int，enum 和 pointer 型变量
4	当 n = 2 时的变量，另外加上 long， unsigned long， float， double 和 long double 型变量
m	结构体、共用体以及数组
忽略时	所有变量 （只有这种情况下才会包括结构体、共用体以及数组仅包括此情况下的结构、集合以及数组）

- 不管以上规范如何，以关键字 sreg 声明的变量总会被分配到 saddr 区域。
- 通过此选项分配到 saddr 区域的变量，和 sreg 变量的处理方法相同。这些变量的函数和限制如 (a) 所述。

规范方法

- 指定 -rs [n] [m] (n : 1, 2, or 4) 选项。

  备注 在 -rs [n] [m][m] 选项中，指定不同 n, m 值的模块之间能相互链接。

(d) 参数 / 自动变量 的 saddr 自动分配选项的使用方法 (-rk)

功能

- 不管是否作出过 sreg 声明 , 参数和自动变量 （除 const 型）自动分配到 saddr 区域。
- 使用 n 的值和根据 m 的设定来指定参数和自动变量的分配。

表 11-9 通过 -rk 选项 , 分配变量到 saddr 区域

n 值	分配到 saddr 区域的变量
1	char 和 unsigned char 型变量
2	当 n = 1 时的变量，另外加上 short，unsigned short，int，unsigned int，enum 和 pointer 型变量
4	当 n = 2 时的变量，另外加上 long， unsigned long， float， double 和 long double 型变量
m	结构体、共用体以及数组
忽略时	所有变量

- 不管以上规范如何，用关键字 sreg 声明的变量分配到 saddr 区域。
- 通过此选项分配到 saddr 区的变量和 sreg 变量处理方法相同。

用法

- 指定 -rk [n] [m] 选项 (当 n 是 1, 2, 或 4).

备注 在 -rk [n] [m][m] 选项中，指定不同 n, m 值的模块之间能相互链接。

限制

- 仅支持静态模式。当未指定 -sm 选项时，输出警告消息，并忽略自动分配。
- 已声明为寄存器变量的参数 / 变量不分配到 saddr 区。
- 当同时还指定 -qv 选项时，优先分配到寄存器 DE。

示例

< C 源代码 >

```
sub(int hsmarg)
{
        int     hsmauto ;
        hsmauto = hsmarg ;
}
```

< 编译器的输出目标 >

```
@@DATS          DSEG    SADDRP
?L0003 :        DS      ( 2 )
?L0004 :        DS      ( 2 )
@@CODE  CSEG
_sub :
                movw    ?L0003 , ax
                movw    ?L0004 , ax     ; hsmauto
                ret
```

(4) 如何使用 sfr 区域 (sfr)

功能

- Sfr 区引用指的是一组特殊函数寄存器，比如 78K/0S 系列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 源代码中进行描述。
- sfr 变量是外部变量，不具有初始值 （未定义）。
- 对只读 sfr 变量执行进行写入检验查。
- 对只写 sfr 变量进行执行读取检查。
- 将数据非法分配到 sfr 变量将会导致编译错误。
- 可以使用的 sfr 名称都为分配在地址 FF00H 至 FFFFH 组成的范围内。

效果

- 对 sfr 区域的操作可以在基于 C 源代码中直接级进行描述。
- 对 sfr 区域操作的指令在长度上短于对内存操作的指令。这有助于缩短目标代码和改进程序执行速度。

用法

- 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 源代码行的开始处说明。然而，如果指定了 #pragma PC （处理器型），则 #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：

  (i) 注释语句

  (ii) 预处理指令 , 其中没有变量或函数的定义 , 也没有引用变量或函数

- 在 C 源程序中，对 sfr 名称的描述按照将设备原有的 sfr 名称进行 （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  __K0S__
        #pragma sfr
#endif

void    main ( )
{
        P0 -= RXB00 ;
        /* RXB00 = 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。

< 编译器的输出目标 >

```
mov     a , P0
sub     a , RXB00
mov     P0 , a
```

兼容性

< 从其他 C 编译器导入到 CC78K0S>

- C 源程序中，和设备或编译器无关的部分无需修改。

< 从 CC78K0S 到其他 C 编译器 >

- 删除 “#pragma sfr”语句或通过 “#ifdef”排序以及添加原来为 sfr 变量的变量声明。示例如下所示。

```
#ifdef  __K0S__
        #pragma sfr
#else
/* 变量的声明 */
unsigned char   P0 ;
#endif

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

- 对于使用 sfr 或其替代功能的设备，必须创建专用库来以访问该区。

(5) noauto 函数 (noauto)

功能

- noauto 函数设置对自动变量加以限制，不输出预处理 / 后处理代码（生成堆栈帧）。
- 所有参数分配到寄存器或 saddr 区域（FEDCH 至 FEDFH），用作寄存器变量。如果有参数不能分配到寄存器，则会出现编译错误。
- 只有在参数分配完成之后，所有的自动变量都能够分配到寄存器或 saddr 区域内时，才可以使用自动变量，用作寄存器变量。
- noauto 函数将参数分配到 sadd 区当作寄存器变量使用，使用 noauto 必须在编译期间指定 -qr 选项时。
- 除分配到寄存器的参数之外的参数会被 noauto 函数存储在 saddr 区作寄存器变量使用，且将参数按升序存储（请参阅 " 附录 A saddr 区域标签列表 ")。
- 调用 noauto 函数的输出代码与调用普通函数的输出代码相同。
- 当指定 -sm 选项时，警告消息仅输出到首次描述 noauto 的行号，然后所有的 noauto 函数按普通函数进行处理。

效果

- 可以缩短目标代码，且并可以改善提高运行速度。

用法

- 在函数声明时，以在函数前加上 noauto 关键字就为函数加上了 noauto 属性声明函数，如下所示 :

```
noauto  类型名          函数名
```

限制

- 当指定 -za 选项时， noauto 函数被禁止。
- noauto 函数的参数有类型和数量上的限制。以下显示 noauto 函数内可以使用的参数类型。除 long/signed long/unsigned long， float/double/long double 之外的参数分配到寄存器 HL。
  - pointer
  - char/signed char/unsigned char
  - int / signed int / unsigned int
  - short / signed short / unsigned short
  - long / signed long / unsigned long
  - float / double / long double
- 可以使用的参数总大小为最多 6 个字节。
- 编译时会检验对这些限制进行检查。
- 如果以 register 声明参数，则忽略 register 声明。

示例

(C 程序)

- 当指定 -qr 选项时

< C 源代码 >

```
noauto  short   nfunc ( short a , short b , short c ) ;
short   l , m ;
void    main ( )
{
        static short    ii , jj , kk ;
        l = nfunc ( ii , jj , kk ) ;
}
noauto  short   nfunc ( short a , short b , short c )
{
        m = a + b + c ;
        return(m) ;
}
```

< 编译器的输出目标 >

```
@@CODE  CSEG
_main :
; line  5 :     static short ii , jj , kk   ;
; line  6 :     l = nfunc ( ii , jj , kk )  ;
        movw    ax , !?L0005    ; kk
        xch     a , x
        mov     a!?L0005 + 1    ; kk
        push    ax
        mov     a , !?L0004     ; jj
        xch     a , x
        mov     a , !?L0004 + 1 ; jj
        push    ax
        mov     a , !?L0003     ; ii
        xch     a , x
        mov     a , !?L0003 + 1 ; ii
        call    !_nfunc         ; 调用函数 nfunc(a , b , c) 函数
        pop     ax
        pop     ax
        movw    ax , bc
        mov     !_l + 1 , a ; 将返回值赋给外部变量 l
        xch     a , x
        mov     !_l , a
; line  7 :     }
        ret
; line  8 :     noauto short nfunc ( short a , short b , short c )
; line  9 :     {
_nfunc :
        push    hl              ; 保存 HL
        xch     a , x           ;
        xch     a , _@KREG12    ; 将参数 a 设置为 _@KREG12
        xch     a , x           ;
        xch     a , _@KREG13    ;
        push    ax              ; 保存 _@KREG14
        movw    ax , _@KREG14   ;
        push    ax              ;
        movw    ax , sp         ;
        movw    hl , ax         ;
        mov     a , [ hl + 10 ] ;
        xch     a , x           ;
        mov     _@KREG14 , ax   ;  将参数 c 分配到 _@KREG14
        mov     a , [ hl + 8 ]  ;
        xch     a , x           ;
        mov     a , [ hl + 9 ]  ;
        movw    hl , ax         ; 将参数 b 分配到 HL
; line  10 :    m = a + b + c   ;
        movw    ax , hl         ;
        xch     a , x           ;
        add     a , _@KREG12    ;
        xch     a , x           ;
        addc    a , _@KREG13    ;
        xch     a , x           ;
        add     a , _@KREG14    ;

        xch     a , x           ;
        addc    a , _@KREG15    ; 把 b(HL) 和 c(_@KREG14) 添加到 a(_@KREG12)
        mov     !_m + l , a     ; 将计算结果赋给外部变量 m
        xch     a , x
        mov     !_m , a
; line  11 :    return ( m )    ;
        xch     a , x
        movw    bc , ax         ; 返回外部变量 m 的内容
```

```
; line  12 :    }
        pop     ax              ;
        movw    _@KREG14 , ax   ; 恢复 _@KREG14
        pop     ax              ;
        movw    _@KREG12 , ax   ; 恢复 _@KREG12
        pop     hl              ; 恢复 HL
        ret
```

说明

- 在以上示例中， noauto 属性标志符加在 C 源代码的函数之前位置。

  声明 noauto 且不执行堆栈帧格式。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用关键字 noauto，则无需修改 C 源程序。
- 要将变量改为 noauto 变量，请根据以上使用方法中描述的程序修改程序。

< 从 CC78K0S 到其他 C 编译器 >

- 必须使用 #define。详情请参阅 “11.6 C 源代码的修改”。

(6) norec 函数 (norec)

功能

- 如果函数自身不调用另一函数，可以改为 norec 函数。
- 通过 norec 函数，不输出预处理和后处理 （栈帧格式）的代码。
- norec 函数的参数分配到寄存器和 saddr 区 （FEE8H 至 FEEFH）。
- 如果自变量参数不能分配到寄存器和 saddr 区域，则出现编译错误。
- 参数存储在寄存器或 saddr 区域 （FEE8H 到 FEEFH ）中，并调用 norec 函数。
- 自动变量分配到 saddr 区域 （FEF0H 到 FEF7H），寄存器变量同样处理。
- 当编译期间指定 -qr 选项时，不能分配到 saddr 区域。
- 如果使用 long/float/double/long double 型之外的参数，则第一个参数存储在寄存器 AX 中，第二个参数存储在寄存器 DE，第三个参数及随后的参数存储在 saddr 区域。请注意，不管参数的类型如何，寄存器 AX 和 DE 中只能存储一个参数。
- 如果在 norec 函数开始时，寄存器 DE 中没有存储在 norec 传递的参数，则将存储在寄存器 AX 中的参数复制到寄存器 DE。如果已存在存储在寄存器 DE 中已经存储了的参数，则将存储在 AX 中的参数的自变量被复制到 _@RTARG6 和 _@RTARG 7。
- 如果使用除了 long/float/double/long double 型之外的自动变量参数，则分配之后剩余的参数按声明的顺序存储到； DE， _@RTARG6 和 _@RTARG 7，和 _@NRARG0， _@NRARG 1...
  如果使用 long/float/double/long double 型的自动变量，则分配之后剩余的参数按声明的顺序存储到 _@NRARG0， 1...
  剩余参数按声明的顺序存储在 saddr 区域 （请参阅 " 附录 A saddr 区域标签列表 ").

效果

- 可以缩短目标代码且并改善程序执行的速度。

用法

- 在函数声明时，在函数前加上 norec 关键字就为函数加上了 norec 属性，如下所示：

```
norec    类型名        函数名
```

- __leaf 还可以代替 norec 来描述。

限制

- 从 norec 函数不可以调用其他函数。
- 对可以在 norec 函数中使用的参数和自动变量的类型和数量有一定限制。
- 当指定 -za 时，禁用 norec 且仅启用 __leaf。
- 当指定 -sm 选项时，警告消息仅输出首次描述 norec 的行号，然后所有的 norec 函数按普通函数普通函数进行处理。
- 编译时 , 对参数和自动变量的限制进行检查，且会出现错误。
- 如果在寄存器中声明参数和自动变量，则忽略寄存器声明。
- 以下显示可以在 norec 函数中使用的参数和自动变量的类型。
  如果类型在 char/signed char/unsigned char 之间，则 norec 函数连续分配到 saddr 区域，然而如果链接其他类型，则以两字节为单元进行分配。
  - pointer
  - char/signed char/unsigned char
  - int / signed int / unsigned int
  - short / signed short / unsigned short
  - long / signed long / unsigned long
  - float / double / long double

(当未指定 -qr 选项时）

- 如果不是 long/float/double/long double 型，则可以在 norec 函数中可以使用的参数的数量为 2 个变量。参数不能用于 long/float/double/long double 型。
- 保留区域的总字节数大小由各种类型组合决定，自动变量可以使用其中参数未使用的剩余空间。如果使用除 long/float/double/long double 之外的类型，则自动变量可以使用最多 4 个字节。参数不能用 long/float/double/long double 型。

(当指定 -qr 选项时）

- 如果使用除 long/float/double/long double 之外的类型，则参数的数量为 6 个变量，且如果使用 long/float/double/long double 型，则为 2 个变量。
- 由使用的各种数据类型决定的总字节数大小以及 saddr 区域保留字节数量，自动变量可以使用其中参数未使用的剩余空间，也可以使用 saddr 区域保留但未使用的剩余空间。如果使用 long/float/double/long double 之外的类型，则自变量可以使用最多 20 个字节，如果使用 long/float/double/long double 型，则自动变量可以使用最多 16 个字节。
- 编译时会检验这些限制，且如果不满足这些限制，则出现错误。

示例

< C 源代码 >

```
norec int rout（int a，int b，int c）;

int     i ,  j ;
void    main ( ) {
        int     k , l , m ;
        i = l + rout ( k , l , m ) + ++k ;
}

norec   int     rout ( int a , int b , int c )
{
        int     x , y ;
        return ( x + ( a << 2 ) ) ;
}
```

- 当指定 -qr 选项时

< 编译器的输出目标 >

```
        EXTRN   _@NRARG0            ; 引用将要使用的 saddr 区域
        EXTRN   _@NRARG1            ;
        EXTRN   _@NRARG6            ;
             :
        _@NRARG0    <- m            ; 将参数存储至 saddr area
             :
        de          <- 1            ;将参数存储至 DE
                 :
        ax          <- k            ; 将参数存储至 AX
        call    !_rout              ; 调用 norec 函数

_rout :
        movw    _@RTARG6 , ax
                                    ; 从 saddr 区域接收参数
        mov     c , #02H
        xch     a , x
        add     a , a
        xch     a , x
        rolc    a , 1
        dbnz    c , $$ - 5
    xch     a , x
        add     a , _@NRARG1        ; 使用 saddr 区域的自动变量
        xch     a , x               ;
        addc    a , _@NRARG1 + 1    ; 使用 saddr 区域的自动变量
        movw    bc , ax             ;
        ret
```

说明

- 在以上示例中，还为在 rout 函数的定义添加了 norec 属性，以声明该函数为 norec 函数。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用关键字 norec，则无需修改 C 源程序。
- 要将变量改为 norec 变量，则根据以上所描述的使用方法程序修改程序。

< 从 CC78K0S 到其他 C 编译器 >

- 必须使用 #define。详情请参阅 “11.6 C 源代码的修改”。

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

功能

- bit 或 boolean 型变量按 1 位数据进行处理，并分配到 saddr 区域。
- 这些变量的处理方法和无初值 （或具有未知的值）的外部变量处理方式相同。
- C 编译器输出以下这些位操作指令。

```
SET1 , CLR1 , NOT1 , BT , BF 指令
```

效果

- 可以在 C 语言中执行基于汇编源代码的程序编程且能够以位为单位访问 saddr 和 sfr 区域。

用法

- 在要使用 bit 或 boolean 型变量的模块内声明 bit 或 boolean 型，如下所示：
- _ _boolean 也可以替代 bit 进行说明。

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

- 在其中要使用 bit 或 boolean 型变量的模块内声明 bit 或 boolean 型，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（除数组元素和结构成员）以及 8- 位 sfr 变量可以自动用当作 bit 型变量。

```
变量名称 .n （其中 n = 0 至 31） )
```

限制

- 通过使用 CY （溢位）标志位来操作标记对两个 bit 或 boolean 型变量。
  出于此原因，不能保证语句之间的进位标志的内容。
- 不能定义位数组，也不能被数组引用。
- bit 或 boolean 型变量不能用作结构体或共用体的成员。
- 此类型变量不能用作函数的参数。
- bit 型变量不能用作自动变量 （除非是静态模式）。
- 仅使用 bit 型变量，每个加载模块最多可以使用 1472 变量 （当使用 saddr 区域 [FE20H 至 FED7H] 时）（正常模式）。
- 仅使用 bit 型变量，每个加载模块最多可以使用 1664 变量 （当使用 saddr 区域 [FE20H 至 FEEEH] 时）（静态模式）。
- 位变量声明时不能赋初值。
- 如果连同 const 关键字声明一起声明变量，则忽略 const 声明。
- 如 表 11-10. 所示，仅可以用 0 和 1 进行的运算操作符和常数的操作。
- *， &（指针引用、地址引用）以及 sizeof 都不能执行 。
- 当指定 -za 选项时，仅启用 __boolean。

表 11-10 仅使用常数 0 或 1 的运算符号操作符 （通过 Bit 型变量）

分类	运算符
赋值表达式	=
按位与表达式	&, &=
按位或表达式	|, |=
按位异或表达式	^, ^=
逻辑与	&&
逻辑或	||
等于	==
不等于	!=

备注 如果使用 sreg 变量，或指定了 -RD， -RS 和 -RK(saddr 自动分配选项) 选项，则可用的 bit 型变量数量减少。

示例

< C 源代码 >

```
#define ON      1
#define OFF     0

extern  bit     data1 ;
extern  bit     data2 ;

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

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

此示例中的内容是当用户为 bit 型变量编写了定义代码的情况。如果未添加 extern 声明，则在编译器输出以下代码。此时不输出 ORG 伪指令。

< 汇编源程序 >

```
PUBLIC  _data1          ; 声明
PUBLIC  _data2

@@BITS  BSEG            ; 分配到字段
        ORG     0FE20H
_data1  DBIT
_data2  DBIT
```

函数中输出以下代码

< 编译器的输出目标 >

```
set1    _data1                  已初始化的
clr1    _data2                  已初始化的
bf_     _data1 , $?L0001        判断
bf      _data1 , $?L0005        逻辑与表达式
bf      _data2 , $?L0005        逻辑与表达式
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果未使用关键字 bit， boolean 或 __boolean ，则无需修改 C 源程序。
- 要将变量改为 bit 或 boolean 型变量，请根据以上使用方法中所描述的程序修改程序。

< 从 CC78K0S 到其他 C 编译器 >

- 必须使用 #define。如需详细信息，请参阅 “11.6 C 源代码的修改”（此情况的结果是， bit 或 boolean 型变量按常规普通变量进行处理。）。

(8) ASM 语句 (#asm #endasm / __asm)

功能

(a) #asm - #endasm

- 通过使用预处理指令 #asm 和 #endasm 可以将用户编写的汇编源代码程序嵌入由 CC78K0S 输出的汇编源代码文件中。
- 将不输出 #asm 和 #endasm 行。

(b) __asm

- 汇编指令将汇编代码描述输出为字符串文字，并嵌入汇编源文件。

效果

- C 源代码的全局变量可以在汇编源代码中操作。
- 可以补充某些无法以 C 源代码来完成的函数。
- 由 C 编译程序编译器产生的汇编源代码可以进行手动优化，并嵌入 C 源代码 （以获得有效的目标对象）。

用法

(a) #asm - #endasm

- 以 #asm 指令指示汇编源代码开始且以 #endasm 指令指示汇编源代码结束。说明在 #asm 与 #endasm 之间进行的汇编源代码的描述。

```
#asm
    :           /* 汇编源程序 */
#endasm
```

(b) __asm

- 如需使用 __asm 的，要在 ASM 语句出现的模块开始处用 #pragma asm 规范进行声明 （#pragma 以后的关键字区分大写字母和小写字母）。
- 以下各项可以在 #pragma asm 之前说明 :

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 既未定义又未引用变量或函数的预处理指令。

- 在 C 源程序中， ASM 声明以下面的格式描述：

```
__asm ( 字符串文字 );
```

- 字符串文字的说明方法符合 ANSI 规范，且使用转义字符 （\n : 换行， \t ： 制表键）或￥可以使一行连续，也可以连接字符串。

限制

- 不允许嵌套 #asm 指令。
- 如果使用 ASM 语句，则不创建目标模块文件。而是创建汇编源代码文件。
- __asm 只能用小写字母来说明。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，仅启用 __asm。
- #asm - #endasm 和 __asm 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 的 CSEG。

示例

(a) #asm - #endasm

< C 源代码 >

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

由用户编写的汇编源代码输出到汇编源代码文件。

< 编译器的输出目标 >

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

说明

- 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。

(b) __asm

< C 源代码 >

```
#pragma asm

int     a , b ;

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

< 汇编源程序 >

```
@@CODE  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，可以根据 C 编译器指定的格式对程序进行修改。
- 如果目标设备不同，则修改程序的汇编源代码部分。

(9) 中断函数修饰符 (__interrupt)

功能

- 以 __interrupt 修饰词声明的函数被视为硬件中断函数，不可屏蔽 / 可屏蔽中断函数的返回指令 RETI 可以返回。
- 以此修饰词限定词声明的被函数视为 （不可屏蔽 / 可屏蔽 / 软件）中断函数，并将寄存器和变量区 （1）以及 （4）以下保存 / 恢复到堆栈或从堆栈恢复寄存器和变量区 （1）以及 （4）以下，其用作编译器的工作区。

  但是，如果函数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) 寄存器变量的 saddr 区域

  (3) norec 函数的参数 / 自动变量的 saddr 区域 (不管用法)

  (4) 运行时间库的 saddr 区域

  备注　如果编译时未指定 -qr 选项 （默认 -qr 选项），则不输出保存 / 恢复代码，因为不使用区 （2）和（3）区。如果编译时指定 -sm 选项，则不输出保存 / 复位代码，因为不使用区 （2），（3）和（4）区，所以不输出保存 / 复位代码。

效果

- 通过此修饰词来的声明函数，向量表和中断函数定义的设置可以在不放在同一个文件中单独的文件中说明。

用法

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

< 对于非可屏蔽不可屏蔽 / 可屏蔽中断函数 >

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

限制

- 因为不存在软件中断，所以不支持 __interrupt_brk。其中第一次出现 _ _ interrupt _brk 时输出警告消息，忽略关键字被忽略，且 _ _ interrupt _brk 按常用普通函数进行处理。
- 不能设定 callt / noauto / norec / __callt / __leaf /__pascal 为中断函数。

注意事项

- 仅通过用此修饰词声明，无法设定向量地址。向量地址必须通过使用 #pragma vect/ interrupt 指令或汇编描述来单独设置。
- Saddr 区域和寄存器都保存到堆栈。
- 即使通过 #pragma vect （或 interrupt） ... 设置了向量地址或改变保存目的地，如果相同文件中不存在对应的函数定义且假定默认堆栈，则忽略保存目的地的变化，且使用默认堆栈。
- 要将相同文件的中断函数定义为 #pragma vect （或 interrupt） ... 规范，即使未加此修饰词，由 #pragma vect （或 interrupt） ... 指定的函数名称也会被判定为中断函数 （如需 #pragma vect/interrupt 的详细信息，请参阅 "(10) 中断函数 (#pragma vect / #pragma interrupt)" 中断函数的使用方法）。

示例

- 按以下格式声明或定义中断函数。设置向量地址的代码通过 #pragma interrupt 生成。

```
#pragma interrupt   INTP0   inter

__interrupt     void    inter ( ) ;             /* 初始声明 */
__interrupt     void    inter ( ) { processing } ;      /* 函数体 */
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 除非支持中断函数，否则无需修改 C 源程序。
- 若需要，请根据以上使用方法中描述的过程对中断函数进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 必须使用 #define 以使得中断修饰词能够按普通函数进行处理。
- 要将中断修饰词用作中断函数，请根据每个编译器的具体规范修改程序。

(10) 中断函数 (#pragma vect / #pragma interrupt)

功能

- 所描述的函数名称的地址被注册到对应于特定中断请求名称的中断向量表中。
- 中断函数会输出代码在函数的开始和结束处的堆栈对以下数据 （除用于 ASM 声明）进行保存或恢复 （如果指定了寄存器组，在该代码之后）

  (1) 寄存器

  (2) 寄存器变量的 saddr 区域

  (3) norec 函数参数 / 自动变量的 saddr 区域 （不管使用参数还是变量）

  (4) 运行时刻库的 saddr 区域 （仅正常模式）

  注，这些具体内容是否执行取决于中断函数的规范或状态，保存 / 复位分别进行，如下所示：
- 如果指定了另外的寄存器组，用来选择寄存器组的代码输出在中断函数的开始处，于是，不对寄存器的内容保存 / 恢复。
- 但是，如果未指定 “不改变”且在中断函数中调用函数，则不管是否指定使用寄存器，对整个寄存器区域进行保存或恢复。

(正常模式)

- 如果编译时未指定 -qr 选项，则 norec 函数中寄存器变量的 saddr 区域和参数 / 自动变量的 saddr 区域不使用；因此，保存 / 恢复的代码不作输出。

  如果保存代码的大小小于恢复代码，则输出恢复代码。

- 表 11-11 总结了以上内容并显示保存 / 恢复区域。

表 11-11 在使用中断函数时的存储 / 恢复区域

保存 / 恢复区	无 BANK	调用函数		不调用函数	
		未指定 -qr	已指定 -qr	未指定 -qr	已指定 -qr
使用的寄存器	NG	NG	NG	OK	OK
所有寄存器	NG	OK	OK	NG	NG
使用的运行时库的 saddr 区域	NG	NG	NG	OK	OK
所有运行时库运的 saddr 区域	NG	OK	OK	NG	NG
使用的寄存器变量的 saddr 区域	NG	NG	OK	NG	OK
norec 函数的参数 / 自动变量的所有 saddr 区域	NG	NG	OK	NG	NG

OK : 保存

NG : 不保存

(静态模式）

- 因为当编译期间指定 -sm 选项时，不使用寄存器变量的 saddr 区域、自动变量或 norec 函数自动变量参数的 saddr 区域和运行时刻库的 saddr 区域都不使用，仅输出寄存器的保存和复位代码；而不输出保存和复位 saddr 区域的代码。

  然而，当设定 leafwork 为 1 至 16 时，在中断函数的开始和结束处，从共享区的高位地址输出堆栈中保存和恢复指定字节数的代码。（当未指定 -ZM 选项时的静态模式，参阅 "(23) 静态模式 "，且在设定 -zm 选项时参阅 "(32) 静态模式展开规范 (-zm)"）。

  备注 如果中断函数中存在 ASM 语句声明，且在此 ASM 声明语句中使用了为编译器的寄存器所保留的区域，则该区域的内容必须由用户自行保存。

效果

- 可以用 C 源代码进行中断函数的描述。

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、寄存器以及是否保存 / 恢复 saddr 区域。在 C 源代码的开始处加上 #pragma 指令描述 （关于中断请求的具体名称，请参阅使用的目标设备的用户手册），关于软件中断 BRK，需要描述 BRK_I。
- 进行 #pragma PC （处理器类型）的语句要在中断相关的 #pragma 指令之前。以下各项可以在此 #pragma 指令之前进行说明。

(i) 注释语句

(ii) 没有定义或引用变量或函数的预处理指令

```
#pragma Δ 向量 ( 或 interrupt) Δ 中断请求名 Δ 函数名 Δ
    [堆栈改变说明 Δ  堆栈使用说明 ]  [ 堆栈使用规范
                                    不改变规范
                                    共享区保存 / 恢复规范
                                    保存 / 恢复目标 ]
```

中断请求名称 :	以大写字母表示。请参阅相关的目标设备的用户手册 (示例： NMI, INTP0 等)。	
函数名称 :	描述中断过程的函数名称	
堆栈改变说明 :	SP = 数组名称 [+ 偏移位置] （示例：SP = buff + 10）。	
	通过 unsigned char 定义数组 （示例：unsigned char buff [10];）。	
堆栈使用说明 :	STACK (默认)	
不改变说明 :	NOBANK	
共享区域保存 / 恢复说明 :	leafwork 1 到 16	
保存 / 恢复目标 :	SAVE_R	限制于寄存器的存储 / 复位目标
	SAVE_RN	保存 / 恢复目标限制于寄存器和 _@NRATxx (当 -sm, -zm 选项被指定)
Δ :	空格	

限制

- 不支持寄存器组规范。
- 中断请求名称必须以大写字母表示。
- 将仅在一个模块内将作出对中断请求名称进行的双重检验。
- 如果在处理向量中断的同时，由于优先级标志寄存器和中断屏蔽标志寄存器的内容出现相同或产生了另一个中断，如果设定没改变，则寄存器的内容可能改变且产生错误结果。然而，编译器无法检验此错误。
- callt/ callf / noauto / norec / __callt/__callf / __leaf/__rtos_interrupt /__pascal/__flash/__flashf/ 函数不能被指定为中
- 因为中断函数不能带参数或返回值，所以指定具有 void 型的中断函数 （示例：void func （void） ;）。
- 即使在中断函数中有 ASM 语句，保存所有寄存器和变量区域的代码也不作输出。因此，如果为编译器所保留的区在中断函数的 ASM 语句中被使用，或在 ASM 语句中进行了函数调用，则用户必须自行保存寄存器和变量区。
- 如果在未指定 -SM 选项时指定 leafwork 为 1 至 16，则输出警告，且忽略共享区的保存 / 恢复规范。
- 当指定堆栈变化时，堆栈指针就被改为数组名称符号加偏移量的位置。数组名称的范围不通过 #pragma 指令保留。数组需要作为全局 unsigned char 型数组单独定义。
- 改变堆栈指针的代码放于函数开始处，且设置堆栈指针返回的代码放在函数结束处。
- 当关键字 sreg/__sreg 添加到数组用于堆栈变化时，假定定义了两个或两个以上具有不同属性和相同名称的变量 ，且出现编译错误。可能通过 -RD 选项将其中一个数组分配于 saddr 区域中，但无法改善代码大小和执行效率，因为数组被用作堆栈。不建议将 saddr 区域作为堆栈使用。
- 在 “不改变”的情况下不能同时指定堆栈变化。如果这样指定，则会出现错误。
- 堆栈改变必须在堆栈使用规范之前说明。如果在堆栈使用规范之后说明堆栈改变，则出现错误。
- 如果某函数被指定为 “不改变”、不在同一模块中定义的寄存器组或堆栈就被作为 #pragma vect/ #pragma interrupt 规范的保存目的地，则输出警告消息且忽略堆栈变化。在此情况下，使用默认堆栈。

示例

- 指定共享区域保存 / 恢复 （仅静态模式）

<C 源代码 >

```
#pragma interrupt INTP0 inter leafwork4
void    func ( ) ;
void    inter ( )
{
        func ( ) ;
}
```

< 编译器的输出目标 >

```
        EXTRN   _@KREG12
        EXTRN   _@KREG14

@@CODE  CSEG
_inter :
        push    ax              ; 保存寄存器值
        push    bc              ;       "
        push    hl              ;       "
        movw    ax , _@KREG12   ; 保存共享区域
        push    ax              ;       "
        movw    ax , _@KREG14   ;       "
        push    ax              ;       "
        call    !_func
        pop     ax              ; 恢复共享区域
        movw    _@KREG14 , ax   ;       "
        pop     ax              ;       "
        movw    _@KREG12 , ax   ;       "
        pop     hl              ; 恢复寄存器
        pop     bc              ;       "
        pop     ax              ;       "
        reti

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

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果根本不使用中断函数，则无需修改 C 源程序。
- 要将普通函数改为中断函数，请根据以上使用方法中描述的过程来修改程序。

< 从 CC78K0S 到其他 C 编译器 >

- 通过删除 #pragma vect 或 #pragma interrupt 指令删除，中断函数可以用作普通函数。
- 当普通常用函数要用作中断函数时，根据每个编译器的具体规范来改变程序。

(11) 中断函数 (#pragma DI, #pragma EI)

功能

- DI 和 EI 代码输出到对象，并创建一个目标文件。
- 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、注释和预处理指示符的声明外），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）。
- 要函数预处理之后输出 DI 对应代码，请在说明 "DI( );" 之前开辟新块 （对 "{" 对此块划分界限 ）。
- 如果 “EI( );”在函数结束处说明 （除注释和预处理指令），则在函数后处理最后输出 EI 对应的代码 （代码 RET 之后立即进行）。
- 要在函数后处理之前输出 EI 对应的代码，请在说明 "EI( );" 之后关闭新块 （对 "{" 对此块划分界限）。

效果

- 创建的函数可以禁止用中断。

用法

- 在 C 源代码开始处说明 #pragma DI 和 #pragma EI 指令。然而，以下语句和指令可以放早于在 #pragma DI 和 #pragma EI 指令之前。

  (i) 注释语句

  (ii) 其他 #pragma 指令

  (iii) 既未定义又未引用变量或函数的预处理指示符
- 在源代码中说明 DI( ); 或 EI( ); 的方式以和与函数调用相同的方式在源代码中说明 DI( ); 或 EI( );。
- DI 和 EI 可以在 #pragma 之后用大写字母或小写字母表示均可。

限制

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

示例

```
#ifdef  __K0S__
        #pragma DI
        #pragma EI
#endif
```

< （C 源代码 1）>

```
#pragma DI
#pragma EI
void    main ( )
{
        DI ( ) ;
        ; 函数体
        EI ( ) ;
}
```

< 编译器的输出目标 >

```
_main :
        di
        ;预处理过程
        ; 函数体
        ; 后处理过程
        ei
        ret
```

- 要在预处理 / 后处理之后和之前输出 DI 和 EI

< （C 源代码 2）>

```
#pragma DI
#pragma EI
void    main ( )
{
        {
                DI ( ) ;
                ; 函数体
                EI ( ) ;
        }
}
```

< 编译器的输出目标 >

```
_main :
        ;预处理过程
        di
        ; 函数体
        ei
        ; 后处理过程程序
        ret
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果根本不使用中断函数，则无需修改 C 源程序。
- 要将普通函数改为中断函数，请根据以上使用方法中描述的程序过程对修改程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 删除 #pragma DI 和 #pragma EI 指令或通过用 #ifdef 分离它们使这些指令无效，且 DI 和 EI 可以用作普通函数名（例如：#ifdef__K0S__ ...#endif).
- 要将普通函数用作中断函数，根据每个编译器的具体规范来修改程序。

(12) CPU 控制指令 (#pragma HALT / STOP / NOP)

功能

- 以下代码输出到该对象以创建一个对象目标文件：

  (1) HALT 操作指令 (HALT)

  (2) STOP 操作指令 (STOP)

  (3) NOP 指令

效果

- 微控制器的待机函数可以通过 C 程序使用。
- 在没有 CPU 运行的情况下时钟可以继续运行。

用法

- 在 C 源代码开始处说明 #pragma HALT， #pragma STOP 和 #pragma NOP 指令。
- 以下各项可以在 #pragma 指令之前说明：

  (i) 注释语句

  (ii) 其他 #pragma 指令

  (iii) 既未定义又未引用变量或函数的预处理指令

- #pragma 以后的关键字可以用大写或小写字母表示。
- 在 C 源代码中用大写字母描述，函数调用的格式相同，如下表示：

  (1) HALT ( ) ;

  (2) STOP ( ) ;

  (3) NOP ( ) ;

限制

- 当使用此函数时， HALT， STOP 和 NOP 不能用作函数名称。
- HALT， STOP 和 NOP 用大写字母表示。如果将其用小写字母表示，则其按普通函数进行处理。

示例

< C 源代码 >

```
#pragma HALT
#pragma STOP
#pragma NOP
void    main ( )
{
        HALT ( ) ;
        STOP ( ) ;
        NOP ( ) ;
}
```

< 编译器的输出目标 >

```
@@CODE  CSEG
_main :
        halt
        stop
        nop
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用 CPU 控制指令，则无需修改 C 源程序。
- 当使用 CPU 控制指令时根据以上使用方法所描述的过程对程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

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

(13) 绝对地址访问函数 (#pragma access)

功能

- 访问普通 RAM 空间的代码输出到目标，通过直接内联扩展而不是通过函数调用方式，且可以创建目标文件。
- 如果未说明 #pragma 指令，则访问绝对地址的函数被视为普通函数。

效果

- 普通存储空间中的特定地址可以很方便的通过 C 语言描述来访问。

用法

- 在 C 源代码开始处说明 #pragma access 指令。
- 在源文件中说明该指令的方式和函数调用的格式相同。
- 以下各项可以在 #pragma access 之前说明 :

  (i) 注释语句

  (ii) 其他 #pragma 指令

  (iii) 既未定义又未引用变量或函数的预处理指令

- #pragma 以后的关键字可以用大写或小写字母表示。

  以下 4 个函数名称在绝对地址访问过程中可用 :

```
peekb , peekw , pokeb , pokew
```

[绝对地址访问函数表]

(a) unsigned char peekb (addr) ;
unsigned int addr ;

返回 1 个字节地址 addr 的内容。

(b) unsigned int peekw (addr) ;
unsigned int addr ;

返回 2 个字节地址 addr 的内容。

(c) void pokeb (addr, data) ;
unsigned int addr ;
unsigned char data ;

将 1 个字节的数据的内容写入由地址 addr 指示的位置。

(d) void pokew (addr, data) ;
unsigned int addr ;
unsigned int data ;

将 2 个字节的数据的内容写入由地址 addr 指示的位置。

限制

- 不要随便使用绝对地址访问函数名称。
- 以小写字母表示绝对地址访问的函数。以大写字母表示的函数按普通函数进行处理。

示例

< C 源代码 >

```
#pragma access

char    a ;
int     b ;

void    main ( )
{
        a = peekb ( 0x1234 ) ;
        a = peekb ( 0xfe23 ) ;
        b = peekw ( 0x1256 ) ;
        b = peekw ( 0xfe68 ) ;

        pokeb ( 0x1234 , 5 ) ;
        pokeb ( 0xfe23 , 5 ) ;
        pokew ( 0x1256 , 7 ) ;
        pokew ( 0xfe68 , 7 ) ;
}
```

< 输出的汇编源程序 >

```
  :       :
mov     a , !01234H
mov     !_a , a
mov     a , 0FE23H
mov     !_a , a
mov     a , !01256H
xch     a , x
mov     a , !01257H
movw    de , #_b
callt   [ @@deist ]
movw    ax , 0FE68H
callt   [ @@deist ]

mov     a , #05H
mov     !01234H , a
mov     0FE23H , #05H
movw    ax , #07H
mov     !01257H , a
xch     a , x
mov     !01256H , a
movw    ax , #07H
movw    0FE68H , ax
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用绝对地址访问的函数，则无需修改源程序。
- 如果使用绝对地址访问的函数，则根据以上使用方法描述的过程对程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 删除 “#pragma access”声明或用 #ifdef 进行屏蔽，绝对地址访问的函数名称可以用作函数名称。
- 要使用绝对地址访问的函数，程序必须根据每个编译器的具体规范来修改（#asm， #endasm， asm 等）。

(14) 位段声明

(a) 类型扩展指示符

功能

- unsigned char 型位段不可跨字节界限进行分配。
- unsigned int 型位段不可跨字界限进行分配，而可以跨字节界限进行分配。
- 相同类型的位段分配在同一个字节单元（或字单元）中。如果类型不同，则位段分配在不同字节单元（或字单元）中。

效果

- 可以保存内存的内容，可以缩短目标代码且可以提高运行速度。

用法

- 作为位段类型说明符，除 unsigned int 型之外可以指定 unsigned char 型。如下声明。

```
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

< 从其他 C 编译器导入到 CC78K0S >

- 无需修改源程序。
- 改变类型说明符以将 unsigned char 用作类型说明符。

< 从 CC78K0S 到其他 C 编译器 >

- 如果 unsigned char 不用作类型说明符，则无需修改源程序。
- 如果用作类型说明符，则将 unsigned char 改为 unsigned int。

(b) 位段的分配方向

功能

- 改变要分配的位段的方向，且当指定 -RB 选项时从 MSB 端开始分配位段。
- 如果未指定 -rb 选项，则从 LSB 端分配位段。

用法

- 在指定编译时间指定的 -rb 选项可以从 MSB 端开始分配位段。
- 不指定该选项以从 LSB 端开始分配位段。

例 1

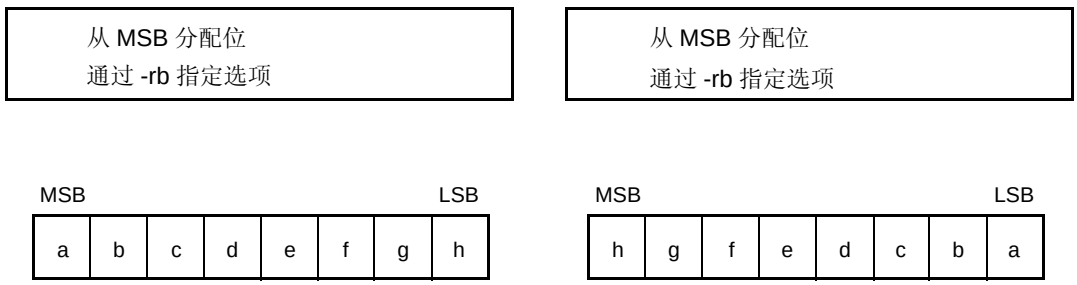
< 位段声明 >

```
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 ;
} ;
```

说明

- 因为 a 到 h 为 8 位或更少的位，所以可以其分配在同一个 1- 字节单元中。

图 11-3 通过位段声明分配位 （示例 1）



例 2

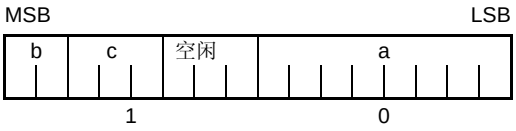
< 位段声明 >

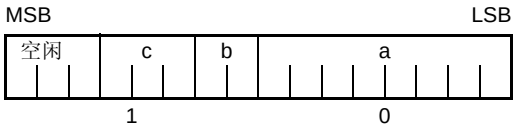
```
struct  t {
        char    a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned char   f : 5 ;
        unsigned char   g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 2 ;
} ;
```

说明

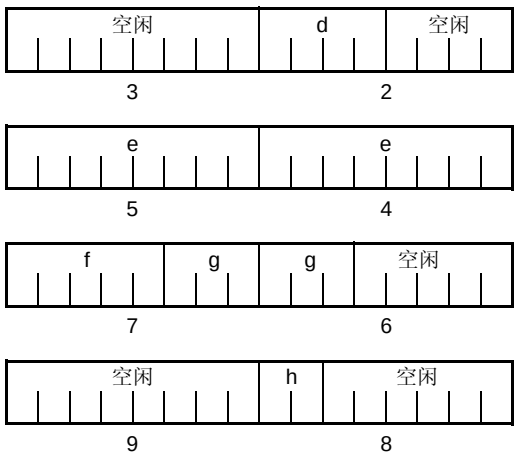
图 11-4 通过位段声明分配位 （示例 2）

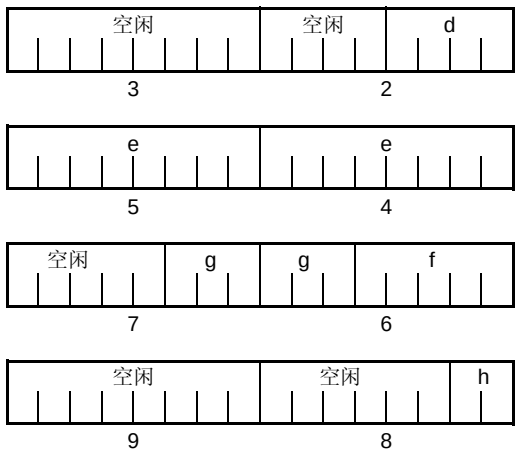
从 MSB 端进行位域分配 当 -rb 选项被指定	从 LSB 端进行位域分配 当 -rb 选项未被指定



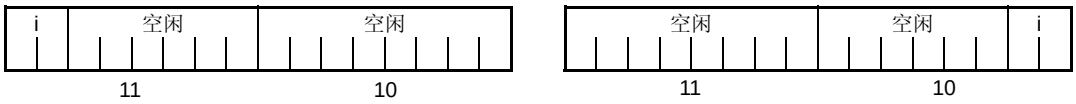


分配 char 型成员 a 到第一字节单元。成员 b 和 c 分配到随后的字节单元中，从第二字节单元开始。如果 a 字节单元没有足够空间来容纳 char 类型成员，则该成员将分配到后面的字节单元中。在此情况下，如果在第二字节单元中仅存在 3 位空间且成员 d 有四位，则其将其分配到第三个字节单元。





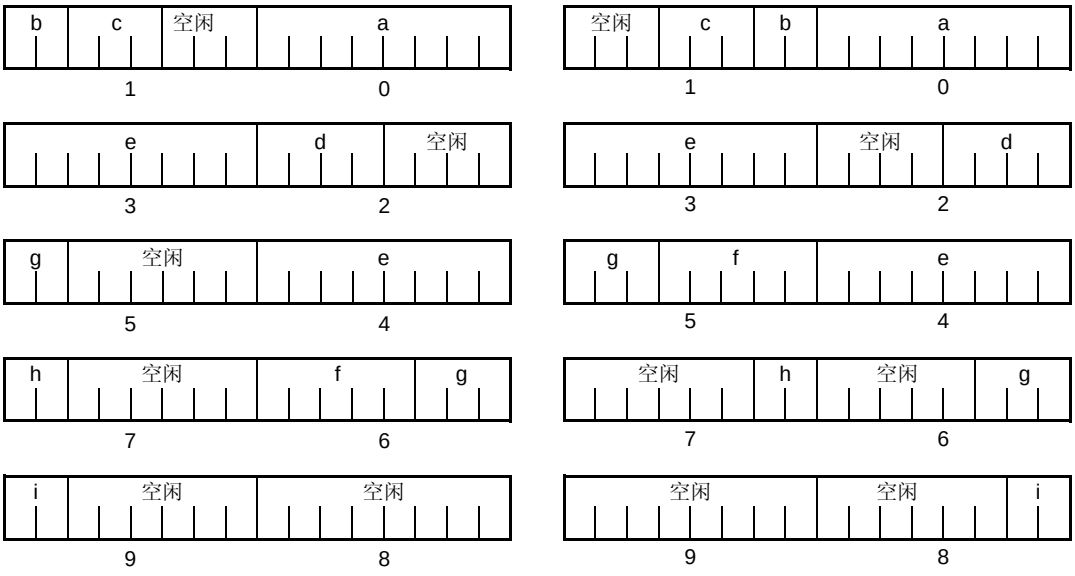
因为成员 g 为 unsigned int 型的位字段位段，所以可以跨字节界限进行分配。因为 h 为 unsigned char 型的位字段位段，所以其不能与和 unsigned int 型的 g 位段分配在相同的字节单元中，而是分配在下一字节单元。



因为 i 为 unsigned int 型的位字段位段，所以其分配在下一字单元。

当指定 -rc 选项时 （以封装结构成员），以上位段成为如下形式。

图 11-5 通过位段声明分配位 (例 2) (当 -rc 选项被指定)



备注　　分配图下方的数字表示从该结构开始的字节偏移值。

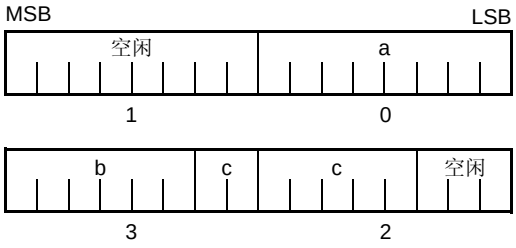
例 3

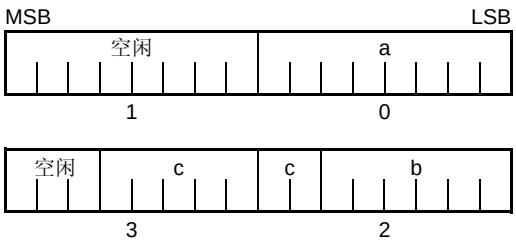
< 位段声明 >

```
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 ;
} ;
```

图 11-6 通过位段声明分配位 （示例 3）

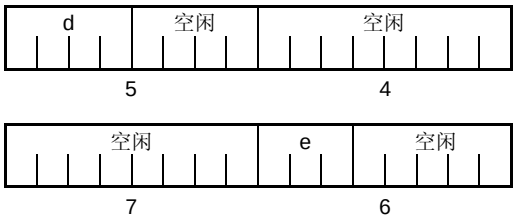
从 MSB 端进行位段分配 当 the -rb 选项被指定	从 LSB 端进行位段分配 当 the -rb 选项未被指定

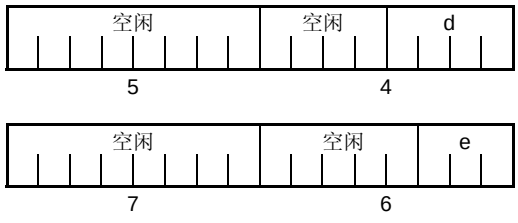




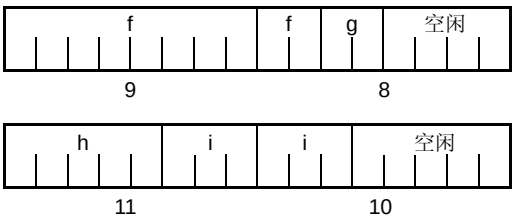
因为 b 和 c 为 unsigned int 型的位段，所以其从下一字单元分配。

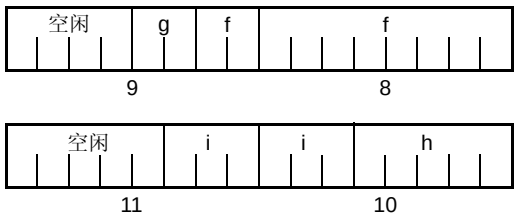
因为 d 也是 unsigned int 型的位段，所以其从下一字单元分配。





因为 e 为 unsigned char 型的位段，所以其分配到下一字节单元。

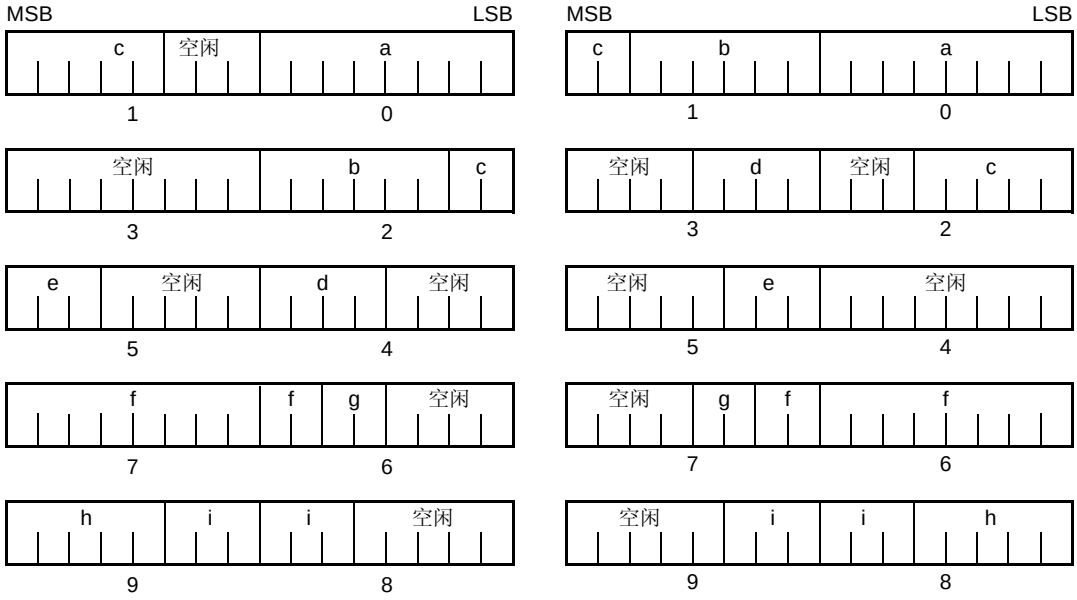




f 和 g、和 h 和 i 每一者分配到单独的字单元。

当指定 -rc 选项时 （对结构成员进行封装），以上位段变为如下形式。

图 11-7 通过位段声明分配位 (例 3) (当 -rc 选项被指定)



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

< 从其他 C 编译器导入到 CC78K0S >

- 无需修改源程序。

< 从 CC78K0S 到其他 C 编译器 >

- 如果使用了 -rb 选项，且代码和考虑到位段分配序列配合进行，则必须修改源程序。

(15) 改变编译器输出区名称的函数 (#pragma section ...)

功能

- 改变编译器输出区域块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 " 附录 B 区段名称列表 "。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。 如需有关连接指令的详细信息，请参阅 RA78K0S 汇编包用户操作手册。
- 要以指定的 AT 起始地址改变块名称 @@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，则该数据位于改变后的区段中。可以用指令进行再次，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的中。此外，对于 static 变量 （函数内）以相同方式同样有效。

效果

- 在一个文件中重复改变编译器输出块，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。

用法

- 通过使用如下所示的 #pragma 指令去指定要改变的区段名称，新区域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。
- 在 #pragma PC （处理器类型）之后说明此 #pragma 指令。
- 以下各项可以在此 #pragma 指令之后说明：

  (i) 注释语句

  (ii) 没有定义或引用变量或函数的预处理指令

  但是，在 BSEG 和 DSEG 中的所有区段，和 CSEG 中的 @@CNST 区段都可以在 C 源代码任意处说明，且可以重复执行改变指令。要返回到原初始区段名称，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：

```
#pragmasection  编译器输出区段名称区段新名称  [AT 起始地址 ]
```

- 有关在 #pragma 之后要说明的关键字，请确保以大写字母表示编译器输出区域块区段名称。Section， AT 可以用大写或小写字母或组合来表示。

- 其中要说明新区域块区段名称的格式符合汇编程序规范 （字段名称最多可以使用八个字母）。
- 仅 C 语言的十六进制数和汇编程序的十六进制数可以用来说明起始地址。

[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 )
```

[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： 要以十六进制数表达值为 255 的一个数值，请在 F 之前加零。因此其为 0FFH。

- 对于 CSEG 中除 @@CNST 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（说明 C 文本之后）说明；否则此指定将会导致错误。
- 如果此 #pragma 指令在 C 文本说明之后执行，则创建汇编源代码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，则仅有此 #pragma 指令且没有 C 文本 （包括变量和函数的外部引用声明）的文件不能作为头文件被包含。这将会导致错误 （ 参照 " 编码错误 示例 1"）。
- #include 语句不能在某文件中出现，因为该文件在执行 C 文本说明之后执行的此 #pragma 指令的文件中说明。如果有此情况发生，则导致错误出现 (参照 " 编码错误 示例 2")。
- 如果 #include 语句在 C 文本之后，则此 #pragma 指令不能出现在此语句之后。如果有此情况发生，则导致错误出现 (参照 " 编码错误 示例 3").

例 1

区域块名称 @@CODE 改为 CC1 且地址 2400H 指定为起始地址。

< C 源代码 >

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

void    main ( )
{
        ; 函数体
}
```

< 输出对象 >

```
CC1     CSEG    AT      2400H
_main :
        ; 预处理过程
        ; 函数体
        ; 后处理过程
        ret
```

例 2

以下为其中主程序 C 代码程序之后的 #pragma 指令的代码示例。内容分配在 “//”ó

```
#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 自动关闭且 ??DATA2 生效
        int         a3 ;                    // ??DATA2
        sreg int    b3 ;                    // ??DATS
        int         c3 = 3 ;                // @@INIT and @@R_INIT
        const int   d3 = 3 ;                // @@CNST
#pragma section @@DATA      @@DATA
        // ??DATA2 关闭，且处理过程返回到默认 @@DATA
#pragma section @@INIT      ??INIT
#pragma section @@R_INIT    ??R_INIT
        // ROMization 无效，除非两个名称 （@@INIT 和 @@R_INIT）都被改变。
        // 这是用户的责任。
        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 和 ??R_INIT 关闭，且处理程序返回到默认设置
#pragma section @@BITS          ??BITS
        __boolean e4 ;                      // ??BITS
#pragma section @@CNST      ??CNST
        char    *const p = " Hello " ;      // p 和 “Hello” 均是 ??CNST
```

例 3

```
#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 和 ??R_INIT3
#pragma section @@DATA      ??DATA4
#pragma section @@INIT      ??INIT4
#pragma section @@R_INIT    ??R_INIT4
```

当此 #pragma 指令在主程序 C 代码之后指定时，所受的限制在以下编码错误示例中说明。

编码错误 示例 1

```
a1.h
        #pragma section @@DATA  ??DATA1   // 文件仅包含 #pragma 区段


a2.h
        extern  int     func1( void ) ;
       #pragma section @@DATA  ??DATA2   // 文件包含 #pragma 指令之后的主程序 C 代码



a3.h
        #pragma section @@DATA  ??DATA3  // 文件仅包含 #pragma 区段。


a4.h
        #pragma section @@DATA  ??DATA3
        extern  int     func2 ( void ) ; // 包括主程序 C 代码的文件。


a.c
        #include " a1.h "
        #include " a2.h "
        #include " a3.h "  // ← 导致错误。
                           // 因为 a2.h 文件包含主程序 C 代码，其之后为此 #pragma 指令，
                           // 不能包括文件 a3.h，其中只有此 #pragma 指令。
        #include " a4.h "
```

编码错误 示例 2

```
b1.h
        const int       i ;

b2.h
        const int       j ;
        #include " b1.h "       // 这不会导致错误，因为这不是文件 （b.c）， b.c 中的
                                 // 主程序 C 代码之后是此 #pragma 指令。

b.c
        const int       k ;
        #pragma section @@DATA  ??DATA1
        #include " b2.h "      // ← 导致错误
                               // 因为 #include 语句随后不能在文件（b.c）之后出现，因为
                               // b.c 中主程序 C 代码之后是此 #pragma 指令。
```

编码错误 示例 3

```
c1.h
        extern int      j ;
        #pragma section @@DATA  ??DATA1 // 这不导致错误出现，因为在 c3.h 处理之前，
                                        // 包含并处理 #pragma 指令。

c2.h
        extern int      k ;
        #pragma section @@DATA  ??DATA2 // ← 导致错误。
                                        // 此 #include 语句在 c3.h 的主程序 C 代码
                                        // 之后指定且此后不能指定 #pragma 指令。

c3.h
        #include " c1.h "
        extern int      i ;
        #include " c2.h "
        #pragma section @@DATA  ??DATA3 // <- 导致错误。
                                        // 此 #include 语句在主程序 C 代码之后指定，
                                        // 且此后不能指定 #pragma 指令。

c.c
        #include " c3.h "
        #pragma section @@DATA  ??DATA4 // ← 导致错误。
                                        // 此 #include 语句在 c3.h 的主程序 C 代码
                                        // 之后指定且此后不能指定 #pragma 指令。
        int     i ;
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 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，请根据以上使用方法中描述的步骤修改源程序。

< 从 CC78K0S 到其他 C 编译器 >

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

限制

- 无法改变给出了向量表相关的字段 （例如， @@VECT02 等）的区段名称。
- 如果在于另一文件中出现具有相同名称的两个或两个以上区段，其中某个区段用 AT 指定了起始地址，则出现链接错误。
- 用来指示该区段用于 bank 函数的区段名称 （@@BANK1 等）不能改变。
- 当改变编译器输出区段名称 @@DATS， @@BITS 和 @@INIS 时，指定地址的范围限制在 0FE20H 至 0FED7H 内。

注意事项

- 区段 （section）等于汇编程序的字段 （segment）。
- 编译器不检验新区段名称是否和另一符号同名。因此，用户必须通过汇编输出汇编列表 , 检查块名是否重复。
- 如果使用 #pragma section 改变了 ROMization 相关的区段名称 （*），则用户必须自行改变启动例程，且职责自负。

  (*) ROM 化相关区段名称

```
@@R_INIT , @@R_INIS , @@INIT , @@INIS
```

  改变有关 ROM 化的区段时要使用的启动例程，下文将说明一个结束模块的示例。

[通过启动例程改变 ROMization 相关区段名称的示例]

下文为改变有关 ROM 化的区段名称，并相应改变启动程序 （cstart.asm 或 cstartn.asm）和结束模块（rom.asm）的示例。

< C 源代码 >

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

如果通过以上所示的 #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：新区段开始第一个标志声明和复制初值的部分，并将结束标签的声明添加到结束模块。

RTT1_S 和 RTT1_E 为区段 RTT1 的开始和结束标签名称，且 TT1_S 和 TT1_E 为区段 TT1 的开始和结束标签名称。

(a) 改变启动例程 cstartx.asm

(i) 添加标签声明，表示改变名称的区段的结束

```
    :
EXTRN   _main , _exi t , _@STBEG
EXTRN   _?R_INIT , _?R_INIS , _?DATA , _?DATS

EXTRN   RTT1_E , TT1_E  <- 添加 RTT1_E 和 TT1_E 的 EXTRN 声明
    :
```

(ii) 添加区段以将初值从具有已改变名称的 RTT1 区段复制到 TT1 区段的程序代码。

```
    :
LDATS1 :
        MOVW    AX , HL
        CMPW    AX , #_?DATS
        BZ      $LDATS2
        MOV     A , #0
        MOV     [ HL ] , A
        INCW    HL
        BR      $LDATS1
LDATS2 :                                  ┐
        MOVW    DE , #TT1_S               │
        MOVW    HL , #RTT1_S              │
LTT1 :                                    │
        MOVW    AX , HL                   │
        CMPW    AX , #RTT1_E              │  添加的代码，用来将初值从已改变名称的
        BZ      $LTT2                     │  RTT1 区段复制到 TT1 区段
        MOV     A , [ HL ]                │
        MOV     [ DE ] , A                │
        INCW    HL                        │
        INCW    DE                        │
        BR      $LTT1                     │
LTT2 :                                    ┘
;
        CALL    !_main ; main ( ) ;
        MOVW    AX , #0
        CALL    !_exit ; exit ( 0 ) ;
        BR      $$
;
```

(iii) 设置已改变名称的区段的开始标签。

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

RTT1            CSEG    ; 指示 RTT1 区段的开始
RTT1_S :                ; 添加标志设定
TT1             DSEG    ; 指示 TT1 区段的开始
TT1_S :                 ; 添加标签设定

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

(b) 改变结束模块 rom.asm 的示例

(i) 添加标签声明，表示已改变名称的区段的结束

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

PUBLIC          RTT1_E , TT1_E          <- 添加 RTT1_E 和 TT1_E

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

(ii) 设置指示结束的标签

```
    :
RTT1            CSEG    ; 添加标签的设置指示 RTT1 区段结束。
RTT1_E :                ;添加标签设置

TT1     DSEG            ; 添加标签的设置指示 TT1 区段结束。
TT1_E :                 ; 添加标签设置

;
        END
```

(16) 二进制常量 （二进制常量 0bxxx）

功能

- 在可以用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。

用法

- 说明 C 源代码中的二进制常量。以下展示二进制常量的使用方法。

```
0b      二进制数
0B      二进制数
```

备注　　二进制数 : 不是 “0” 就是 “1”

- 二进制常量以 0b 或 0B 开始，且随后是数 0 或 1 的表。
- 二进制常量的值以 2 为基底进行计算。
- 二进制常量的类型为可以在下表中表示的值的第一个。

(i) 下标二进制数 : 整型
无符号整型
长整型
无符号长整型

(ii) 下标为 u 或 U : 无符号整数型
无符号长整型

(iii) 下标为 l 或 L : 长整数型
无符号长整型

(iv) 下标 u 或 U 和下标 l 或 L
: 无符号长整型

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

< 从其他 C 编译器导入到 CC78K0S >

- 无需修改。

< 从 CC78K0S 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。

(17) 模块名更改函数 (#pragma name)

功能

- 将指定的模块名称的前八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八个字母输出到汇编表文件指定时作为符号信息（MOD_NAM），且当指定 -ng 选项时作为 NAME 准伪指令。
- 如果模块名称指定了九个或九个以上字母，则输出警告消息。
- 如果说明描述中出现未经认可的字母，则出现错误且处理程序异常终止。
- 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。

用法

- 以下展示说明方法。

```
#pragma name     模块名
```

模块名称必须由 OS 授权为文件名称的字符组成，除 “（“ ”）” 之外区分大写 / 小写字母。

示例

```
#pragma name    module1
    :
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 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。

< 从 CC78K0S 到其他 C 编译器 >

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

(18) 循环移位函数 (#pragma rot)

功能

- 输出对表达式值进行移位的代码到目标中，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

- 即使未对移位的处理过程进行描述，循环移位函数还是可以通过 C 源代码或 ASM 说明语句来实现。

用法

- 在源文件中描述的方法和以与函数调用的格式相同。共有以下 4 个函数名。

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

[循环移位函数表]

(a) unsigned char rorb (x , y) ;
unsigned char x ;
unsigned char y ;

将 x 向右移位 y 次。

(b) unsigned char rolb (x , y) ;
unsigned char x ;
unsigned char y ;

将 x 向左移位 y 次。

(c) unsigned int rorw (x , y) ;
unsigned int x ;
unsigned char y ;

将 x 向右移位 y 次。

(d) unsigned int rolw (x , y) ;
unsigned int x ;
unsigned char y ;

将 x 向左移位 y 次。

备注　　上述函数声明不受 -zi 选项影响。

- 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。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 不会产生变量定义 / 调用或函数的定义 / 调用的预处理指令。

- #pragma 之后的关键字可以用大写或小写字母表示。

示例

< C 源代码 >

```
#pragma rot
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned char   c ;
void    main ( ) {
        c = rorb ( a , b ) ;
}
```

< 输出的汇编源程序 >

```
mov     a , !_b
mov     c , a
mov     a , !_a
ror     a , 1
dbnz    c , $$-1
mov     !_c , a
```

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

< 从其他 C 编译器导入到 CC78K0S >

- 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对源程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- #pragma rot 语句被 #ifdef 删除或者屏蔽。
- 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。

(19) 乘法函数 (#pragma mul)

功能

- 输出表达式的值乘以对象的代码到目标中，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

- 生成的代码与 CC78K0 兼容，且利用会自动对应乘法指令 I/O 的数据大小的代码。因此，可以生成的代码量具有小于普通乘法表达式描述大小的代码。

使用方法

- 在源文件中描述的方法和函数调用的格式相同。

```
mulu
```

[乘法函数列表]

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。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 不会产生变量定义 / 调用或函数的定义 / 调用的预处理指令。

- #pragma 之后的关键字可以用大写或小写字母表示。

限制

- 不内联展开乘法函数 , 而是通过库调用。

例

< C 源代码 >

```
#pragma mul
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned int    i ;
void    main ( )
{
        i = mulu ( a , b ) ;
}
```

< 编译器的输出目标 >

```
mov     a , !_b
mov     x , a
mov     a , !_a
callt   [ @@mulu ]
movw    de , #_i
callt   [ @@deist ]
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

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

< 从 CC78K0S 到其他 C 编译器 >

- #pragma mul 语句被 #ifdef 删除或者屏蔽。乘法函数的名称可以用作函数名称。
- 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。

(20) 除法函数 (#pragma div)

功能

- 输出从对象除以表达式值的代码，通过直接内联展开而不是函数调用，并生成对象文件。
- 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 CC78K0 兼容，且会自动对应除法指令 I/O 的数据大小。因此，生成的代码量小于普通除法表达式描述。

使用方法

- 在源文件中描述的方法和函数调用的格式相同共有以下 2 个除法函数名。

```
divuw , moduw
```

[除法函数表]

(a) 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。

(b) unsigned char moduw (x , y) ;
unsigned int x ;
unsigned char y ;

运算 x 和 y 的无符号除法并返回余数。

备注　　上述函数声明不受 -zi 选项影响。

- 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。
  (i) 注释
  (ii) 其他 #pragma 指令
  (iii) 不会产生变量定义 / 调用或函数的定义 / 调用的预处理指令。
- #pragma 之后的关键字可以用大写或小写字母表示。

限制

- 不内联展开除法函数，而通过库调用。

例

< C 源代码 >

```
#pragma div
unsigned int    a = 0x1234 ;
unsigned char   b = 0x12 ;
unsigned char   c ;
unsigned int    i ;
void    main ( ) {
        i = divuw ( a , b ) ;
        c = moduw ( a , b ) ;
}
```

< 编译器的输出目标 >

```
mov     a , !_b
mov     c , a
movw    de , #_a
callt   [ @@deilo ]
callt   [ @@divuw ]
movw    de , #_i
callt   [ @@deist ]
mov     a , !_b
mov     c , a
movw    de , #_a
callt   [ @@deilo ]
callt   [ @@divuw ]
mov     a , c
mov     !_c , a
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果编译器不使用除法函数，则无需修改。
- 要改为除法函数，请根据以上的方法对源程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- #pragma div 语句被 #ifdef 删除或者屏蔽。除法函数名称可以用作函数名称。
- 要用作除法函数，请根据每个编译器的具体规范修改程序 （#asm， #endasm 或 asm() ; 等）。

(21) BCD 运算函数 (#pragma bcd)

功能

- 输出对象中的表达式的值执行 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。
- 如果没有对应的 #pragma 指令，则 BCD 操作循环移位函数被视为普通函数。

效果

- 即使未对 BCD 运算的过程进行描述， BCD 运算函数也可以通过 C 源代码或 ASM 语句来实现。

用法

- 在源文件中描述的方法和函数调用的格式相同。存在 BCD 运算的 13 类函数名称，如下列出。 如需更多信息，请参阅本章稍后介绍的 [BCD 运算函数表]。

```
adbcdb, sbbcdb, adbcdbe, sbbcdbe, adbcdw, sbbcdw, adbcdwe,
sbbcdwe, bcdtob, btobcde, bcdtow, wtobcd, btobcd
```

- 除法函数的用法通过模块的 #pragma bcd 指令声明。然而，以下各项可以在 #pragma bcd 之前编码。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 预处理指令不能产生变量定义 / 引用或函数定义 / 引用。

- 可以在 #pragma 之后用大写或小写字母用于作为关键字来说明。

限制

- BCD 运算函数名称不能用作函数名称。
- BCD 运算函数用小写字母编码。如果使用大写字母，则这些函数被视为普通函数。
- 静态模式不支持 adbcdwe 和 sbbcdwe。

示例

< C 源代码 >

```
#pragma  bcd
unsigned char   a = 0x12 ;
unsigned char   b = 0x34 ;
unsigned char   c ;
void    main ( )
{
        c = adbcdb ( a , b ) ;
        c = sbbcdb ( b , a ) ;
}
```

< 输出的汇编源程序 >

```
mov     a , !_a
add     a , !_b
adjba
mov     !_c , a
mov     a , !_b
sub     a , !_a
adjbs
mov     !_c , a
```

[BCD 运算函数表]

(a) unsigned char adbcdb (x , y) ;

unsigned char x ;

unsigned char y ;

通过 BCD 调整指令实现十进制加法运算。

(b) unsigned char sbbcdb (x , y) ;

unsigned char x ;

unsigned char y ;

通过 BCD 调整指令实现十进制减法运算。

(c) unsigned int adbcdbe (x , y) ;

unsigned char x ;

unsigned char y ;

通过 BCD 调整指令实现十进制加法 （带结果展开）运算。

(d) unsigned int sbbcdbe (x , y) ;

unsigned char x ;

unsigned char y ;

通过 BCD 调整指令实现十进制减法 （带结果展开）运算。如果出现借位，则高位数据位被置为 0x99。

(e) unsigned int adbcdw (x , y) ;

unsigned int x ;

unsigned int y ;

通过 BCD 调整指令实现十进制加法运算。

(f) unsigned int sbbcdw (x , y) ;

unsigned int x ;

unsigned int y ;

通过 BCD 调整指令实现十进制减法运算。

(g) unsigned long adbcdwe (x , y) ;

unsigned int x ;

unsigned int y ;

通过 BCD 调整指令实现十进制加法 （带结果展开）运算。

(h) unsigned long sbbcdwe (x , y) ;
unsigned int x ;
unsigned int y ;

通过 BCD 调整指令实现十进制减法 （带结果展开）运算。如果出现借位，则更高的位设定为 0x9999。

(i) unsigned char bcdtob (x) .
unsigned char x ;

十进制数转换成二进制数。

(j) unsigned int btobcde (x) ;
unsigned char x ;

二进制数转换成十进制数。

(k) unsigned int bcdtow (x) ;
unsigned int x ;

十进制数转换成二进制数。

(l) unsigned int wtobcd (x) ;
unsigned int x ;

十进制数转换成二进制数。然而，如果 x 的值超过 10000，则返回 0xffff。

(m) unsigned char btobcd (x) ;
unsigned char x ;

十进制数转换成相应二进制数。然而，舍弃溢位。

备注　　上述函数声明不受 -zi 和 -zl 选项影响。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

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

< 从 CC78K0S 到其他 C 编译器 >

- 通过 #ifdef 可对 #pragma bcd 语句进行删除或隔离。 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）。

(22) 数据插入函数 (#pragma opc)

功能

- 将数据常量插入到当前地址。
- 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插入到代码区域而不使用 ASM 声明。

  当 ASM 使用时，如果没有汇编器中间件则不能获得对象。但是，如果使用数据插入函数，则对象可以在没有汇编器的情况下获得。

用法

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

  [数据插入函数列表]

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

  将参数中描述的常量值插入到当前地址。

  参数只能使用常量。

- 通过 #pragma opc 指令声明数据插入函数的用法。

  然而，以下各项可以在 #pragma opc 之前说明。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 不会产生变量定义 / 调用或函数的定义 / 调用的预处理指令。

- #pragma 之后的关键字可以用大写或小写字母表示。

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。如果将其用小写字母表示，则其按普通函数进行处理。

示例

< C 源代码 >

```
#pragma opc
void    main ( ) {
        __OPC ( 0xBF ) ;
        __OPC ( 0xA1 , 0x12 ) ;
        __OPC ( 0x10 , 0x34 , 0x12 ) ;
}
```

< 编译器的输出目标 >

```
_main :
; line  4 : __OPC ( 0xBF ) ;
        DB      0BFH
; line  5 : __OPC ( 0xA1 , 0x12 ) ;
        DB      0A1H
        DB      012H
; line  6 : __OPC ( 0x10 , 0x34 , 0x12 ) ;
        DB      010H
        DB      034H
        DB      012H
; line  7 : }
        ret
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 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，根据以上使用方法源程序进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- #pragma opc 语句被 #ifdef 删除或者限制。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)。

(23) 静态模式

功能

- 所有参数通过寄存器传递 (参阅 "11.7.5 静态模式函数调用接口 ")。
- 通过寄存器传输的函数参数，并将其分配在函数 - 特定的静态区域。
- 自动变量分配到函数 - 特定的静态区域。
- 在使用 leaf 函数注情况下，参数和自动变量分配到 saddr 区域内 0FEDFH 以下的位置，按照描述顺序从高位地址开始存放。因为 addr 区域通常被所有模块中的 leaf 函数使用，所以此区称作共享区。共享区的最大尺寸通过 -SM 选项的参数来指定。

```
-sm [ nn ] : nn = 0-16
```

nn 个字节分配给共享区，和剩余部分分配到函数 - 专用静态区。如果指定 nn = 00 或忽略此指定，则不使用共享区。

注　对于函数体内部未调用函数的情况，不需要说明 norec/__leaf，因为编译器会自动执行判定，所以不需要说明 norec/_ _leaf。

- 可能以将 sreg/__sreg 关键字添加到函数参数和自动变量之前。被声明为 sreg/__sreg 的函数参数和自动变量会被分配到 saddr 区域。结果，可以很方便的进行位处理。
- 通过指定 -rk 选项，函数参数和自动变量 （除函数的静态变量）分配到 saddr 且可以进行位处理 （请参阅“(3) 如何使用 saddr 区域 (sreg / __sreg)”）。
- 编译器自动执行以下宏定义。

```
#define __STATIC_MODEL__        1
```

效果

- 通常情况下，访问静态区域的指令比访问静态帧的指令更短、更快。因此，可能在缩短对象代码量的同时改善运行速度。
- 不执行 saddr 区的自动变量和变量 （中断函数的寄存器变量、 norec 函数参数 / 自动变量、运行时库参数）的保存 / 恢复操作，而使用 saddr 区域来保存 / 恢复，结果，可以提高中断处理的速度。
- 因为数据区通常由几个 leaf 函数使用，所以可以节约存储空间。

用法

- 编译期间指定 -sm 选项。

  静态模式调用在此情况下的对象，而正常模式调用不具有 -sm 选项规范的对象。

示例

- -sm4 规范的示例如下所示。

  < C 源代码 >

```
void    sub ( char , char , char ) ;
void    main ( )
{
        char    i = 1 ;
        char    j , k ;
        j = 2 ;
        k = i + j ;
        sub ( i ,  j , k ) ;
}
void    sub ( char p1 , char p2 , char p3 )
{
        char    a1 , a2 ;
        a1 = 1<<p1 ;
        a2 = p2 + p3 ;
}
```

< 编译器的输出目标 >

```
@@DATA          DSEG
?L0003 :        DS      ( 1 )           ;main 函数的自动变量 i
?L0004 :        DS      ( 1 )           ;main 函数的自动变量 j
?L0005 :        DS      ( 1 )           ;main 函数的自动变量 k
?L0008 :        DS      ( 1 )           ;sub 函数的自动变量 a2

; line  1 :     void sub ( char , char , char ) ;
; line  2 :     void main ( )
; line  3 :     {

@@CODE  CSEG
_main :
; line  4 :     char    i = 1 ;
                mov     a , #01H        ; 1
                mov     !?L0003 , a     ; i     ; 自动变量 i
; line  5 :     char    j , k ;
; line  6 :     j = 2 ;
                inc     a
                mov     !?L0004 , a     ; j     ; 自动变量 j
; line  7 :     k = i + j ;
                add     a , !?L0003     ; i     ; i 加 j
                mov     !?L0005 , a     ; k     ; 代替 k
; line  8 :     sub ( i , j , k ) ;
                movw    hl , ax                 ; 通过寄存器 H 传输 k
                mov     a , !?L0004     ; j
                movw    bc , ax                 ; 通过寄存器 B 传输 j
                movw    a , !?L0003     ; i     ; 通过寄存器 A 传输 i
                call    !_sub
; line  9 :     }
                ret
; line  10 :    void sub ( char p1 , char p2 , char p3 )
; line  11 :    {
_sub :
                mov     _@KREG15 , a            ; 分配第 1 个参数到
                                                ; 共享区
                movw    ax , bc
                mov     _@KREG14 , a            ; 分配第 2 个参数到
                                                ; 共享区
                movw    ax , hl
                mov     _@KREG13 , a            ; 分配第 1 个参数到
                                                ; 共享区
; line  12 :    char       a1 , a2 ;
; line  13 :    a1 = p1 ;
                mov     a , _@KREG15    ; p1    ; 第 1 个参数 p1
                mov     _@KREG12 , a    ; a1    ; 自动 a1 是在
                                                ; 共享区
; line  14 :    a2 = p2 + p3 ;
                mov     a , _@KREG14    ; p2    ; 第 2 个参数 p2
                add     a ,_@KREG13     ; p3    ; 添加第 3 个参数 p3
                mov     !?L0008 , a     ; a2    ; 自动变量 a2 在
                                                ; 专用函数区
; line   15 :      }
                ret
```

限制

- 静态模式模块不能与正常模式相链接。然而，即使共享区的最大尺寸不同，静态模式模块也可以彼此连接。
- 不支持浮点数。如果出现 float 和 double 关键字，则出现致命错误。
- 参数限制为最多 3 个参数，总共 6 个字节。
- 因为不通过堆栈传输参数，所以不能使用可变长度参数。使用可变变量长度自变量参数导致错误。
- 不能使用结构体 / 共用体结构 / 集合的参数和返回值。这些种类型的参数和返回值会导致错误。
- 不能使用 noauto/norec/__leaf 函数。输出警告消息并忽略说明 (参阅 "(5) noauto 函数 (noauto)", "(6) norec 函数 (norec)")。
- 不能使用递归函数。因为函数参数和自动变量区域是静态保留的，所以不能使用递归函数。 编译器可以检测的递归函数，并产生错误。
- 不能忽省略原型声明。 如果既没有函数的真实定义也没有原型声明，则不管是否存在函数调用，都会产生错误。
- 由于参数的限制和不能使用递归函数，因此某些标准库不能无法使用。
- 如果未指定 -ZL 选项，则输出警告并且处理过程和指定了 -ZL 选项一样。因此 long 型始终视为 int 型 （请参阅 “(24) 类型更改 (-ZI)”）。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 当创建正常模式对象时，除非指定 -sm 选项，否则无需修改源文件。
- 要创建静态模式对象，根据以上方法进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 如果由另一编译器进行重新编译，则无需修改源文件。

注意事项

- 因为静态保护参数 / 自变量自动变量静态保留，所以可能破坏递归函数中参数 / 自动变量的内容。当函数自身直接调用时，出现错误。然而，因为编译器无法检测此过程，所以当在调用其他函数之后又进行函数自身调用时，不出现错误报警。
- 中断期间，如果通过中断服务 （中断函数和由中断函数调用的函数）调用处理的函数，则可能破坏参数 / 自动变量的内容。
- 中断期间，即使处理中的函数正在使用共享区时，也不进行共享区保存 / 返回。

(24) 类型更改 (-ZI)

(a) 从 int/short 型改为 char 型

功能

- int 和 short 型视为 char 型。换句话说， int 和 short 描述等于 char 描述。
- 下文给出了类型调整的详细内容 （某些 -qu 选项受影响）。

表 11-12 类型修改类型调整的详细信息 （从 int 和 short 型改为 char 型）

C 源代码中的类型说明	选项	修改之后的类型
short， short int， int	具有 -qu	unsigned char
short， short int， int	不具有 -qu	signed char
unsigned short, unsigned short int, unsigned, unsigned int	-	unsigned char
signed short, signed short int, signed, signed int	-	signed char

- 将警告消息会指向 int 或 short 关键字在 C 源代码中第一次出现的行号。
- 不管是否指定， -QC 选项都会有效。当未进行 -qc 选项规范时，输出警告消息且 -qc 选项变得有效。
- 如果同时指定 -za 选项 （诸如 -zai 选项），输出警告消息 （仅当指定 -w2 警告等级时）。
- 以下类型的语句可以通过类型声明符说明，如果没有特别声明，即视为 char 型。

  (i) 函数的参数和返回值

  (ii) 类型声明符忽略变量 / 函数声明

- 编译器自动执行以下宏定义。

```
#define __FROM_INT_TO_CHAR__    1
```

- 某些标准库不能使用。

用法

- 指定 -ZI 选项。

限制

- 指定的 -ZI 和未指定的 -ZI 模块无法连接。

(b) 从 long 型改为 int 型

功能

- long 型视为 int 型。换句话说， long 描述等于 int 描述明。
- 下文给出了类型调整的详细内容。

表 11-13 类型调整的详细内容 （从 long 型改为 int 型）

C 源代码中的类型说明	修改之后的类型
unsigned long, unsigned long int	unsigned int
long, long int, signed long, signed long int	signed int

- 警告消息会指向 long 关键字在 C 源代码中第一次出现的行号。
- 如果同时指定 -za 选项 （-zal），则输出警告消息 （仅当指定 -w2 警告等级时）。
- 编译器自动执行以下宏定义。

```
#define __FROM_LONG_TO_INT__    1
```

- 某些标准库不能使用。

用法

- -zl 选项被选定。

限制

- 指定的 -zl 和未指定的 -zl 模块无法连接。

(25) Pascal 函数 (__pascal)

功能

- 函数调用期间生成将参数放入堆栈的代码由被调用方产生，而不是由发起函数调用的一方负责。

效果

- 如果在程序中多次出现函数调用，则可以缩短目标代码。

用法

- 当声明函数时，将 __pascal 属性修饰词添加到开始处。

限制

- pascal 函数不支持可变长度参数。如果定义可变长度参数，则输出警告且并忽略 __pascal 关键字。
- pascal 函数中 , 关键字 norec / __interrupt 不能被指定。如果在 norec 关键字的情况下指定它们 , 将忽略 __pascal 关键字 , 如果在 __interrupt / __interrupt_brk / __rtos_interrupt 关键字的情况下 , 则输出错误。
- 如果原型声明不完整，则不能正常操作，因此当 pascal 函数缺少定义或原型声明缺失时，输出警告消息。
- 当指定静态模式选项 （-sm）时，不支持 Pascal 函数。如果当使用 pascal 函数时指定 -sm 选项，则输出的警告消息指向 __ pascal 关键字第一次出现的位置，且忽略输入文件中的 __ pascal 关键字。

说明

- 指定 -zr 选项，使所有函数都变成 pascal 函数。但是，如果 pascal 函数被调用的次数很少，则可能增加目标代码。

示例

< C 源代码 >

```
__pascal        int     func ( int a , int b , int c ) ;
void    main ( )
{
        int     ret_val ;

        ret_val = func ( 5 , 10 , 15 ) ;
}
__pascal        int     func ( int a , int b , int c )
{
        return ( a + b + c ) ;
}
```

< 编译器的输出目标 >

```
_main :
        push    hl
        movw    ax , #02H
        callt   [ _@cprep ]
        movw    ax , #0FH       ; 15
        push    ax
        mov     x , #0AH        ; 10
        push    ax
        mov     x , #05H        ; 5
        call    !_func
        movw    ax , bc         ; 不能在这儿修改堆栈。
        mov     [ hl + 1 ] , a  ; ret_val
        xch     a , x
        mov     [ hl ] , a      ; ret_val
        pop     ax
        pop     hl
        ret

_func :
        push    hl
        push    ax
        movw    ax , sp
        movw    hl , ax
        mov     a , [hl]        ; a
        mov     a , [ hl + 6 ]  ; b
        xch     a , x
        mov     a , [ hl + 1 ]  ; a
        addc    a , [ hl + 7 ]  ; b
        xch     a , x
        add     a , [ hl + 8 ]  ; c
        xch     a , x
        addc    a , [ hl + 9 ]  ; c
        movw    bc , ax
        pop     ax
        pop     hl
        pop     de              ; 获得返回地址
        pop     ax              ;
        pop     ax              ; 调用方所消耗的 4 字节堆栈在此调整
        pop     de              ; 重新加载返回地址
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用保留字 __ pascal，则无需修改。
- 要改为 Pascal 函数，根据以上方法进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 通过使用 #define 保持兼容性。
- 通过这样转换， pascal 函数被视为普通函数。

(26) 函数调用接口的自动 pascal 函数化 (-zr)

功能

- 除了 nore / __interrupt 可变长度参数函数，所有函数都被添加 __pascal 属性。

用法

- 编译期间指定 -zr 选项。

限制

- 指定了 -ZR 选项的模块和其中未指定 -ZR 选项的模块无法连接。如果进行链接，则导致链接错误
- 静态模式规范选项 （-sm）和 -zr 选项不能同时指定。如果同时指定，则输出警告消息并且忽略 -zr 选项。
- 因为数学函数标准库不支持 pascal 函数，所以当使用数学函数标准库时， -zr 选项无法使用。

  备注： 有关 pascal 函数调用接口的详细信息 , 参阅 "11.7.6 Pascal 函数调用接口 "。

(27) 参数 / 返回值的 int 扩展限制方法 (-ZB)

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。
- 函数参数的原型已经定义为 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。

用法

- 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 ( )
{
        c = func1 ( d , e ) ;
        c = func2 ( d , e ) ;
}
unsigned char   func1 ( unsigned char x , unsigned char y )
{
        return  x + y ;
}
```

- 当指定 -ZB 时

< 编译器的输出目标 >

```
_main :
; line  5 :     c = func1 ( d , e ) ;
        mov     a , !_e
        xch     a , x           ; 不在表达式中执行
        push    ax
        mov     a , !_d
        xch     a , x           ; 不在表达式中执行
        call    !_func1
        pop     ax
        mov     a , c
        mov     !_c , a
; line  6 :     c = func2 ( d , e ) ;
        mov     a , !_e
        xch     a , x
        xch     a , a           ; 因为没有原形声明，执行 int 扩展
        push    ax
        mov     a , !_d
        xch     a , x
        xor     a , a           ; 因为没有原形声明，执行 int 扩展
        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
; line  10 :    return x + y ;
        mov     a , [ hl ]      ; x
        add     a , [ hl + 6 ]  ; y
        mov     c , a
; line  11 :    }
        pop     ax
        pop     hl
        ret
        END
```

限制

- 如果此函数的函数体定义与原型声明不同，则程序可能错误操作。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果所有函数体定义的原型声明都无法正确运行，则进行校正原型声明。或者 , 不指定 -ZB 选项。

< 从 CC78K0S 到其他 C 编译器 >

- 无需修改。

(28) 数组偏移量计算的简便方法 (-qw2 / -qw4)

功能

- 当计算 char/unsigned char/unsigned int/short/unsigned short 型数组的偏移量，且索引为 unsigned char 型变量时，生成的代码仅对低位字节进行计算 ，默认为不存在进位。
- 当指定 -qw2 选项时，仅当使用 unsigned char 变量引用的 saddr 区域配置序列时，生成的代码仅对低位字节进行计算来产生偏移量。
- 当指定 -qw4 选项时，仅当使用 unsigned char 变量引用 saddr 区域配置序列时，通过基于大小的优先权生成仅计算偏移量的低字节的代码。

效果

- 因为简化了偏移量计算的代码，所以实现目标代码的缩减和运行速度的提高。

用法

- 编译期间指定 -qw2 和 -qw4 选项。

示例

< C 源代码 >

```
unsigned char   c ;
unsigned char           ary [ 10 ] ;
sreg unsigned char      sary [ 10 ] ;
void    main ( )
{
        unsigned char   a ;

        a = ary [ c ] ;
        a = sary [ c ] ;
}
```

- 当 -qw2 被指定

< 编译器的输出对象 >

```
_main :
        push    hl
        push    ax
        movw    ax , sp
        movw    hl , ax
; line  6 :     unsigned char   a ;
; line  7 :     a = ary [ c ] ;
        mov     a , !_c
        xch     a , x
        xor     a , a
        addw    ax , #_ary
        movw    de , ax
        mov     a , [ de ]
        mov     [ hl + 1 ] , a          ; a
; line  8 :     a = sary [ c ] ;
        mov     a , !_c
        add     a , #low ( _sary )
        mov     e , a                   ; 反计算低字节
        mov     d , #0FEH               ; 254
        mov     a , [ de ]
        mov     [ hl + 1 ] , a          ; a
; line  9 : }
        pop     ax
        pop     hl
        ret
        END
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 无需修改

< 从 CC78K0S 到其他 C 编译器 >

- 无需修改。

(29) 寄存器直接引用函数 (#pragma realregister)

功能

- 输出访问对象寄存器的代码，通过直接内联展开而不是函数调用，并生成对象文件。
- 如果没有对应的 #pragma 指令时，寄存器直接引用函数被视为普通函数。

效果

- 可以通过 C 源文件中的描述方便的进行，寄存器的访问。

用法

- 在源文件中描述的方法和函数调用的格式相同 （请参阅本章稍后介绍的 [寄存器直接引用函数表]）。

  存在 21 种类型的寄存器直接引用函数名称。

```
__geta , __seta , __getax , __setax , __getcy , __setcy , __set1cy ,
__clr1cy , __not1cy , __inca , __deca , __rora , __rorca , __rola ,
__rolca , __shla , __shra , __ashra , __nega , __coma , __absa
```

- 在模块中使用 #pragma realregister 指令，声明寄存器直接引用函数。

  以下可以在 #pragma realregister 指令之前说明。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 预处理指令不能产生变量定义 / 引用或函数定义 / 引用。

示例

<C 源代码 >

```
#pragma realregister
unsigned char   c = 0x88 , d , e ;
void    main ( )
{
        __seta ( c ) ;          /* 设置 A 寄存器中 C 变量的值 */
        __shla ( ) ;            /* 逻辑左移 1 位 */
        d = __geta ( ) ;        /* 设置变量 d 中 A 寄存器的值 */
        if ( __getcy ( ) ) {    /* 引用 CY（检验溢出） */
                e = 1 ;         /* 当 CY = = 1 时将 e 设定为 1 */
        }
}
```

< 编译器的输出目标 >

```
_main :
; line  5 :   __seta ( c ) ;      /* 设置 A 寄存器中 C 变量的值 */
        mov   a , !_c
; line  6 :   __shla ( ) ;        /* 逻辑左移 1 位 */
        add   a , a
; line  7 :   d = __geta ( ) ;    /* 设置变量 d 中 A 寄存器的值 */
        mov   !_d , a
; line  8 :   if ( __getcy ( ) ) {    /* 引用 CY（检验溢出） */
        bnc   $?L0003
; line  9 :   e = 1 ;             /* 当 CY = = 1 时设置 e 的值为 1 */
        mov   a , #01H ;1
        mov   !_e , a
?L0003 :
; line  10 :    }
; line  11 :    }
        ret
```

[寄存器直接引用函数表]

(1) unsigned char __geta (void) ;

获得 A 寄存器的值。

(2) void __seta (unsigned char x) ;

设置 A 寄存器的 x。

(3) unsigned int __getax (void) ;

获得 AX 寄存器的值。

(4) void __setax (unsigned int x) ;

设置 AX 寄存器的 x。

(5) bit __getcy (void) ;

获得 CY 标志值。

(6) void __setcy (unsigned char x) ;

设置 CY 标志的 x 的低 1 位。

(7) void __set1cy (void) ;

生成 set1 CY 指令。

(8) void __clr1cy (void) ;

生成 Clr1 CY 指令。

(9) void __not1cy (void) ;

生成 not1 CY 指令。

(10) void __inca (void) ;

生成 inc a 指令。

(11) void __deca (void) ;

生成 dec a 指令。

(12) void __rora (void) ;

生成 1 ror a, 指令。

(13) void __rorca (void) ;

生成 1 rorc a, 指令。

(14) void __rola (void) ;

生成 1 rol a, 指令。

(15) void __rolca (void) ;

生成 1 rolc a, 指令。

(16) void __shla (void) ;

生成执行 A 寄存器逻辑左移 1 位的代码。

(17) void __shra (void) ;

生成执行 A 寄存器逻辑右移 1 位的代码。

(18) void __ashra (void) ;

生成执行 A 寄存器算术右移 1 位的代码。

(19) void __nega (void) ;

生成获得 A 寄存器二补数的代码。

(20) void __coma (void) ;

生成获得 A 寄存器一补数的代码。

(21) void __absa (void) ;

生成获得 A 寄存器绝对值的代码。

限制

- 寄存器直接引用的函数名称只能用作函数名称。寄存器直接引用函数以小写字母表示。以大写字母表示的函数被视为普通函数。
- 通过 __seta, __setax, 和 __setcy 函数设定的 A 和 AX 寄存器和 CY 标志的值，在下一次代码的生成过程中不保留。
- 由 A 和 AX 寄存器和 __geta, __getax, and __getcy 函数的 CY 标志引用的时序对应于表达式的评估序列。

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 如果不使用寄存器直接引用函数，则无需修改。
- 要改为寄存器直接引用函数，请使用上述方法。

< 从 CC78K0S 到其他 C 编译器 >

- "#pragma realregister" 语句应该用 #ifdef 语句删除或者屏蔽。寄存器直接引用函数名称能用作为函数名称。
- 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)。

注意事项

- 在执行寄存器直接引用函数之前不保证 CY， A， AX 的内容将按照设计意图所希望的进行保存。因此，建议在其通过第一次表达式说明改变值之前，使用此函数，方法是在表达式的开头项中使用。

(30) 内存控制函数 (#pragma inline)

功能

- 输出由标准库内存操控函数 memcpy 和 memset 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。
- 如果没有对应的 #pragma 指令时，生成调用标准库函数的代码。

效果

- 与调用标准库函数时相比，提高了运行速度。
- 如果指定的字符数量为常量，则可以缩短目标代码。

用法

- 在源文件中描述的方法和函数调用的格式相同。
- 以下各项可以在 #pragma inline 之前说明。

  (i) 注释

  (ii) 其他 #pragma 指令

  (iii) 不产生变量定义 / 引用或函数定义 / 引用的预处理指令

示例

< C 源代码 >

```
#pragma inline
char    ary1 [ 100 ] , ary2 [ 100 ] ;
void    main ( )
{
        memset ( ary1 , ' A ' , 50 ) ;
        memcpy ( ary1 , ary2 , 50 ) ;
}
```

- 当未指定 -SM 时

< 编译器的输出对象 >

```
_main :
        push    hl
; line  5 :     memset ( ary1 , ' A ' , 50 ) ;
        movw    de , #_ary1
        mov     a , #041H       ; 65
        mov     c , #032H       ; 50
        mov     [ de ] , a
        incw    de
        dbnz    c , $$-2
; line  6 :     memcpy ( ary1 , ary2 , 50 ) ;
        movw    de , #_ary1
        movw    hl , #_ary2
        mov     c , #032H       ; 50
        mov     a , [ hl ]
        mov     [ de ] , a
        incw    de
        incw    hl
        dbnz    c , $$-4
; line  7 :    }
        pop     hl
        ret
```

- 当指定 -SM 时

```
_main :
        push    de
; line  5 :     memset ( ary1 , ' A ' , 50 ) ;
        movw    hl , #_ary1
        mov     a , #041H       ; 65
        mov     c , #032H       ; 50
        mov     [ hl ] , a
        incw    hl
        dbnz    c , $$-2
; line  6 :     memcpy ( ary1 , ary2 , 50 ) ;
        movw    hl , #_ary1
        movw    de , #_ary2
        mov     c , #032H       ; 50
        mov     a , [ de ]
        mov     [ hl ] , a
        incw    de
        incw    hl
        dbnz    c , $$-4
; line  7 :    }
        pop     de
        ret
```

兼容性

< 从另一个 C 编译器到 CC78K0S >

- 如果不使用内存运算函数，则无需修改。
- 在改变内存运算函数，使用方法如上。

< 从 CC78K0S 到另一个 C 编译器 >

- "#pragma inline" 指令应该用 #ifdef 语句删除或者限制。

(31) 绝对地址分配规范 (__directmap)

功能

- 由函数中 __directmap 和 static 变量声明的外部变量，其初值被视为分配的地址，且该变量分配到指定地址。
- C 源代码中的 __directmap 变量按常规变量进行处理。
- 因为初值视为分配 地址规范，所以不能定义初值，并保留一个未定义的值。
- 下文给出了可指定的地址规范范围、保留区域范围是通过的模块链接的为特定地址保留的范围，变量双重检验范围。

地址规范范围	保留区域范围	双重检验范围
0x80 - 0xffff	0xfd00 - 0xfeff	0xf000 - 0xfeff

- 如果指定的地址规范在地址规范范围之外，则输出一个错误。
- 如果复制由 __directmap 声明的变量的分配地址重复，且在双重检验范围内，则输出 W762 警告消息并且显示有重复的变量名称。
- 如果地址规范范围在 saddr 区域之内，则自动作出 __sreg 声明，并生成 saddr 指令。
- 如果按二进制引用由 __directmap 声明的 char/unsigned char/short/unsigned short/int/unsigned int/long/ unsigned long 型变量，则 sreg/__sreg 与 __directmap 必须同时指定。如果不指定，则引用时会发生错误。

效果

- 一个或一个以上的变量可以分配到相同属性的地址。

用法

- 在其中将定义要分配在绝对地址中的变量的模块中声明 __directmap。

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

- 如果为结构体 / 共用体 / 数组声明 __directmap，则将地址放入用括号 {} 中。
- 在 __directmap 外部变量被引用的模块中， __directmap 不需要声明，仅声明 extern 即可。

```
extern                  类型名称      变量名称 ;
extern      __sreg      类型名称      变量名称 ;
```

- 如果模块中有某个位于要 saddr 区域内的 __directmap 外部变量被引用，为了模块中生成 saddr 指令，则必须一起使用 __sreg 来进行以 “extern __sreg ” 类型名称型命名变量名称。

示例

< C 源代码 >

```
__directmap     char            c = 0xfe00 ;
__directmap     __sreg char     d = 0xfe20 ;
__directmap     __sreg char     e = 0xfe21 ;
__directmap     struct  x {
        char    a ;
        char    b ;
} xx = { 0xfe30 } ;
void    main ( )
{
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

< 输出对象 >

```
        PUBLIC  _c
        PUBLIC  _d
        PUBLIC  _e
        PUBLIC  _xx
        PUBLIC  _main
_c      EQU     0FE00H      ; __directmap 声明的变量的地址
_d      EQU     0FE20H      ; 通过 EQU 定义
_e      EQU     0FE21H      ;
_xx     EQU     0FE30H      ;
        EXTRN   __mmfe00    ; 链接保留区域模块的 EXTRN 输出
        EXTRN   __mmfe20    ;
        EXTRN   __mmfe21    ;
        EXTRN   __mmfe30    ;
        EXTRN   __mmfe31    ;
@@CODE CSEG
_main :
; line  10 :    c = 1 ;
        mov     a , #01H        ;1
        mov     !_c , a
; line  11 :    d = 0x12 ;
        mov     _d , #012H      ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line  12 :     e.5 = 1 ;
        set1     _e.5           ; 因为使用了 __sreg，可以进行位处理
; line  13 :     xx.a = 5 ;
        mov     _xx , #05H      ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line  14 :    xx.b = 10 ;
        mov     _xx + 1 , #0AH  ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line  15 :    }
        ret
```

限制

- 不能将函数参数、返回值或自动变量指定为 __directmap。如果有类型指定，则出现错误。
- 如果 short/unsigned short/int/unsigned int/long/unsigned long 型变量分配到在奇数地址，则将在由 __directmap 声明的文件中会生成校正代码，而如果由 extern 声明从外部文件引用这些变量，则将生成非法代码。
- 如果指定的地址位于在保护区留区域范围之外，则变量区域不作保留，有必要指定指令文件或为保留该区域创建单独的模块。

兼容性

< 从其他 C 编译器到 e CC78K0S >

- 如果不使用关键字 __directmap，则无需修改。
- 要改为 __directmap 变量，根据以上方法进行修改。

< 从 CC78K0S 到其他 C 编译器 >

- 使用 #define 可以实现兼容性 （请参阅 “11.6 C 源代码的修改”）。
- 当 __directmap 用作绝对地址分配规范，请根据每个编译器的具体规范来修改程序。

(32) 静态模式展开规范 (-zm)

功能

- 编译器保留 _@NRAT00 至 _@NRAT07 的 8 个字节 saddr 区域，为参数和工作使用。
- 通过声明将 参数和自动变量声明的 __temp 就可以使用临时变量 （如需详细信息，请参阅 “(33) 临时变量 (__temp)”）。
- 可以声明的参数数量范围为 3 到 6 个整型变量，对于字符型变量为 3 至 9 个。第四个及以后的参数被发起调用方设置在 _@NRAT00 至 _@NRAT05 的区域，并由被调用方拷贝到单独区域。但是，如果已为 leaf 函数或参数声明了 __temp，被调用方将不会进行参数拷贝，且其中存放参数的 _@NRATxx 区将按其正常用途使用。
- 参数可以使用 2 个字节以内的结构体 / 共用体。
- 函数返回值可以使用结构体 / 共用体。如果结构体 / 共用体为 2 个字节或更少字节，则返回该值。如果结构体 / 共用体为 3 个字节或更多字节，则将返回值存储在专门保存返回值的保留静态区，并返回该区的最高地址。
- _@NRAT00 至 _@NRAT07 的 8 个字节区还用作 leaf 函数共享区。在共享区的分配过程中，首先分配到 _@NRAT00 至 _@NRAT07 的 8 个字节区，然后再使用指定 -SM 选项指定的 _@KREGxx 区。
- 数组、结构体 / 共用体也可以分配到 _@NRATxx 和 _@KREGxx，只要其大小适合可以放入 _@KREGxx 区， _@KREGxx 区通过 _@NRATxx 和 -SM 选项来指定 。
- 中断函数需要保存的目标在下表表 11-14 中给出。

表 11-14 保存的中断函数目标

恢复 / 保存区	无 BANK	调用函数		不调用函数	
		-zm1	-zm2	-zm1	-zm2
使用的寄存器	NG	NG	NG	OK	OK
所有寄存器	NG	OK	OK	NG	NG
整个 _@NRATxx 区	NG	OK	OK	NG	NG
整个 _@KREGxx 区	NG	OK	NG	NG	NG
使用的 @KREGxx 区	NG	NG	OK	NG	OK

OK : 已保存

NG : 没有被保存

注 然而，当指定 #pragma interrupt 时，中断函数的存储目标可以通过如下指定来限制。

SAVE_R （保存 / 恢复目标限制于寄存器中）

SAVE_RN （保存 / 恢复目标限制于寄存器和 _@NRATxx）。

- -ZM1 与 -ZM2 选项的差别，只是在处理 -SM 选项保留的 _@KREGxx 区有所不同。

  当指定 -ZM1 选项时， _@KREGxx 区仅用于 leaf 函数共享区。

  当指定 -ZM2 选项时，对 _@KREGxx 区进行保存 / 恢复，且参数和自动变量分配也在此 （正常模式下与 -QR 选项兼容）。

- 如果指定 -ZM 选项的同时未指定 -SM 选项，则输出 W055 警告消息且忽略 -ZM 选项规范。

效果

- 可以减少对现有静态模式的限制，提高描述的方便性。

用法

- 编译期间指定 -SM 选项及 -ZM 选项。

示例 1

< C 源代码 >

```
char    funcl ( char a , char b , char c , char d , char e ) ;
char    func2 ( char a , char b , char c , char d ) ;
void    main ( )
{
        char    a = 1 , b = 2 , c = 3 , d = 4 , e = 5 , r ;
        r = func1 ( a , b , c , d , e ) ;
}
char    func1 ( char a , char b , char c , char d , char e )
{
        char    r ;

        r = func2 ( a , b , c , d ) ;
        return  e + r ;
}
char    func2 ( char a , char b , char c , char d )
{
        return  a + b + c + d ;
}
```

- 当指定 -SM， -ZM 和 -QC 时

< 输出对象 >

```
_main :
; line  5 :     char    a = 1 , b = 2 , c = 3 , d = 4 , e = 5 , r ;
        mov     a , #01H        ; 1
        mov     !L0003 , a      ; a
        inc     a
        mov     !L0004 , a      ; b
        inc     a
        mov     !L0005 , a      ; c
        inc     a
        mov     !L0006 , a      ; d
        inc     a
        mov     !L0007 , a      ; e
; line  6 :
; line  7 :     r = func1 ( a , b , c , d , e ) ;
        mov     _@NRAT01 , a    ; 将第五个参数设定到 saddr 区域以便接收和传输参数
        mov     a , !L0006      ; d
        mov     _@NRAT00 , a    ; 将第四个参数设定到 saddr 区域以便接收和传输参数
        mov     a , !L0005      ; c
        movw    hl, ax
        mov     a , !L0004      ; b
        movw    bc , ax
        mov     a , !L0003      ; a
        call    !_func1
        mov     !L0008 , a      ; r
; line  8 :     }
        ret
; line  9 :    char    func1 ( char a , char b , char c , char d , char e )
; line  10 :     {
_func1 :
        mov     !L0011, a
        movw    ax , bc
        mov     !L0012 , a
        movw    ax , hl
        mov     !L0013 , a
        mov     a , _@NRAT00    ; 复制到静态区
        mov     !L0014 , a      ;
        mov     a , _@NRAT01    ; 复制到静态区
        mov     !L0015 , a
; line  11 :    char    r ;
; line  12 :
; line  13 :    r = func2 ( a , b , c , d )
        mov     a , !L0014      ; d
        mov     _@NRAT00,a      ; 将第四个参数设定到 saddr 区域以便接收和传输参数
        mov     a , !L0013      ; c
        movw    hl, ax
        mov     a , !L0012      ; b
        movw    bc , ax
        mov     a , !L0011      ; a
        call    !_func2
        mov     !L0016 , a      ; r
; line  14 :    return  e + r ;
        add     a , !L0015      ; e
L0010 :
; line  15 :     }
        ret
```

```
; line  16 :    char    func2 ( char a , char b , char c , char d )
; line  17 :     {
_func2 :
        mov     _@NRAT01 , a
        movw    ax , bc
        mov     _@NRAT02 , a
        movw    ax , hl
        mov     _@NRAT03 , a
; line  18 :    return  a + b + c + d ;
        mov     a , _@NRAT01    ; a
        add     a , _@NRAT02    ; b
        add     a , _@NRAT03    ; c
        add     a , _@NRAT00    ; d 使用 leaf 函数的 _@NRAT00
L0018 :
; line  19 :     }
        ret
```

- 当指定 -SM， -ZM 和 -QC 时

< 输出对象 >

```
@@CODE  CSEG
_main :
        movw    ax , _@KREG10   ;
        push    ax              ; 保存 _@KREG10 至 _@KREG15 区
        movw    ax , _@KREG12   ;
        push    ax              ;
        movw    ax , _@KREG14   ;
        push    ax              ;
; line  5 :     char    a = 1 , b = 2 , c = 3 , d = 4 , e = 5 , r ;
        mov     _@KREG15 , #01H ; a,1 将变量分配到 _@KREG11 至 _@KREG15
        mov     _@KREG14 , #02H ; b , 2
        mov     _@KREG13 , #03H ; c , 3
        mov     _@KREG12 , #04H ; d , 4
        mov     _@KREG11 , #05H ; e , 5
; line  6 :
; line  7 :     r = func1 ( a , b , c,  d , e ) ;
        mov     a , _@KREG11    ; e
        mov     _@NRAT01 , a    ; 将第五个参数设定到 saddr 区域以便接收和传输参数
        mov     a , _@KREG12    ; d
        mov     _@NRAT00 , a    ; 将第四个参数设定到 saddr 区域以便接收和传输参数
        mov     a , _@KREG13    ; c
        movw    hl, ax
        mov     a , _@KREG14    ; b
        movw    bc , ax
        mov     a , _@KREG15    ; a
        call    !_func1
        mov     _@KREG10 , a    ; r
; line  8 :     }
        pop     ax              ;
        movw    _@KREG14,ax     ; 恢复 _@KREG10 到 _@KREG15 的区域
        pop     ax              ;
        movw    _@KREG12,ax     ;
        pop     ax              ;
        mov     w_@KREG10 , ax  ;
        ret
```

```
; line  9 :     char func1 ( char a , char b , char c , char d , char e )
; line  10:     {
_func1 :
        mov     _@NRAT06 , a    ; 保存 a 寄存器
        movw    ax , _@KREG10   ;
        push    ax              ; 保存 _@KREG10 至 _@KREG15 区
        movw    ax , _@KREG12   ;
        push    ax              ;
        movw    ax , _@KREG14   ;
        push    ax              ;
        mov     a , _@NART06    ; 恢复寄存器 a
        mov     _@KREG15 , a
        movw    ax , bc
        mov     _@KREG14 , a
        movw    ax , hl
        mov     _@KREG13 , a
        mov     a , _@NART00    ; 复制到 _@KREG12
        mov     _@KREG12 , a
        mov     a , _@NART01    ; 复制到 _@KREG11
        mov     _@KREG11 , a
; line  11 :    char r ;
; line  12 :
; line  13 :    r = func2 ( a , b , c , d )
        mov     a , _@KREG12    ; d
        mov     _@NRAT00 , a    ; 将第四参数设定到 saddr 区域以便传输参数
        mov     a , _@KREG13    ; c
        movw    hl, ax
        mov     a , _@KREG14    ; b
        movw    bc , ax
        mov     a , _@KREG15    ; a
        call    !_func2
        mov     _@KREG10 , a    ; r
; line  14 :    return e + r ;
        add     a , _@KREG11    ; e
L0004 :
; line  15 :     }
        movw    hl , ax         ; 保存寄存器 a
        pop     ax              ;
        movw    _@KREG14 , ax   ; 恢复 _@KREG10 到 _@KREG15 区域
        pop     ax              ;
        movw    _@KREG12 , ax   ;
        pop     ax              ;
        movw    _@KREG10 , ax   ;
        movw    ax , hl         ; 寄存器 a 被保存
        ret
; line  16 :    char func2 ( char a , char b , char c , char d )
; line  17 :     {
_func2 :
        mov     _@NRAT01 , a
        movw    ax , bc
        mov     _@NRAT02 , a
        movw    ax , hl
        mov     _@NRAT03 , a
; line  18 :    return a + b + c + d ;
        mov     a , _@NRAT01    ; a
        add     a , _@NRAT02    ; b
        add     a , _@NRAT03    ; c
        add     a , _@NRAT00    ; d 使用 leaf 函数的 _@NRAT00
L0006 :
; line  19 :     }
        ret
```

例 2

< C 源代码 >

```
__sreg  struct  x {
        unsigned char   a ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
} xx , yy ;
__sreg  struct  y {
        int     a ;
        int     b ;
} ss , tt ;
struct  x       func1 ( struct x ) ;
struct  y       func2 ( ) ;
void main ( )
{
        yy = func1 ( xx ) ;
        tt = func2 ( ) ;
}
struct  x       func1 ( struct x aa )
{
        aa.a = 0x12 ;
        aa.b = 0 ;
        aa.c = 1 ;
        return aa ;
}
struct  y       func2 ( )
{
        return  tt ;
}
```

- 当指定 -SM 和 -ZM 时

< 输出对象 >

```
@@CODE CSEG
_main :
; line  14 :    yy = func1 ( xx ) ;
        movw    ax , _xx
        call    !_func1
        movw    _yy , ax
; line  15 :    tt = func2 ( ) ;
        call    !_func2
        movw    hl, ax
        push    de
        movw    de , #_tt
        mov     a , #041H       ; 4
        mov     a , [ hl ]
        mov     [ de ] , a
        incw    hl
        incw    de
        dbnz    c , $$-4
        pop     de
; line  16 :    }
        ret
; line  17 :    struct   x       func1 ( struct x aa )
; line  18 :    {
_func1 :
        movw    _@NRAT00 , ax
; line  19 :    aa.a = 0x12 ;
        mov     _@NRAT00 , #012H        ; aa ,18
; line  20 :    aa.b = 0 ;
        clr1    _@NRAT01.0
; line  21 :    aa.c = 1 ;
        set1    _@NRAT01.1
; line  22 :    return  aa ;
        movw    ax , _@NRAT00           ; aa 因为 2 个字节或更少的字节返回的值
; line  23 :    }
        ret
; line  24 :    struct  y       func2 ( )
; line  25 :    {
; line  26 :    return  tt ;
        movw    hl , #_tt               ; 因为 3 个字节或更多字节
        push    de                      ; 复制到保护静态区的返回值
        movw    de , #L0007
        mov     a , #041H       ; 4
        mov     a , [ hl ]
        mov     [ de ] , a
        incw    hl
        incw    de
        dbnz    c , $$-4
        pop     de
        movw    ax , #L0007         ; 返回静态区域的最高地址
; line  27 :    }
        ret
```

兼容性

< 从其他 C 编译器导入到 CC78K0S >

- 无需修改源程序。

< 从 CC78K0S 到其他 C 编译器 >

- 无需修改源程序。

(33) 临时变量 (__temp)

功能

- 参数和自动变量分配到 _@NRAT00 至 _@NRAT07 区域，而不管其是否需要 对应于 leaf 函数。如果参数和自动变量未分配到 _@NRAT00 至 _@NRAT07 区，则处理方法将与未声明 __temp 时的处理方式相同。
- 函数调用时舍弃由 __temp 声明的参数和自动变量的值。
- 不能将外部和静态变量声明为的 __temp。
- 如果同时还声明了 __sreg，则可以按位运算 char/unsigned char/short/unsigned short/int/unsigned int 变量。
- 如果当未指定 -SM 和 -ZM 选项时声明 _ _temp，则输出 W0339 警告消息，且忽视文件中的 __temp 声明。

效果

- 因为由 __temp 声明的参数和自变量自动变量共享 _@NRAT00 至 _@NRAT07 区域，所以可以保留参数和自动变量区域。
- 如果可以清晰标识包含参数的区段和包含自动变量的区段，并且 __temp 声明应用于函数调用前后不需要保证值匹配的变量，则可以保留内存。

用法

- 编译期间指定 -SM 和 -ZM 选项，并将声明参数和自变量自动变量声明为的 __temp。

示例

< C 源代码 >

```
void    func1 ( __temp char a , char b , char c , __sreg __temp char d ) ;
void    func2 ( char a ) ;
void    main ( )
{
        func1 ( 1 , 2 , 3 , 4 ) ;
}
void    func1 ( __temp char a , char b , char c , __sreg __temp char d )
{
        __temp  char    r ;

        d.1 = 0 ;
        r = a + b + c + d ;
        func2 ( r ) ;
}
void    func2 ( char r )
{
        int     a = 1 , b = 2 ;
        r++ ;
}
```

- 当指定 -sm， -zm 和 -qc 时

< 输出对象 >

```
@@CODE CSEG
_main :
; line  5 :     func1 ( 1 , 2 , 3 , 4 ) ;
        mov     a , #04H                ; 4
        mov     _@NRAT01 , a
        mov     h , #03H                ; 3
        mov     b , #02H                ; 2
        mov   a , #01H ;1
        call    !_func1
; line  6 :    }
        ret
; line  7 :     void func1 ( __temp char a , char b , char c , __sreg __temp
char d )
; line  8 :    {
_func1 :
        mov     _@NRAT01 , a            ; 分配到 _@NRAT01
        mov     a , b
        mov     !L0005 , a
        mov     a , h
        mov     !L0006 , a
                                        ; 分配到 _@NRAT00 的参数
                                        ; 没有改变
; line  9 :     __temp  char    r ;
; line  10 :
; line  11 :    d.1 = 0 ;
        clr1    _@NRAT00.1              ; 可能会进行位操作
; line  12 : r = a + b + c + d ;
        mov     a , _@NRAT01            ; a
        add     a , !L0005              ; b
        add     a , !L0006              ; c
        add     a , _@NRAT00            ; d
        mov     _@NRAT02 , a            ; r 使用 _@NRAT02
; line  13 :    func2 ( r ) ;
        call    !_func2
; line  14 :    }
                                        ;  _@NRAT00 到 _@NRAT02 的值
                                        ; 返回后被改变
        ret
; line  15 :    void     func2 ( char r )
; line  16 :    {
_func2 :
        mov     _@NRAT01 , a
; line  17 :    int     a = 1 , b = 2 ;
        movw    _@NRAT02 , #01H         ; a , 1
        movw    _@NRAT04 , #02H         ; b , 2
; line  18 :    r++ ;
        inc     _@NRAT00
; line  19 :    }
        ret
```

< 输出对象 >

```
@@CODE  CSEG
_main :
; line  5 :     func1 ( 1 , 2 , 3 , 4 ) ;
        mov     a , #04H                ; 4
        mov     _@NRAT00 , a
        mov     h , #03H                ; 3
        mov     b , #02H                ; 2
        mov     a , #01H        ; 1
        call    !_func1
; line  6 :     }
        ret
; line  7 :     void    func1 ( __temp char a , char b , char c , __sreg
__temp char d )
; line  8 :     {
_func1 :
        mov     _@NRAT01 , a            ; 分配到 _@NRAT01
        movw    ax , bc
        mov     !L0005 , a
        movw    ax , hl
        mov     !L0006 , a
                                        ; 分配到 _@NRAT00 的变量
                                        ; 没有改变
; line  9 :     __temp  char    r ;
; line  10 :
; line  11 :    d.1 = 0 ;
        clr1    _@NRAT00.1              ; 位操作可能
; line  12 :    r = a + b + c + d ;
        mov     a , _@NRAT01    ; a
        add     a , !L0005              ; b
        add     a , !L0006              ; c
        add     a , _@NRAT00            ; d
        mov     _@NRAT02 , a            ; r _@NRAT02 used
; line  13 :    func2 ( r ) ;
        call    !_func2
; line  14 :     }
                                        ; _@NRAT00 到 _@NRAT02 的值
                                        ; 返回后被改变
        ret
; line  15 :    void    func2 ( char r )
; line  16 :     {
_func2 :
        mov     _@NRAT00 , a
; line  17 :    int     a = 1 , b = 2 ;
        movw    ax , #01H               ; 1
        movw    _@NRAT02 , ax           ; a
        incw    ax
        movw    _@NRAT04 , ax           ; b
; line  18 :    r++ ;
        inc     _@NRAT00
; line  19 :     }
        ret
```

限制

- 如果调用的函数参数小于等于 3 个，则针对函数调用的自动变量可以将参数和自变量声明为 __temp。如果存在 4 个或更多的参数，则在参数估计期间可能会舍弃参数的值，所以不能保证描述的值。

兼容性

< 从另一个 C 编译器到 CC78K0S >

- 如果不使用保留字 __temp，则无需修改。
- 要为一个临时变量，根据以上方法进行修改。

< 从 CC78K0S 到另一个 C 编译器 >

- 使用 #define 可以实现兼容性 （请参阅 “11.6 C 源代码的修改”）。
  这一个修改意味着 __temp 变量按照常规变量进行处理。

(34) 支持序言 / 尾声的库 (-zd)

功能

- 指定模式的序言 / 结尾代码可以通过库调用来替换。
- 用户可以使用的 callt 的入口数量在正常模式下减少两个，且在静态模式下最多可能占用十个。
- 在正常模式下库替换模式如下所示。

  HL,_@KREGxx 保存 / 复制，堆栈帧保护 -> callt [@@cprep2]

  HL,_@KREGxx 恢复，堆栈帧释放 -> callt [@@cdisp2]
- 在静态模式下，参数分配到 _@NRATxx 和 _@KREGxx，使得前 3 个参数符合以下说明的模式。当 char 和 int 混合使用时，调整分配间隔按照多个 int 型参数模式处理。
- 在静态模式下库替换模式如下所示。

< 对于 char 2 参数 >

```
mov     _@NRAT00 , a            ->  callt [ @@nrcp2 ]
movw    ax , bc
mov     _@NRAT01 , a

mov     _@KREG15 , a            ->  callt [ @@krcp2 ]
movw    ax , bc
mov     _@KREG14 , a
```

< 对于 char 3 参数 >

```
mov     _@NRAT05 , a            ->  callt [ @@nrcp3 ]
movw    ax , bc
mov     _@NRAT06 , a
movw    ax , hl
mov     _@NRAT07 , a

mov     _@KREG15 , a            ->  callt [ @@krcp3 ]
movw    ax , bc
mov     _@KREG14 , a
movw    ax , hl
mov     _@KREG13 , a

mov     _@NRAT06 , a            ->  call !@@nkrc3
movw    ax , bc
mov     _@NRAT07 , a
movw    ax , hl
mov     _@KREG15 , a
```

< 对于 int 2 参数 >

```
movw    _@NRAT00 , ax           ->  callt [ @@nrip2 ]
movw    ax , bc
movw    _@NRAT02 , ax

movw    _@KREG14 , ax           ->  callt [ @@krip2 ]
movw    ax , bc
movw    _@KREG12 , ax
```

< 对于 int 3 参数 >

```
movw    _@NRAT02 , ax           ->  callt [ @@nrip3 ]
movw    ax , bc
movw    _@NRAT04 , ax
movw    ax , hl
movw    _@NRAT06 , ax

movw    _@KREG14 , ax           ->  callt [ @@krip3 ]
movw    ax , bc
movw    _@KREG12 , ax
movw    ax , hl
movw    _@KREG10 , ax

movw    _@NRAT04 , ax           ->  call !@@nkri31
movw    ax , bc
movw    _@NRAT06 , ax
movw    ax , hl
movw    _@KREG14 , ax

movw    _@NRAT06 , ax           ->  call !@@nkri32
movw    ax , bc
movw    _@KREG14 , ax
movw    ax , hl
movw    _@KREG12 , ax
```

< 对于保存 / 恢复 >

```
_@NRAT00 to _@NRAT07 save       ->  callt [ @@nrsave ]

_@NRAT00 to _@NRAT07 restore    ->  callt [ @@nrload ]

_@KREG14 to 15 save             ->  call !@@krs02

_@KREG12 to 15 save             ->  call !@@krs04
                                ->  call !@@krs04i
_@KREG10 to 15 save             ->  call !@@krs06
                                ->  call !@@krs06i

_@KREG08 to 15 save             ->  call !@@krs08
                                ->  call !@@krs08i
_@KREG06 to 15 save             ->  call !@@krs10
                                ->  call !@@krs10i

_@KREG04 to 15 save             ->  call !@@krs12
                                ->  call !@@krs12i

_@KREG02 to 15 save             ->  call !@@krs14
                                ->  call !@@krs14i

_@KREG00 to 15 save             ->  call !@@krs16
                                ->  call !@@krs16i

_@KREG14 to 15 restore          ->  call !@@krl02

_@KREG12 to 15 restore          ->  call !@@krl04
                                ->  call !@@krl04i

_@KREG10 to 15 restore          ->  call !@@krl06
                                ->  call !@@krl06i

_@KREG08 to 15 restore          ->  call !@@krl08
                                ->  call !@@krl08i

_@KREG06 to 15 restore          ->  call !@@krl10
                                ->  call !@@krl10i

_@KREG04 to 15 restore          ->  call !@@krl12
                                ->  call !@@krl12i

_@KREG02 to 15 restore          ->  call !@@krl14
                                ->  call !@@krl14i

_@KREG02 to 15 restore          ->  call !@@krl14
                                ->  call !@@krl16i
```

效果

- 通过以库替换开端和结尾代码，可以缩短目标代码。

用法

- 编译期间指定 -zd 选项。

示例 1

< C 源代码 >

```
int     func1 ( int a , int b , int c ) ;
int     func2 ( int a , int b , int c ) ;
void main ( )
{
        int     r;

        r = func1(1，2，3）;
}
int     func1 ( int a , int b , int c )
{
        return  func2 ( a + 1 , b + 1 , c + 1 ) ;
}
int     func2 ( int a , int b , int c )
{
        return  a + b + c ;
}
```

- 当指定 -sm8, -zm2d, 和 -qc 时

```
@@CODE  CSEG
_main :
        movw    ax , _@KREG14
        push    ax
; line  5 :     int     r ;
; line  6 :
; line  7 :     r = func1 ( 1 , 2 , 3 ) ;
        movw    hl , #03H               ; 3
        movw    bc , #02H               ; 2
        movw    ax , #01H               ; 1
        call    !_func1
        movw    _@KREG14 , ax           ; r
; line  8 :    }
        pop     ax
        movw    _@KREG14 , ax
        ret
; line  9 :     int     func1 ( int a , int b , int c )
; line  10 :    {
_func1 :
        call    !@@krs06
        callt   [ @@krip3 ]
; line  11 :    return  func2 ( a + 1 , b + 1 , c + 1 ) ;
        movw    ax , _@KREG10           ; c
        incw    ax
        movw    hl, ax
        movw    ax , _@KREG12           ; b
        incw    ax
        movw    bc , ax
        movw    ax , _@KREG14           ; a
        incw    ax
        call    !_func2
L0004 :
; line  12 :    }
        call    !@@krl06
        ret
; line  13 :    int     func2 ( int a , int b , int c )
; line  14 :    {
_func2 :
        callt   [ @@nrip3 ]
; line  15 :    return  a + b + c ;
        movw    ax , _@NRAT02           ; a
        xch     a , x
        add     a , _@NRAT04            ; b
        xch     a , x
        addc    a , _@NRAT05            ; b
        xch     a , x
        add     a , _@NRAT06            ; c
        xch     a , x
        addc    a , _@NRAT07            ; c
L0006 :
; line  16 :    }
        ret
```

示例 2

< C 源代码 >

```
int     func ( register int a , register int b ) ;
void main ( )
{
        register int    a = 1 , b = 2 , c = 3 , r ;
        r = func ( a , b ) ;
}
int     func ( register int a , register int b )
{
        register int    r ;

        r = a + b ;
        return  r ;
}
```

- 当指定 -qr 和 -zd 时

< 输出对象 >

```
@@CODE  CSEG
_main :
        movw    de , #03100H
        callt   [ @@cprep2 ]
; line  4 :     register int    a = 1 , b = 2 , c = 3 , r ;
        movw    hl , #01H               ; 1
        movw    ax , hl
        incw    ax
        movw    _@KREG14 , ax           ; b
        incw    ax
        movw    _@KREG12 , ax           ; c
; line  5 :
; line  6 :     r = func ( a , b ) ;
        movw    ax , _@KREG14           ; b
        push    ax
        movw    ax , hl
        call    !_func
        pop     ax
        movw    ax , bc
        movw    _@KREG10 , ax           ; r
; line  7 :     }
        movw    ax , #03100H
        callt   [ @@cdisp2 ]
        ret
; line  8 :     int     func ( register int a , register int b )
; line  9 :     {
_func :
        movw    de , #0E840H
        callt   [ @@cprep2 ]
; line  10 :    register int    r ;
; line  11 :
; line  12 :    r = a + b ;
        movw    ax , hl
        xch     a , x
        add     a , _@KREG12            ; a
        xch     a , x
        addc    a , _@KREG13            ; a
        movw    _@KREG14 , ax           ; r
L0004 :
; line  14 :     }
        movw    ax , #0E840H
        callt   [ @@cdisp2 ]
        ret
```

限制

- 不能同时指定优化规范选项 -ql4 和 -zd 选项。如果同时指定，则输出 w0052 警告消息并用以 -ql3 选项替换 -ql4 选项进行处理。

注意事项

- 仅当前 3 个参数没有任何一个被指定 register，或将前 3 个参数都指定为 __temp 时，在静态模式下的参数拷贝模式是模式 - 匹配的。因此，如果指定 -QV 选项，或为前 3 个参数有某个被指定为 register/__temp，则不进行模式 - 匹配，所以不可能替换 -ZD 选项。

兼容性

< 从另一个 C 编译器到 CC78K0S >

- 无需修改源程序。
- 要以库替换序言 / 结尾代码，请根据以上使用方法描述的过程对源程序进行修改。

< 从 CC78K0S 到另一个 C 编译器 >

- 无需修改源程序。

11.6 C 源代码的修改

通过使用本 C 编译器的扩展函数，可以产生效率更高的目标。但是，希望这些扩展函数仅适用于 CC78K0S 系列。因此，要将其用于其他设备，可能需要对 C 源代码进行修改。此处，介绍了如何使 C 源代码可以从另一 C 编译器移植到本 C 编译器以及相反操作。

< 从其他 C 编译器导入到 CC78K0S >

- #pragma 注

  如果另一 C 编译器支持 #pragma 预处理器指令，则必须修改 C 源代码。修改 C 源代码的方法和修改的程度取决于另一 C 编译器的规范。

- 扩展的规范

  如果另一 C 编译器已扩展了规范，诸如添加了新的关键字，则必须修改 C 源代码。修改 C 源代码的方法和修改的程度取决于另一 C 编译器的规范。

  注 #pragma 为 ANSI 支持的预处理指令之一。编译器会把 #pragma 之后的字符串识别为命令。如果编译程序编译器不支持此指令，则忽略 #pragma 指令，且将继续进行编译直到其完全结束。

< 从 CC78K0S 到其他 C 编译器 >

- 由于 CC78K0S 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。

示例

(1) 要使关键字无效 （同样应用于 callf， sreg， noauto 和 norec 等）

```
#ifndef __K0S__
#define callt           /*  将 callt 作为普通函数 */
#endif
```

(2) 从一种类型转换为另一种类型

```
#ifndef __K0S__
#define bit     char    /* 将 bit 型变量改为 char 型变量 */
#endif
```

11.7 函数调用接口

以下将说明有关函数调用的接口。

(1) 返回值 (所有函数通用)

(2) 普通函数调用接口

  (a) 传递参数

  (b) 存储参数的位置和顺序

  (c) 自动变量存储的位置和顺序

(3) noauto 函数调用接口 （仅普通模式）

  (a) 参数传递

  (b) 存储参数的位置和顺序

  (c) 存储自动变量的位置和顺序

(4) norec 函数调用接口 （普通模式）

  (a) 参数传递

  (b) 存储参数的位置和顺序

  (c) 存储自动变量的位置和顺序

(5) 静态模式函数调用接口

  (a) 参数传递

  (b) 存储参数的位置和顺序

  (c) 存储自动变量的位置和顺序

(6) Pascal 函数调用接口

11.7.1 返回值

调用的函数将返回值存储在寄存器和进位标志中，如 表 11-15 所示。

表 11-15 存储返回值的位置

类型	存储方法	
	普通模式	静态模式
1 个字节的整数	BC	A
2 个字节的整数		AX
4 个字节的整数	BC （低）， DE （高）	不支持
指针	BC	AX
结构体 , 共用体	BC （如果复制拷贝到函数专用区，则为结构体 / 共用体结构或集合的起始地址）	不支持
1 位	CY （进位标志)	CY （进位标志)
浮点数 （float 类型）	BC （低）， DE （高）	不支持
浮点数 （double 类型）	BC （低）， DE （高）	不支持

11.7.2 普通函数调用接口

当所有参数分配到寄存器且未使用自动变量时，普通函数调用接口与 noauto 函数调用接口相同。

(1) 传递参数

- 存在两类参数：分配到寄存器的参数，普通参数。
- 只要寄存器和 _@KREGxx 还有剩余空间，分配到寄存器的参数预先有 register 声明并且被成功分配到寄存器或 _@KREGxx。但是，仅当指定 -QR 时，参数才会被分配到 _@KREGxx。下文称分配到寄存器或 _@KREGxx 的参数为寄存器参数。
- 关于 _@KREGxx，请参考 “附录 A saddr 区域标签列表”。
- 剩余参数分配到栈堆栈。
- 在函数发起调用方端，以 register 声明的参数和普通参数以相同方式传输。第二个参数及以后的参数通过堆栈传输且第一个参数通过寄存器或堆栈传输。
- 在函数定义方，通过寄存器或栈堆栈传输的参数被保存在参数分配的位置。
- 寄存器参数拷贝到寄存器或 _@KREGxx。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同。
- 普通参数通过堆栈传递。当参数通过堆栈传输时，其中参数被传入的区域就成为参数分配到的区域。
- 对存放参数的寄存器进行保存和恢复，这个工作在函数定义方进行。

- 其中第一个参数传输入的位置如 表 11-16 所示。

表 11-16 类型修改类型调整的详细信息 （从 int 和 short 型改为 char 型）

类型	存储方法
1 个字节的数据[注] 2 个字节的数据[注]	AX
3 个字节的数据[注]	AX, BC
4 个字节的数据[注]	AX, BC
浮点数 （float 类型）	AX, BC
浮点数 （double 类型）	AX, BC
其它	通过堆栈传输

注 1 至 4 字节的数据可以包括结构体、共用体和指针。

(2) 存储参数的位置和顺序

- 存在两类参数：分配到寄存器的参数和普通参数。当 -qv 时，分配到寄存器的参数为以寄存器和参数声明的参数。
- 未分配到寄存器的参数分配到堆栈。分配到堆栈的参数从最后一个参数开始按顺序放于堆栈上。
- 将寄存器值保存和恢复至参数分配位置的工作是由函数定义方执行所分配的参数。
- 在函数定义端定义方，通过寄存器或堆栈传输的参数存储在将参数所分配到的区域。
- 寄存器参数被复制拷贝到寄存器或 _@KREGxx。仅当在指定了 -qr 选项时才会拷贝执行复制到 _@KREGxx 中的操作。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同即使当参数通过寄存器传输时，因为有关函数调用程序 （传输端）的寄存器不同于函数定义端 （接收端）的寄存器，所以还需要进行寄存器复制。
- 在函数调用方，寄存器参数和常规参数使用相同方法。
  第二个或随后的参数通过堆栈传输。第一个参数通过寄存器和堆栈传输。
  请参阅表 11-16 如需了解其中传输第一个参数的传入位置。

（要使用的寄存器）

HL

当存在堆栈帧时，自动变量不分配到 HL。

（要使用的 saddr 区域）

_@KREG12 至 15

（分配序列分配顺序）

- 寄存器

char 类型 :	顺序为 L-H。
int, short, and enum 类型 :	HL

- saddr 区域

char type :	顺序为 _@KREG12, _@KREG13, _@KREG14, and _@KREG15.
int, short, and enum 类型 :	顺序为 _@KREG12 to 13 and _@KREG14 to 15.
long, float, double 类型 :	顺序为 _@KREG12 to 13 (低位)- _@KREG14 到 15 (高位)。

(3) 自动变量存储的位置和顺序

- 存在两类自动变量：分配到寄存器的自动变量和常用普通自动变量。分配到寄存器的自动变量以用 register 声明的自动变量和指定 -QV 选项时的自动变量。只要存在可分配的寄存器和 _@KREGxx 存在空间，其就尽量分配到寄存器和 _@KREGxx。然而，仅当指定 -qr 选项时，自动变量才会分配到 _@KREGxx。

  下文称分配到寄存器和 _@KREGxx 的自动变量为寄存器变量。

- 关于 _@KREGxx，请参考 “附录 A saddr 区域标签列表”。

- 在寄存器参数分配完成之后，再分配寄存器变量。因此，当在寄存器参数分配完成之后如果存在过剩过多的寄存器时，寄存器变量分配到寄存器。

- 未分配到寄存器的自动变量分配到堆栈。

- 寄存器的保存和恢复，且在函数定义方的 _@KREGxx 进行自动变量的分配。

(a) 自动变量分配序列

将自动变量分配到 _@KREGxx 的顺序如下所示。

（要使用的寄存器）

HL

当存在堆栈帧时，自动变量不分配到 HL。

（要使用的 saddr 区域）

_@KREG00 至 15

（分配顺序）

- 寄存器

  char 类型 : 顺序为 L and H.

  int, short, and enum 类型 :HL

- saddr 区域

  char 类型 : 顺序为 _@KREG00, _@KREG01 ..., 和 _@KREG11.

  int, short, 和 enum 类型 : 顺序为 _@KREG00 至 01, _@KREG02 至 03 ... 以及 _@KREG10 至 15。

  long, float, double 类型 : 顺序为 _@KREG00 至 03, _@KREG04 至 07, 以及 _@KREG12 至 15。

- 分配到堆栈的自动变量按声明的顺序分配到堆栈。

[示例]

< （C 源代码 1）>

```
void    func0 ( register int , int ) ;
void    main ( )
{
func0 ( 0x1234 , 0x5678 ) ;
}
void    func0 ( register int p1 , int p2 )
{
        register int    r ;
        int             a ;
        r = p2 ;
        a = p1 ;
}
```

< 输出代码 >

```
_main :
; line  4 :     func0 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push    ax                      ; 通过堆栈传输的参数
        movw    ax , #01234H    ; 4660  ; 第一个传输的数据
                                        ; 通过寄存器
        call    !_func0                 ; 函数调用
        pop     ax                      ;通过堆栈传输的参数
; line  5 :    }
        ret
; line  6 :     void     func0 ( register int p1 , int p2 )
; line  7 :    {
_func0 :
        push    hl
        xch     a , x
        xch     a , _@KREG12
        xch     a , x
        xch     a , _@KREG13            ; 分配寄存器参数 p1 到
                                        ; _@KREG12
        push    ax                      ; 保存寄存器的 saddr 区域
                                        ; 参数
        movw    ax , _@KREG00
        push    ax                      ; 保存寄存器的 saddr 区域
                                        ; 变量
        push    ax                      ; 为自动变量保留区域
                                        ; 变量 a
        movw    ax, sp
        movw    hl, ax
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line  10 :    r = p2 ;
        mov     a , [ hl + 10 ] ; p2    ; 通过堆栈传递的参数 p2
        xch     a , x
        mov     a , [ hl + 11 ] ; p2
        movw    _@KREG00 , ax   ; r     ; 分配寄存器变量
                                        ; _@KREG00
; line  11 :    a = p1 ;
        movw    ax , _@KREG12   ; p1    ; 寄存器参数 _@KREG12
        mov     [ hl + 1 ] , a  ; a
        xch     a , x
        mov     [ hl ] , a      ; a     ; 分配为自动变量 a
; line  12 :    }
        pop     ax                      ; 释放自动变量的区域
                                        ; 变量 a
        pop     ax
        movw    _@KREG00 , ax           ; 恢复寄存器变量 saddr 区域
        pop     ax
        movew    _@KREG12 , ax          ; 恢复寄存器参数 saddr 区域
        pop     hl
        ret
```

< （C 源代码 2）>

```
void    func1 ( int , register int ) ;
void    main ( )
{
        func1 ( 0x1234 , 0x5678 ) ;
}
void    func1 ( int p1 , register int p2 )
{
        register int    r ;
        int             a ;
        r = p2 ;
        a = p1 ;
}
```

< 输出代码 >

```
_main :
; line  4 :     func1 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push    ax                      ; 通过堆栈传输的参数
        movw    ax , #01234H    ; 4660  ; 第一个传输的数据
                                        ; 通过寄存器
        call    !_func1                 ; 函数调用
        pop     ax                      ; 通过堆栈传输的参数
; line  5 :    }
        ret
; line  6 :     void     func1 ( int p1 , register int p2 )
; line  7 :    {
_func1 :
        push    hl
        push    ax                      ; 载入第 1 个参数 p1 在堆栈
        movw    ax , _@KREG00
        push    ax                      ; 保存寄存器的 saddr 区域变量
        movw    ax , _@KREG12
        push    ax                      ; 保存寄存器的 saddr 区域变量
        push    ax                      ; 为自动变量 a 保留区域
        movw    ax, sp
        movw    hl, ax
        mov     a , [ hl + 12 ]         ; 通过堆栈传递的参数 p2
                                        ; 通过 saddr 区域接收
        xch     a , x
        mov     a , [ hl + 13 ]
        movw    _@KREG12 , ax           ; 将寄存器参数分配到
                                        ; _@KREG12.
; line  8 :     register int    r ;
; line  9 :     int a ;
; line  10 :    r = p2 ;
        movw    ax , _@KREG12   ; p2
        movw    _@KREG00 , ax   ; r     ; 寄存器参数 _@KREG00
; line  11 :    a = p1 ;
        mov     a , [ hl + 6 ]  ; p1    ; 传递数 p1 ( 低位 ) 通过
                                        ; 堆栈且通过寄存器接收
        mov     [ hl ] , a      ; a     ; 自动变量 a ( 低位 )
        xch     a , x
        mov     a , [ hl + 7 ]  ; p1    ; 传递数 p1 ( 高位 ) 通过
                                        ; 堆栈且通过寄存器接收
        mov     [ hl + 1 ] , a  ; a     ; 自动变量 a ( 高位 )
; line  12 :    }
        pop     ax                      ; 释放自动变量的区域
                                        ; 变量 a
        pop     ax
        movw    _@KREG12 , ax           ; 恢复寄存器的 saddr 区域
                                        ; 变量
        pop     ax
        movw    _@KREG00 , ax           ; 恢复寄存器的 saddr 区域
                                        ; 变量
        pop     ax
        pop     hl
        ret
```

11.7.3 noauto 函数调用接口 （仅普通模式）

(1) 参数传递

- 在函数调用程序中，参数传输的方式与普通函数中的方式相同。参考 “11.7.2 普通函数调用接口”。
- 在函数定义方，通过寄存器或堆栈传输的参数被拷贝到寄存器以及 _@KREG12 至 15 中。仅当指定 -QR 选项时，才会有参数拷贝到 _@KREG12 至 15 的操作。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同。
- 将寄存器值保存和恢复至参数分配位置的工作是由函数定义方执行所分配的参数。

(2) 存储参数的位置和顺序

- 在函数定义方，所有参数被分配到寄存器和 _@KREG12 至 15。但是，仅当指定 -QR 选项时，参数才会分配到 _@KREG12 至 15。
- 如果有参数既没有分配到寄存器也没有分配到或 _@KREG12 至 15 的，则将出现错误。
- 在函数调用程序中，参数传输的方式与普通函数中的方式相同 (参考 “11.7.2 普通函数调用接口”)。
- 在函数定义端定义方，通过寄存器或堆栈传递的参数复制到寄存器及 _@KREG12 至 15 中。即使在参数通过寄存器传递时，也必须要进行寄存器复制，因为函数调用方 （发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 将寄存器值保存和恢复至参数分配位置的工作是由函数定义方执行所分配的参数。

（分配顺序）

- 分配队列与普通函数相同 (参阅 "11.7.2 普通函数调用接口 ")。

(3) 存储自动变量的位置和顺序

- 自动变量分配到寄存器和 _@KREG12 至 15 中。然而，仅当指定 -QR 选项时，自动变量才会分配到 _@KREG12 至 15。关于 _@KREG12 至 15, 请参考 “附录 A saddr 区域标签列表”。
- 当参数分配到寄存器完成之后，如果还有空闲的寄存器，则自动变量也分配到寄存器。当指定 -QR 选项时，自动变量也被分配到 _@KREG12 至 15。
- 如果有自动变量既没有分配到寄存器也没有分配 _@KREG12 至 15，则出现错误。
- 对存放自动变量的寄存器 _@KREG12 至 15 进行保存和恢复，这个工作在函数定义方进行。

（分配顺序）

- 将自动变量分配到寄存器的顺序和分配参数分配的顺序相同。
- 分配到 _@KREG12 至 15 的自动变量按声明的顺序分配进行。

[示例]

< C 源代码 >

```
noauto  void    func2 ( int , int ) ;
void    main ( )
{
        func2 ( 0x1234 , 0x5678 ) ;
}
noauto  void    func2 ( int p1 , int p2 )
{
    :
}
```

< 输出代码 >

```
_main :
; line  4 :     func2 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push    ax                      ; 通过堆栈传输的参数
        movw    ax , #01234H    ; 4660  ; 第一个传输的数据
                                        ; 通过寄存器
        call    !_func2                 ; 函数调用
        pop     ax                      ; 通过堆栈传输的参数
; line  5 :    }
        ret
; line  6 :     noauto  void    func2 ( int p1 , int p2 )
; line  7 :     {
_func2 :
        push    hl                      ; 保存参数的寄存器
        xch     a , x
        xch     a , _@KREG12            ; 分配参数 p1 到 _@KREG12
                                        ; ( 低位 )
        xch     a , x
        xch     a , _@KREG13            ; 分配参数 p1 到 _@KREG13
                                        ; ( 高位 )
        push    ax                      ; 保存参数的 saddr 区域
        movw    ax, sp
        movw    hl, ax
        mov     a , [ hl + 6 ]          ; 参数 p2 ( 低位 ) 通过 a 传递
                                        ; 堆栈和通过寄存器接收
        xch     a , x
        mov     a , [ hl + 7 ]          ; 参数 p2 ( 高位 ) 通过 a 传递
                                        ; 堆栈和通过寄存器接收
        movw    hl , ax                 ; 分配参数到 HL
            :
        pop     ax
        movw    _@KREG12 , ax           ; 恢复自动变量的 saddr 区域
        pop     hl                      ; 恢复参数寄存器
        ret
```

11.7.4 norec 函数调用接口 （普通模式）

(1) 参数传递

所有参数分配到 _@NRARGx 和 _@RTARG6 至 7。在函数调用方，参数通过寄存器 _@NRARGx 传输递。在函数定义方，通过寄存器传输的参数拷贝到寄存器，或拷贝到 _@RTARG6 至 7 （请参阅 “附录 A saddr 区域标签列表”）。

(2) 存储参数的位置和顺序

- 在函数定义端定义方，所有参数分配到寄存器， _@NRARGx， _@RTARG6 至 7。仅当指定了 -QR 选项时，参数才会分配到 _@NRARGx。
- 仅当 DE 中存在参数时，自动变量分配到 _@RTARG6 至 7 （请参阅 “附录 A saddr 区域标签列表”）。
- 如果有参数既没有分配到寄存器也没有分配， _@NRARGx， _@RTARG6 至 7，则将发生错误。
- 在函数调用，参数通过寄存器和 _@NRARGx 传递。
- 在函数定义方，通过寄存器传输的参数复制到寄存器或 _@RTARG6 至 7。即使参数通过寄存器传递，也必须要进行寄存器复制，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同。如果参数通过寄存器传输，则参数传入的区域成为为其分配的区域。
- 如果参数可以不再通过寄存器传输，则其可以分配到 _@NRARGx 且通过该处传输。在此情况下，通过寄存器和 _@NRARGx 进行联合传输。

（参数分配顺序）

- 分配到 _@NRARGx 的参数按照声明的顺序分配。
- 分配到寄存器的参数根据以下规则分配到寄存器 _@RTARG6 至 7。

（要使用的寄存器）

- 当一个参数为 char, int, short, enum, 或指针型 : AX 传递 , DE 接收
- 在两个或更多个参数为 char, int, short, enum, 或指针型时 : AX 和 DE 传递 _@RTARG6, 7 和 DE 接收

（分配顺序）

- char, int, short, enum, 和指针型： 顺序为 DE, _@RTARG6 至 7

(3) 存储自动变量的位置和顺序

- 只要存在可分配的寄存器和 _@NRARGx 还有空间，自动变量就尽可能分配到寄存器和 _@NRARGx。
  如果不存在空闲可分配的寄存器，则其分配到 _@NRATxx。
  然而，仅当指定 -QR 选项时，自动变量才会分配到 _@NRARGx 和 _@NRATxx。
  关于 _@KREGxx，请参考 “附录 A saddr 区域标签列表”。
  如果有自动变量既没有分配到寄存器也没有分配如果存在不能分配到寄存器， _@NRARGx 和 _@NRATxx 的自动变量，则出现错误。
- 对存放自动变量的寄存器进行保存和恢复工作在函数定义方进行。

（分配序列分配顺序）

- 将分配自动变量分配到至寄存器，_@RTARG6 至 7 的顺序与分配参数的顺序相同。
- 分配到，_@NRARGx, _@NRATxx 的自动变量按照声明的顺序分配。

[示例]

- 在正常模式下

< C 源代码 >

```
norec   void    func3 ( char , int , char , int ) ;
void    main ( )
{
        func3 ( 0x12 , 0x34 , 0x56 , 0x78 ) ;
}
norec   void    func3 ( char p1 , int p2 , char p3 , int p4 )
{
        int             a ;
        a = p2 ;
}
```

- 当指定 -QR 时

< 输出代码 >

```
_main :
; line  4 :     func3 ( 0x12 , 0x34 , 0x56 , 0x78 ) ;
        movw    _@NRARG1 , #078H ; 120 ; 参数通过 _@NRARG1 传输
        movw    _@NRARG1 , #078H ; 86  ; 参数通过 _@NRARG1 传输
        movw    de , #034H       ; 52  ;  参数通过寄存器 DE 传输
        mov     a , #012H        ; 18  ; 参数通过寄存器 A 传输
        call    !_func3                ; 函数调用
        ret
; line  6 :    norec   void    func3 ( char p1 , int p2 , char p3 , int p4 )
; line  7 :    {
_func3 :
        mov     _@RTARG6 , a     ; 配置参数 p1 到 _@RTARG6
; line  8 :     int a ;
; line  9 :     a = p2 ;
        movw    ax , de          ; 参数 p2
        movw    _@NRARG2 , ax    ; a   ; 自动变量 a
        ret
```

11.7.5 静态模式函数调用接口

(1) 参数传递

- 在函数调用方，寄存器参数的传输方式和普通参数相同。
  所有参数要通过寄存器传输，最多可以存在有最多三个参数共 6 个字节。
- 在函数定义方，通过寄存器传输的参数被存储为其分配的区域。寄存器参数拷贝到寄存器。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同。
- 普通函数分配到函数专用区。

(2) 存储参数的位置和顺序

(a) 参数存储位置

- 存在两类自动变量：分配到寄存器的参数和普通参数。
- 分配到寄存器的参数就是使用 register 声明过的参数。
- 在函数定义方，通过寄存器或堆栈传输的参数存储参数所分配的区域。
  寄存器参数复制拷贝到寄存器。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方 （发送方）的寄存器和函数定义方 （接收方）的寄存器不同。普通参数分配到函数专用区。
- 对存放参数 / 自动变量的寄存器进行保存和恢复，这个工作在函数定义方进行。
- 剩余的自动变量分配到函数专用区。
- 在函数调用方，寄存器参数的传输方式和普通参数相同所有参数要通过寄存器传输，最多可以有三个参数共 6 个字节。表 11-17 展示了参数传入的区位置。

表 11-17 静态模式下传输参数的区

数据大小	第一个参数	第二个参数	第三个参数
1 字节的数据[注]	A	B	H
2 字节的数据[注]	AX	BC	HL
4 字节的数据[注]	分配到 AX 和 BC 和剩余参数分配到 H 或 HL。		

注 在 1 至 4- 字节数据中不能包括有结构体或共用体。

(b) 参数分配顺序

- 分配到函数专用区的参数按照顺序从最后一个参数分配。
- 在符合以下规则时，寄存器参数分配到寄存器 DE。

（要使用的寄存器）

DE

（分配序列分配顺序）

char 型：	序列为 D, E
int, short, enum 型：	DE

(3) 存储自动变量的位置和顺序

(a) 自动变量的存储位置

- 存在两类自动变量：分配到寄存器的自动变量和普通自动变量。
- 分配到寄存器中的自动变量是寄存器声明的自动变量以及由 -qv 选项设定的自动变量。
- 在分配寄存器参数分配完成之后，再分配寄存器变量。出于这种原因，仅当寄存器参数分配完成之后，寄存器还有空闲，才会将寄存器变量分配至寄存器。
- 剩余的自动变量分配到函数专用区。
- 将寄存器值保存和恢复至参数分配位置的工作是由函数定义方执行所分配的参数。

(b) 自动变量分配顺序

- 在符合以下规则时，自动变量分配到寄存器 DE。

（要使用的寄存器）

DE

（分配顺序）

char 型：	序列为 D, E
int, short, enum 型：	DE

- 分配到函数专用区的自动变量按声明的顺序进行分配。

[示例]

< C 源代码 >

```
void    func4 ( register int , char ) ;
void    func ( void ) ;
void    main ( )
{
        func4 ( 0x1234 , 0x56 ) ;
}
void    func4 ( register int p1 , char p2 )
{
        register char   r ;
        int             a ;
        r = p2 ;
        a = p1 ; func ( ) ;
}
```

< 输出代码 >

```
@@DATA          DSEG
L0005   : DS    ( 1 )                   ; 参数 p2
L0006   : DS    ( 1 )                   ; 自动变量 r
L0007   : DS    ( 2 )                   ; 自动变量 a

; line  1 :     void    func4 ( register int , char ) ; void func ( void ) ;
; line  2 :     void main ( )
; line  3 :    {

@@CODE  CSEG
_main :
; line  4 :     func4 ( 0x1234 , 0x56 ) ;
        mov     b , #056H     ; 86    ; 通过寄存器 B 传输第二个参数
        movw    ax , #01234H  ; 4660  ; 通过寄存器 AX 传输第一个参数
        call    !_func4               ; 函数调用
; line  5 :     }
        ret
; line  6 :     void    func4 ( register int p1 , char p2 )
; line  7 :     {
_func4 :
        push    de                    ; 保存寄存参数的寄存器
        movw    de , ax               ; 为 DE 分配寄存器参数 p1
        movw    a , b

        mov     !L0005 , a            ; 复制参数 p2 到 L0005
; line  8 :     register char r ;
; line  9 :     int           a ;
; line  10 :    r = p2 ;
        mov     !L0006 , a    ; r     ; 自动变量 r
; line  11 :    a = p1 ; func ( ) ;
        movw    ax , de               ; 寄存器参数 p1
        movw    !L0007 , ax   ; a     ; 自动变量 a
        callt   [ @@hlist ]
; line  12 :     }
        pop     de                ;返回寄存器参数到寄存器
        ret
```

[示例 2]

< C 源代码 >

```
void    func5 ( int , register char ) ;
void    func ( void ) ;
void    main ( )
{
        func5 (0x1234 , 0x56 ) ;
}
void    func5 ( int p1, register char p2 )
{
        register char   r ;
        int             a ;
        r = p2 ;
        a = p1 ; func ( ) ;
}
```

- 当指定 -QW3 时

< 输出代码 >

```
@@DATA  DSEG
L0005   : DS  ( 2 )
L0006   : DS  ( 2 )

; line  1 :   void  func5 ( int , register char ) ; void func ( void ) ;
; line  2 :     void main ( )
; line  3 :     {

@@CODE  CSEG
_main :
; line  4 :   func5 ( 0x1234 , 0x56 ) ;
        mov   b , #056H     ; 86    ; 通过寄存器 B 传输第二个参数
        movw  ax , #01234H  ; 4660  ; 通过寄存器 AX 传输第一个参数
        call  !_func5               ; 函数调用
; line  5 :     }
        ret
; line  6 :   void    func5 ( int p1 , register char p2 )
; line  7 :     {
_func5 :
        push    de                  ; 保存寄存变量的寄存器
                                    ; 和寄存器参数
        movw  !L0005 , ax           ; 复制参数 p1 到 L0005
        callt   [ @@hlist ]
        movw    ax , bc
        mov   de , ax               ; 将寄存器参数 p2 分配到 d。
; line  8 :   register char r ;
; line  9 :     int           a ;
; line  10 :  r = p2 ;
        movw  ax , de               ; 寄存器参数 p2
        mov   e , a                 ; 自动变量 r
; line  11 :  a = p1 ; func ( ) ;
        movw  hl , #L0005   ; p1    ; 参数 p1
        callt [ @@hlilo ]
        movw  hl , #L0006   ; a     ; 自动变量 a
        callt   [ @@hlist ]
        call  !_func
; line  12 :     }
        pop     de                ; 恢复寄存参数的寄存器
        ret
```

11.7.6 Pascal 函数调用接口

此函数接口与其他函数接口之间的不同在于：当调用函数时，为了加载参数对校正操作在被调用方进行，而不是在发起函数调用方进行。其他所有操作和的函数指定属性相同。

[为参数分配的区域]

[参数分配的顺序]

[为自动变量分配的区域]

[自动变量分配的顺序]

- 如果同时指定有 noauto 属性，则与调用 noauto 函数时的特征相同 （请参阅 “11.7.3 noauto 函数调用接口 （仅普通模式）”）。
- 如果同时指定有 noauto 属性，则与调用普通函数时的特征相同 （请参阅 “11.7.2 普通函数调用接口”）。

[例 1]

< C 源代码 >

```
__pascal        void    func0 ( register int , int ) ;
void    main ( )
{
        func0 ( 0x1234 , 0x5678 ) ;
}
__pascal        void    func0 ( register int p1 , int p2 )
{
        register int    r ;
        int             a ;
        r = p2 ;
        a = p1 ;
}
```

- 当指定 -qr 选项时

< 输出代码 >

```
_main :
; line  4 :   func0 ( 0x1234 , 0x5678 ) ;
        movw  ax , #05678H    ; 22136
        push  ax                        /* 堆栈通过参数传输 */
        movw  ax , #01234H    ; 4660    /* 传输的第一个参数 */
                                        /* 通过寄存器 */
        call  !_func0                   /* 函数调用 */
                                        /* 此处不校正堆栈 */
; line  5 :   }
        ret
; line  6 :   __pascal        void    func0 ( register int p1 , int p2 )
; line  7 :     {
_func0 :
        push    hl
        xch     a , x
        xch   a , _@KREG12
        xch     a , x
        xch   a , _@KREG13    /* 分配寄存器参数 p1 到 _@KREG12 */
        push  ax              /* 保存寄存器参数使用的 saddr 区域 */
        movw  ax, _@KREG14
        push  ax              /* 保存寄存器变量使用的 saddr 区域 */
        push  ax              /* 保留自动变量 a 区域 */
        movw    ax , sp
        movw    hl , ax
; line  8 :    register int    r ;
; line  9 :     int           a ;
; line  10 :  r = p2 ;
        mov   a , [ hl+10 ]   ; p2  /* 堆栈传输参数 p2 */
        xch     a , x
        mov   a , [ hl+11 ]   ; p2
        movw  _@KREG14 , ax   ; r   /* 分配到寄存器参数 _@KREG14 */
; line  11 :  a = p1 ;
        movw  ax , _@KREG12   ; p1  /* 寄存器参数 _@KREG12 */
        mov   [ hl+1 ] , a    ; a
        xch     a , x
        mov   [ hl ] , a      ; a   /* 分配到自动变量 a */
; line  12 :     }
        pop   ax            /* 释放自动变量 a 区域 */
        pop     ax
        movw  _@KREG14 , ax /* 恢复寄存参数的 saddr 区域 */
        pop     ax
        movw  _@KREG12 , ax /* 恢复寄存参数的 saddr 区域 */
        pop     hl
        pop   de            /* 包含返回地址 */
        pop   ax            /* 通过传递参数用完了堆栈 */
                            /* 堆栈被校正 */
        push  de            /* 重新载入返回地址 */
        ret
```

[例 2]

< C 源代码 >

```
__pascal        noauto  void    func2 ( int , int ) ;
void    main ( )
{
        func2 ( 0x1234 , 0x5678 ) ;
}
__pascal        noauto  void    func2 ( int p1 , int p2 )
{
    :
}
```

- 当指定 -qr 选项时

< 输出代码 >

```
_main :
; line  4 :     func2 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H    ; 22136
        push  ax                      /* 通过堆栈传输的参数 */
        movw  ax , #01234H  ; 4660    /* 第一个传输的数据 */
                                      /* 通过寄存器 */
        call  !_func2                 /* 函数调用 */
                                      /* 此处不校正堆栈 */
; line  5 :    }
        ret
; line  6 :   __pascal     noauto  void    func2 ( int p1 , int p2 )
; line  7 :   {
_func2 :
        push  hl              /* 保存参数的寄存器 */
        xch     a , x
        xch   a , _@KREG12    /* 分配参数到 _@KREG12 */
                              /* ( 低位 ) */
        xch     a , x
        xch   a , _@KREG13    /* 分配参数 p1 到 _@KREG13 */
                              /* ( 高位 ) */
        push  ax              /* 保存自变量参数使用的 saddr 区域 */
        movw    ax, sp
        movw    hl, ax
        mov   a , [ hl + 6 ]  /* 参数 p2 ( 低位 ) 通过 a 传输 */
                              /* 堆栈和通过寄存器接收 */
        xch     a , x
        mov   a , [ hl + 7 ]  /* 参数 p2 ( 高位 ) 通过 a 传输 */
                              /* 堆栈和通过寄存器接收 */
        movw  hl , ax         /* 分配参数到 HL */
             :
        pop     ax
        movw  _@KREG12 , ax   /* 恢复自动变量使用的 saddr 区域 */
        pop   hl              /* 恢复参数寄存器 */
        pop   de              /* 包含返回地址 */
        pop   ax              /* 通过传递参数用完了堆栈 */
                              /* 通过校正的堆栈 */
        push  de              /* 重新载入返回地址 */
        ret
```

第 12 章　引用汇编程序

本章描述如何链接用汇编语言编写的程序。

如果在 C 源程序中调用由其它编程语言编写的函数，那么这两种目标模块要通过链接器进行链接。本章就是描述在 C 源程序中调用其他编程语言的程序的过程步骤，以及在其他语言程序中调用 C 语言程序的过程步骤。

按这个顺序来介绍通过使用 RA78K0S 汇编程序包和 CC78K0S 如何与其它语言相衔接：

(1) 在 C 语言程序中调用汇编语言程序

(2) 由汇编语言程序调用 C 语言程序

(3) 引用 C 语言定义的变量

(4) 由 C 语言程序引用汇编语言定义的变量

(5) 注意事项

12.1 访问参数 / 自动变量

CC78K0S 访问参数和自动变量的过程描述如下。

12.1.1 普通模式

- 在函数调用方，寄存器参数的传递方式和普通函数参数传递方式相同。
  函数第一个参数使用如下寄存器和堆栈进行传递，后续的其它参数使用堆栈进行传递。

表 12-1 参数传递 (函数调用方）

类型	单元传递位置 (第一个参数）	传递单元传递位置 (第二个及更靠其后的参数）
1 字节 , 2 字节数据	AX	堆栈传递
3 字节 , 4 字节数据	AX, BC	堆栈传递
浮点数	AX, BC	堆栈传递
其它数据	堆栈传递	堆栈传递

  注　　1 至 4 字节的数据可以包括结构体和共用类型数据。

- 在函数定义方，通过寄存器或堆栈传递的参数都储存在参数分配单元内。
  寄存器参数拷贝到寄存器中或 saddr 区域 (_@KREGxx）中。即使参数传递已经由寄存器完成传递，由于函数调用方 (（参数传递方)）与函数定义方 (（参数接受方)）的寄存器不同，所以必须要执行寄存器拷贝。
  一般参数通过寄存器来传递，并在函数的定义方压入堆栈。如果通过堆栈完成传递，则传递位置简单地变为了参数分配的位置。
  将寄存器值保存和恢复即分配参数是在函数定义方执行的。
- 函数的参数和在函数内部声明的自动变量的值被存储在如下寄存器中，或者使用选项可以将其存储到 saddr 区域或通过选项存储在堆栈帧中。当存储在栈帧中时，使用 HL 寄存器作为基址指针。
  如果函数参数被声明为寄存器类型，或由 QV 选项和 QR 选项指定函数参数时，该参数将分配至 saddr 区域。

表 12-2 参数 / 自动变量存储 (被调用函数内）

选项	参数 / 自动变量	存储位置	优先级
-qv （寄存器分配选项）	声明的参数或自动变量	HL 寄存器 （仅当不需要基址指针时)）	字符型 : L, H, 按这个顺序 int, short, enum 型 : HL
-qr （saddr 分配选项）	声明为寄存器的参数或自动变量	HL 寄存器 （仅当不需要基址指针时)） 参数 : _@KREG12 至 15 [0FEE4H 至 0FEE7H] 自动变量 a; _@KREG00 至 11 [0FED8H 至 0FEE3H]	按照出现的顺序，只分配参数或变量的字节数量。 按这个顺序，分配寄存器为 char 型 : L, H int, short, enum 型 : HL
-qrv	声明的参数或自动变量	HL 寄存器 （仅当不需要基址指针时)） 参数 : _@KREG12 至 15 [0FEE4H 至 0FEE7H] 自动变量 ; _@KREG00 至 11 [0FED8H 至 0FEE3H]	按照出现的顺序，只分配参数或变量的字节数量。 按这个顺序，分配寄存器为 char 型 : L, H int, short, enum 型 : HL
Default	声明的参数，自动变量	堆栈帧	出现顺序

下面示例显示了函数的调用情况。

- C 源代码：在 -qrv 设定的一般模式上

```
void    func0 ( register int , int ) ;
void    main ( ) {
        func0 ( 0x1234 , 0x5678 ) ;
}
void    func0 ( register int p1 , int p2 ) {
        register int    r ;
        int             a ;
        r = p2 ;
        a = p1 ;
}
```

< 输出的汇编源程序 >

```
    EXTRN   _@KREG12
    EXTRN   _@KREG13
    EXTRN   _@KREG10
    EXTRN   _@KREG14
    PUBLIC  _func0
    PUBLIC  _main

@@CODE  CSEG
_main :
    movw    ax , #05678H    ; 22136
    push    ax                      ; 传递参数至堆栈上
    movw    ax , #01234H    ; 4660  ; 传递第 1 个参数至寄存器上
    call    !_func0                 ; 函数调用
    pop     ax                      ; 传递参数至堆栈上
    ret
_func0 :
    push    hl                      ; 保存参数寄存器
    xch     a , x
    xch     a , _@KREG12
    xch     a , x
    xch     a , _@KREG13    ; 分配寄存器参数 p1 至 _@KREG12
    push    ax              ; 保存寄存器参数使用的 saddr 区域
    movw    ax , _@KREG10
    push    ax              ; 保存寄存器参数使用的 saddr 区域
    movw    ax , _@KREG14
    push    ax              ; 保存寄存器自动变量使用的 saddr 区域
    movw    ax , sp
    movw    hl , ax
    mov     a , [ hl + 10 ]         ; 堆栈传输的参数 p2
    xch     a , x
    mov     a , [ hl + 11 ]
    movw    hl , ax                 ; 分配至 HL
    movw    ax , hl                 ; 参数 p2
    movw    _@KREG10 , ax   ; r     ; 分配至寄存器参数 r
    movw    ax, _@KREG12    ; p1    ; 寄存器参数 p1
    movw    _@KREG14 , ax   ; a     ; 分配至自动变量 a
    pop     ax
    movw    _@KREG14 , ax   ; 恢复寄存器变量的 saddr 区域
    pop     ax
    movw    _@KREG10 , ax   ; 恢复自动变量的 saddr 区域
    pop     ax
    movw    _@KREG12 , ax   ; 恢复寄存器变量的 saddr 区域
    pop    hl                       ; 恢复参数寄存器
    ret
    END
```

12.1.2 静态模式

- 在函数调用方，寄存器参数的传递方式和普通参数的传递方式相同。
- 通过一个寄存器就可以传递多达 3 个参数，或总数为 6 个字节的参数。

表 12-3 参数传递 (函数调用方）

类型	传递地址 (第一个参数)	传递地址 (第二个参数)	传递地址 (第三个参数)
1 字节数据	A	B	H
2 字节数据	AX	BC	HL
4 字节数据	分配到 AX 和 BC, ，其余分配至 H 或 HL		

注 1 至 4 字节数据不包括结构体和共用体。

- 在函数定义方，通过寄存器传递的参数储存储在参数分配单元内。
  声明为寄存器的参数 (（寄存器参数)）总是尽可能分配至寄存器中去，普通参数则分配至为特殊函数保留用的区域中去。
- 所有的寄存器参数都是通过寄存器来进行传递的，但是由于函数调用方 (（参数传递方)）与函数定义方 (（参数接受方)）的寄存器不同，所以，必须要执行寄存器拷贝。
- 将分配给寄存器的参数 / 自动变量的保存与恢复是在函数定义方执行的。
- 通过使用选项，将函数参数及函数内部所声明的自动变量值存储在如下所列的特殊函数区域区。特殊函数区域是在 RAM 中为每个函数预留的静态区域。

表 12-4 参数 / 自动变量存储 (被调用函数内）

选项	参数 /auto 自动变量	存放位置	优先级
-qv （寄存器分配选项）	声明的参数或自动变量	DE 寄存器	参数 :： char 型 :D, E, 按这个顺序 int, short, enum 型 : DE 自动变量 : char 类型 :E, D, 按这个顺序 int, short, enum 型 : DE
Default	声明的参数，自动变量	特殊函数区域	从第一个参数起进行分配，自动变量按照其出现的先后次序进行分配
Default	参数声明为寄存器变量的寄存器变量	DE 寄存器	根据被引用次数，分配的字节数量仅等于变量或参数大小。 除了变量或参数必需的字节数量，其它都被分配该函数专用区域。

下面示例显示了函数的调用情况。

< C 源程序：静态模式 -SM 和 -QV 选项指定时 >

```
void    sub ( ) ;
void    func ( register int , char ) ;
void    main ( ) {
        func ( 0x1234 , 0x56 ) ;
}
void    func ( register int  p1 , char  p2 ) {
        register char   r ;
        int             a ;
        r = p2 ;
        a = p1 ;
        sub ( ) ;
}
```

< 输出的汇编源程序 >

```
    PUBLIC  _func
    PUBLIC  _main
        :
@@DATA          DSEG
?L0005 :    DS      ( 1 )           ; 参数 p2
?L0006 :    DS      ( 1 )           ; 寄存器变量 r
?L0007 :    DS      ( 2 )           ; 自动变量 a
        :
@@CODE  CSEG
_main :
    mov     b , #056H       ; 86    ; 通过寄存器 B 传输第二个参数
    movw    ax , #01234H    ; 4660  ; 通过寄存器 AX 传输第一个参数
    call    !_func                  ; 函数调用
    ret
_func :
    push    de                      ; 保存寄存参数的寄存器
    movw    de , ax                 ; 为 DE 分配寄存器参数 p1
    movw ax,bc
    mov     !?L0005 , a             ; 复制参数 p2 到 L0005
    mov     !?L0006 , a     ; r     ; 寄存器变量 r
    movw    ax , de                 ; 寄存器参数 p1
    mov     !?L0007 + 1 , a ; a
    xch     a , x
    mov     !?L0007 , a     ; a     ; 分配自动变量 a
    call    !_sub
    pop     de                ;恢复寄存器参数使用的寄存器
    ret
    END
```

12.2 返回值的存储

函数调用期间的返回值存储在寄存器中及进位标志位中。

返回值存储位置如下表所示。

表 12-5 返回值的存储位置

类型	普通模式	静态模式
1 个字节的整数	BC	A
2 个字节的整数		AX
4 个字节的整数	BC (低字节）， DE (高字节）	不支持
指针型	BC	AX
结构体，共用体	BC (结构体或共用体的起始地址拷贝至特殊函数区域）	不支持
1 位	CY （进位标志）	CY （进位标志）
浮点数	BC (低端）， DE (高端）	不支持

12.3 在 C 语言程序中调用汇编语言程序

本节显示的是使用普通模式 (缺省）时的示例。如果设定 -qv 选项 , -qr 选项 , 以及 -qrv 选项 , 参数作为在 表 12-2 的表示存储。然而，只在不需要基址指针时才分配 HL 寄存器 （当不使用基址指针时）。

由 C 语言程序调用汇编语言程序过程描述如下。

- C 语言函数调用过程
- 汇编语言程序的数据保存和调用返回

12.3.1 C 语言函数调用过程

本例为一 C 语言程序调用汇编语言程序的示例。

```
extern  int     FUNC ( int , long ) ;   /* 函数原型 */

void    main ( )
{
        int     i ,  j ;
        long    l ;

        i = 1 ;
        l = 0x54321 ;
        j = FUNC ( i , l ) ;            /* 函数调用 */
}
```

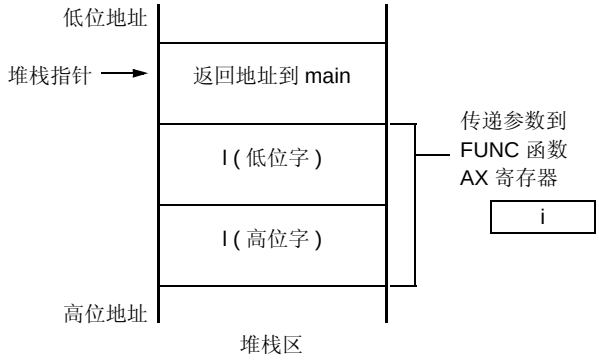
在该示例程序中，程序接口及执行程序执行的流程控制流如下所述。

(i) 将由 main 函数传给递至 FUNC 函数的第一个参数放置在寄存器中，将第二个及其后续参数都放置在堆栈中。

(ii) 使用 CALL 指令将控制权移交传递至给 FUNC 函数。

下图所显示的是上例中在控制权移交给传递至 FUNC 函数中后的即时堆栈情况。

图 12-1 调用之后的堆栈区域



12.3.2 汇编语言程序的数据保存和调用返回

被 main 函数调用 FUNC 函数中的具体执行过程如下。

(1) 保存基址指针，工作寄存器。

(2) 将堆栈指针 （SP）拷贝至基址指针 （HL）。

(3) 执行 FUNC 函数中描述的处理过程。

(4) 设置返回值。

(5) 恢复所保存的寄存器值。

(6) 返回 main 函数。

接下来，让我们通过一汇编语言程序示例来进行解释。

```
$PROCESSOR ( 9026 )

        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2

@@DATA          DSEG
?L0003 :        DS      ( 2 )
_DT2 :  DS      ( 4 )

@@CODE  CSEG
_FUNC :
        PUSH    HL                      ; 保存基址指针       (1)
        PUSH    AX
        MOVW    AX , SP                 ; 复制堆栈指针     (2)
        MOVW    HL , AX
        MOV     A , [ HL ]              ; arg1
        MOVW    !_DT1 , A               ; 移动第 1 个参数 ( i )
        XCH     A , X
        MOV     A , [ HL + 1 ]          ; arg1
        MOV     !_DT1 + 1 , A
        MOV     A , [ HL + 8 ]          ; arg2
        XCH     A , X
        MOV     A , [ HL + 9 ]          ; arg2
        MOVW    BC , AX
        MOV     A , [ HL + 6 ]          ; arg2
        XCH     A , X
        MOV     A , [ HL + 7 ]          ; arg2
        MOVW    DE , #_DT2
        XCH     A , X
        MOV     [ DE ],A                ; 移动第 2 个参数 ( l )
        XCH     A , X
        INCW    DE
        MOV     [ DE ] , A
        XCHW    AX , BC
        INCW    DE
        XCH     A , X
        MOV     [ DE ] , A
        XCH     A , X
        INCW    DE
        MOV     [ DE ] , A
        XCHW    AX , BC
        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，工作寄存器

首先，在 C 源程序中描述的函数名称前加上标签前缀 ‘_’。基址指针及工作寄存器按照 C 源程序内定义的函数名进行保存。

在描述的标签之后，对 HL 寄存器 (（基址指针)）进行保存。

在 C 编译器处理程序情况下，调用其他函数时并不会自动保存为寄存器变量保存的寄存器。因此，如果这些用于被调函数使用的工作寄存器而发生改变时，必须确保预先对这些寄存器值进行保存。但是，如果在函数调用方没有使用这些工作寄存器的值，则无需对其进行保存。

(2) 将基址指针 (HL）拷贝至堆栈指针 (SP）

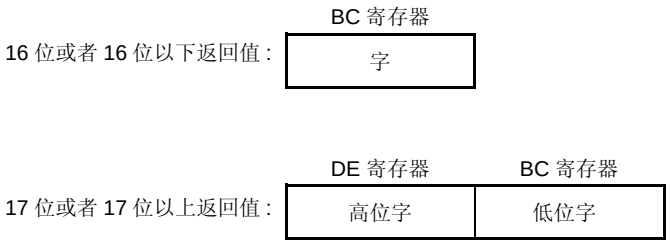
由于函数中的 "PUSH, POP" 指令改变了堆栈指针 （SP）的值。因此，要将堆栈指针拷贝至 "HL" 寄存器，用做参数的基址指针。

(3) FUNC 函数基本处理过程

在处理 （1）和 （2）执行后，执行调用函数的基本处理。

(4) 设置返回值

如果有返回值，则将其存储在 "BC" 和 "DE" 寄存器中。如果没有返回值，则无需进行存储。

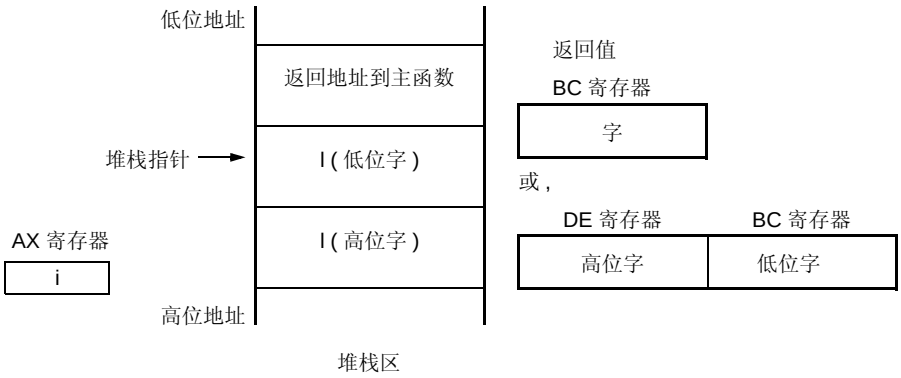


(5) 恢复寄存器值

恢复所保存的基址指针和工作寄存器。

(6) 返回至主函数

图 12-2 返回后的堆栈区域



12.4 由汇编语言程序调用 C 语言程序

12.4.1 由汇编语言程序调用 C 语言函数

由汇编语言程序调用 C 语言函数的执行过程为以下几步：

(1) 将参数保存到堆栈。

(2) 保存 C 工作寄存器 （AX, BC, 及 DE）。

(3) 调用 C 语言函数。

(4) 根据参数所占的字节数增加堆栈指针 （SP）值。

(5) 引用 C 语言函数的返回值 （在 BC 或 DE 及 BC 中）。

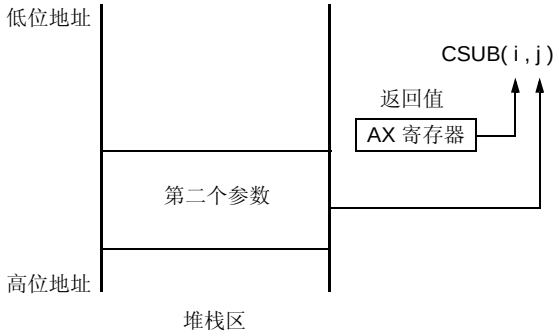
下例为汇编语言程序示例。

```
$PROCESSOR ( 9216 )

        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2

@@CODE  CSEG
_FUNC2 :
        movw    ax, #20H        ; 设置第二个参数 ( j )
        push    ax              ;
        movw    ax , #21H       ; 设置第一个参数 ( i )
        call    !_CSUB          ; 调用 " CSUB ( i , j ) "
        pop     ax              ;
        ret
        END
```

(1) 参数入栈

放置任何参数在堆栈上。图 12-3 显示参数传递

图 12-3 在堆栈上放置参数



(2) 保存工作寄存器 (（AX, BC 及 DE 寄存器)）

AX,BC 和 DE 三组寄存器用于 C 语言。在返回时，它们的值不能恢复。因此，如果需要寄存器中的值，在调用方上保存它们。

在参数传递前后，要保存或恢复寄存器。当在 C 语言函数中使用了 HL 寄存器时，该寄存器始终需要保存在 C 语言函数方进行保存。

(3) 调用 C 语言函数

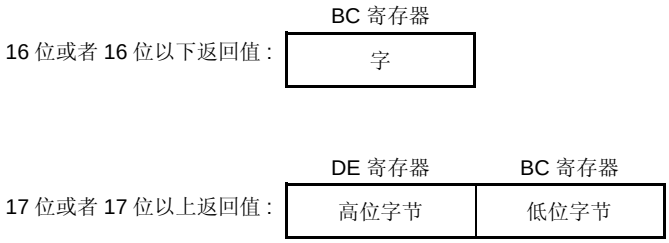
用 CALL 指令调用 C 语言函数。如果该 C 语言函数为 “callt” 函数，那么就由 callt 指令执行函数调用。

(4) 恢复堆栈指针 (SP）

根据参数所占用的字节数来恢复堆栈指针。

(5) 引用返回值 (BC 和 DE）

C 语言函数返回值的返回情况如下。

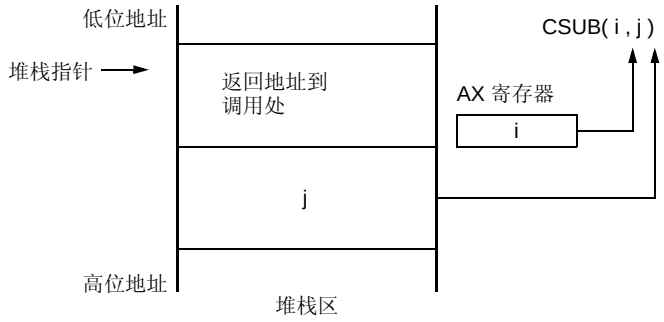


(6) 引用在 C 语言函数中引用的参数

为了正确地将参数 “i” 和 “j” 传至如下所示 C 语言程序中，将其按图 12-4 所示放入堆栈中。

```
void    CSUB ( i , j )
int     i ,  j ;
{
        i += j ;
}
```

图 12-4 传递参数到 C 语言



12.5 引用其它语言定义的变量

12.5.1 引用 C 语言定义的变量

如果在汇编语言程序中引用 C 语言定义的外部变量，就必须使用 EXTRN 进行外部声明。要引用在汇编语言程序中定义的变量，在变量名称前要加一个下划线 “_”。

<C 语言程序示例 >

```
extern  void    subf ( ) ;

char    c = 0 ;
int     i = 0 ;
void    main ( )
{
        subf ( ) ;
}
```

下面示例程序为 RA78K0S 汇编编译器下的汇编程序。

```
$PROCESSOR ( 9216 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf :
        MOV     a , #04H
        MOV     !_c , a
        MOVW    ax , #07H       ; 7
        MOVW    de , #_i
        INCW    DE
        MOV     [ DE ] , A
        DECW    DE
        XCH     A , X
        MOV     [ DE ] , A
        RET
        END
```

12.5.2 由 C 语言程序引用汇编语言定义的变量

由 C 语言程序引用汇编语言定义的变量按照如下这种方式进行。

< C 语言程序示例 >

```
extern  char    c ;
extern int       i ;

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

下面示例程序为 RA78K0S 汇编编译器下的汇编程序。

```
        NAME    ASMSUB

        PUBLIC  _c
        PUBLIC  _i

ABC     DSEG
_c :    DB      0
_i :    DW      0

        END
```

12.6 注意事项

(1) “_”(下划线)

CC78K0S 对外部定义及输出的目标模块中的引用名称前加一个下划线 “_”(ASCII 码 “5FH”)。在下面 C 语言程序示例中，“j = FUNC(（i, l)）;”就被认为是 “对作为外部函数名称 _FUNC 的一个引用”。

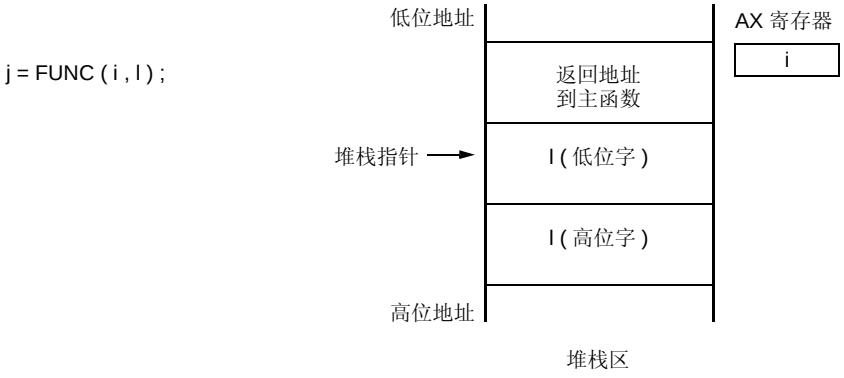
```
extern  int     FUNC ( int, long );     /* 函数原型 */

void main ( )
{
        int     i , j ;
        long    l ;

        i = 1 ;
        l = 0x54321 ;
        j = FUNC ( i, l );              /* 函数调用 */
}
```

RA78K0S 中的程序名称为写作 “_FUNC”。

(2) 参数在堆栈中的位置

参数在堆栈中的存放位置是按照从后缀参数到前缀参数，由高地址向低地址方向顺序存放的顺序进行的。

图 12-5 参数的堆栈位置



第 13 章　编译器的有效应用

本章介绍如何有效地使用 CC78K0S。

13.1　有效编码

在开发 78K0S 串行微机 - 应用的产品 , 通过利用 saddr 区域，调用表或设备调用区域使用 CC78K0S 能实现有效目标的生成。

- 使用外部变量
  - 如果 （saddr 区域可用）
    - 使用 sreg/__sreg 变量 /
      指定编译器选项 （-RD）
- 使用 1 位数据
  - 如果 （saddr 区域可用）
    - 使用 bit/boolean/ __boolean 型变量
- 函数定义
  - 如果 （函数被多次调用）
    - 如果 （callt 区可用）
      - 用作 __callt/callt 函数 （可以有效降低代码大小）
    - 如果 （没有递归调用）
      - 用作 __leaf/norec 函数
  - 如果 （没有使用自动变量）
    - 用作 noauto 函数
  - 如果 （使用自动变量 并且 saddr 区域可用）

(1) 使用外部变量

如果 saddr 区域可用，则在定义外部变量时，将变量定义为 sreg/__sreg 变量。 sreg/__sreg 变量的相关指令比分配到内存的那些变量的指令在代码长度上要短。这有助于缩短目标代码也改进程序执行速度。（不使用 sreg 变量，通过指定 -RD 选项也可以获得相同效果。）

```
sreg/__sreg 变量：    extern sreg int 变量名称 ;
                                    extern __sreg int 变量名称 ;
```

备注　　参考 “11.5 (3) 如何使用 saddr 区域 (sreg / __sreg)”。

(2) 1 位型数据

只使用一位数据的数据对象应该声明为 bit 型变量 （或者声明为 boolean/__boolean 型变量）。对 bit/ boolean/_ _boolean 型变量操作，都将通过位操作指令来完成。由于 saddr 区域的使用方式和 sreg 变量相同，于是便缩短了对象目标代码，也进而提高了程序执行速度。

```
声明 bit/boolean 型变量 ::      bit 变量名称 ;
                                              boolean 变量名称 ;
                                              __boolean 变量名称 ;
```

备注　　参考 "11.5 (7) bit 位变量，boolean 型变量 (bit / boolean / __boolean)"。

(3) 函数的定义

对于需要反复调用的函数，要求其目标代码应该更短或提供可以高速调用的结构。如果这些被频繁调用函数可以使用 callt 区域，那么这类函数就应该定义为 callt 函数。由于对 callt/callf 函数的调用是使用设备的 callt 区进行的，所以，对 callt/callf 函数的调用比普通函数调用速度要快，而且代码长度也要短。

```
定义 callt 函数的定义 :  callt   int     tsub ( )  {
                :
}
```

备注　　参考 "11.5 (1) callt 函数 (callt / __callt)", 和 "11.5 (6) norec 函数 (norec)"。

除了使用 saddr 区 saddr 区域之外，通过使用最优化选项编译来，也可以生成一些无需 C 源程序修改的 C 源程序目标代码对象。 如果用户需要了解每个 -Q 子选项详情，敬请参阅 CC708KOS C 编译器操作篇。

(4) 优化选项

特别强调目标代码大小的最优选项如下。

< 特别强调目标代码大小 >

```
-qx3
```

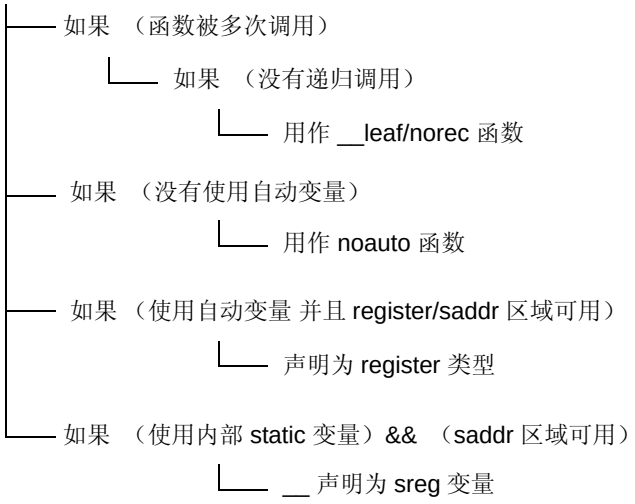
将变量定义为 __sreg，可以进一步缩减代码大小并提高程序执行的速度。但是，这种方法限于 saddr 区域可用情况下，如果区域空间不足以及 saddr 区域不能用，就会出现编译错误。

如果需要更进一步提高程序执行的速度，请指定 -QX2 为缺省选项。

另外，还可以通过将 CC78K0S 支持的扩展函数添加至 C 源程序中去的方法，这样可以来提高目标代码的效率。

(5) 使用扩展描述

- 函数定义



- 函数没有被递归调用

反复调用函数，没有递归式的使用它们，则应该定义为 __leaf/norec 函数。 norec 函数没有预处理 / 后处理的函数 （栈帧）。因此，相对于普通函数，目标代码可以缩短从而提高执行速度。

备注 有关 norec 函数的定义 (norec int rout ( )...), 参阅 "11.5 (6) norec 函数 (norec)" 和 "11.7.4 norec 函数调用接口 （普通模式）"。

- 不使用自动变量的函数

对那些不使用自动变量的函数应该将其定义为 noauto 函数。这些函数不能以栈帧格式输出代码，且它们的参数将尽可能的传递到寄存器中。这函数有助于缩短目标代码也改进程序执行速度。

备注 有关 noauto 函数定义的 （noauto int sub1 (int i) ...)，参阅 "11.5 (5) noauto 函数 (noauto)", "11.7.3 noauto 函数调用接口 （仅普通模式） "。

- 使用自动变量的函数

  如果 saddr 区域可以为用于那些不使用自动变量的函数之用，用 register 存储类型限定符来进行函数声明。通过这个 register 声明，声明的目标将分配到寄存器中。程序使用寄存器比使用存储体运行速度要更快，同时也能精简目标代码。

  备注　　有关 register 变量的定义 (register int i; ...)，参阅 "11.5 (2) 寄存器变量 (register)" 。

- 使用内部静态变量的函数

  如果函数能使用 saddr 区域，则使用内部静态变量，声明函数使用 __sreg 或者指定 -rs 选项与 sreg 变量方式一样，缩短了目标代码长度，提高程序执行速度。

  备注　　参考 "11.5 (3) 如何使用 saddr 区域 (sreg / __sreg)"。

除此之外，还可以通过如下方法来提高编码效率和程序执行速度。

- 使用 SFR 名（或 SFR 位名）。

  #pragma sfr

- 对那些只有一位成员的位段，使用 __sreg 声明。（unsigned char 型数据可以用作成员）。

```
__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 ;
```

- 使用乘法和除法嵌入式函数。

  #pragma mul

  #pragma div

- 仅描述汇编语言中那些执行速度需要提高的模块。

附录 A　saddr 区域标签列表

在 CC78K0S 中 , 通过以下标签名来引用 saddr 区域。因此，在 C 语言源程序及汇编语言源程序中就不能使用如下所列的标签名称。

A.1　普通模式

(a) 寄存器变量

表 A-1 寄存器变量 (普通模式)

标签名	地址
_@KREG00	0FED8H
_@KREG01	0FED9H
_@KREG02	0FEDAH
_@KREG03	0FEDBH
_@KREG04	0FEDCH
_@KREG05	0FEDDH
_@KREG06	0FEDEH
_@KREG07	0FEDFH
_@KREG08	0FEE0H
_@KREG09	0FEE1H
_@KREG10	0FEE2H
_@KREG11	0FEE3H
_@KREG12	0FEE4H 注
_@KREG13	0FEE5H 注
_@KREG14	0FEE6H 注
_@KREG15	0FEE7H 注

注　当函数参数由 register 声明时，或者指定了 -QV 选项和 -QR 选项时，那么参数便被分配至 saddr 区域。

(b) norec 函数的参数

表 A-2 norec 函数的参数 (普通模式)

标签名	地址
_@NRARG0	0FEE8H
_@NRARG1	0FEEAH
_@NRARG2	0FEECH
_@NRARG3	0FEEEH

(c) norec 函数的自动变量

表 A-3 norec 函数的自动变量

标签名	地址
_@NRAT00	0FEF0H
_@NRAT01	0FEF1H
_@NRAT02	0FEF2H
_@NRAT03	0FEF3H
_@NRAT04	0FEF4H
_@NRAT05	0FEF5H
_@NRAT06	0FEF6H
_@NRAT07	0FEF7H

(d) 运行时刻库的参数

表 A-4 运行时刻库的参数

标签名	地址
_@RTARG0	0FEF8H
_@RTARG1	0FEF9H
_@RTARG2	0FEFAH
_@RTARG3	0FEFBH
_@RTARG4	0FEFCH
_@RTARG5	0FEFDH
_@RTARG6	0FEFEH
_@RTARG7	0FEFFH

A.2 静态模式

(a) 共享区域

表 A-5 共享区域 (静态模式)

标签名	地址
_@KREG00	0FEF0H
_@KREG01	0FEF1H
_@KREG02	0FEF2H
_@KREG03	0FEF3H
_@KREG04	0FEF4H
_@KREG05	0FEF5H
_@KREG06	0FEF6H
_@KREG07	0FEF7H
_@KREG08	0FEF8H
_@KREG09	0FEF9H
_@KREG10	0FEFAH
_@KREG11	0FEFBH
_@KREG12	0FEFCH
_@KREG13	0FEFDH
_@KREG14	0FEFEH
_@KREG15	0FEFFH

(b) 对于参数 , 自动变量和工作区

表 A-6 对于参数 , 自动变量 , 和工作区

标签名	地址
_@NRAT00	0FExxH 注
_@NRAT01	_@NRAT00 + 1
_@NRAT02	_@NRAT00 + 2
_@NRAT03	_@NRAT00 + 3
_@NRAT04	_@NRAT00 + 4
_@NRAT05	_@NRAT00 + 5
_@NRAT06	_@NRAT00 + 6
_@NRAT07	_@NRAT00 + 7

注 在 saddr 区域的任意地址

附录 B　区段名称列表

这章说明了全部的编译器输出段以及其位置。

(1) 和 （2）显示选项以及在表中使用的重定位属性。

本节介绍所有段和通过编译程序输出的分配。

(1) CSEG 重定位属性

CALLT0 :	对于已定义段的重新分配，使其起始地址为范围在 30H 至 7FH 内的偶数。
AT 绝对表达式 :	分配指定段到绝对地址处 (在 0000H 到 FEFFH 的范围内)。
FIXED :	在 0800H 到 0FFFH 范围内分配指定段。
UNITP :	分配指定段使其起始地址为 (在 80H 到 0FA7EH 范围内) 的任意一个偶地址处。

(2) DSEG 重定位属性

SADDRP :	分配指定段使其起始地址为 (在 FE20H 至 FEFFH 范围内) 的 saddr 区域。
UNITP :	分配指定段使其起始地址为 （默认是在 RAM 区域内）的任意一个偶地址处。

B.1 段名称列表

B.1.1 程序区域和数据区域

表 B-1 段名列表 (程序区域和数据区域) ...

区段名称	区段类型	重定位属性	说明
@@CODE	CSEG		存放代码的区段
@@LCODE	CSEG		存放库代码的区段
@@CNST	CSEG		存放常量的区段
@@R_INIT	CSEG		初始化数据区段 （带初始值）
@@R_INIS	CSEG	UNITP	初始化数据区段 （带初始值的 sreg 变量）
@@CALT	CSEG	CALLT0	callt 函数表区段
@@VECTnn	CSEG	AT 00nnH	向量表区段[注]
@@INIT	DSEG		数据区域段 （带初始值）
@@DATA	DSEG		数据区域段 （无初始值）
@@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）
@@BITS	BSEG		布尔运算类型和比特类型变量区段

注 根据中断类型改变 nn 的值。

B.2 区段地址

表 B-2 区段地址

区段类型	所定位的目标区 （默认）
CSEG	ROM
BSEG	RAM 的 saddr 区域
DSEG	RAM

B.3 C 源程序示例

```
#pragma INTERRUPT       INTP0   inter   /* 中断向量 */

void    inter ( void ) ;        /* 中断函数原型声明 */
const   int     i_cnst = 1 ;        /* 常量 */
callt   void    f_clt ( void ) ;    /* callt 函数原型声明 */
boolean b_bit ;                     /* boolean 型变量 */
long    l_init = 2 ;                /* 带初始值的外部变量 */
int     i_data ;                /* 不带初始值的外部变量 */
sreg    int     sr_inis = 3 ;       /* 带初始值的 sreg 变量 */
sreg    int     sr_dats ;           /* 带初始值的 sreg 变量 */

void    main ( )                    /* 函数定义 */
{
        int     i ;
        i = 100 ;
}

void    inter ( )                   /* 中断函数定义 */
{
        unsigned char   uc = 0 ;
        uc++ ;
        if ( b_bit )
        b_bit = 0 ;
}

callt   void    f_clt ( )           /* callf 函数定义 */
{
}
```

B.4 输出汇编程序模块的示例

根据设备的不同在汇编程序源代码中设置类似指令和命令。

详情参阅 RA78K0S 汇编程序包操作用户手册。

```
; 78K/0S 序列 C 编译程序 V1.60 汇编源程序
;                              日期 : xx xxx xxxx 时间 : xx : xx : xx

; 命令   : -c9026 sampk0s.c -sa -ng
; In-file   : sampk0s.c
; Asm-file  : sampk0s.asm
; Para-file :

$PROCESSOR ( 9026 )
$NODEBUG
$NODEBUGA
$KANJICODE    SJIS
$TOL_INF      03FH , 0130H , 00H , 00H

              EXTRN   _@cprep
              PUBLIC  _inter
              PUBLIC  _i_cnst
              PUBLIC  ?f_clt
              PUBLIC  _i_cnst
              PUBLIC  _b_bit
              PUBLIC   _l_init
              PUBLIC  _i_data
              PUBLIC  _sr_inis
              PUBLIC  _sr_dats
              PUBLIC  _main
              PUBLIC  _f_clt
              PUBLIC  _@vect06

@@BITS        BSEG                              ; boolean 型变量段
_b_bit        BIT

@@CNST        CSEG                              ; const 变量段
_i_cnst :     DW      01H                       ; 1

@@R_INIT      CSEG                              ; 初始化数据段
                                                ; ( 有初值的外部变量 )
              DW      00002H , 00000H           ; 2

@@INIT        DSEG                              ; 数据区段
                                                ; ( 有初值的外部变量 )
_l_init :     DS      ( 4 )
@@DATA        DSEG    UNITP                     ; 数据区段
                                                ; ( 无初值的外部变量 )
_i_data :     DS      ( 2 )

@@R_INIS      CSEG                              ; 初始化数据段
                                                ; ( 有初值的 sreg 变量 )
              DW      03H                       ; 3

@@INIS        DSEG    SADDRP                    ; 数据区段
                                                ; ( 有初值的 sreg 变量 )
_sr_inis :    DS      ( 2 )
```

```
@@INIS        DSEG    SADDRP                      ; 数据区段
                                                  ; ( 无初值的 sreg 变量 )
_sr_dats :    DS      ( 2 )

@@CALT        CSEG    CALLT0                      ; callt 函数的段
?f_clt :      DW      _f_clt

; line  1 :   #pragma INTERRUPT   INTP0   inter   /* 中断向量 */
; line  2 :
; line  3 :   void    inter ( void ) ;            /* 中断函数 */
                                                  /* 原型声明 */
; line  4 :   const   int     i_cnst = 1 ;        /* 常量 */
; line  5 :   callt   void    f_clt ( void ) ;    /* callt 函数原型 */
                                                  /* 声明 */
; line  6 :   boolean b_bit ;                     /* boolean 型变量 */
; line  7 :   long    l_init = 2 ;                /* 带初始值的外部变量 */
; line  8 :   int     i_data ;                    /* 外部变量 */
                                                  /* 无初始值 */
; line  9 :   sreg    int     sr_inis = 3 ;       /* 带初始值的 sreg 变量 */
; line  10 :  sreg    int     sr_dats ;           /* 不带初始值的 sreg 变量 */

; line  11 :
; line  12 :  void    main ( )                    /* 函数定义 */
; line  13 :  {

@@CODE  CSEG                                      ; 代码部分区段
_main :
        push  hl                                  ; [ INF ] 1 , 4
        movw  ax , #02H                           ; [ INF ] 3 , 6
        callt [ _@cprep ]                         ; [ INF ] 1 , 8
; line  14 :  int     i ;
; line  15 :  i = 100 ;
        movw  ax , #064H      ; 100               ; [ INF ] 3 , 6
        mov   [ hl + 1 ] , a  ; i                 ; [ INF ] 2 , 6
        xch   a , x                               ; [ INF ] 1 , 4
        mov   [ hl ] , a      ; i                 ; [ INF ] 1 , 6
; line  16 :  }
        pop   ax                                  ; [ INF ] 1 , 6
        pop   hl                                  ; [ INF ] 1 , 6
        ret                                       ; [ INF ] 1 , 6
; line  17 :
; line  18 :  void    inter ( )                   /* 中断函数定义 */
; line  19 :  {
_inter :
        push  ax                                  ; [ INF ] 1 , 4
        push  de                                  ; [ INF ] 1 , 4
        push  hl                                  ; [ INF ] 1 , 4
        movw  ax , #02H                           ; [ INF ] 3 , 6
        callt [ _@cprep ]                         ; [ INF ] 1 , 8
; line  20 :  unsigned char   uc = 0 ;
        xor   a , a                               ; [ INF ] 2 , 4
        mov   [ hl + 1 ] , a  ; uc                ; [ INF ] 2 , 6
; line  21 :  uc++ ;
        inc   a                                   ; [ INF ] 2 , 4
        xch   a , [ hl + 1 ]  ; uc                ; [ INF ] 2 , 8
; line  22 :  if ( b_bit )
        bf    _b_bit , $L0005                     ; [ INF ] 4 , 10
; line  23 :  b_bit = 0 ;
        clr1  _b_bit                              ; [ INF ] 3 , 6
```

```
L0005 :
; line  24 :  }
        pop   ax                                    ; [ INF ] 1 , 6
        pop   hl                                    ; [ INF ] 1 , 6
        pop   de                                    ; [ INF ] 1 , 6
        pop   ax                                    ; [ INF ] 1 , 6
        reti                                        ; [ INF ] 1 , 8
; line  26 :
; line  27 :  callt   void f_clt ( )                 /* callt 函数定义 */
; line  28 :     {
_f_clt:
; line  29 :     }
        ret                                         ; [ INF ] 1 , 6

@@VECT06      CSEG    AT      0006H                 ; 中断向量
_@vect06 :
        DW    _inter
        END

; *** 编码信息 ***
;
; $FILE C:\NECTools32\work\sampk0s.c
;
; $FUNC main ( 13 )
;       void = ( void )
;       CODE SIZE = 15 字节 , CLOCK_SIZE = 58 clocks , STACK_SIZE = 6 字节
;
; $FUNC inter ( 19 )
;       void =  ( void )
;       CODE SIZE = 27 字节 , CLOCK_SIZE = 96 clocks , STACK_SIZE = 10 字节
;
; $FUNC f_clt ( 27 )
;       void = ( void )
;       CODE SIZE = 1 字节 , CLOCK_SIZE = 6 clocks , STACK_SIZE = 0 字节
;
; Target chip : uPD789026
; Device file : Vx.xx
```

附录 C　运行时间库列表

表 C-1 显示运行时间库列表。

通过在函数名称的开头附加 @@ 等格式，调用这些操作指令。

因此 , 通过在函数名称的开头附加 _@ 的格式调用 cstart, cprep, 和 cdisp。

没有库支持可以不在 表 C-1 中执行。编译程序执行内联展开

long 型的加和减，与 / 或 / 异或，和移位也会内联展开。

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
自增	lsinc	OK	-	有符号 long 型加 1
	luinc	OK	-	无符号 long 型加 1
	finc	OK	-	float 型加 1
自减	lsdec	OK	-	有符号 long 型减 1
	ludec	OK	-	无符号 long 型减 1
	fdec	OK	-	float 型减 1
符号取反	lsrev	OK	-	有符号 long 型的符号取反
	lurev	OK	-	无符号 long 型的符号取反
	frev	OK	-	float 型符号取反
1 的补码	lscom	OK	-	得到有符号 long 型 1 的补码
	lucom	OK	-	得到无符号 long 型 1 的补码
逻辑非	lsnot	OK	-	有符号 long 型取反
	lunot	OK	-	无符号 long 型取反
	fnot	OK	-	float 型取反
乘法	csmul	OK	OK	在有符号 char 数据间进行乘法运算
	cumul	OK	OK	在无符号 char 数据间进行乘法运算
	ismul	OK	OK	在有符号 int 型数据间进行乘法运算
	iumul	OK	OK	在无符号 int 型数据间进行乘法运算
	lsmul	OK	-	在有符号 long 型数据间进行乘法运算
	lumul	OK	-	在无符号 long 型数据间进行乘法运算
	fmul	OK	-	进行 float 型数据间的乘法运算

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
除法	csdiv	OK	OK	在有符号 char 型数据间进行除法运算
	cudiv	OK	OK	在无符号 char 型数据间进行除法运算
	isdiv	OK	OK	在有符号 int 型数据间进行除法运算
	iudiv	OK	OK	在无符号 int 型数据间进行除法运算
	lsdiv	OK	-	在有符号 long 型数据间进行除法运算
	ludiv	OK	-	在无符号 long 型数据间进行除法运算
	fdiv	OK	-	进行 float 型数据间的除法运算
取余	csrem	OK	OK	在有符号 char 型数据型间进行取余运算
	curem	OK	OK	在无符号 char 型数据间进行取余运算
	isrem	OK	OK	在有符号 int 型数据间进行取余运算
	iurem	OK	OK	在无符号 int 型数据间进行取余运算
	lsrem	OK	-	在有符号 long 型数据间进行取余运算
	lurem	OK	-	在无符号 long 型数据间进行取余运算
加法	lsadd	OK	-	在有符号 long 型数据间进行加法运算
	luadd	OK	-	在无符号 long 型数据间进行加法运算
	fadd	OK	-	进行 float 型数据间的加法运算
减法	lssub	OK	-	在有符号 long 型数据间进行减法运算
	lusub	OK	-	在无符号 long 型数据间进行减法运算
	fsub	OK	-	进行 float 型数间的减法运算
向左移位	islsh	OK	OK	把有符号 int 型数据朝左移动
	iulsh	OK	OK	把无符号 int 型数据朝左移动
	lslsh	OK	-	把有符号 long 型数据朝左移动
	lulsh	OK	-	把无符号 long 型数据朝左移动
向右移位	isrsh	OK	OK	把有符号 int 型数据朝右移动
	iursh	OK	OK	把无符号 int 型数据朝右移动
	lsrsh	OK	-	把有符号 long 型数据朝右移动
	lursh	OK	-	把无符号 long 型数据朝右移动

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
比较	cscmp	OK	OK	比较有符号 char 型数据
	iscmp	OK	OK	比较无符号 int 型数据
	lscmp	OK	-	比较有符号 long 型数据
	lucmp	OK	-	比较无符号 long 型数据
	fcmp	OK	-	比较 float 型数据
位与	lsband	OK	-	在有符号 long 型数据间进行与运算
	luband	OK	-	在无符号 long 型数据间进行与运算
位或	lsbor	OK	-	在有符号 long 型数据间进行或运算
	lubor	OK	-	在无符号 long 型数据间进行或运算
位异或	lsbxor	OK	-	在有符号 long 型数据间进行异或运算
	lubxor	OK	-	在无符号 long 型数据间进行异或运算
逻辑与	fand	OK	-	在 2 个 float 型数据之间进行逻辑与运算
逻辑或	当	OK	-	在 2 个 float 型数据之间进行逻辑或运算
转换浮点数	ftols	OK	-	从 float 型转换到有符号 long 型
	ftolu	OK	-	从 float 型转换到无符号 long 型
转换为浮点数	lstof	OK	-	从有符号 long 型转换到 float 型
	lutof	OK	-	从无符号 long 型转换到 float 型
位的转换	btol	OK	-	从 bit 型转换为 long 型

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	

启动程序	cstart	OK	OK	启动模块
				- 在 atexit 函数预留的函数注册区 (2 × 32 字节) 之后，设置其起始标签为 _@FNCTBL。
				- 保留中断区 (32 字节), 并设置 _@MEMTOP 起始标签，然后将该区域的下一个对应标签设置为 _@MEMBTM 。
				- 按如下方式定义复位矢量表的段，并且设置启动模块的起始地址。
				`@@VECT00 CSEG AT 0000H`
				`DW _@cstart`
				- 设置寄存器组为 RB0。
				- 将输入错误编号的变量 _errno 设置为 0。
				- 将 _@FNCENT 变量设置为 0，该变量用于存放由 atexit 函数注册的函数目。
				- 将 _@MEMTOP 地址赋给 _@BRKADR 变量，当作初始中断值。
				- 设置 _@SEED 变量的初始值为 1，该变量为 rand 函数的伪随机数发生源。
				- 执行对数据进行初始化的拷贝过程，同时对无初值的外部数据清 0。
				- 调用 main 函数 （用户程序 ）。
				- 通过参数 0 来调用 exit 函数。

函数的预处理和后处理	cprep	OK	-	函数的预处理
	cdisp	OK	-	函数的后处理
	cprep2	OK	-	函数的预处理 （包含寄存变量 saddr 区域）
	cdisp2	OK	-	函数的后处理 （包含寄存变量 saddr 区域）

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
函数的预处理和后处理	nrcp2	-	OK	用于复制参数
	nrcp3	-	OK	
	krcp2	-	OK	
	krcp3	-	OK	
	nkrc3	-	OK	
	nrip2	-	OK	
	nrip3	-	OK	
	krip2	-	OK	
	krip3	-	OK	
	nkri31	-	OK	
	nkri32	-	OK	
	nrsave	-	OK	用于保存 _@NRATxx
	nrload	-	OK	用于恢复 _@NRATxx
	krs02	-	OK	用于保存 _@KREGxx
	krs04	-	OK	
	krs04i	-	OK	
	krs06	-	OK	
	krs06i	-	OK	
	krs08	-	OK	
	krs08i	-	OK	
	krs10	-	OK	
	krs10i	-	OK	
	krs12	-	OK	
	krs12i	-	OK	
	krs14	-	OK	
	krs14i	-	OK	
	krs16	-	OK	
	krs16i	-	OK	

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
函数的预处理和后处理	krl02	-	OK	用于恢复 _@KREGxx
	krl04	-	OK	
	krl04i	-	OK	
	krl06	-	OK	
	krl06i	-	OK	
	krl08	-	OK	
	krl08i	-	OK	
	krl10	-	OK	
	krl10i	-	OK	
	krl12	-	OK	
	krl12i	-	OK	
	krl14	-	OK	
	krl14i	-	OK	
	krl16	-	OK	
	krl16i	-	OK	
	hdwinit	OK	OK	在 CPU 复位后，立即进行外围设备 (sfr) 的初始化处理。
BCD- 类型转换	bcdtob	OK	OK	转换 1 字节 bcd 码为 1 字节 2 进制数
	btobcd	OK	OK	转换 1 字节 2 进制数为 2 字节 bcd 码
	bcdtow	OK	OK	转换 2 字节 bcd 码为 2 字节 2 进制数
	wtobcd	OK	OK	转换 2 字节 2 进制数为 2 字节 bcd 码
	bbcd	OK	OK	转换 1 字节 1 进制数为 2 字节 bcd 码

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
辅助函数	mulu	OK	OK	兼容 mulu 指令
	divuw	OK	OK	兼容 divuw 指令
	clra0	OK	OK	用来替换固定类型指令模式
	clra1	OK	OK	
	clrax0	OK	OK	
	clrax1	OK	OK	
	clrbc0	OK	OK	
	clrbc1	OK	OK	
	cmpa0	OK	OK	
	cmpa1	OK	OK	
	cmpc0	OK	OK	
	cmpax0	OK	OK	
	cmpax1	OK	OK	
	movca	OK	OK	
	movac	OK	OK	
	ctoi	OK	OK	
	uctoi	OK	OK	
	adjba	OK	OK	
	adjbs	OK	OK	
	addrde	OK	OK	
	addrhl	OK	OK	
	shl4	OK	OK	
	shr4	OK	OK	
	tabled	OK	OK	
	tableh	OK	OK	
	apdecd	OK	OK	
	apdech	OK	OK	
	apincd	OK	OK	
	apinch	OK	OK	
	deilo	OK	OK	
	deist	OK	OK	
	deiinc	OK	OK	
	deidec	OK	OK	

表 C-1 运行时间库

分类	函数名称	支持模式		功能
		普通模式	静态模式	
辅助函数	hlilo	OK	OK	用来替换固定类型指令模式
	hlist	OK	OK	
	hliinc	OK	OK	
	hlidec	OK	OK	
	dellab	OK	-	
	dell03	OK	-	
	della4	OK	-	
	delsab	OK	-	
	dels03	OK	-	
	hlllab	OK	-	
	hlll03	OK	-	
	hllla4	OK	-	
	hllsab	OK	-	
	hlls03	OK	-	
	hliadd	OK	OK	
	hlisub	OK	OK	
	hlicmp	OK	OK	
	hliand	OK	OK	
	hlior	OK	OK	
	hlixor	OK	OK	
	imule	OK	OK	
	isdive	OK	OK	
	iudive	OK	OK	
	isreme	OK	OK	
	iureme	OK	OK	
	iadde	OK	OK	
	isube	OK	OK	
	iande	OK	OK	
	iore	OK	OK	
	ixore	OK	OK	

附录 D　库堆栈消耗表

表 D-1 显示从标注库中消耗的堆栈数量。

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longjmp	2	2
stdarg.h	va_arg	0	-
	va_start	0	-
	va_end	0	-

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
stdio.h	sprintf	52 (72) 注 1	-
	sscanf	290 (304) 注 1	-
	printf	54 (72) 注 1	-
	scanf	294 (304) 注 1	-
	vprintf	52 (72) 注 1	-
	vsprintf	52 (72) 注 1	-
	getchar	0	0
	gets	6	6
	putchar	0	0
	puts	4	4

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
stdlib.h	atoi	4	4
	atol	10	-
	strtol	20	-
	strtoul	20	-
	calloc	14	14
	free	8	8
	malloc	6	6
	realloc	12	12
	abort	0	0
	atexit	0	0
	exit	2 + n 注 2	2 + n 注 2
	abs	0	0
	div	6	-
	labs	2	-
	ldiv	16	-
	brk	0	0
	sbrk	4	4
	atof	33	-
	strtod	33	-
	itoa	10	10
	ltoa	16	-
	ultoa	16	-
	rand	14	-
	srand	0	-
	bsearch	32 + n 注 4	-
	qsort	16 + n 注 5	-
	strbrk	0	0
	strsbrk	4	4
	stritoa	10	10
	strltoa	16	-
	strultoa	16	-

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
string.h	memcpy	4	6
	memmove	4	8
	strcpy	2	4
	strncpy	4	6
	strcat	2	4
	strncat	4	6
	memcmp	2	4
	strcmp	2	2
	strncmp	2	4
	memchr	2	2
	strchr	2	0
	strcspn	6	6
	strpbrk	4	4
	strrchr	4	4
	strspn	6	6
	strstr	4	4
	strtok	4	4
	memset	4	4
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	4

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
math.h	acos	24	-
	asin	24	-
	atan	20	-
	atan2	21	-
	cos	24 (34) 注 6	-
	sin	24 (34) 注 6	-
	tan	26 (34) 注 6	-
	cosh	24	-
	sinh	25	-
	tanh	30	-
	exp	22	-
	frexp	2 (10) 注 6	-
	ldexp	2 (10) 注 6	-
	log	24 (34) 注 6	-
	log10	24 (34) 注 6	-
	modf	2 (10) 注 6	-
	pow	25 (35) 注 6	-
	sqrt	18	-
	ceil	2	-
	fabs	0	-
	floor	2	-
	fmod	2 (10) 注 6	-
	matherr	0	-
	acosf	24	-
	asinf	24	-
	atanf	20	-
	atan2f	21	-
	cosf	24 (34) 注 6	-
	sinf	24 (34) 注 6	-
	tanf	26 (34) 注 6	-
	coshf	24	-
	sinhf	25	-

表 D-1 标准库堆栈消耗表

分类	函数名称	普通模式	静态模式
math.h	tanhf	30	-
	expf	22	-
	frexpf	2 (10) 注 6	-
	ldexpf	2 (10) 注 6	-
	logf	24 (34) 注 6	-
	log10f	24 (34) 注 6	-
	modff	2 (10) 注 6	-
	powf	25 (35) 注 6	-
	sqrtf	18	-
	ceilf	2	-
	fabsf	0	-
	floorf	2	-
	fmodf	2 (10) 注 6	-
assert.h	__assertfail	66 (84) 注 7	-

注 1. 括号中的值适用于使用支持浮点数的版本。

注 2. n 为外部函数总的堆栈占用空间，这些外部函数由 atexit 函数注册。

注 3. 括号中的值只有在使用乘法 / 除法时使用的。

注 4. n 是 bsearch 函数调用的外部函数占用堆栈空间。

注 5. n 是 (20 + 从 qsort 调用的外部函数的堆栈占用空间) x (1 + 递归调用次数)。

注 6. 括号内的数值是指有操作异常发生是的数值。

注 7. 括号内的数值是指适用支持浮点数的 printf 版本时的数值。

表 D-2 显示从运行时间库中消耗的堆栈数量。

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
递增	lsinc	0	-
	luinc	0	-
	finc	12 (22) 注 1	-
递减	lsdec	0	-
	ludec	0	-
	fdec	12 (22) 注 1	-
符号取反	lsrev	0	-
	lurev	0	-
	frev	0	-
1 的补码	lscom	0	-
	lucom	0	-
逻辑非	lsnot	0	-
	lunot	0	-
	fnot	0	-
乘法	csmul	4 (1) 注 2	4 (1) 注 2
	cumul	4 (1) 注 2	4 (1) 注 2
	ismul	6 (5) 注 2	6 (5) 注 2
	iumul	6 (5) 注 2	6 (5) 注 2
	lsmul	6(7) 注 2	-
	lumul	6(7) 注 2	-
	fmul	8 (18) 注 1	-
除法	csdiv	8	8
	cudiv	4	4
	isdiv	8	12
	iudiv	4	6
	lsdiv	10	-
	ludiv	6	-
	fdiv	8 (18) 注 1	-

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
余数	csrem	8	8
	curem	4	4
	isrem	8	12
	iurem	4	6
	lsrem	10	-
	lurem	6	-
加法	lsadd	0	-
	luadd	0	-
	fadd	8 (18) 注 1	-
减法	lssub	0	-
	lusub	0	-
	fsub	8 (18) 注 1	-
向左移位	islsh	0	0
	iulsh	0	0
	lslsh	2	-
	lulsh	2	-
向右移位	isrsh	0	0
	iursh	0	0
	lsrsh	2	-
	lursh	2	-
比较	cscmp	0	2
	iscmp	0	2
	lscmp	2	-
	lucmp	2	-
	fcmp	4 (14) 注 1	-
按位与	lsband	0	-
	luband	0	-
按位或	lsbor	0	-
	lubor	0	-
按位异或	lsbxor	0	-
	lubxor	0	-
逻辑与	fand	0	-
逻辑或	for	0	-

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
浮点数的转换	ftols	4	-
	ftolu	4	-
转换为浮点数	lstof	12 (22) 注 1	-
	lutof	12 (22) 注 1	-
位的转换	btol	0	-
启动例程	cstart	2	2
函数的预处理和后处理	cprep	2 + n 注 3	-
	cdisp	0	-
	cprep2	自动变量大小 + 寄存器变量	-
	cdisp2	0	-
	nrcp2	-	0
	nrcp3	-	0
	krcp2	-	0
	krcp3	-	0
	nkrc3	-	0
	nrip2	-	0
	nrip3	-	0
	krip2	-	0
	krip3	-	0
	nkri31	-	0
	nkri32	-	0
	nrsave	-	8
	nrload	-	0
	krs02	-	2
	krs04	-	4
	krs04i	-	4
	krs06	-	6
	krs06i	-	6
	krs08	-	8
	krs08i	-	8
	krs10	-	10
	krs10i	-	10
	krs12	-	12

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
函数的预处理和后处理	krs12i	-	12
	krs14	-	14
	krs14i	-	14
	krs16	-	16
	krs16i	-	16
	krl02	-	0
	krl04	-	0
	krl04i	-	0
	krl06	-	0
	krl06i	-	0
	krl08	-	0
	krl08i	-	0
	krl10	-	0
	krl10i	-	0
	krl12	-	0
	krl12i	-	0
	krl14	-	0
	krl14i	-	0
	krl16	-	0
	krl16i	-	0
	hdwinit	0	0
BCD- 类型转换	4	4	4
	8	8	8
	4	4	4
	10	10	10
	8	8	8

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
辅助函数	mulu	4	4
	divuw	6	6
	clra0	0	0
	clra1	0	0
	clrax0	0	0
	clrax1	0	0
	clrbc0	0	0
	clrbc1	0	0
	cmpa0	0	0
	cmpa1	0	0
	cmpc0	0	0
	cmpax0	0	0
	cmpax1	0	0
	movca	0	0
	movac	0	0
	ctoi	0	0
	uctoi	0	0
	adjba	2	2
	adjbs	1	1
	addrde	0	0
	addrhl	0	0
	shl4	0	0
	shr4	0	0
	tabled	0	0
	tableh	0	0
	apdecd	0	0
	apdech	0	0
	apincd	0	0
	apinch	0	0
	deilo	0	0
	deist	0	0
	deiinc	0	0
	deidec	0	0
	hlilo	0	0

表 D-2 运行时间库堆栈消耗表

分类	函数名称	普通模式	静态模式
辅助函数	hlist	0	0
	hliinc	0	0
	hlidec	0	0
	dellab	2	-
	dell03	0	-
	della4	0	-
	delsab	0	-
	dels03	2	-
	hlllab	0	-
	hlll03	0	-
	hllla4	0	-
	hllsab	0	-
	hlls03	0	-
	hliadd	0	0
	hlisub	0	0
	hlicmp	0	0
	hliand	0	0
	hlior	0	0
	hlixor	0	0
	imule	10	10
	isdive	12	16
	iudive	8	10
	isreme	12	16
	iureme	8	10
	iadde	0	0
	isube	2	2
	iande	0	0
	iore	0	0
	ixore	0	0

注 1. 圆括号中的数值适用于操作异常发生时 (当适用了编译器自带的 matherr 函数时)。

注 2. 括号中的数值时为乘法 / 除法时使用的。

注 3. n 为需要保留的自动变量的安全空间大小。

附录 E　库响应中断的最长时间列表

按数序使用乘数库提供的中断禁用时间，在中断期间不破坏操作内容。

表 E-1 显示了在使用乘法时，为库禁用中断的最长时间。

在不使用乘数库提供的中断禁用期间，没有周期。

表 E-1 库的最大中断禁用时间 （时钟数）

分类	函数名称	模式支持	
		普通模式	静态模式
乘法	csmul	52	52
	cumul	52	52
	ismul	60	60
	iumul	60	60
	lsmul	80	-
	lumul	80	-

附录 F 索引

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