用户手册

NEC

CC78K0R 第 1 版

C 编译器

语言篇

目标设备

78K0R 系列

文档编号 U17837CA1V0UM00 (第 1 版)
出版日期 2007 年 12 月 CP(K)

© 日本电气电子株式会社 **2007**

日本印刷

[备忘录]

[备忘录]

- **本文档所刊登的内容有效期截至 2007 年 12 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

CC78K0R C 编译器（下文简称为该编译器）是在**美国信息系统国家标准草案— C 语言编程**（1998 年 12 月 7 日）中的**第 2 章 环境**和**第 3 章 语言**的基础上开发的。因此，使用 CC78K0R 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 78K0R 系列微控制器的应用产品。

CC78K0R C 编译器语言篇（本手册）是为了让那些使用 CC78K0R 编译器进行软件开发的用户能够正确了解 CC78K0R 编译器的基本功能及语言规格而编写的。

本手册不介绍如何操作 CC78K0R 编译器。因此，在您掌握了本手册的内容后，请阅读 **CC78K0R C 编译器 操作篇 (U17838E)**。

关于 78K0R 系列的体系结构，敬请参阅 78K0R 系列微控制器各产品的用户手册。

[目标设备]

可以使用 CC78K0R 编译器开发 78K0R 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]

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 CC78K0R 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（编译器指令）

详细介绍预处理指令的类型，以及如何使用各种预处理指令。

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开发目标设备的性能。

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 CC78K0R 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、库调用时最长中断禁止时间列表，及索引列表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 CC78K0R 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：
  CC78K0R 编译器的语言规格符合 **ANSI** 标准 **C**。因此，您可以从**第 11 章**开始，该章介绍了 CC78K0R 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 78K0R 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）

资料名		资料编号
CC78K0R C 编译器 V 1.00 版本	操作篇	U17838E
	语言篇	本文档
RA78K0 汇编程序包 V 1.00 版本	操作篇	U17836E
	汇编语言	U17835E
SM+ 系统模拟器	操作篇	U18010E
PM plus V6.20 版本		U17990E
ID78K0R-QB 集成调试器 V3.20 版本	操作篇	U17839E

[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 = **78K0R** 系列实时操作系统 **RX78K0R**

[约定]

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。

目录

前言......6
目录......9
插图列表......12
表格列表......13
第 1 章 概述......14
1.1 C语言与汇编语言......14
1.2 使用C编译器的程序开发过程......16
1.2.1 所需软件......16
1.2.2 产品开发流程......16
1.3 C源程序的基本结构......18
1.3.1 程序格式......18
1.4 该C编译器的最佳性能特性......21
1.5 该C编译器的特色......23
第 2 章 C语言的结构......27
2.1 字符集......28
2.1.1 字符集......28
2.1.2 多字节字符......28
2.1.3 转义字符序列......29
2.1.4 三字符序列......29
2.2 关键字......30
2.2.1 ANSI-C关键字......30
2.2.2 CC78K0R增加的关键字......31
2.3 标识符......32
2.3.1 标识符的作用范围......33
2.3.2 标识符的连接......34
2.3.3 标识符名称空间......34
2.3.4 对象的生存期......35
2.4 数据类型......36
2.4.1 基本类型......37
2.4.2 字符类型......41
2.4.3 不完全类型......41
2.4.4 派生类型......41
2.4.5 标量类型......42
2.4.6 兼容类型......42
2.4.7 复合类型......43
2.5 常量......44
2.5.1 浮点型常量......44
2.5.2 整型常量......45
2.5.3 枚举常量......46
2.5.4 字符常量......47
2.6 字符串文字......48
2.7 运算符......49
2.8 定界符......50
2.9 头文件名......51
2.10 注释......52
第 3 章 数据类型与存储类的声明......53
3.1 存储类声明符......54
3.2 类型声明符......55
3.2.1 结构体指定符与共用体指定符......57
3.2.2 枚举声明符......59
3.2.3 标记......60
3.3 类型修饰词......61
3.4 声明符......62
3.4.1 指针声明符......62
3.4.2 数组声明符......63
3.4.3 函数声明符（包括原型声明）......63
3.5 类型名......64
3.6 typedef 声明......65
3.7 初始化......67
3.7.1 具有静态存储生存期的对象的初始化......67
3.7.2 具有自动存储生存期的对象的初始化......67
3.7.3 字符数组的初始化......68

3.7.4 聚合或共用体型对象的初始化 69
第 4 章 类型转换 71
4.1 算术运算数 73
4.2 其他运算数 75
第 5 章 运算符与表达式 76
5.1 基本表达式 78
5.2 后缀运算符 79
5.3 单目运算符 86
5.4 类型转换运算符 93
5.5 算术运算符 95
5.6 移位运算符 101
5.7 关系运算符 104
5.8 按位逻辑运算符 111
5.9 逻辑运算符 115
5.10 条件运算符 118
5.11 赋值运算符 120
5.12 逗号运算符 123
5.13 常量表达式 125
第 6 章 C语言的控制结构 127
6.1 带标签的语句 129
6.2 复合语句或块 133
6.3 表达式语句和空语句 134
6.4 条件控制语句 135
6.5 循环语句 138
6.6 分支语句 142
第 7 章 结构体和共用体 147
7.1 结构体 147
7.2 共用体 151
第 8 章 外部定义 154
8.1 函数定义 155
8.2 外部对象定义 157
第 9 章 预处理指令（编译器指令） 158
9.1 条件编译 158
9.2 源文件包含指令 166
9.3 宏替换 170
9.4 行控制 175
9.5 #error预处理指令 176
9.6 #pragma指令 177
9.7 空指令 178
9.8 预定义的宏名 179
第 10 章 库函数 181
10.1 函数之间的接口 181
10.1.1 参数 181
10.1.2 返回值 182
10.1.3 保存单独库（Individual Libraries）所用的寄存器 183
10.2 头文件 185
10.3 可重入性（re-entrantability）（仅适用于正常模式） 192
10.4 标准库函数 193
10.4.1 为参数和返回值使用优化库 198
10.5 字符和字符串函数 199
10.6 程序控制函数 204
10.7 特殊函数 206
10.8 I/O函数 210
10.9 应用函数 231
10.10 字符串/存储器函数 254
10.11 数学函数 272
10.12 诊断函数 319
10.13 更新启动例程及库函数所需的批处理文件 321
10.13.1 使用批处理文件 322
第 11 章 扩展函数 324
11.1 宏名称 324
11.2 关键字 325
11.3 存储器 327
11.4 #pragma指令 329
11.5 如何使用扩展函数 331
(1) callt 函数（callt/_ _callt） 334

(2) 寄存器变量（register） 337
(3) 如何使用saddr区域（sreg/_ _sreg） 339
(4) 如何使用sfr区域（sfr） 344
(6) norec 函数（norec） 346
(7) bit型变量，boolean型变量（bit/ Boolean/_ _boolean） 349
(8) ASM 语句（#asm，#endasm/ _ _asm） 353
(9) Kanji（2-字节字符）（/* kanji */, //kanji） 355
(10) 中断函数（#pragma vect/#pragma interrupt） 358
(11) 中断函数修饰词（__ interrupt, __ interrupt_brk） 365
(12) 中断函数（#pragma DI，#pragma EI） 367
(13) CPU控制指令（#pragma HALT/STOP/BRK/NOP） 370
(14) 位域声明 372
(15) 改变编译器输出区段名称（#pragma section…） 379
(16) 二进制常量（二进制常量 0bxxx） 389
(17) 模块名称改变函数（#pragma name） 391
(18) 循环移位函数（#pragma rot） 392
(19) 乘法函数（#pragma mul） 394
(20) 除法函数（#pragma div） 396
(21) 数据插入函数（#pragma opc） 398
(22) 实时操作系统（RTOS）的中断函数（#pragma rtos_interrupt ...） 400
(23) 实时操作系统（RTOS）的中断函数修饰词（__rtos_interrupt ...） 403
(24) 实时操作系统（RTOS）的任务函数（#pragma rtos_task） 405
(25) Flash区域分配方法（-ZF） 407
(26) Flash区域跳转表（#pragma ext_table） 408
(27) 从boot区域到flash区域的函数调用（#pragma ext_func） 411
(28) 固件ROM函数（__flash） 414
(29) 参数/返回值的int展开限制方法（-ZB） 415
(30) 内存操控函数（#pragma inline） 417
(31) 绝对地址分配规格（__directmap） 419
(32) near/far区域规格 423
(33) 存储器模式规格 427
11.6 C源代码的修改 431
11.7 函数调用接口 432
11.7.1 返回值 433
11.7.2 普通函数调用接口 434
11.7.3 norec函数调用接口 439
第 12 章 汇编程序的引用 440
12.1 访问参数/自动变量 440
12.2 返回值的存储 441
12.3 在C语言程序中调用汇编语言程序 442
12.3.1 C 语言函数调用过程 442
12.3.2 汇编语言程序的数据保存和调用返回 443
12.4 由汇编语言程序调用C语言程序 445
12.4.1 由汇编语言程序调用C语言函数 445
12.5 引用其它语言定义的变量 447
12.5.1 引用其它语言中定义的变量 447
12.5.2 由C语言程序引用汇编语言定义的变量 448
12.6 注意事项 449
第 13 章 编译器的有效应用 450
13.1 高效编码 450
附录A 用于saddr区域的标签列表 454
附录B 区段名称列表 456
B.1 区段名称列表 457
B.1.1 程序区域和数据区域 457
B.1.2 flash存储器区域 458
B.2 区段位置 459
B.3 C 源程序示例 460
B.4 输出汇编模块示例 461
附录C 运行时刻库列表 470
附录D 库占用的堆栈列表 474
附录E 库响应中断的最长时间列表 485
附录F 索引 486

插图列表

插图编号　插图标题 .... 页码

图 1-1　编译流程 15
图 1-2　使用CC78K0R编译器进行程序开发的流程 17
图 2-1　类型的划分 36
图 4-1　通常算术类型转换 74
图 6-1　条件控制语句的控制流程 135
图 6-2　循环语句的控制流程 138
图 6-3　分支语句的控制流程 142
图 11-1 存储空间的利用 328

表格列表

表格编号 表格标题..........页码

表 1-1 该C编译器的性能指标最大值..........21
表 1-2 改善执行速度的方法..........23
表 1-3 扩展函数列表..........23
表 2-1 字符集中可以使用的字符列表..........28
表 2-2 转义字符序列列表..........29
表 2-3 三字符序列列表..........29
表 2-4 ANSI-C关键字列表..........30
表 2-5 CC78K0R追加的关键词列表..........31
表 2-6 标识符列表..........32
表 2-7 基本数据类型列表..........45
表 2-8 整型常量和可表示的类型..........46
表 2-9 运算符列表..........49
表 3-1 存储类声明符..........54
表 3-2 类型声明符..........56
表 3-3 类型名称举例..........64
表 4-1 类型转换列表..........71
表 5-1 运算符计算优先级..........77
表 5-2 除号/求余运算结果符号..........95
表 5-3 移位运算..........101
表 8-1 外部对象定义示例..........157
表 9-1 宏名称列表..........179
表 10-1 第一参数传递列表..........182
表 10-2 返回值存储列表..........182
表 10-3 标准库函数列表..........193
表 10-4 更新库函数的批处理文件..........321
表 11-1 添加的关键字列表..........325
表 11-2 #pragma指令列表..........329
表 11-3 类型调整细节（从整形和短整形改变为char型）..........434
表B-1 区段位置..........459
表C-1 运行时刻库列表..........470
表 D-1 标准库占用堆栈列表..........474
表 D-2 运行时刻库堆栈占用空间列表..........480
表 E-1 为库禁止中断的最长时间（clock数量）..........485

第 1 章 概述

本章介绍 CC78K0R 在系统开发中的作用，也让读者对该 C 编译器的功能概述有所了解。

CC78K0R 系列 C 编译器是一个语言处理程序，不论编写的 C 语言源程序符合 78K0R 系列规格，或是符合 ANSI-C 规格，CC78K0R 系列 C 编译器都会将 C 语言转换为机器语言。使用 CC78K0R 系列 C 编译器，可以得到适用于 78K0R 系列的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。程序和数据必须由人（程序员）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，只不过是被称为机器语言的二进制数组合而已。

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。正是由于这种对应关系，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，这意味着程序员必须对计算机的每一步操作都给出具体指令。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，而且程序员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，它使得程序员在编程时无需考虑计算机的体系结构。

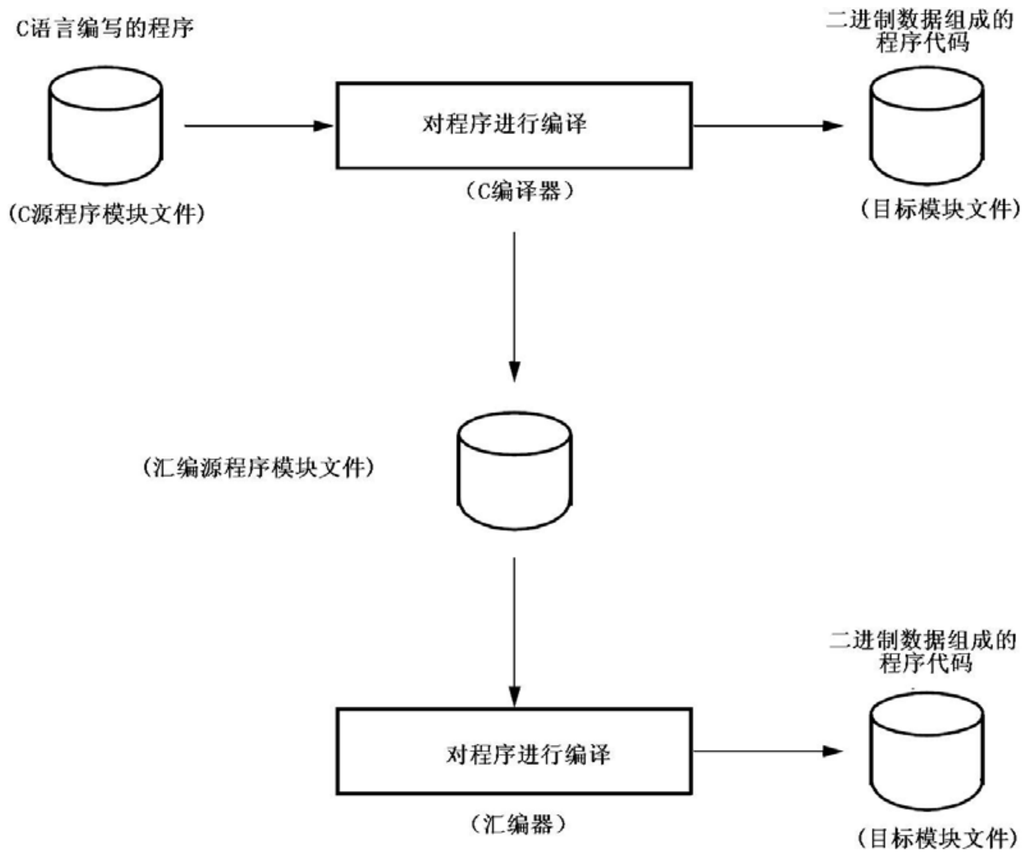
与汇编语言程序相比，可以认为 C 语言编写开发的程序逻辑结构更加易于理解。

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

C 语言的特点是便于人们理解。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被称为 C 编译器。

C 编译器的将 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，如果程序员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。下图所示为 C 编译器的转换流程。

图 1-1 编译流程



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，需要连接器来将编译器生成的目标模块文件组织起来，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 所需软件

和该 C 编译器有关的软件如下。

- 编辑器　　　　　　　　　　　用于创建源程序模块文件
- RA78K0R 汇编程序包

程序名称	功能
汇编器	用于将汇编语言转换为机器语言
连接器	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
PM plus	集成化开发环境平台

- 调试器（用于 78K0R）　　　　用于调试 C 源程序模块文件

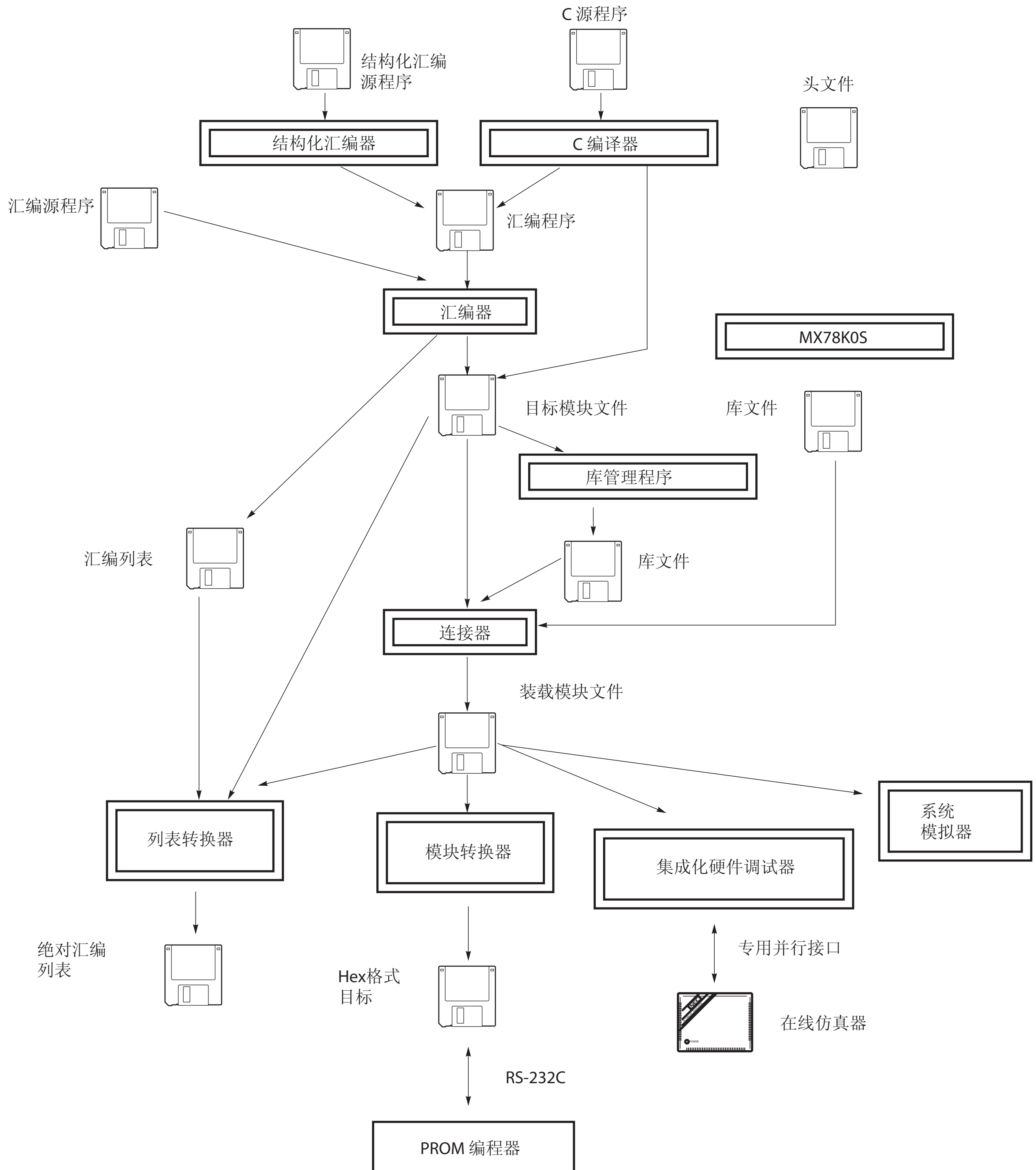
1.2.2 产品开发流程

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 源程序模块进行翻译转换。
<4>将常用的模块注册到库中。
<5>连接相应的目标模块文件。
<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 CC78K0R 编译器进行程序开发的流程

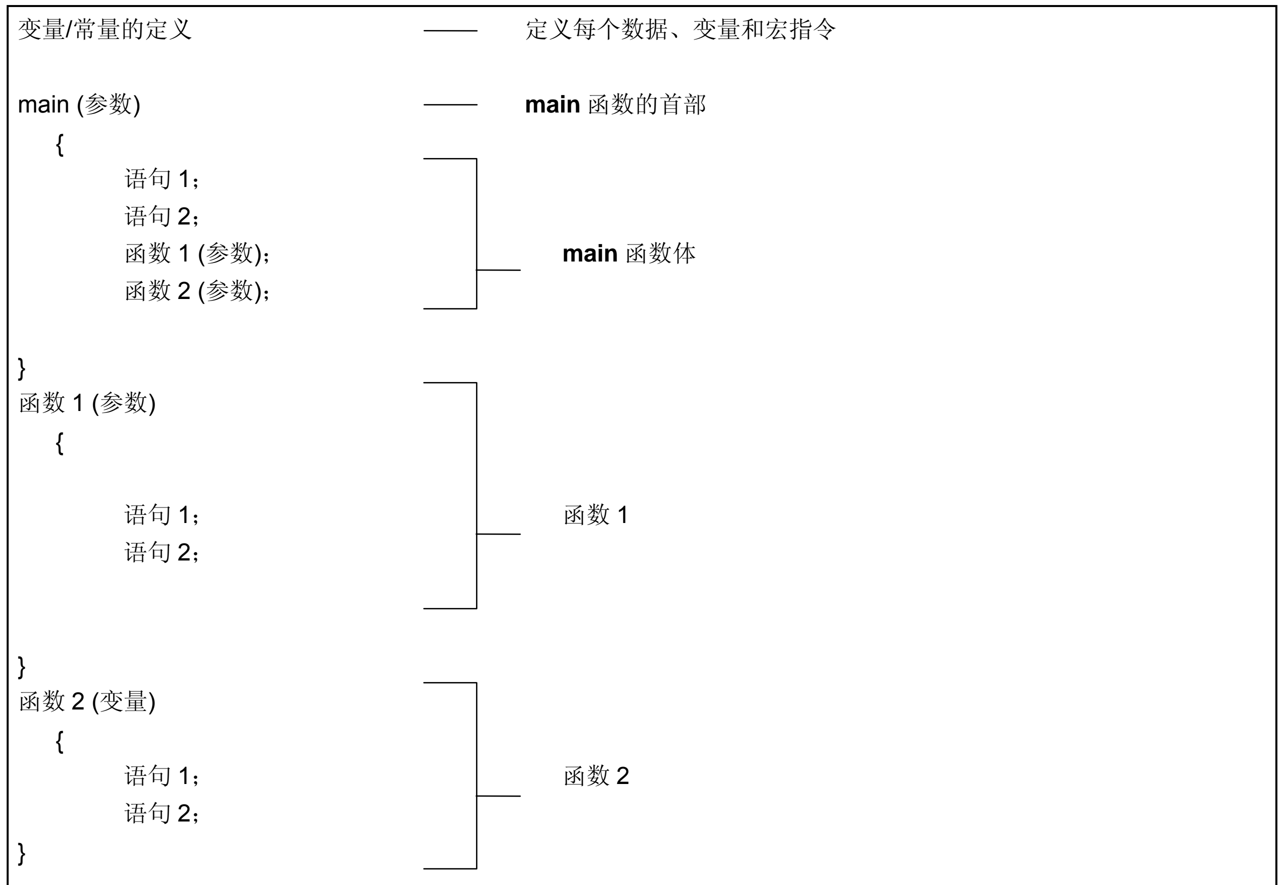


1.3 C源程序的基本结构

1.3.1 程序格式

C 程序是函数的集合。必须创建这些函数，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。



一个实际的 C 源程序示例。

```
#define TRUE 1                          /* #define xx xx： 预处理程序指令（定义宏）*/
#define FALSE 0                         /* #define xx xx： 预处理程序指令（定义宏） */
#define SIZE 200                        /* #define xx xx： 预处理程序指令（定义宏） */
void printf(char*, int);                /* xx xx ( xx , xx )： 函数原型声明符*/
void putchar( char);                    /* xx xx ( xx )：  函数原型声明符*/
char mark [ SIZE + 1 ] ;                /* char xx： 类型声明符，外部定义*/
                                        /* xx [ xx ]： 操作符*/
main ( )
{
        int i , prime , k , count ;     /* int xx： 类型声明符*/
        count = 0 ;                     /* xx = xx： 操作符 */
        for ( i = 0 ; i <= SIZE ; i ++ )  /* for ( xx ; xx ; xx ) xx ; ： 控制结构*/
                mark [ i ] = TRUE ;
        for ( i = 0 ; i <= SIZE ; i ++ ) {
                if ( mark [ i ] )
                {
                        prime = i + i + 3 ;          /* xx = xx + xx + xx： 操作符*/
                        printf ( "%6d" , prime ) ;   /* xx ( xx ) ; ： 操作符*/
                        count ++ ;
                                        /* if ( xx ) xx ; ： 控制结构*/
        if ( ( count%8 ) == 0 ) putchar ( '\n' ) ;
        for ( k = i + prime ; k <= SIZE ; k += prime )
                mark [ k ] = FALSE ;
                }
        }
        printf ( "\n%d primes found." , count ) ;   /* xx ( xx ) ; ： 操作符*/
}

void printf ( char *s , int i )
{
                int j ;
                char *ss ;
                j = i ;
                ss = s ;
}
void putchar ( char c )
{
                char d ;
                d = c ;
}
```

<1> 类型与存储类的声明

标识符的数据类型与存储类表明声明了对应的数据目标。具体细节敬请参阅第 3 章 类型与存储类的声明。

<2> 操作符与表达式

它们是指示编译器进行运算的语句，比如算术运算、逻辑运算或赋值运算。具体细节敬请参阅第 5 章 运算符与表达式。

<3> 控制结构

这是用来控制程序流程的语句。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。具体细节敬请参阅第 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享相同的内存时，可以定义为一个共用体。具体细节敬请参阅第 7 章 结构体与共用体。

<5> 外部定义

声明一个函数或外部目标。当 C 语言源程序被划分为多个独立的特殊用途或特征动作时，函数就是一个元素，C 源程序就是这些函数的集合。 具体细节敬请参阅第 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。具体细节敬请参阅第 9 章 预处理指令（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 该C编译器的最佳性能特性

在进行程序开发工作之前，请牢记在以下**表 1-1** 中总结的要点（限制值或最大保证值）。

表 1-1 该 C 编译器的性能指标最大值

项目	限制值/最大保证值
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	1024 个符号[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 个符号[注 1]
每个源程序模块文件的宏的数量	10000 个宏
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	65535 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 30000 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	和存储器模型有关[注 2]
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 使用该 C 编译器的最低性能指标

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
宏的嵌套层数	200
由 -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。

2. 最大值如下所示，由选择的存储器模式决定。

存储器模式	最大值
小型模式（Small model）	代码部分：64KB，数据部分：64KB；总共 128KB
中型模式（Medium model）	代码部分：1 MB，数据部分：64KB
紧凑模式（Compact model）	代码部分：64KB，数据部分：1 MB
大型模式（Large model）	代码部分：1 MB，数据部分：1 MB

1.5 该C编译器的特色

该 C 编译器支持用于指导 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 78K0R 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。

表 1-2 改善执行速度的方法

方法	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**
可以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	**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
可以按位访问 **saddr** 或 **sfr** 区域	**bit** 型变量，**boolean/_ _boolean** 型变量
可以用**无符号字符**型指定位域	位域声明
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
特定的数据和指令可以直接嵌入到代码区域中	数据插入函数
memcpy 和 **memset** 直接内联扩展并输出	存储器操控函数

下面概括介绍该编译器的扩展函数。关于每个扩展函数的详细情况，请参阅第 11 章 扩展函数。

表 1-3 扩展函数列表

扩展函数	概述
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。这使得目标代码比使用通常的调用指令 call 的目标代码要短。

扩展函数	概述
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。
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。
sfr 区域（sfr）	通过声明使用 sfr 名称，可以在 C 源文件中对 sfr 区域进行操作。
norec 函数（norec）	被声明为 **norec/_ _leaf** 的函数不输出代码的预处理和后处理（堆栈帧信息）。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。详细情况，请参阅第 11.5 (6)节 norec 函数。
位变量与布尔型变量（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。
ASM 语句（#asm #endasm / __asm）	用户编写的汇编源程序可以嵌入到该 C 编译器输出的汇编源文件中。
中断函数（#pragma vect / #pragma interrupt）	预处理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。
中断函数修饰词（__interrupt, __interrupt_brk）	该修饰词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
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。
CPU 控制指令（#pragma HALT / STOP / BRK / NOP）	以下指令都要嵌入到目标代码中： halt 指令 stop 指令 brk 指令 nop 指令
位域声明	将位域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。

扩展函数	概述
更改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。
二进制常量描述函数（Binary constant 0bxxx）	可以在 C 源代码中描述二进制常量。
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。
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。
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。
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。
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。
实时操作系统（RTOS）的中断处理程序（#pragma rtos_interrupt ...）	可以描述 RX78K0R 的中断处理程序。可以用#pragma 指令来设置向量（设置中断请求名称，处理程序的函数名称，堆栈切换）。
实时操作系统（RTOS）的中断处理程序修饰词（__rtos_interrupt）	该修饰词允许描述中断处理程序，为 RX78K0R 在另外的文件中设置向量。
实时操作系统（RTOS）的任务函数（#pragma rtos_task）	用#pragma 指令指定的函数被解释为 RX78K0R 的任务。这样描述实时操作系统（RTOS）的任务函数可以用 C 语言实现更高的代码效率
Flash 区域分配方法（-ZF）	通过在编译时指定-ZF 选项，可以把程序分配在 flash 区域。或者程序和创建在 boot 区域未指定-ZF 选项的目标结合使用。
Flash 区域跳转表（#pragma ext_table）	启动例程，终端函数分配到 flash 区域，boot 区域发起的函数调用
从 boot 区域到 flash 区域的函数调用（#pragma ext_func）	可以通过#pragma 指令来指定函数名称和 flash 区域的 ID 值，可以进行从 boot 区域到 flash 区域的函数调用。
固件 ROM 函数（__flash）	在接口库的原形声明时，将__flash 属性添加到首地址，就可以在 C 源程序中对固件 ROM 函数进行操控。
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。

扩展函数	概述
内存操控函数（#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。
绝对地址分配规格（__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。
near/far 区域规格	在声明函数或变量时，添加__near 或__far 类型修饰符，该函数或变量的位置可以单独指定。
存储器模式规格	在编译时指定-ms,-mm,-mc 或-ml 选项，函数或变量的位置可以通过指定存储器模式来指定。

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
字符串文字	运算符	定界符
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 ;                               /* { } : 定界符            */
        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同时使用。

所有全字符和半角片假名（包括半角的标点符号）都可以出现在注释中。

表 2-1 字符集中可以使用的字符列表

26 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 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

10 个数字

0 1 2 3 4 5 6 7 8 9

29 图形字符

! “ # % & ‘ ( ) * + , - . / :
; < = > ? [\ ] ^ — { | } ~

以及用于指示空格、横向制表、垂直制表、换页等的非打印控制字符。

2.1.2 多字节字符

注释中只能出现转义日本工业标准编码（shift JIS）和 EUC 编码格式。

在注释之外的位置，只能使用 0x7F 之内的 ASCII 字符。

在注释之外的任何位置，都不能出现全字符或半角片假名（包括半角的标点符号）。

2.1.3 转义字符序列

非图形的转义字符序列可以用来表示控制字符，比如提示报警或换页符等。每个转义字符序列都由 \ 符号和一个字母字符组成。

由转义字符序列表示的非图形字符如下表所示。

表 2-2 转义字符序列列表

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	09H
\v	垂直制表	0BH

2.1.4 三字符序列

当源文件中包含下表中左列的三字符时（称为“三字符”序列），这些列出的三字符将被转换为右列中的单个字符。

当指定了编译选项**-ZA**（该选项禁止不符合 ANSI 规格的函数，并使能一部分 ANSI 规格的函数）时，可以使用三字符序列。

表 2-3 三字符序列列表

<table>
<tr><th>三字符序列</th><th>含义</th></tr>
<tr><td>??=</td><td>#</td></tr>
<tr><td>??(</td><td>[</td></tr>
<tr><td>??/</td><td>\</td></tr>
<tr><td>??)</td><td>]</td></tr>
<tr><td>??’</td><td>^</td></tr>
<tr><td>??<</td><td>{</td></tr>
<tr><td>??!</td><td>|</td></tr>
<tr><td>??></td><td>}</td></tr>
<tr><td>??-</td><td>~</td></tr>
</table>

2.2 关键字

2.2.1 ANSI-C关键字

以下标记在 CC78K0R 编译器中用作关键字，因此不可用作标签或变量名。

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 CC78K0R增加的关键字

在 CC78K0R 编译器中，增加了以下标记作为关键字，来实施其扩展的功能。（当包含大写字母时，此标记将不被视为关键字）

这些标记不可用作标签或变量名，**ANSI C** 对这些标记不兼容。

不是以"_ _"开始的关键字可以通过指定（**-ZA**）选项使其失效，（**-ZA**）选项仅允许 **ANSI-C** 语言规格。

callf, _ _callf、noauto、__banked、non__banked、__BANK0-15、__pascal、__temp 及__mxcall 也被当作是关键字，是为了与 CC78K0 保持兼容。

表 2-5 CC78K0R 追加的关键词列表

关键词	用法
_ _callt/callt	声明 **callt** 函数
_ _callf/callf	声明 **callf** 函数
_ _sreg/sreg	声明 **sreg** 变量
noauto	声明 **noauto** 函数
_ _leaf/norec	声明 **norec** 函数
bit	声明 **位** 型变量
_ _boolean/boolean	声明 **布尔** 型变量
_ _interrupt	硬件中断函数
_ _interrupt_brk	软件中断函数
_ _banked, _ _non_banked	Bank 接口[注 1]
__BANK0-15	常量地址上的 bank 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
_ _flashf	**_ _flashf** 函数[注 2]
_ _directmap	绝对地址分配规格
_ _temp	临时变量
__near, __far	存储器分配区域规格
_ _mxcall	**_ _mxcall** 函数[注 3]

注　1．为函数信息文件保留的关键词。不要在 C 源程序中使用该关键词。

2．CC78K0R 保留的关键词。用户不能使用该关键字。

3．用于与 MX 接口的保留关键字。用户不能使用该关键字。

2.3 标识符

标识符是用于以下变量的名称：

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 结构体成员、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合而成。
以下字符可以用作标识符：
标识符的最大长度没有限制。不过，在本编译器中，只能识别前 249 个字符。

_（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 标识符的作用范围

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 : 文件作用域*/
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所有位置。

具有函数作用域的标识符可以从指定的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）单元内的整体。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。具有文件作用域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声明的函数从头至尾的范围。在函数原型的参数列表中声明的所有标识符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，多个文件中关联引用的相同的标识符就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

外部连接是指作为翻译（编译）单元进行连接的标识符，标识符构成了整个程序，并组成一系列的库的集合。

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** 却未指定存储类型的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

内部连接是指在一个翻译（编译）单元内将要进行连接的标识符。

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，本身就是一个实体。

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
- 在块内未指定存储类型 **extern** 的对象的标识符

2.3.3 标识符名称空间

所有的标识符可以归入以下的“名称空间”。

名称空间	解释
标签名	由一个标签声明来标识
结构体、共用体或枚举的标记名	由关键字 **struct**,**union** 或 **enum** 来标识
结构体或共用体的成员名	由点（**.**）运算符或箭头（**->**）运算符来标识。
普通标识符（上述情况以外的标识符）	声明为普通标识符或枚举型常量

2.3.4 对象的生存期

每个对象都有一个决定其生存期的生存期（它在内存中存在的时间）。

生存期分为以下两类：静态存储和自动存储。

(1)静态存储生存期

在执行具有静态存储的目标程序前，为存储的对象和值保留一个区域，并进行初始化。

在整个程序的执行过程中都持续存在并保持最后存储的值。

具有静态存储生命期的对象如下所示。

- 具有外部连接的对象
- 具有内部连接的对象
- 由存储类别修饰词 **static** 声明的对象

(2)自动存储生存期

对于具有自动生存期的对象，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处要进行初始化。在这种情况下，如果有任何对象进入程序块的方式是从外部跳转到程序块内的标签，则该对象不进行初始化。

对于具有自动存储周期的对象，当声明的程序块执行完毕时，保留的区域将无法保证。

具有自动存储生命期的对象如下所示。

- 无连接的对象
- 在程序块内声明，但没有加存储类修饰词 **static** 的对象

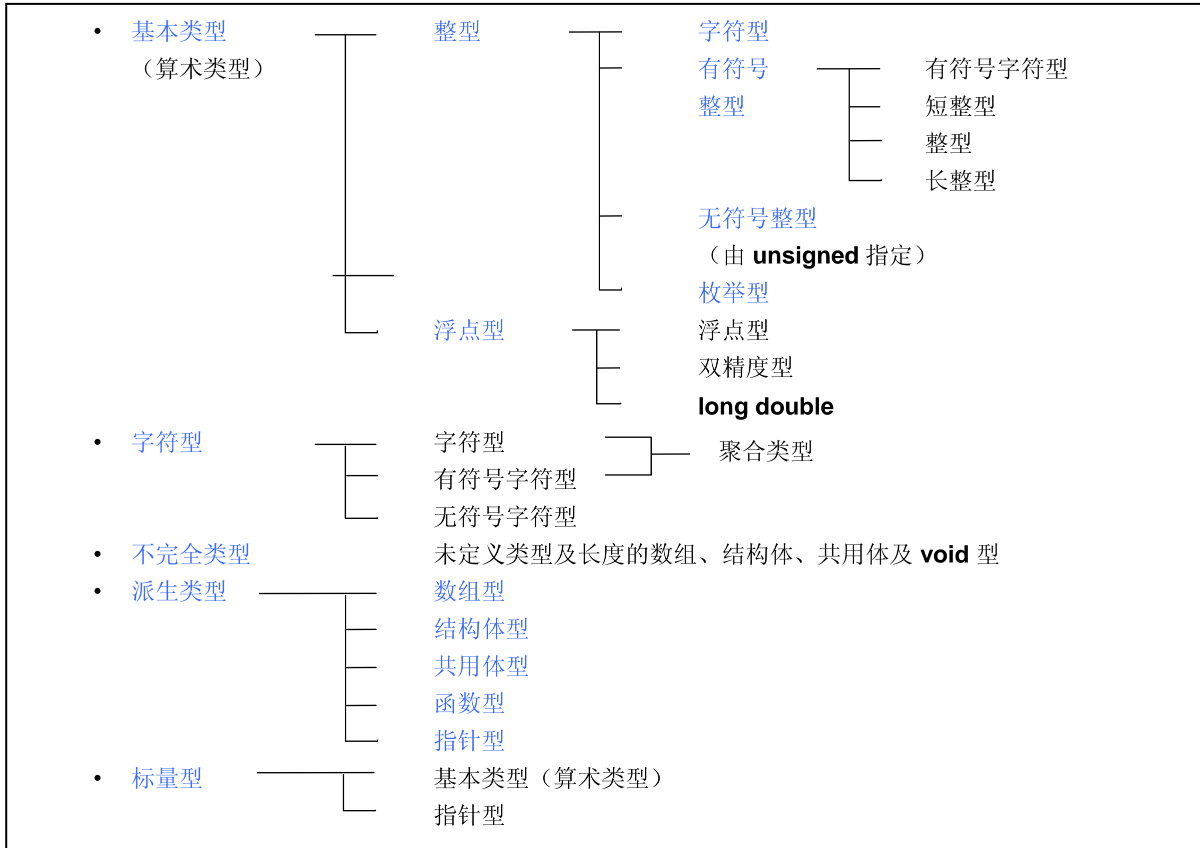
2.4 数据类型

数据类型决定存储在各个对象中的值的含义，
根据变量声明的不同，数据类型分为以下 3 类。

- 对象类型 .... 表示具有大小信息的对象的类型
- 函数类型 .... 表示函数的类型
- 不完全类型 .... 表示不具有大小信息的对象的类型

这些类型的分类如下所示。

图 2-1 类型的划分



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四类。每一类都有二进制数 0 和 1 表示的值。

- 字符型（**char**）
- 有符号字符型（**signed char**）
- 无符号字符型（**unsigned char**）
- 枚举型（**enumeration**）

(a) char 型

字符型具有足够的长度来存储执行字符集中的任何基本字符。存储在 **char** 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，溢出的部分将被忽略。

(b) 有符号整型

有符号整型将进一步分为以下四种类型：

- 有符号字符型（**signed char**）
- 短整形（**short int**）
- 整形（**int**）
- 长整形（**long int**）

声明为 **signed char** 型的对象，其存储区与无修饰词的 **char** 型大小相同。

无修饰词的 **int** 型对象的大小与执行环境的 CPU 体系结构相适应。每个有符号整型数据都有其对应的无符号整型数据。两者的存储区域大小相同。大于零的有符号整型数据是无符号整型数据的一个子集。

(c) 无符号整型

无符号整型数据用关键字 **unsigned** 来定义。

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，如果无符号整型数据参与计算的结果是一个不能用整型表示的值，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，或是已知的整型常数列表。

枚举型由一组或多组枚举数据组成。

(2) 浮点型

浮点型又细分为三类：

- **float**
- **double**
- **long double**

在该编译器中，将 **double** 、**long double** 型及 **float** 型都作为单精度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，**float** 、 **double** 、 **long double** 型的值具有相同的范围。

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

备注 1　**signed** 关键字可以省略。不过，对于 **char** 型，具体将被作为 **signed char** 还是 **unsigned char** 来处理，要根据编译时的条件决定。

备注 2　**short int** 数据和 **int** 数据将作为值范围相同、但具有不同类型的数据来处理。

备注 3　**unsigned short int** 数据和 **unsigned int** 数据将被作为值范围相同、但具有不同类型的数据来进行处理。

备注 4　**float**，**double** 和 **long double** 数据将作为值范围相同、但具有不同类型的数据来处理。

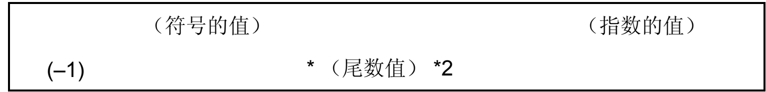
备注 5　**float**，**double** 和 **long double** 类型的值范围都是绝对的。

[浮点数（float 型）规格]

(a)　　格式

浮点数的格式如下所示。

这种格式的数值如下。



<table>
<tr><td>s</td><td>符号（1 位）
0 表示正数，1 表示负数。</td></tr>
<tr><td>e</td><td>指数以底数为 2 的指数表示一个 1 字节的整数（在负数情况下，用 2 的补码表示），并在增加 7FH 偏移量后使用。这些关系如下表所示。
<table>
<tr><th>指数（十六进制）</th><th>指数的值</th></tr>
<tr><td>FE</td><td>127</td></tr>
<tr><td>:</td><td>:</td></tr>
<tr><td>81</td><td>2</td></tr>
<tr><td>80</td><td>1</td></tr>
<tr><td>7F</td><td>0</td></tr>
<tr><td>7E</td><td>–1</td></tr>
<tr><td>:</td><td>:</td></tr>
<tr><td>01</td><td>–126</td></tr>
</table>
</td></tr>
<tr><td>m</td><td>尾数（23 位）
尾数以绝对值形式表示，其第 22 位至 0 位相当于二进制数的第 1 位至 23 位。除非浮点值是 0，否则指数值总是进行调整，使尾数在 1~2 的范围内（标准化）。结果 1 的位置（比如值 1）始终是 1，并在此格式下忽略表示。</td></tr>
</table>

(b)　　0 的表示

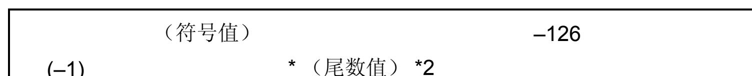
当指数为 0，尾数为 0 时， ±0 的表示如下。

(c) 无穷大的表示

当指数为 FFH，尾数为 0 时， ±∞表示如下。

(d) 非标准化的值

当指数为 0，尾数不为 0 时，非标准化的值表示如下。



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。

(e) 非数值（**NaN**）表示

当指数为 FFH，尾数不为 0 时，无论符号是多少，均表示为 NaN。

(f) 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都可以表示，则舍入至一个偶数（二进制最低位为 0 的数）。

(g) 运算异常

有 5 种运算异常，如下表所示。

异常	返回值
下溢	非标准化数
不准确	±0
上溢	±∞
除 0	±∞
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 字符类型

字符数据类型包括以下三种。

- 字符型（**char**）
- 有符号字符型（**signed char**）
- 无符号字符型（**unsigned char**）

2.4.3 不完全类型

不完全数据类型包括以下四种。

- 未指定类型及大小的数组
- 结构体
- 共用体
- **void** 型

2.4.4 派生类型

派生类型分为以下三种。

- 数组类型
- 结构体型
- 共用体型
- 函数型
- 指针型

(1) 数组类型

数组类型连续分配一组成员对象，成员对象此处被称为元素类型，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成员对象，每个对象的大小可以不同。指定各个成员对象需要给定一个名称。

(3) 共用体举型

共用体类型是一组共用内存的成员对象。这些成员对象在大小和名称上均可以不同，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引用类型的函数型对象类型中创建的，也可以由不完全类型创建的。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。

由被引用的类型 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：

- **char** 型
- 有符号整型（Signed integral type）
- 无符号整型（Unsigned integral type）
- 枚举型（Enumeration type）
- 浮点型（Floating-point type）
- 指针型（Pointer type）

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带有参数类型列表（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。
- 如果两个类型都有参数类型列表（即有原型的函数），则复合类型就具有如下原型：包括从这两个原型中提取的所有信息。

[复合原型示例]

假设有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变量，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（Floating-point constants）
- 整型常量 （Integer constants）
- 枚举型常量（Enumeration constants）
- 字符型常量（Character constants）

2.5.1 浮点型常量

一个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 （**float**） l/L （**long double**） 备注　如果省略（**double**）

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包括在效数字。而且，小数点还是指数部分不可或缺（例如：1.23F，2e3）。

2.5.2 整型常量

整型常量以一个数字开始，不包括小数点或指数部分。

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。

无符号后缀
　　u U
长整型后缀
　　l L

共有以下三种整型常量。

- 十进制常量：
- 八进制常量：
- 十六进制常量：

表 2-7 基本数据类型列表

十进制常量	十进制常量是一个以 10 为底数（基数）的整数。 [规格] 该数必须以非 0 的数值开始，其后跟随的数字可以是从 0 至 9 的任何数字（例如：56U）。 十进制数以一个非 0 数字开始 十进制数字 = 1 2 3 4 5 6 7 8 9
八进制常量	八进制常量是一个以 8 为底数（基数）的整数。 [规格] 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）。 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	十六进制常量是一个以 16 为底数（基数）的整数。 [规格] 该数必须以 0x 或 0X 开始，其后跟随的数字可以是从 0 至 9、a 至 f 或 A 至 F 的数的任何数字，a 至 f 或 A 至 F 表示从 10 至 15 的数（例如：0xF3）。 整型前缀 0x 或 0X + 十六进制数字 十六进制 数字 = 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

这个类型的整型常量被视为首选的“可表示类型”，如下所示。

在该编译器中，根据编译的条件（选项），无下标的常量类型可以更改为 **char** 或 **unsigned char**。

表 2-8 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	**int，long int，unsigned long int**
无后缀八进制、十六进制数	**int，unsigned int，long int，unsigned long int**
后缀 u 或 U	**unsigned int，unsigned long int**
后缀 l 或 L	**long int，unsigned long int**
后缀 u 或 U，l 或 L	**unsigned long int**

2.5.3 枚举常量

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

枚举型（enum）可以是以下列出的三种类型的任何一种，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（**signed char**）
- 无符号字符型（**unsigned char**）
- 有符号整型　（**signed int**）

它的描述方法是“**enum** 枚举型{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。当没有“= 1”的标识时，每个常量的值分别为 0，1，2，3，4， 5。

2.5.4 字符常量

字符常量是括在单引号对中的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（¥ 或\）和换行符（¥n）。要表示这些字符，需要使用由转义字符序列。有以下三种转义字符序列。

- 简单转义字符序列：　\’　\”　\?　\¥
  \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]
  （示例：\012，\0 注 1）
- 十六进制转义字符序列：　\x 十六进制数
  （示例：\xFF 注 2）

注 1.　空字符

注 2.　在该编译器中，**\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 字符串文字

字符串文字是放在双引号对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 转义序列\”来表示。
数组元素可以是 **char** 型的字符串文字，并由给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 2-9 运算符列表

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

[ ]、( )和?:运算符必须成对使用。

一个表达式必须在方括号“[ ]”、圆括号“( )”或“?”和“:”之间描述。

和 **##**运算符只能用于在预处理指令中定义宏。（有关介绍，请参阅第 **5** 章 运算符与表达式。）

2.8 定界符

定界符是一个具有独立的句法或意义的符号。但是它绝不会产生一个值。
在 C 语言中使用的定界符如下所示。

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

一个表达式声明或语句可以在方括号"[ ]"、圆括号"( )"或花括号"{ }"中进行描述。这些定界符必须如上述所示的那样成对使用。 定界符 ##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** 指令的详细情况，请参阅 **9.2 节 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 源程序模块中提供信息的语句。它以“/*”开始，以“*/”结束。也可以使用 **-ZP** 选项将“//”后至换行符之间的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。通过声明可以为标识符命名的对象或函数保留一个适当的存储区，被称为“定义”。

下面是一个声明的示例。

<数据类型与存储类的声明示例>

```
#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，之间由逗号分开。每个声明符都可以有附加的类型信息或一个初始化符，或者二者兼有。

如果一个对象的标识符声明它无连接，则该对象的类型必须是恰当的（对象具有大小相关的信息），位置处于声明符或初始化声明符（如果有的话）尾部。

3.1 存储类声明符

存储类声明符指定对象的存储类别。它说明对象的作用域和对象具有的值的存储位置。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 3-1 存储类声明符

声明符类型	意义
typedef	**typedef** 声明符声明一个指定的类型的替代名。关于 **typedef** 声明符的详细情况，请参阅第 3.6 节 typedef 声明符。
extern	**extern** 声明符说明（告诉编译器）在紧随该声明符之后的这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生存期。对于具有静态存储生存期的对象，在程序执行前就会为其保留一个存储区，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，并保持其最后存储的值。
auto	**auto** 声明符说明对象具有自动存储生存期。 对于具有自动存储生存期的对象，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。如果对象是通过调转到程序块内的一个标签的方式进入程序块时，该对象将不进行初始化。 当声明语句所在的程序块执行完毕时，为自动存储生存期的对象所保留的存储区将无法保证。
register	**register** 声明符将一个对象指定分配给 CPU 的寄存器。对于该 C 编译器，它将被分配到 CPU 的寄存器或 **saddr** 区域。关于寄存器变量的详细情况，请参阅第 11 章 扩展函数。

3.2 类型声明符

类型声明符指定（或表示）一个对象的类型。有以下的类型声明符可供使用。

- void(空类型)
- char(字符型)
- short(短整型)
- int(整型)
- long(长整型)
- float(浮点型)
- double(双精度型)
- long double(长精度型)
- signed(有符号型)
- unsigned(无符号型)
- Structure or union specifier(结构体或共用体指定符)
- Enumeration specifier(枚举声明符)
- typedef 名称

在该 C 编译器中，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 CC78K0R 编译器中可以表示的极限值（圆括号中的值）加以说明。对于浮点运算，由于本编译器只支持 IEEE Std 754-1985 标准的单精度，因此 **double** 和 **long doublefloat** 数据被认为具有与 **float** 数据相同的格式。

表 **3-2** 类型声明符

类型声明符	意义	限制
void	空值集合	
char	可以存储的基本字符集数量	
signed char	有符号整数	（–128 ~ +127）
unsigned char	无符号整数	（0 ~ 255）
short, signed short, **short int, signed short int**	有符号整数	（–32768 ~ +32767）
unsigned short, **unsigned short int**	无符号整数	（0 ~ 65535）
int, **signed**, **signed int**	有符号整数	（–32768 ~ +32767）
unsigned, **unsigned int**	无符号整数	（0 ~ 65535）
long, signed long, long int, **signed long int**	有符号整数	（–2147483648 ~ +2147483647）
unsigned long, **unsigned long int**	无符号整数	（0 ~ 4294967295）
float	单精度浮点数	（1.17549435E–38F ~ 3.40282347E+38F）
double	双精度浮点数	（1.17549435E–38F ~ 3.40282347E+38F）
long double	扩展精度浮点数	（1.17549435E–38F ~ 3.40282347E+38F）
Structure/union specifier	成员对象集合	
Enumeration specifier	**int** 型常量集合	
typedef	指定类型的替代名	
bit, **boolean**, **_ _boolean**	表示一位的整数	（0 至 1）

用斜线分开的声明符具有相同大小。

注　　绝对值范围。

3.2.1 结构体指定符与共用体指定符

结构体指定符和共用体指定符均说明一组命名的成员（对象）。这些成员对象的类型可以互不相同。

(1)结构体指定符

结构体指定符将两个或多个不同类型的一组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。并为成员按照声明顺序保留连续的存储区。

不过，由于 78K0R 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。默认情况下代码大小优先级更高，可以插入对齐数据以确保 2 字节或多字节的成员被分配到偶地址。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。在这种情况下，尽管减小了数据的大小，但是，分配到奇地址 2 字节或多字节的成员是通过使用单字节读/写代码来实现读/写的，这样会增加代码的大小。

结构体的声明如下。但是，并不会为声明分配内存地址，因为它没有结构体变量列表。关于结构体变量的定义，请参阅第 7 章 结构体与共用体。

```
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)共用体指定符

共用体指定符将两个或多个不同类型的一组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，并不会为声明分配内存地址，因为它没有共用体变量列表。关于共用体变量的定义，请参阅第 7 章 结构体与共用体。

```
共用体标识符{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域。

本编译器中增加了与位域声明有关的扩展函数。详细情况，请参阅 **11.5 (15)** 位域声明。

(3)位域

位域是一个整型区域，由指定数量的位组成。可以在位域中指定 **int** 型、 **unsigned int** 型及 **signed int** 型的数据[注 1]。

无修饰词的 **int** 域的最高有效位或 **signed int** 域的最高有效位将被认定为符号位[注 2]。

如果存在两个或多个位域，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域将被压缩到相邻的位位置。通过放置具有零宽度的未命名位域，则下一个位域将不被压缩到相同内存单元内的空间中。未命名的位域没有声明符，仅声明一个冒号和宽度。

单目运算符&（取地址）不能应用到位域对象。

注 1. 在本编译器中，还可以指定 **char** 型、 **unsigned char** 型及 **signed char** 型。它们均被视为 **unsigned** 型，因为本编译器不支持 **signed** 型位域。

注 2. 在本编译器中，可以使用编译器选项 **-RB** 来更改位域分配的方向（详细情况，请参阅第 **11** 章 扩展功能）。

<位域示例>

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

3.2.2 枚举声明符

枚举型声明符是按顺序放置的对象列表。用 **enum** 声明符声明的对象将被声明为 **int** 型的常量。

枚举型声明符的声明如下所示。

```
enum 标识符{成员列表}
```

对象使用枚举符列表进行声明。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，第二个及后续对象的值等于前一个对象的值加 1。还可以使用“=”指定一个常量值。

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 ;
        if ( *cp != burgundy ) {
                :
        } else {
                :
        }
                :
}
```

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。在具有相同作用域的后续声明中，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，结构体或共用体为不完全类型。

```
structure/union      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全。

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"声明为一个由标记（tnode）指定类型的对象，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"表示指向"sp"所指对象左侧的"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 类型修饰词

有两个类型修饰词可供使用：**const** 及 **volatile**。

这两个修饰词只影响左侧的值。

使用非 **const** 型修饰词的左侧值不能更改 **const** 型修饰词定义的对象。使用非 **volatile** 型修饰词的左侧值不能引用 **volatile** 型修饰词定义的对象。

一个用 **volatile** 修饰词定义的对象可能会被编译器无法识别的方法更改，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 C 语言程序抽象调整执行的顺序规则。此外在每个序列点，最后存储在对象中的值必须与程序决定的值一致，除非出现上述的编译器无法识别的因素引起的更改。

如果使用类型修饰词指定了一个数组类型，则修饰词适用于数组元素，而非数组本身。在指定函数类型时不能包括类型修饰词。不过，在 2.2 关键字中提到的本编译器特有的类型修饰词 **callt**, **__ callt**, **callf**, **_ _ callf**, **noauto**, **norec**, **_ _ leaf**, **_ _ interrupt**, **_ _ interrupt_brk**, **_ _rtos_interrupt**, **_ _pascal** 都可以作为修饰词来修饰函数类型。

sreg, **_ _sreg**, **_ _directmap** 和 **_ _temp** 也是类型修饰词。

在下面的例子中，“real_time_clock”可以被硬件更改，但是诸如赋值、递增、递减等运算则无法更改其值。

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

使用类型修饰词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 可以更改  */
typedef int A[2][3];
const A a={{4,5,6},{7,8,9}};/*  const 整型二维数组  */
int *pi;
const int *pci;

ncs=cs;                     /*  正确  */
cs=ncs;                     /*  左值的 volatile 限制，左值具有可调整的赋值运算符  */
pi=&ncs.mem;                /*  正确*/
pi=&cs.mem;                 /*  赋值运算符  =  类型的 volatile 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

声明符用于说明一个标识符。

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

指针声明符表示待声明的标识符是一个指针。

指针指向（指示）一个值的存储位置。指针声明如下所示。

```
* 类型修饰词列表 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 **const** 修饰词并指向 **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说明要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。

```
类型　标识符　[常量表达式]
```

通过该声明，标识符成为具有声明类型的数组。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[ ]”，还声明了有 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，就指定了一个标识符，它用来表示参数及其类型。在头文件“**stdarg.h**”中定义的宏将括号（, ...）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

类型名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。

表 3-3 类型名称举例

类型名称举例	解释
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，该指针指向一个数组，该数组具有三个 **int** 型变量元素。
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，这些数组中的函数具有一个 **unsigned int** 型的参数，还有一个指向各个的函数的常量指针，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 名称的句法如下所示。

```
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域成员的结构体。位域成员如下所示。

- 位域成员名称为 **t**，值为 0 ~ 15
- 位域成员没有名称，**const** 限定的值为–16 ~ +15（如果访问）
- 位域成员名称为 **r**，值为 –16 ~ +15

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

在本例中，这两个位域声明的差别在于，第一个位域声明有 unsigned 作为类型声明符（因此，**t** 成为结构体成员的名称），第二个位域声明有 **const** 作为类型声明符（修饰词 **t** 可以被称为 **typedef** 名）。在此声明之后，如果发现有在有效范围内，则函数 **f** 被声明为“具有一个参数并返回 **signed int** 型值的函数”，函数的这个参数被声明为“指向函数的指针类型，该函数具有一个参数并返回 **signed int** 型值”。标识符 **t** 被声明为 **long** 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都一样效果，指定的函数和未使用 **typedef** 的第一种函数声明方法相同。

```
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

初始化指的是为对象中预先设置一个值。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。
对于具有静态存储生存期的对象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 具有静态存储生存期的对象的初始化

如果未对具有静态存储生存期的算术型对象进行初始化处理，则对象的值将隐性地被初始化为 0。
同样地，具有静态存储生存期的指针型对象将被默认初始化为一个 **null** 空指针常量。

<示例>

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

3.7.2 具有自动存储生存期的对象的初始化

如果未进行初始化处理，则具有自动存储生存期的对象的值将变得不确定，并且无法保证。

```
示例        void func(void){
                char aval;              /*在此点定义*/
                        :
                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，一系列字符串形式的文字可以用来对数组的成员或元素进行初始化。

在下面的例子中，定义了无类型修饰词的数组对象“**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";
```

下面的例子如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，所以长度指示一个“**char** 数组”型对象。

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

3.7.4 聚合或共用体型对象的初始化

（1） 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进行。待指定的初始化符列表必须用括号括起来。

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生存期的对象。

对于大小未知的数组，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。

（2） 共用体型

用括号中的第一个成员作为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，大小未知的数组“x”将变为一维数组，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，因为未指定它的值。

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

下面的例子产生同上例相同的结果。

```
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操作。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 给出了一个类型转换的列表。

表 4-1 类型转换列表

转换前 ＼ 转换后		（有符号）char	无符号 char	（有符号）short int	无符号 short int	（有符号）int	无符号 int	（有符号）long int	无符号 long int	float	double	long double
（有符号）char	+	\	○	○	○	○	○	○	○	○	○	○
	–	\	N	○	N	○	N	○	N	○	○	○
无符号 char		Δ	\	○	○	○	○	○	○	○	○	○
（有符号）short int	+			\	○	\	○	○	○	○	○	○
	–			\	N	\	N	○	N	○	○	○
无符号 short int				Δ	\	Δ	\	○	○	○	○	○
（有符号） int	+			\	○	\	○	○	○	○	○	○
	–			\	N	\	N	○	N	○	○	○
无符号 int				Δ	\	Δ	\	○	○	○	○	○
（有符号） long int	+							\	○	○	○	○
	–							\	N	○	○	○
无符号 long int								Δ	\	○	○	○
float										\	○	○
double											\	○
long double												\

○: 可以正常执行类型转换

\: 不可以执行类型转换

N： 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方式转换。但是，如果一个正数不足以表示它，将不会产生一个正确的值。（被视为一个无符号整数）

空白： 转换结果中的溢出将被截断。根据转换后的类型，数据的 + 或 – 符号可能会被改变。

备注 **signed** 关键字可以省略。但是，对于 **char** 型数据，它将被视为 **signed char** 或 **unsigned char** 型，具体情况要根据编译时的条件（选项）决定。

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** 型位域的数据类型（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 **unsigned int** 型。这种隐式的转换称为"一般整型提升"。

所有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。

在本编译器中，无类型修饰词的 **char** 型数据通常被当作 **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，除非有符号整数是小于零的负值。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，然后为其加上无符号整数可表示的最大值加 1，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，则溢出的值将被忽略。有关转换模式，请参见表 4-1。

下面列出了从有符号整型向无符号整型之间的转换运算。

		无符号	
		值范围更小	值范围更大
有符号	+	/	O
	–	/	+

O:可以正常进行类型转换

+: 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

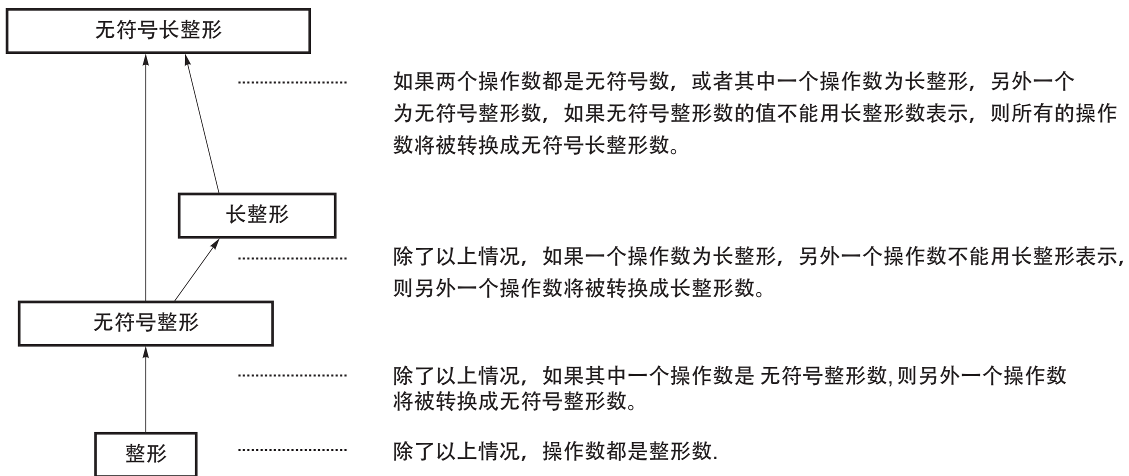
对算术型数据进行运算得到类型可以有各种值。

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 **long double** 型，则另一个将被转换为 **long double** 型。
- 如果运算数中有任一个是 **double** 型，则另一个将被转换为 **double** 型。
- 如果运算数中有任一个是 **float** 型，则另一个将被转换为 **float** 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** 通常算术类型转换



在本编译器中，可以使用编译条件（优化选项）禁止向 **int** 型的转换

详细情况，敬请参阅 **CC78K0R C** 编译器操作篇用户手册。

4.2 其他运算数

(1)左值与函数定位符

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

不包括数组类型、不完全类型或 **const** 修饰词类型的左值，以及没有加 **const** 修饰词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，则表达式或说明符的值将被假定为不存在。

(3)指针

void 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 **void** 型指针。在这两种情况下，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成为"空指针常量"。如果用另一种指针来替代，或赋值，或与空指针常量比较，则空指针常量将被转换为该指针类型。

第 5 章 运算符与表达式

本章介绍 C 语言中使用的运算符和表达式。

C 语言支持很多算术、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位运算和地址运算的运算符。

表达式是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动作，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（设计意图之外的结果）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;                                    /*  = : 赋值运算符  */
        for(i=0;i<=SIZE;i++)                        /*  ++ : 后缀运算符  */
                mark[i]=TRUE;                       /*  < = : 关系运算符  */

        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)                    /*  == : 关系运算符  */
                    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;){
                int j;
                char *ss;
                j=i;
                ss=s;
        }
        void puttchar(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 运算符计算优先级

<table>
<tr><th>表达式类型</th><th>运算符</th><th>结合方向</th><th>优先级</th></tr>
<tr><td>后缀表达式</td><td>[ ] ( ) . – > ++ – –</td><td>→</td><td>最高</td></tr>
<tr><td>单目表达式</td><td>++ – – & * + – ~ ! sizeof</td><td>←</td><td></td></tr>
<tr><td>类型转换表达式</td><td>（类型）</td><td>←</td><td></td></tr>
<tr><td>乘法表达式</td><td>* / %</td><td>→</td><td></td></tr>
<tr><td>加法表达式</td><td>+ –</td><td>→</td><td></td></tr>
<tr><td>按位移位表达式</td><td><< >></td><td>→</td><td></td></tr>
<tr><td>关系表达式</td><td>< > <= >=</td><td>→</td><td></td></tr>
<tr><td>等式表达式</td><td>== !=</td><td>→</td><td></td></tr>
<tr><td>按位与表达式</td><td>&</td><td>→</td><td></td></tr>
<tr><td>按位异或表达式</td><td>^</td><td>→</td><td></td></tr>
<tr><td>按位或表达式</td><td>|</td><td>→</td><td></td></tr>
<tr><td>逻辑与表达式</td><td>&&</td><td>→</td><td></td></tr>
<tr><td>逻辑或表达式</td><td>| |</td><td>→</td><td></td></tr>
<tr><td>条件表达式</td><td>? :</td><td>←</td><td></td></tr>
<tr><td>赋值表达式</td><td>= *= /= %= += –=
<<= >>= &= ^= | =</td><td>←</td><td></td></tr>
<tr><td>逗号表达式</td><td>,</td><td>→</td><td>最低</td></tr>
</table>

同一行内的操作运算具有相同的优先级。

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。

- 声明为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）

如果声明了一个对象，则成为基本表达式的标识符是表达式的左值；如果声明了一个函数，则成为基本表达式的标识符是一个函数定位符。正如在第 2.5 节 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。字符串文字成为左侧值，其具有的数据类型在第 2.6 节 字符串文字中详细介绍。

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。
下文中将介绍基本表达式。

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

下标运算符

语法

```
后缀运算符[下标运算符]
```

功能

下标说明符[ ]指定或引用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（n – 1）维数组的指针，数组元素由 j*...*k 组成。

注

后缀表达式必须有一个“指向对象的....指针”。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“.....”型。

函数调用运算符

语法

```
后缀表达式(函数表达式列表)
```

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。这称为“默认实参扩展”。执行函数原型声明可以避免默认实参扩展，并能检测类型错误、参数数量是否匹配及返回值的类型。

如果调用的函数既未指定存储类型也没有说明数据类型，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，以下的声明将会隐含进行。

```
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

使用该运算符，调用的函数其返回值必须是非数组类型。后缀表达式必须是一个指向该函数的指针类型。

在包括原型的函数调用时，调用的参数类型必须是能够兼容定义中的参数类型。参数的数量也必须一致。

结构体与共用体成员（.）

语法

```
后缀表达式 . 标识符
```

功能

.（点）运算符（也称为成员运算符）指定结构体或共用体中的某个单个成员。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。

结构体与共用体成员（→）

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）指定结构体或共用体的某个单个成员。

后缀表达式是指向特定结构体或共用体的指针的名称，标识符是成员的名称。

<'.'，'->'运算符示例>

```
#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);
}
```

后缀自增运算符(++)

语法

```
后缀表达式 ++
```

功能

后缀自增运算符将对象的值增加 1。

这种自增运算在执行时会根据对象的数据类型而自动调整。

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。

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（说明的或隐含的）。

5.3 单目运算符

单目运算符会对一个对象或参数（即运算数）进行运算。
支持的单目运算符有以下几种。

- 前缀自增运算符 （++）
- 前缀自减运算符 （– –）
- 单目&运算符 （&）
- 单目*运算符 （*）
- 单目算术运算符 （+ – ~ !）
- **sizeof** 运算符

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”产生的效果与以下表达式相同。

```
E = E + 1
  或
E += 1
```

前缀自减运算符（- -）

语法

```
– – 单目表达式
```

功能

前缀自减运算符将对象的值减 1。

前缀自减运算符的表达式“– –E”产生的效果与以下表达式相同：

```
E = E – 1
  或
E –= 1
```

单目&运算符（&）

语法

```
& 运算数
```

功能

单目 & 运算符返回指定对象的指针（即，其后所描述的变量的地址）。

单目 * 运算符（*）

语法

```
    * 运算数
```

功能

单目 * 运算符返回特定指针所指示的值（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一个左值，该左值所引用的对象不支持使用寄存器存储类说明符进行声明。函数定位符及位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为指针类型。

单目算术运算符（+ – ~ !）

语法

```
+  运算数
–  运算数
~  运算数
!  运算数
```

功能

- +（单目加）运算符对其运算数执行正整型提升。
- –（单目减）运算符对其运算数执行负整型提升。
- ~（否定号）运算符是一个按位求补的运算符，它将运算数字中的所有位取反。
- 如果运算数是 1，! 非或逻辑取反运算符返回 0，否则，返回 1。换句话说，该运算符将 0 变为 1，将 1 变为 0。

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指定对象所占的大小。返回值由对象的数据类型控制，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。对于数组型对象，返回值是数组中的字节总数。对于结构体或共用体对象，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

下面的示例通过一个元素的大小除数组中的总字节数来求得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];
}
char array[]=
```

注：

- 如果表达式具有函数类型或不完全类型，并且左值引用位域对象，则该表达式不能用作该操作符的运算数。

5.4 类型转换运算符

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种数据类型。
类型转换运算符主要用于转换指针类型。
有下列几种类型转换运算符可以使用。

类型转换运算符（类型名）

类型转换运算符（类型名称）

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;           /* 转换常量 */
}
```

5.5 算术运算符

- 乘法运算符 *
- 除法运算符 /
- 取余运算符 %
- 加法运算符 +
- 减法运算符 –

算术运算符分为乘法运算符和加法运算符，其中乘法运算符的优先级高于加法。

乘法运算符可以求两个运算数的乘积、商及余数。加法运算符求两个运算数的和与差。

表 5-2 除号/求余运算结果符号

a/b		b: +	b: –
a	+	+	–
	–	–	+

a % b		b: +	b: –
a	+	+	+
	–	–	–

备注 a 和 b 表示运算数。

除法是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 靠拢截取。同样，求余或取模运算是对两个整数进行的，这两个整数的符号（如果有）会通过正常的算术转换去掉。表 5-2 分别以符号给出了两个运算数进行除法和求余运算的计算结果。下面介绍乘法和加法运算符。解释语法中的 E1 和 E2 表示运算数或表达式。

乘法运算符（*）

语法

```
E1 * E2
```

功能

* 运算符对两个运算数执行正常的乘法操作，并返回乘积。

除法运算符（/ ）

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。

取余运算符（%）

语法

```
E1 % E2
```

功能

% 运算符对两个运算数执行求余（或取模）运算，并返回结果中的余数。

加法运算符（+）

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。

减法运算符（–）

语法

```
E1 – E2
```

功能

– 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 移位运算符

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）移动指定位数，移动多少次由第二个运算数决定。共有以下两个移位运算符。

- 移位运算符 <<
- 移位运算符 >>

表 5-3 移位运算

a<<b		b[注]
a	+	0
	–	0

a>>b		b[注]
a	+	0
	–	–1

注　该表说明了当右操作数大于左操作数中的位数，或当移位运算时发生溢出的情况。
如果右操作数是一个负数，则其值将被作为一个无符号正数处理。

备注　a 和 b 表示运算数。

移位运算符（<<）

语法

```
E1 << E2
```

功能

二进制<<（左移）运算符将左操作数向左移动，右操作数指定移动的位数，并将空出的位置填 0。如果在“E1 << E2”中左操作数 E1 是无符号类型，则结果的值就是 E1 乘以 2 的 E2 次幂。

移位运算符（>>）

语法

```
E1 >> E2
```

功能

二进制>>（右移）运算符将左操作数向右移动，右操作数指定移动的位数。

如果左操作数是无符号型的，则空出的位将被填充 0（逻辑移位）。

如果左操作数是带符号型的，则将符号位填充到移动后空出的位中。

如果在“E1 >> E”中左操作数 E1 是无符号形，或 E1 是带符号型的非负值，则结果的值就是 E1 除以 2 的 E2 次幂。

5.7 关系运算符

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"等式运算符"。

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

关系运算符和等式运算符如下所示。

- < 运算符
- > 运算符
- <= 运算符
- >= 运算符
- = = 运算符
- != 运算符

关系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 CC78K0R 编译器中，如果指定的关系是真，则关系运算符和等式运算符产生一个"1"；如果为假，则产生一个"0"。 得到的结果是 int 型的。

下文中将介绍关系运算符和等式运算符。其中的 E1 和 E2 表示运算数或表达式。

关系运算符（<）

语法

```
E1 < E2
```

功能

如果左操作数小于右操作数， < 运算符返回 1；否则，返回 0。

关系运算符（>）

语法

```
E1 > E2
```

功能

如果左操作数大于右操作数， > 运算符返回 1；否则，返回 0。

关系运算符（<=）

语法

```
E1 <= E2
```

功能

如果左操作数小于或等于右操作数，<= 运算符返回 1；否则，返回 0。

关系运算符（**>=**）

语法

```
E1 >= E2
```

功能

如果左操作数大于或等于右操作数，>= 运算符返回 1；否则，返回 0。

等式运算符（= =）

语法

```
E1 == E2
```

功能

如果两个运算数相等， = = 运算符返回 1；否则，返回 0。

等式运算符（**!=**）

语法

```
E1 != E2
```

功能

如果两个运算数不相等，!= 运算符返回 1；否则，返回 0。

5.8 按位逻辑运算符

按位逻辑运算符以位为单位，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 |

下面将介绍按位逻辑运算符。其中的 E1 和 E2 表示运算数或表达式。

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。

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

按位异或运算符（^）

语法

```
E1 ^ E2
```

功能

^ （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”的位置上的值得到 0。

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

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”的位置上的值得到 0。

		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**运算符"&&"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"。

下面详细介绍各个逻辑运算符。其中的 E1 和 E2 表示运算数或表达式。

逻辑与运算符（&&）

语法

```
E1 && E2
```

功能

&& 运算符对两个运算数执行逻辑与运算，如果两个运算数的值均非 0，则返回一个“1”；否则，返回 0。返回结果的类型是 **int** 型。

		左操作数的值	
		0	非 0
右操作数的值	0	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。如果左操作数的值为“0”，则右侧的运算数将不再进行计算。

逻辑或运算符（||）

语法

```
E1 || E2
```

功能

| | 运算符对两个运算数执行逻辑或运算，如果两个运算数的值均为 0，则返回一个“0”；否则，返回 1。返回结果的类型是 int 型。

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">左运算数各位的值</td></tr>
<tr><td>0</td><td>非 0</td></tr>
<tr><td rowspan="2">右运算数各位的值</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>非 0</td><td>1</td><td>1</td></tr>
</table>

注

该运算符始终从左向右对运算数进行求解。如果左操作数的值为非 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将要执行的操作。条件运算符用“?”和“:”来进行判断。下面介绍条件运算符。

条件运算符？ ：

条件运算符（? :）

语法

```
第一运算数 ? 第二运算数 : 第三运算数
```

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。结果的类型是通用类型。

如果两种运算数类型均是结构体或共用体，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（*= /= %= += –= <<= >>= &= ^= |=）

下面将介绍赋值运算符。其中的 E1 和 E2 表示运算数或表达式。

简单赋值运算符（=）

语法

```
E1 = E2
```

功能

= 运算符将右操作数（表达式）转换为左操作数的类型，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"在声明时未加修饰词，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;
if ( ( c = f ( ) ) == -1 ) {
:
} else {
:
}
```

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，并将结果保存到左侧的对象中。保存到左侧对象中的值的类型将被转换为和左侧对象的类型一致。

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示。

逗号运算符 （，）

逗号运算符（，）

语法

```
E1 , E2
```

功能

逗号运算符将左操作数作为 **void** 型来计算（即忽略其值），然后计算右操作数。逗号表达式结果的类型和值就是右操作数的类型和值。

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符，但是下列任何运算符都不能获取目标的值。

- 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。另外还有一种控制结构：分支，它用来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由上而下逐条执行程序中的语句。

(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，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。当处于控制语句的状态条件满足时，可执行语句会重复执行，执行的次数由控制语句决定。

(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。程序从指定标签的下一条语句处开始执行。

在 C 语言中使用如下六类语句。

- 带标签的语句 .... 根据 **switch** 语句的值或 **goto** 语句的目的地跳往一个分支
- 复合语句（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............................ 包括表达式和分号的语句
- 条件控制语句 .... 根据表达式的值从几个备选语句中选择一个符合条件的
- 循环语句.. 调用循环体中的语句被反复执行，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

下面是这些语句的一个说明示例。

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

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

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

        count = 0;
        for(i = 0 ; i <= SIZE ; i++)                    /* for  循环语句 */
            mark [i] = TRUE ;
        for(i = 0 ; i <= SIZE ; i++) {                  / * for  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 带标签的语句

带标签语句指定 **switch** 或 **goto** 语句的跳转目的地。由控制表达式从 **switch** 语句包含的两个（或更多）选项语句中选择指定的语句。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对应的标签。

带标签语句的语法如下所述。

> **case** 标签
> **default** 标签

case标签

语法

```
        case 常量表达式 : 语句
```

功能

case 标签只在 **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），执行表达式“**i=i+4**”。与此类似，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。上面例子中的各个 **break** 语句可以用来退出 **switch** 语句。

如本例所示，在包含两个（或更多）选项时使用 **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** 语句处开始处理。因为 **case** 语句中没有包含 **break** 语句，所以继续顺序执行第三条语句。

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。

default标签

语法

```
default : 语句
```

功能

default 标签是仅用于 **switch** 语句内部的特殊情况标签。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。

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** 分句（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( )的返回值不是 1、2 或 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

表达式语句包括一个表达语句和分号。空语句只包含分号，在需要有语句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，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

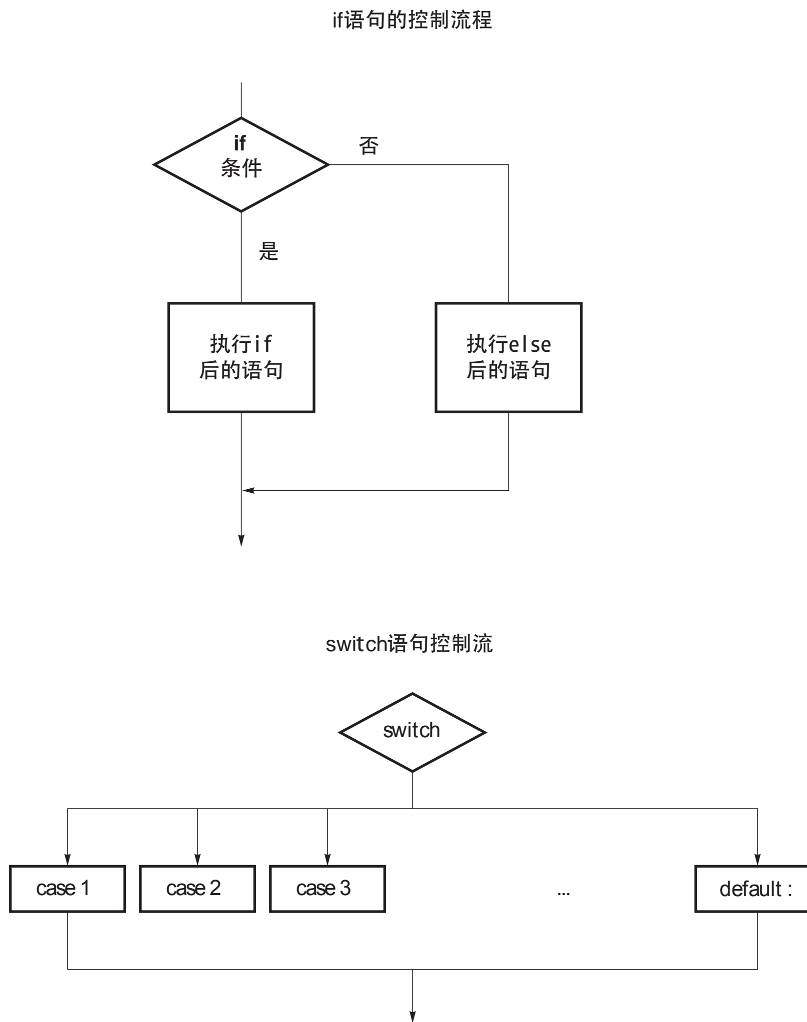
条件控制（选择）语句包括 **if** 和 **switch** 语句。根据括号中的控制表达式的值，**if** 或 **switch** 语句让程序能够在几组备选语句中选择其中一组执行。

有下列条件控制语句类型可以使用。

if 和 if ... else 语句 switch 语句

if 和 **switch** 语句的控制流程如下面的图 6-1 所示。

图 6-1 条件控制语句的控制流程



if 和 **if ... else** 语句

语法

```
if（表达式）语句
if（表达式）语句-1else 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之后的括号内的语句。

if ... else 语句，如果控制表达式的值非零，那么就执行紧随在控制表达式之后的语句-1；如果控制表达式的值为零，就执行 **else** 后面的语句-2。

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在"{ }"中时，只会把 **if** 语句/**if...else** 语句后面的第一行作为执行体。

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。

如果控制表达式值的对应 **case** 标签不存在，那么就会执行 **default** 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** 标签，那么就不会执行任何语句。

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** 标签的值不能相等。**switch** 语句中只能有一个 **default** 标签。

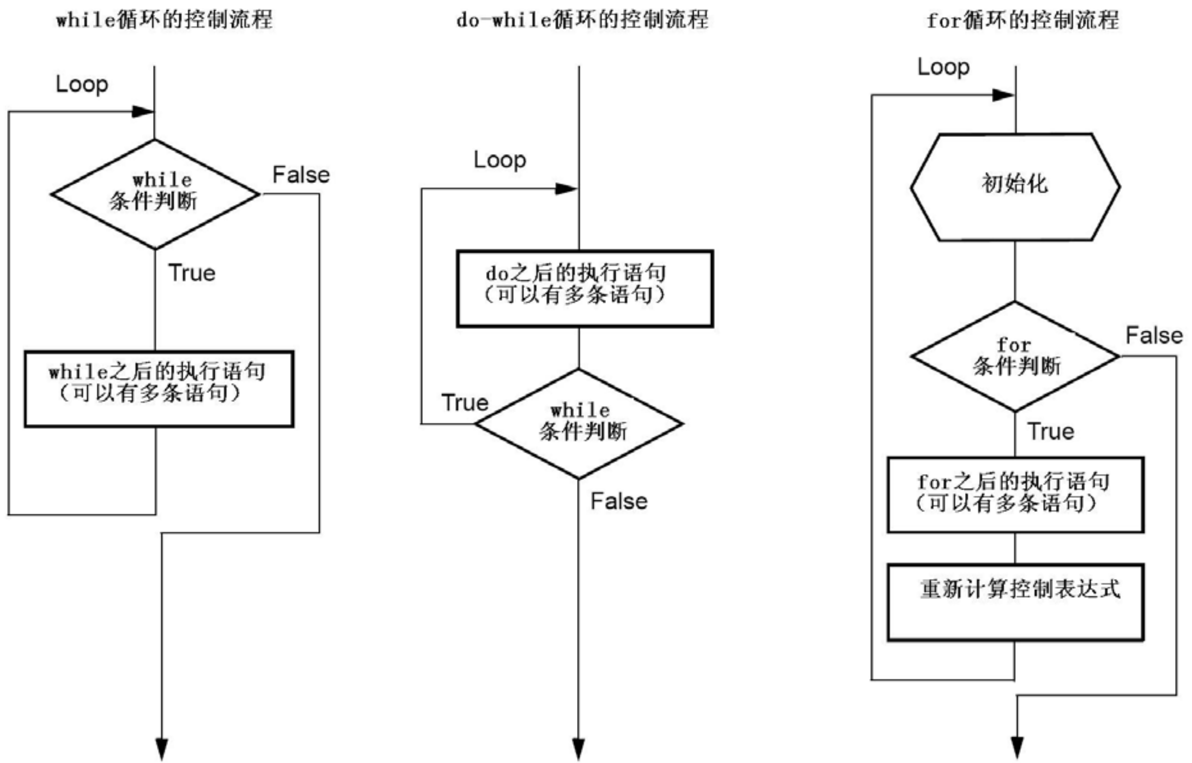
6.5 循环语句

只要括号中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 语言规格中有下列三种类型的循环语句。

- while 语句
- do 语句
- for 语句

各类循环语句的控制流程如下面的图 6-2 所示。

图 6-2 循环语句的控制流程



while语句

语法

```
while（表达式）语句
```

功能

只要括号中控制表达式的值为真（非零），**while** 语句就会反复多次执行一个（或更多）语句（**while** 循环体）。**while** 语句在执行循环体之前先计算控制表达式的值。

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** 循环的主体。如果 i 的值变为 11，那么控制表达式“i<11”返回 0。由于这个原因，只要 i 的值小于 11（在 1 到 10 之间），就会重复执行循环体。

“**while**(1) {语句}”用来无休止地执行循环语句。

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。

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，那么控制表达式"i<11"返回 0。由于这个原因，只要 i 的值小于 11（在 1 到 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，所以循环体至少会执行一次。

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 **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。

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，对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，会对变量进行重新计算。

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，那么控制表达式“i<11”返回 0。由于这个原因，只要 i 的值小于 11（在 1 到 10 之间），就会反复执行循环体。

注

如果 **for** 语句后面的处理内容没有被放在“{ }”中，只会把 **for** 语句后面的第一行当作 **for** 语句的循环体。

for 语句的第一和第三表达式可以省略。当第二表达式被省略时，它会由一个非 0 常量替代。“**for (; ;)** 语句”用来永不停息地执行循环体。

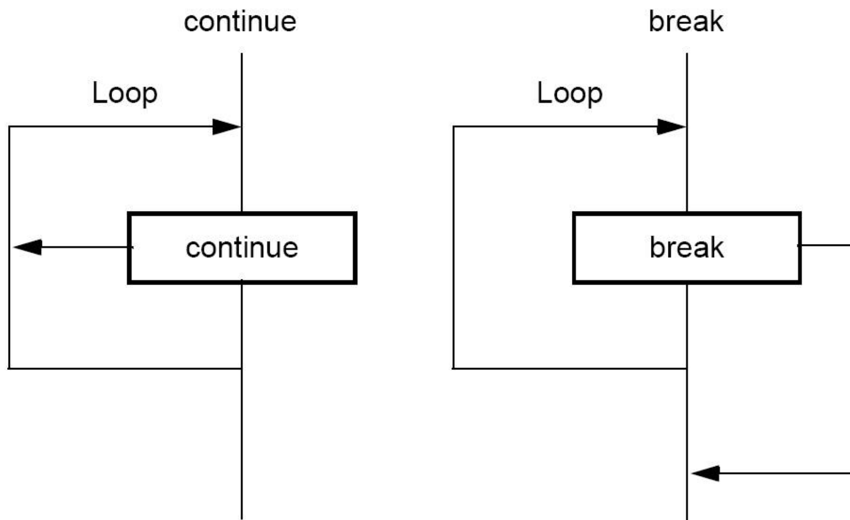
6.6 分支语句

分支语句用来从当前控制流程中退出，并把控制权转移到程序的其它地方。分支语句共有下列四种表达语句。

- 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



goto语句

语法

```
goto  标识符;
```

功能

goto 语句导致程序从当前函数中无条件跳转 **goto** 语句指定的标签名称处。

例

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

说明

在上面的例子中，当程序执行到 **goto** 语句时，C 语言无条件地跳出当前 **do ... while** 循环的处理，并把控制转到"point"标签之后的语句处。

注

在 **goto** 语句指定的标签名称（分支目的地）必须已经在包含 **goto** 语句的当前文件中指定过。换句话说，**goto** 只能从当前函数中跳转到某个标签位置，而不能从一个函数转到另一个函数。

continue语句

语法

```
continue ;
```

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。

例

```
while(i<11){
        :
        continue;
        :
contin : ;
}
```

说明

在上面的例子中，当 C 语言对 **while** 循环的处理到达 **continue** 语句位置时，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 语句只能用在循环体中。

break语句

语法

```
break ;
```

功能

break 语句可能出现在循环语句或 **switch** 语句中，会导致控制转移到循环语句或 **switch** 语句之后的语句位置。

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 循环 */
        }
}
```

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 **switch** 语句时发现相应的 **case** 标签，那么 **case** 标签之后的 **break** 语句会让 C 语言从 **switch** 语句中退出。

注

break 语句只能用在循环体或 switch 语句内。

return语句

语法

```
return 表达式 ;
```

功能

return 语句退出当前函数并将控制转到对此函数发起调用的位置，然后返回 **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“}”与执行不带表达式的纯 **return** 语句具有相同的结果。

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** 类型的函数来说，**return** 语句是不能带有返回值的，返回值表达式不能在 **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一个给定名称的集合中。
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。
结构体声明列表中可以赋予任何标记名称。
在声明之后，就可以使用该标记名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，在第一个结构体声明当中，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。

[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

结构体声明列表指定要声明的结构体类型的内在结构。

结构体声明列表中的每个单独元素都是可以被调用的成员，按照声明的次序给其中各个成员分配存储区。在下面的 **[结构体声明列表举例]** 中，按照变量 a、数组 b、二维数组 c 的顺序分配存储区。

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

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。还可以用位域方式来指定各成员的位数。

如果变量取二进制值"0"或"1"，则位域所需的最少位数指定为 1。通过这种对位域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];
```

[位域声明举例]

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

(3)数组和指针

结构体变量可以声明为数组，也可以用指针进行引用。

[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针，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（或结构体元素）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）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 ;
        c=no[0].name[1];
}
```

[用指向结构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。任何可以被称为标记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。

共用体声明列表中的每个单独元素都是可以被调用的成员，存储区按照声明的次序分配给其中各成员。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。不会为其它成员分配新存储区，而是使用同一块区域。

不完整类型（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

共用体变量可以声明为数组，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**]**

在利用指向共用体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。这些声明被称为“外部声明”，因为表面看来它们处于函数外部而且有效范围可以跨越文件范围。

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** 运算符的运算对象外），那么在整个程序中的某处必定存在该标识符的一个外部定义。

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];                    /* 外部对象定义 */

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，则默认定义为 **extern**。外部函数定义意味着定义的函数可能会从其它文件被引用。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，因此可以省略。函数声明部分为“max(int a, int b)”，函数体为“{return 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，所以编译器对参数类型会进行强制转换。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标识符列表的示例。

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

函数地址可以在函数调用中作为参数传递。使用表达式中的函数名，可以生成指向该函数的指针。

```
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** 只能以下列两种方式之一进行定义。

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

或

```
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。

如果对象标识符的声明有文件作用域而无对应初始化，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初始为 0。

如下所示为外部对象定义的示例。

表 8-1 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;.........................      /* 违反连接规则 */
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;.........................      /* 违反连接规则 */
extern int i1;..................      /* 引用先前的具有外部连接的声明 */
extern int i2;..................      /* 引用先前的具有内部连接的声明 */
extern int i3;..................      /* 引用先前的具有外部连接的声明 */
extern int i4;..................      /* 引用先前的具有外部连接的声明 */
extern int i5;..................      /* 引用先前的具有内部连接的声明 */
```

第 9 章 预处理指令（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。

预处理指令指定源文件编译前进行的处理工作。预处理指令包括的操作有：根据条件的判定来处理或跳过部分源文件、从其它源文件获取代码、将原来的源代码替换为其它文本（与宏扩展相同）。

预处理指令在下文中进行详细说明。

9.1 条件编译

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）。在任何条件编译指令的常量表达式中都不能使用 **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

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

在预处理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。

[例]

在本例中，因为 SYM 已定义，所以单目表达式返回 1，并对**#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
```

#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 编译器在翻译阶段中跳过（丢弃）一段源代码。

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"是否成立，以此来确定是否在翻译阶段中使用**#if** 和**#endif** 之间的若干语句（即源代码）。如果"FLAG"的值为非零，则**#if** 和**#endif** 之间的源代码会被丢弃。如果值为零，则**#if** 和**#endif** 之间的源代码会被翻译。

#elif指令

语法

```
#elif 常量表达式 ˛ 换行 [组]
```

功能

#elif 指令通常跟在**#if** 指令之后。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）。

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。

#ifdef 指令

语法

```
#ifdef 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。

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。

#ifndef指令

语法

```
#ifndef 标识符. 换行 [组]
```

功能

#ifndef 指令等同于：

#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。

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，不会翻译**#ifndef** 和**#endif** 之间的程序。若从未定义标识符“ON”，则会翻译**#ifndef** 和**#endif** 之间的程序。

#else 指令

语法

#else 换行 [组]

功能

在前面的条件编译指令的标识符为非零的情况下，**#else** 指令使 C 编译器在翻译阶段时丢弃**#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。

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。若从未定义标识符“ON”，则会翻译**#else** 和**#endif** 之间的源代码。

#endif指令

语法

```
#endif  换行
```

功能

#endif 指令指示**#ifdef** 块的结尾。

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，在本编译器中对**#include** 指令的嵌套层数是有限制的。关于具体的限制情况，请参见**表 1-1**。

备注 预处理记号字符串：由**#define** 指令定义的字符串

#include < >指令

语法

```
#include <文件名> 换行
```

功能

若指令格式如 **#include <>**，则 CC78K0R 编译器会在相关目录中搜索尖括号中指定的头文件，这些目录包括编译器选项**-i** 指定的目录、INC78K0R 环境变量指定的目录和“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路经），用指定文件的全部内容来替换**#include** 指令行语句。

例

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

说明

在上面的例子中，C 编译器在 INC78K 环境变量指定的目录和“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路经）中搜索文件 **stdio.h**，用指定文件 **stdio.h** 的全部内容来替换指令行**#include <stdio.h>**。

备注 上面所述的目录可能会因安装方法的不同而有所差异。

#include“ ”指令

语法

```
#include "文件名" 换行
```

功能

若指令形如**#include**“ ”，则首先在当前工作目录下搜索双引号内指定的头文件。如果没有找到，则会在编译器选项**-i** 指定的目录、INC78K 环境变量指定的目录和“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路径）中进行搜索。然后，编译器用搜索到的指定文件的全部内容来替换**#include** 指令行。

例

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

说明

在上面的例子中，C 编译器在当前工作目录、INC78K 环境变量指定的目录、“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路径）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”**。

备注　　上面所述的目录可能会因安装方法的不同而有所差异。

#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。

例

```
#define INCFILE "myprog.h"
#define 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 编译器在编译器选项**-i** 指定的目录、INC78K 环境变量指定的目录、“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路经）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。如果未能找到指定文件，则会在编译器选项**-i** 指定的目录、INC78K 环境变量指定的目录和“..\NECTools32\INC78K0R”目录（CC78K0R 的启动路经）中进行搜索。

备注 上面所述的目录可能会因安装方法的不同而有所差异。

9.3 宏替换

宏替换指令**#define** 和**#undef** 用"替换列表"来替换标识符指定的字符串（宏名）。**#define** 指令有两种形式：对象格式和函数格式：

- 目标格式
  #define 指令

- 函数格式
  #define（）指令

(1)实参替换

函数调用时的实际参数替换，必须在标识了函数形式宏调用的参数之后执行的。如果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，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，如果替换列表中的参数前有该预处理记号，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

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），即使在扫描替换列表时再次发现，也不会进行替换。

(5)宏定义的作用域

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

#define指令

语法

```
#define 标识符 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替换为给定替换列表的对应内容，且对源代码中此指令定义后出现的相同标识符都进行相同操作。

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”。

#define ( )指令

语法

#define 标识符 ([标识符表]) 替换列表 换行

功能

函数形式的**#define** 指令格式为"#define name (name,…name) 替换列表"，将指定标识符替换为给定替换列表的对应内容。

只要相同标识符在源代码中该指令定义位置之后出现，就会进行替换。函数形式宏替换还包括对参数的替换。

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。

#undef指令

语法

```
#undef 标识符 换行
```

功能

#undef 指令终止由对应**#define** 指令设置的标识符作用域。

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

行控制预处理指令**#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。

[更改行号]

要想更改行号，应进行如下的指定。

不能指定为 0 和大于 32767 的数。

```
#line 数字字符串 换行
```

[例]

```
#line 10
```

[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
```

[例]

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

[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，指定的预处理记号字符串在所有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
```

[例]

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

9.5 #error预处理指令

#error 预处理指令输出信息，信息中包括已指定的预处理记号，并使编译过程不完全终止。
此预处理用来终止编译。此预处理指令的说明如下。

```
#error "预处理记号字符串" 换行
```

[例]

```
#if _ _K0R_ _
        :
#else
#error "not for 78K0R"
        :
#endif
```

在本例中使用了指示本编译器设备系列的宏名称**__K0R __**。若设备为 78K0R 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信息“not for 78K0R（不适用于 78K0R）”。

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 CC78K0R 编译器中有若干个用来使产生的代码适应 78K0R 系列的**#pragma** 指令。
关于**#pragma** 指令的详情请参见 第 11 章 扩展函数。

[例]

在本例中，**#pragma NOP** 指令使该描述语句能在 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 预定义的宏名

在 CC78K0R 编译器中已经定义了下列宏名。

表 9-1 宏名称列表

_ _LINE_ _	当前源程序行的行号（十进制常量）
_ _FILE_ _	源文件名（字符串字面量（字符文字））
_ _DATE_ _	源文件编译日期（字符串文字，形如“Mmm dd yyyy”）
_ _TIME_ _	源文件编译时间（字符串文字，形如“hh:mm:ss”）
_ _STDC_ _	十进制常量“1”，表示与 ANSI [注]规格兼容

注　　ANSI 是美国国家标准协会的缩略词。

绝对不能用**#define** 或**#undef** 预处理指令定义这些宏名或者将其定义为标识符。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于指示设备序列和指示具体设备名称的宏名。要想针对目标设备输出目标码，必须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。

- 指示设备序列的宏名

```
'_ _K0R_ _'
```

- 指示设备名称的宏名
  在设备类型名称前添加了‘_ _’，名称后添加了‘_’

<例>

```
_ _F1166A0_      _ _F1166A0Y_
```

注意其中的英文字母为大写。

备注　　设备类型名称与-C 选项指定的类型名称相同。关于设备类型名称，请参见设备文件相关的参考资料。

CC78K0R 编译器具有指示存储模型的宏名。
当指定静态模型时，定义如下

<当指定小型模式时>

```
#define _ _K0R_SMALL _ _          1
```

<当指定中等模式时>

```
#define _ _K0R_MEDIUM _ _         1
```

<当指定紧凑模式时>

```
#define _ _K0R_COMPACT_ _         1
```

<当指定大型模式时>

```
#define _ _ K0R_LARGE_ _          1
```

通过在命令行中加入下列内容可指定编译的目标设备类型
“-c 设备类型名”

<例>

```
cc78k0r  –cF1166A0Y  prime.c
```

不需要在编译时指定设备类型，可以在 C 源程序的开头进行指定。

“#pragma PC (设备类型)”

<例>

```
#pragma PC(F1166A0Y)
            :
```

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
- 不会产生对变量或函数的定义/引用的预处理指令。

第 10 章 库函数

在 C 语言中，没有专门的指令用于和外部源（外围器件和设备）进行传输（输入输出）数据的指令。这是因为 C 语言的设计者希望这种函数的个数尽可能保持最少。但是，对于实际系统开发来说，I/O 操作是必须的。因此，CC78K0R 中必定会含有进行 I/O 操作的库函数。

CC78K0R 编译器中含有的库函数如 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 CC78K0R 编译器所提供的库函数。

10.1 函数之间的接口

要想使用库函数，必须进行调用。对库函数的调用需要通过调用指令来完成。函数的参数和返回值分别由栈和寄存器进行传递。

但是，第一参数也会尽可能的通过寄存器传递。

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方（发起调用的函数）进行的。被调用方（被别的函数调用的函数）只引用参数值。

但是，当参数由寄存器传递时，被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。

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。

第一个参数的传递列表如下所示。

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

第一参数的类型	传递方法
1 字节、2 字节整型	AX
near 数据指针	AX
3 字节整型	AX，BC
4 字节整型	AX，BC
函数指针，far 数据指针	AX，BC
浮点数（**float** 类型）	AX，BC
浮点数（**double** 类型）	AX，BC
其它	通过堆栈传递

备注 在上述类型中，1 到 4 字节整型中包括结构体和共用体。

10.1.2 返回值

函数的返回值从它的最低有效位（LSB）开始，以 16 位为单位进行存储，顺序是从寄存器 BC 到寄存器 DE。当返回结构体时，结构体的首地址储存在寄存器 BC，DE 中。

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。

表 10-2 返回值存储列表

返回值的类型	存储方法
1 位	CY
1 字节、2 字节整型	BC
near 指针	BC
4 字节整型	BC（低）、DE（高）
far 指针	BC（低）、DE（高）
浮点数（**float** 类型）	BC（低）、DE（高）
浮点数（**double** 类型）	和 float 型同样处理
结构体	复制结构体，返回到函数专用区域，把地址存储到 BC，DE 中。

10.1.3 保存单独库（Individual Libraries）所用的寄存器

使用 HL 的库把自己使用的寄存器保存到堆栈中。

每个使用 **saddr** 区域的库，把自己使用的 **saddr** 区域地址保存到堆栈中。堆栈区被各个库当作工作区使用。

传递参数和返回值的过程示例如下所示（指定为小型模式或者中等模式）。

<被调用函数>

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

<a> （由调用方）把参数放入栈中

参数“c”和“b”的高 16 位及参数“b”的低 16 位按照命名的顺序放入栈中。参数“a”由 AX 寄存器传递。

<b> （由调用方）通过 **call** 指令调用 **func**

返回地址在堆栈中的位置处于参数“b”的低 16 位之后，控制流程转移到函数 **func**。

<c> （由被调用方）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 原来的值就被压入堆栈中。

<d> （由被调用方）把由寄存器传递的第一个参数放入堆栈中。

<e> （由被调用方）处理 **func** 并把返回值存储在寄存器中。

返回值“long”的低 16 位存储在 BC 中，返回值的高 16 位存储在 DE 中。

<f> （由被调用方）恢复存储的第一个参数

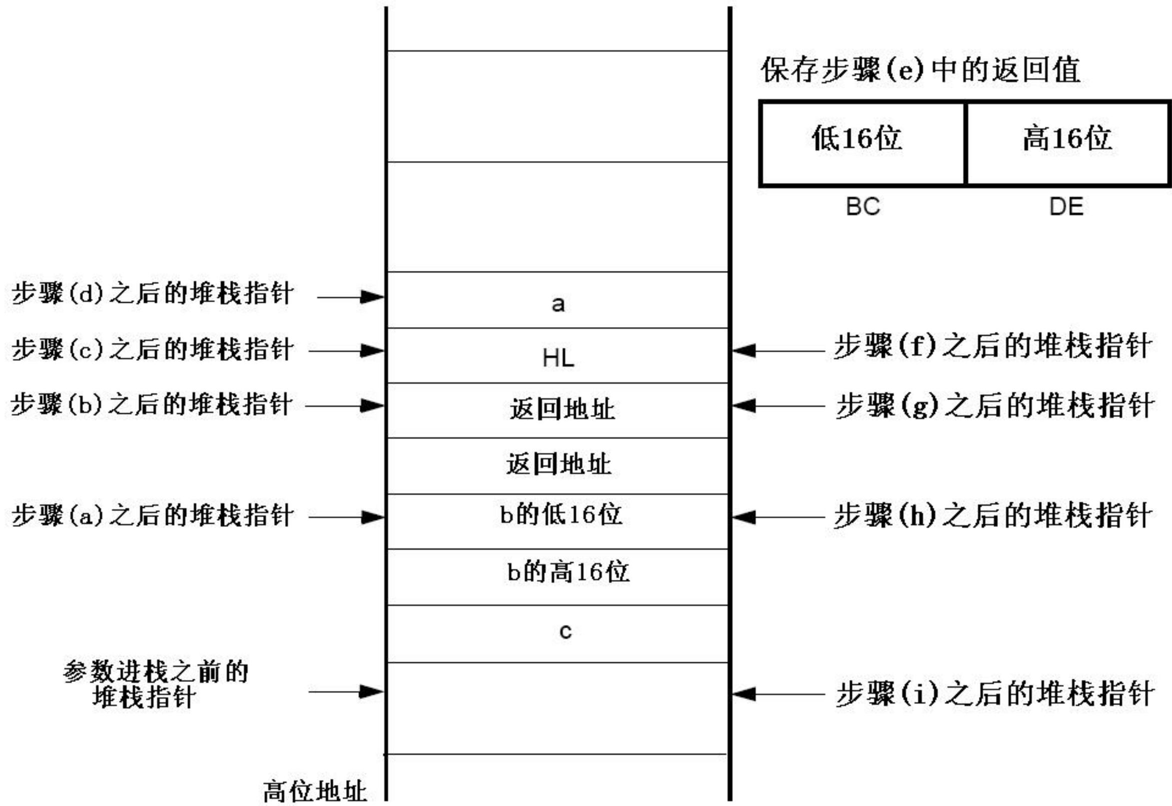
<g> （由被调用方）恢复保存的寄存器

<h> （由被调用方）通过 **ret** 指令把控制权返回给调用方

<i> （由调用方）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。

给堆栈指针加 6。



10.2 头文件

CC78K0R 编译器具有 13 个头文件。每个头文件对标准库函数、数据类型名和宏名称进行定义或声明。
CC78K0R 编译器的头文件如下所示。

ctype.h	setjmp.h	stdarg.h	stdio.h	stdlib.h
string.h	error.h	errno.h	limits.h	stddef.h
math.h	float.h	assert.h		

(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如下库函数。
但是，在指定了编译选项**-ZA**（此选项禁止使用不符合 ANSI 规格的函数，允许使用符合 ANSI 规格的一部分函数）时，**_toupper** 和**_tolower** 将不会被定义，定义了 **tolow** 和 **toup** 作为替代。当未指定**-ZA** 时，**tolow** 和 **toup** 不做定义。声明的函数随选项和指定模式的不同而有所差异。

isalnum	isalpha	iscntrl	isdigit	isgraph
islower	isprint	ispunct	isspace	isupper
isxdigit	tolower	toupper	isascii	toascii
_toupper	_tolower	tolow	toup	

(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。将要声明的函数随选项和指定模式的不同而有所差异。

Setjmp	longjmp

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]

```
typedef int  jmp_buf[12];
```

(3) stdarg.h

此头文件用来定义特殊函数。

在此头文件中定义了下列三个函数。

```
va_arg      va_start      va_starttop      va_end
```

在头文件 **stdarg.h** 中声明了下列对象。

[把 char 声明为指针类型 va_list]

```
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。

将要声明的函数随选项和指定模式的不同而有所差异。

```
sprintf      sscanf      printf      scanf      vprintf
vsprintf     getchar     gets        putchar    puts
__putc
```

声明了下列宏名称。

```
#define EOF  (–1)
```

(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，在指定了编译选项**-ZA**（此选项禁止使用不符合 ANSI 规格的函数，允许使用符合 ANSI 规格的一部分函数）时，**brk**、**sbrk**、**itoa**、**ltoa**、**ultoa** 无定义。定义 **strbrk**、**strsbrk**、**stritoa**、**strltoa** 和 **strultoa** 作为替代。当未指定**-ZA** 时，这些函数不作定义。

atoi	atol	strtol	strtoul	calloc
free	malloc	realloc	abort	atexit
exit	abs	div	labs	ldiv
brk	sbrk	atof	strtod	itoa
ltoa	ultoa	rand	srand	bsearch
qsort	strbrk	strsbrk	stritoa	strltoa
strultoa				

在头文件 **stdlib.h** 中定义了下列对象。

[对结构体类型 div_t 进行声明，其成员 quot 和 rem 为 int 类型]

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

[对宏名称 **RAND_MAX** 的定义]

```
#define RAND_MAX  32767
```

[对宏名称的定义]

```
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的不同而有所差异。

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

(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。

[对宏名称"EDOM"、"ERANGE"、"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（它把未指定修饰词的 **char** 当作 **unsigned char**）时，通过由编译器声明的宏 **__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)
```

(10) stddef.h

在此头文件中声明和定义了下列对象。

[将 int 类型声明为 ptrdiff_t]

```
typedef int                ptrdiff_t;
```

[将 unsigned int 类型声明为 size_t]

```
typedef unsigned int       size_t;
```

[对宏名称 NULL 的定义]

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

[对宏名称 offsetof 的定义]

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

备注　**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）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。当指定的成员为位域时，操作无法保证。

(11) math.h

math.h 定义了下列函数。

```
acos      asin      atan      atan2     cos
sin       tan       cosh      sinh      tanh
exp       frexp     ldexp     log       log10
modf      pow       sqrt      ceil      fabs
floor     fmod      matherr   acosf     asinf
atanf     atan2f    cosf      sinf      tanf
coshf     sinhf     tanhf     expf      frexpf
ldexpf    logf      log10f    modff     Powf
Sqrtf     Ceilf     Fabsf     Floorf    Fmodf
```

定义了下列对象。

[对宏名称 HUGE_VAL 的定义]

```
#define HUGE_VAL DBL_MAX
```

(12) float.h

float.h 定义了下列对象。

当 **double** 类型的长度为 32 位时，由编译器声明的宏_ _**DOUBLE_IS_32BITS**_ _将对要定义的宏进行排序。

```
#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             1.1749435E–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

assert.h 定义了下列函数。

```
_ _assertfail
```

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**，此宏名称不在 **assert.h** 头文件中定义。如果当 assert.h 被加载到源文件时，**NDEBUG** 被定义为宏，那么 **assert.h** 头文件就会简单地对 assert 宏做如下声明，而不定义**__assertfail**。

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

10.3 可重入性（re-entrantability）（仅适用于正常模式）

重入（re-entrant）是一种状态，指由一个程序调用的函数能够被另一个程序继续调用。

CC78K0R 编译器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) 中所示的函数是不可重入的。

- 不可重入的函数

setjmp、 longjmp、 atexit、 exit

- 下列函数所使用的区域在启动例程中被特意保留

div、 ldiv、 brk、 sbrk、 rand、 srand、 strtok

- 处理浮点数的函数

sprintf、 sscanf、 printf、 scanf、 vprintf、 vsprintf [注]、 atof、 strtod、所有数学函数

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

10.4 标准库函数

本节按照下列函数分类方式说明 CC78K0R 编译器的标准库函数。即使在指定了**-ZF** 参数时也支持所有的标准库函数。

表 10-3 标准库函数列表

函数类型	函数
Character & String Functions（字符函数和字符串函数）	is-
	toupper, tolower
	toascii
	_toupper/toup, _tolower/tolow
Program Control Functions（程序控制函数）	setjmp, longjmp
Special Functions （特殊函数）	va_start, va_starttop, va_arg, va_end
I/O Functions（I/O 函数）	sprintf
	sscanf
	printf
	scanf
	vprintf
	vsprintf
	getchar
	gets
	putchar
	puts
	__putc

表 10-3 标准库函数列表

函数类型	函数
Utility Functions（应用函数）	atoi, atol
	strtol, strtoul
	calloc
	free
	malloc
	realloc
	abort
	atexit, exit
	abs, labs
	div, ldiv
	brk, sbrk
	atof, strtod
Utility Functions（应用函数）	itoa, ltoa, ultoa
	rand, srand
	bsearch
	qsort
	strbrk
	strsbrk
	stritoa, strltoa, strultoa

表 10-3 标准库函数列表

函数类型	函数
Character String/Memory Functions（字符串/存储器函数）	memcpy, memmove
	strcpy, strncpy
	strcat, strncat
	memcmp
	strcmp, strncmp
	memchr
	strchr, strrchr
	strspn, strcspn
	strpbrk
	strstr
	strtok
	memset
	strerror
	strlen
	strcoll
	strxfrm
Mathematical Functions（数学函数）	acos
	asin
	atan
	atan2
	cos
	sin
	tan
	cosh
	sinh

表 10-3 标准库函数列表

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

表 10-3 标准库函数列表

函数类型	函数
Mathematical Functions（数学函数）	ceilf
	fabsf
	floorf
	fmodf
Diagnostic Functions（诊断函数）	__assertfail

10.4.1 为参数和返回值使用优化库

通过可以将指针指定为参数和返回值的标准库，将会根据所指定的存储器模式链接一个优化库。

如果某指针不是存储器模式的默认指针，为了处理这个指针，需要用下列标准函数名称调用函数；就会为该指针链接优化库。

<函数名称>_n ：该指针总是被当作 near 指针处理。

<函数名称>_f ：该指针总是被当作 far 指针处理。

例如，当使用小型（small）模式时，far 指针可以指定为 strcmp 函数的一个参数。

〈例〉

```
#include <string.h>
__far char * sf1;
__far char * sf2;
func ( ) {
          :
   r = strcmp_f ( sf1 , sf2 );
          :
}
```

[注意]

- 当使用小型模式或中等模式时，I/O 函数 sprintf, printf, vprintf, vsprintf, sscanf 和 scanf 的指针参数被当作 near 指针处理，该指针参数和变量参数同样对待。不能使用函数指针。
- 当使用函数指针或者 far 指针时，使用 printf_f, 并且所有的变量参数指针都必须转换为 far 指针。
- 当使用紧凑模式或大型模式时，I/O 函数 sprintf, printf, vprintf, vsprintf, sscanf 和 scanf 的指针参数被当作 far 指针处理，该指针参数和变量参数同样对待。
- 当使用小型模式或中等模式时，特殊函数 va_start, va_starttop, va_arg 和 va_end 的指针函数被当作 near 指针处理，该指针参数和变量参数同样对待。不能使用函数指针。

10.5 字符和字符串函数

有如下的字符/字符串函数可用。

- is-
- toupper, tolower
- toascii
- _toupper/toup, _tolower/tolow

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	字符 c 处于字符范围内时为 1 字符 c 不处于字符范围内时为 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

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.. 要进行转换的字符	如果 c 是可转换字符，就会返回对应的大写字母/小写字母。 如果 c 不可转换，字符“c”就会原样返回。

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。

toascii

功能

字符函数 **toascii** 将“c”转换成 ASCII 码。

头文件

ctype.h

函数原型

int toascii(**int c**);

函数	参数	返回值
toascii	c.. 要转换的字符	将“c”中超出 ASCII 码范围之外的位转换为 0，然后将该值返回。

说明

toascii 函数将“c”在 ASCII 码范围（0 至 6 位）之外的位（7 至 15 位）转换为“0”并返回转换后的位值。

_toupper/toup，_tolower/tolow

功能

字符函数**_toupper/toup** 从“c”中减去“a”再加上“A”得到结果。

字符函数**_tolower/tolow** 从“c”中减去“A”再加上“a”得到结果。

（**_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.. 要转换的字符	从“c”中减去“a”再加上“A”得到的值
_tolower/tolow		从“c”中减去“A”再加上“a”得到的值

备注 a：小写；A：大写

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。

10.6 程序控制函数

有如下程序控制函数可用

- setjmp，longjmp

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** ... 用于保存环境信息的数组	• 若直接调用，返回 0 • 若从相应 **longjmp** 处返回，则返回由“val”给出的值；若“val”为 0 则返回 1
longjmp	**env** ... 由 **setjmp** 保存环境信息的数组 **val** ... **setjmp** 的返回值	**longjmp** 不会返回，因为程序继续执行将会恢复到保存环境的 **setjmp** 之后语句的状态。

说明

setjmp

- 在直接调用时，**setjmp** 将 **saddr** 区域、**SP** 及函数的返回地址（作为 **HL** 寄存器或寄存器变量使用）保存到 **env** 中，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 **SP**）中。程序继续执行，就好像对应的 **setjmp** 返回了 **val** 一样（但是，如果 **val** 为 0，则返回 1）。

10.7 特殊函数

有如下特殊函数可用

- va_start, va_starttop, va_arg, va_end

va_start，va_starttop，va_arg，va_end

功能

va_start 函数（宏）用来启动变量参数列表。
va_starttop 函数（宏）用来设置可变参数的数量处理。
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**);
va_starttop (**va_list ap**,**parmN**);
type va_arg(**va_list ap**,**type**);
void va_end(**va_list ap**);

函数	参数	返回值
va_start **va_starttop**	**ap** ... 要进行初始化并在 **va_arg** 和 **va_end** 中使用的变量 **parmN** ... 变量参数前面的参数	无
va_arg	**ap** ... 处理参数列表的变量 **type**... 指向变量参数相应位置的类型（**type** 是变量长度的类型；例如，若说明为 **va_arg** (va_list ap, int) 则为 **int** 类型，或者，若说明为 **va_arg** (**va_list ap**, **long**) 则为 **long** 类型）	通常情况下 ... 变量参数相应位置的值 若 **ap** 为空指针 ... 返回 0
va_end	**ap** ... 用来处理参数变量数量的变量	无

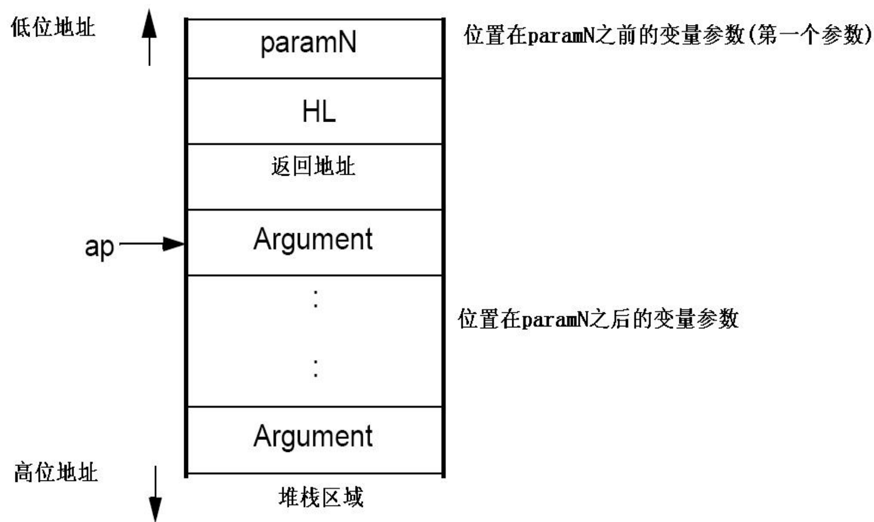
说明

va_start

- 在 **va_start** 宏中，参数 **ap** 必须是 **va_list** 类型（**char***类型）的对象。
- 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。
- 如果 **parmN** 是第一个参数，这个函数的正常操作无法保证，这种函数可以使用 **va_starttop**。

va_starttop

- 参数 **ap** 必须是 **va_list** 类型（**char***类型）的对象。
- 指向 **parmN** 中下个参数的指针存储在 **ap** 中。
- **parmN** 是最右侧的参数名称，也是函数原型中指定的第一个参数。
- 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。
- 如果 **parmN** 不是第一个参数，这个函数的正常操作无法保证。



va_arg

- 在 **va_arg** 宏中，参数 **ap** 必须与和 **va_start** 初始化所使用的 **va_list** 类型对象相同（否则无法保证其他正常操作）。
- **va_arg** 在变量参数相应位置返回一个 **type** 类型的值。
  此处所指的对应位置就是紧跟在 **va_start** 之后的第一个变量参数以及之后的每个 **va_arg**。
- 如果参数指针 **ap** 为空指针，那么 **va_arg** 就返回 0（**type** 类型）。
- 在 CC78K0R 编译器中，当指定一个指针作为参数列表，在中等模式下必须指定 near 数据指针（2 字节长度），在大型模式下必须指定 far 数据指针（4 字节长度）。
  在各种模式下，函数指针长度固定为 4 字节，但是在指定指针作为参数列表时，各种模式下的指针长度必须指定为 2-或 4-字节。

va_end

- **va_end** 宏在参数指针 **ap** 中设置一个空指针，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。

10.8 I/O函数

有如下 I/O 函数可用

- sprintf
- sscanf
- printf
- scanf
- vprintf
- vsprintf
- getchar
- gets
- putchar
- puts
- __putc

sprintf

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** ... 指向将输出写入内容的字符串的指针 **format** ... 该指针指向的字符串用来说明格式命令的规格 **...** ... 零个或更多个要转换的参数	在 **s** 中写入的字符数量（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量，那么就无法保证操作。如果实际的参数数量多于格式中所定义的数量，那么只会对多余的实际参数进行评估并忽略。
- 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。
- 每个格式命令都以一个%字符开头，后面跟的内容可以是：
  (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。）默认用空格进行空洞填充。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，不会因为指定了最小字段宽度而被截取。
- 可选精度（小数位数）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。
  此精度说明必须采用(.整数)的形式。若省略整数部分，则假定已指定为 0。
  精度规格指定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**、**l** 和 **L** 修饰词
  h 修饰词要求 **sprintf** 函数以短整型或无符号短整型类型来进行此修饰词后面的 **d**、 **i**、 **o**、 **u**、 **x** 或 **X** 类型的转换。**h** 修饰词要求 **sprintf** 函数用短整型指针进行此修饰词后面的 **n** 类型转换。
  l 修饰词要求 **sprintf** 函数以长整型（**long int**）或无符号长整型（**unsigned long int**）类型进行此修饰词后面的 **d**、 **i**、 **o**、 **u**、 **x** 或 **X** 类型的转换。**h** 修饰词要求 **sprintf** 函数用长整型指针进行此修饰词后面的 n 类型转换。
  其它类型说明符， **h**、**l** 或 **L** 修饰词均被忽略。

- 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）标志之后的正字段。忽略所有负精度。

下列标志用来修饰格式命令：

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。
space	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，将空格标志被忽略。
#	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。

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。

如果没有指定精度，则假定默认指定为 1。

若用 0 精度转换 0，则什么也看不到。

精度代码	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。当省略精度时，默认精度为 6。
e	将 **double** 型参数用“[-] d.dddd e [符号] ddd”格式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。ddd 肯定为三位十进制数，其符号为+或–。 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	将整型参数转换为无符号字符型并将结果写为单个字符。
s	相关参数是一个指向字符串的指针，其中的字符会持续写入，直到遇到终止空字符（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，其中放置已经写入字符串“s”中的字符数量。 不进行任何转换。
%	打印一个 % 符号。 不转换相关参数（但标志及最小字段宽度的说明是有效的）。

- 无效的转换说明符在操作时无法保证。
- 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。

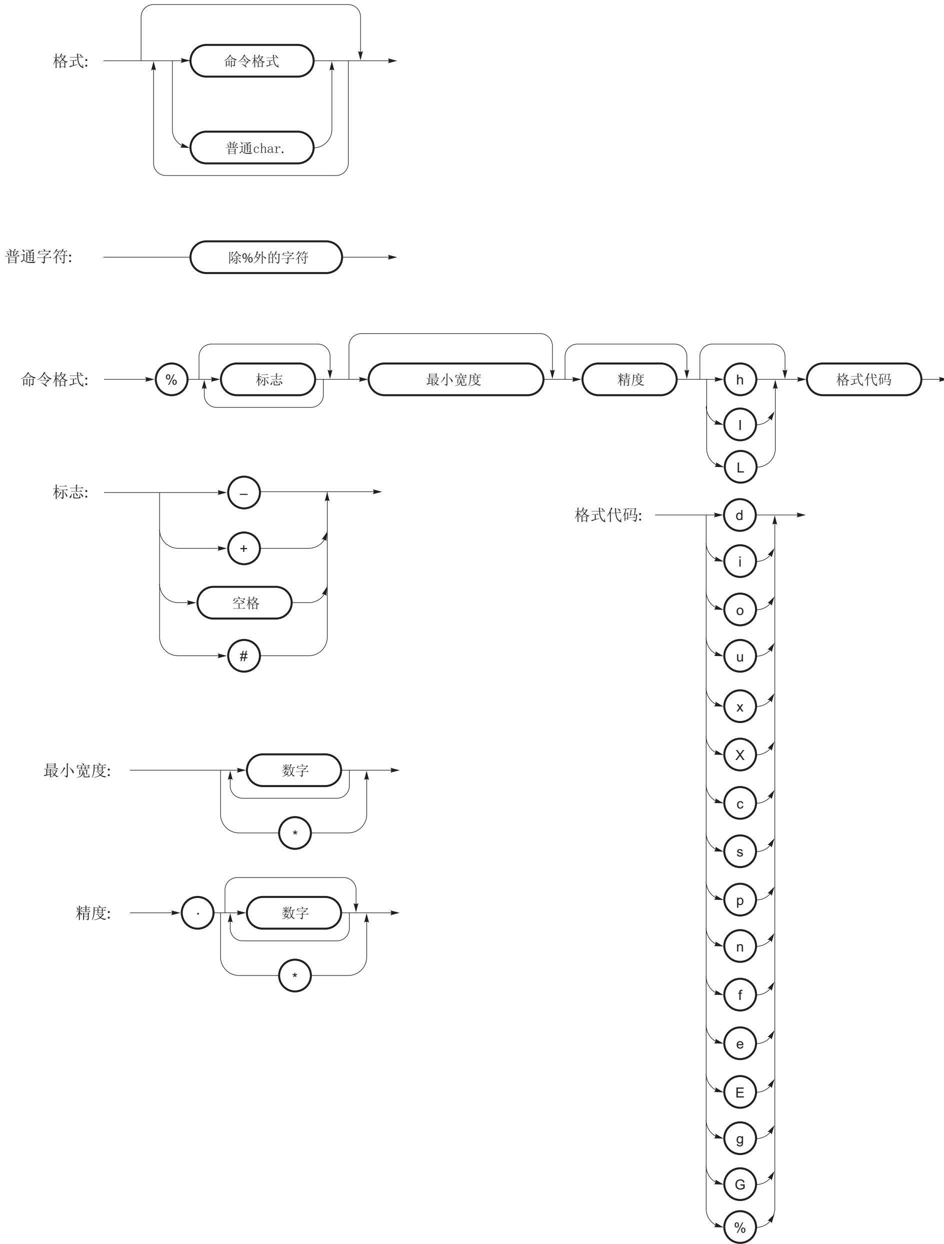
  非数值 → “(NaN)”

  +∞ → “(+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 命令的语法如下所示。



- 在 CC78K0R 编译器中，在指定指针作为参数的转换说明符 s, p 和 n 使用中等模式时必须指定 near 数据指针（2 字节长度），在大型模式下必须指定 far 数据指针（4 字节长度）。
  在各种模式下，函数指针长度固定为 4 字节，但在将指针作为参数时，各种模式下的指针长度必须指定为 2- 或 4-字节。

sscanf

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 **format** ... 该指针指向的字符串用来说明输入格式命令的规格 **...** ... 指向转换值完成后存储对象的指针，以及零个（或更多）的参数	若字符串 **s** 为空则返回–1。 若字符串 s 非空，则返回输入数据项指定的分配数量

说明

- **sscanf** 从 **s** 所指的字符串输入数据。由 **format** 所指的字符串才可以被指定为允许进行输入的字符串。**format** 后面的零个（或更多）参数用作指向对象的指针。**format** 指定如何从输入字符串进行数据的转换。
- 如果参数的数量少于由 **format** 所指向的格式命令，那么就无法保证编译器的正确运行。
  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，会进行表达式评估但不会有数据输入。
- 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** 修饰词，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** 在所有的格式说明符之前，那么参数就是指向 **long int** 的指针。

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：用来说明相应转换类型的字符（见后文描述）

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（见后文）。格式命令按照下列步骤执行。
    (a) 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。
    (b) 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。
    (c) 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。

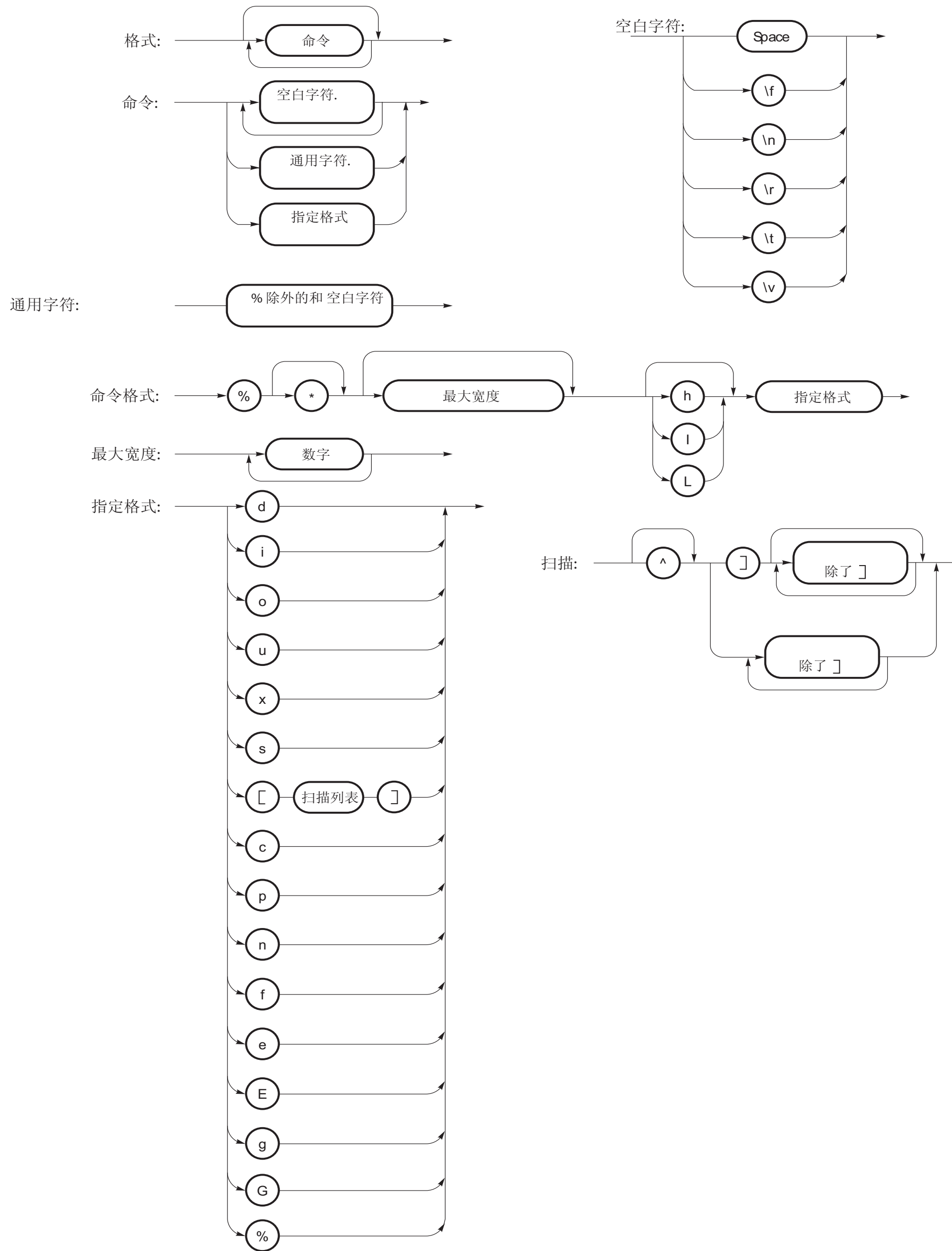
可用下列类型说明符：

转换指定符	内容
d	转换十进制整数（可能带符号）。 对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。

转换指定符	内容
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一个空的字符串结束符。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数组首元素字符的指针，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（]）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音调符号（^）时除外。 当该字符为音调符号时，在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	输入字符串中的字符数量由字段宽度指定。（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数组首字符的指针，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。
P	读取无符号十六进制整数。 对应参数必须是指向 **void** 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。由该函数从字符串“s”中读取且已经存储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。

输入格式命令的语法如下所示：



- 在 CC78K0R 编译器中，在指定指针作为参数的转换说明符 s, p 和 n 使用中等模式时必须指定 near 数据指针（2 字节长度），在大型模式下必须指定 far 数据指针（4 字节长度）。
  在各种模式下，函数指针长度固定为 4 字节，但在将指针作为参数时，各种模式下的指针长度必须指定为 2-或 4-字节。

printf

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	**format** ... 该指针指向的字符串用来说明输出转换的规格 **...** ... 要转换的 0 个或更多参数	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。

scanf

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	**format** ... 该指针指向的字符串用来说明输入转换的规格 **...** ... 指针指向用来分配转换值的对象（0 个或更多）参数	当字符串 **s** 非空时... 返回被赋值的输入项的个数

说明

- 用 **getchar** 函数进行输入。指定由 **format** 指示的字符串所许可的输入字符串。用 **format** 之后的参数作为指向对象的指针。**format** 指定如何由输入字符串进行转换。
- 当没有足够的参数供 **format** 使用时，无法保证正常操作。当参数个数大于 **format** 指定的参数数量时，会对表达式评估但不会有实际输入。
- **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指定的输入字符相冲突，则冲突的输入字符被向下舍去。转换的各种指示与 **sscanf** 函数的相同。

vprintf

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	**format** ... 该指针指向的字符串用来说明输出转换的规格 **p** ... 指向参数列表的指针	输出字符的个数（末尾的空字符不计数）

说明

- 按照格式规格中的输出转换说明，用 **putchar** 函数转换并输出参数列表中的指针所指参数。
- 各转换说明与 **sprintf** 函数的情况相同。

vsprintf

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** ... 指针指向作为输出的字符串 **format** ... 该指针指向的字符串用来说明输出转换的规格 **p** ... 指向参数列表的指针	输出到 **s** 的字符的个数（末尾空字符不计数）

说明

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

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	无	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。
- 要想改变在 SFR 中的读取位置，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 **getchar** 函数。

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	通常 ... **s** 若在未读取字符时就发现文件末尾 ... 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。

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** ... 要输出的字符	字符输出

说明

- 把 **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。

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	通常 ... 0 当 **putchar** 函数返回–1 时 ... –1

说明

- 用 **putchar** 函数将 **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。
- 当返回值正常时，返回 0；当 **putchar** 函数返回–1 时，返回–1。

__putc

功能

__putc 输出一个字符到输出设备。

头文件

stdio.h

函数原型

int __putc(int c, void *opaque);

函数	参数	返回值
__putc	**c** : 待输出的字符. opaque.: 指向字符输出目标地址的指针	已经输出的字符.

说明

- 用**__putc** 函数将 c 指定的字符（转换为无符号字符型）输出到 opaque 所指向的目标地址。opaque 所指向的目标地址增加的单位是一个字节。
- 如果 opaque 为 0，会调用 putchar 函数，返回 putchar 函数的返回值。

10.9 应用函数

有如下应用函数可用

- atoi, atol
- strtol, strtoul
- calloc
- free
- malloc
- realloc
- abort
- atexit, exit
- abs, labs
- div, ldiv
- brk, sbrk
- atof, strtod
- itoa, ltoa, ultoa
- rand, srand
- bsearch
- qsort
- strbrk
- strsbrk
- stritoa, strltoa, strultoa

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	• 若正确转换，返回 **int** 值。 • 若出现正上溢，返回 **INT_MAX**（32767） • 若出现负上溢，返回 **INT_MIN**（–32768） • 若字符串无效，返回 0
atol		• 若正确转换，返回 **long int** 值； • 若为正上溢，返回 **LONG_MAX**（2147483647）； • 若为负上溢，返回 **LONG_MIN**（–2147483648）； • 若字符串无效，返回 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字字符号或空字符串结束符为止）。如果在字符串中未发现可以转换的数字符号，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字字符号或空字符串结束符为止）。如果在字符串中未发现可以转换的数字符号，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）。

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** ... 存储指向不可识别段的指针的指针 **base** ... 指定的基数	• 若正确转换，返回 **long int** 值 • 出现正溢出时返回 **LONG_MAX**（2147483647） • 出现负溢出时返回 **LONG_MIN**（–2147483648） • 若未转换则返回 0
strtoul		• 若正确转换，返回 **unsigned long** • 出现上溢时返回 **ULONG_MAX**（4294967295U） • 若未转换则返回 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。
  (1) 空白字符串，可能为空（由 **isspace** 来判定）
  (2) 以由 base 值确定的基来表示的整型
  (3) 无法识别的一个或多个字符组成的字符串（包括空字符串结束符）
  备注 strtol 函数将字符串的第(2)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），这些基被用来代表 10 至 35。
  若基为 16，则会忽略在前的 0x 或 0X。

- 若 **endptr** 不是空指针，则指向该字符串第(3)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 (2)**。
- 若字符串(2)为空或字符串(2)的首个非空白字符在给定的基下不适合当作整型使用，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。
  (1) 空白字符串，可能为空（由 **isspace** 来判定）
  (2) 以由 **base** 值确定的基来表示的整型
  (3) 无法识别的一个或多个字符组成的字符串（包括空字符串结束符）

  strtoul 函数将此字符串的第(2)部分转换为无符号整型，并返回该无符号整型的值。
- 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(3)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** (2)。
- 若字符串(2)为空或字符串(2)的首个非空白字符在给定的基下不适合当作整型使用，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。

calloc

功能

存储器函数 **calloc** 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** ... 数组中成员的个数 **size** ... 各成员的大小	• 若分配了所请求的大小，则返回指向已分配区域开始处的指针 • 若未分配所请求的大小，返回空指针

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。如果新的间断值是奇数，calloc 函数会将其修正为偶数。存储器函数 **brk** 的间断值设置见 **brk, sbrk**。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 calloc_m 和 calloc_f 函数都不可用。

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	无

说明

- **free** 函数释放由 **ptr** 指向的分配空间（在间断值前）。（在 **free** 之后调用 **malloc**、**calloc** 或 **realloc** 函数将从 **ptr** 中分配空间。）
- 若 **ptr** 未指向已分配空间，则 **free** 不会有任何效果。（通过将 **ptr** 设置为新的间断值，对已分配的空间进行释放。）
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 free_m 和 free_f 函数都不可用。

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	**size** ... 要分配的存储块的大小	• 若分配了所请求的大小，则返回指向已分配区域开始处的指针 • 若未能分配所请求的大小，返回空指针

说明

- **malloc** 函数会分配一个存储块，其大小由 **size** 指定的字节数决定，并返回一个指向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。如果新的间断值是奇数，calloc 函数会将其修正为偶数。存储器函数 **brk** 的间断值设置见 **10.9 应用函数 （11）brk, sbrk**。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 malloc_m 和 malloc_f 函数都不可用。

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** ... 指针指向先前分配的存储块开头 **size** ... 要给此存储块设定的新长度	• 若重新分配了所请求的大小，则返回指向重新分配空间开头的指针 • 若 **ptr** 为空指针，则返回指向已分配空间开头的指针 • 若所请求的大小未得到重新分配，或“ptr”不是空指针，则返回空指针

说明

- **realloc** 函数将 **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，那么该函数就无法执行，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间断值。如果新的间断值是奇数，calloc 函数会将其修正为偶数。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 realloc_m 和 realloc_f 函数都不可用。

abort

功能

程序控制函数 **abort** 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	无	无返回值

说明

- **abort** 函数循环运行，永远无法返回到其调用方。
- 用户必须创建 **abort** 处理例程。

atexit，exit

功能

atexit 对在正常终止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	• 若函数注册为完整结束（wrap-up）函数，则返回 0 • 若函数无法注册，则返回 1
exit	**status** ... 指示终止的状态值	exit 永远无法返回。

说明

atexit

- **atexit** 函数注册由 **func** 指向的完整结束函数，该函数可以在正常程序终止时无参数调用，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 个完整结束函数。若完整结束函数可以注册，则 **atexit** 返回 0。若因已注册满了 32 个完整结束函数而不能再注册更多完整结束函数，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方。
- 用户必须创建 **exit** 处理例程。

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**);

<table>
<tr><th>函数</th><th>参数</th><th>返回值</th></tr>
<tr><td>abs</td><td rowspan="2">j ... 要得到的任意有符号整形数的绝对值</td><td>• 若 j 在下列范围内，则返回 j 的绝对值
–32767 ≤ j ≤ 32767
• 若 j 为–32768，则返回–32768（0x8000）</td></tr>
<tr><td>labs</td><td>• 若 j 在下列范围内，则返回 j 的绝对值
–2147483647 ≤ j ≤ 2147483647
• 若 j 的值为–2147483648，则返回–2147483648（0x80000000）</td></tr>
</table>

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。

div，ldiv

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 **denom** ... 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 **numer** 的绝对值除以 **denom** 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 **numer** 和 **denom** 符号相同则为正；否则为负）。
- 余数为 **numer – denom***商的值。
- 若 **denom** 为 0，则商为 0，余数为 numer。
- 若 **numer** 为–32768 且 denom 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 **numer** 的绝对值除以 **denom** 的绝对值所得的最大 long int 类型整数。余数的符号总是与除法结果相同（若 **numer** 和 **denom** 符号相同则为正；否则为负）。
- 余数为（**numer – denom *** 商）的值。
- 若 **denom** 为 0，则商为 0，余数为 **numer**。
- 若 **numer** 为–2147483648 且 **denom** 为–1，则商为–2147483648，余数为 0。

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** ... 用来设置存储块释放位置的中断值	• 若正确设置中断值，返回 0 • 若无法改变中断值，返回–1
sbrk	**incr** ... 设置的中断值要增加/减小的值（字节）	• 若正确进行增加或减小，则返回原中断值 • 若原中断值无法增加或减小，则返回–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)。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 brk_m 和 brk_f 函数都不可用。

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。）

如果为 incr 指定的是奇数，sbrk 函数会将其修正为偶数。

- 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)**。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 sbrk_m 和 sbrk_f 函数都不可用。

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	• 若正确转换，则返回转换后的值 • 若出现正上溢，则返回 **HUGE_VAL**（带上溢值的符号） • 若出现负上溢，则返回 0 • 若字符串无效，返回 0
strtod	**nptr** ... 要转换的字符串 **endptr** … 存储指向不可识别块的指针的指针	• 若正确转换，则返回转换后的值 • 若出现正上溢，则返回 **HUGE_VAL**（带上溢值的符号） • 若出现负上溢，则返回 0 • 若字符串无效，返回 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）。
- 当正确转换时返回一个浮点数。
- 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，那么分别会返回非标准数和±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）。
  如果字符串的开始字符不满足这个格式要求，scan 过程就结束。如果 endptr 不是空指针，以某字符开始的指针被存放在 endptr 中，该字符可能为空（blank）。
- 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。并且此时 endptr 存储的指针指向下一个字符串。
- 如果转换无法进行，则返回 0。

itoa，ltoa，ultoa

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** ... 整数要转换成的字符串 **string** ... 指向转换结果的指针 **radix** ... 输出字符串所使用的基	• 若正确转换，返回指向转换后字符串的指针 • 若未正确转换，返回空指针

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。
- 输出字符串的基由 **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。

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	无	从 0 至 **RAND_MAX** 的伪随机整数
srand	**seed** ... 伪随机数发生器的初始值	无

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** 用来设置随机数计算过程的起点，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**，与在调用 **srand** 之后且 **seed** = 1 时再调用 **rand** 相同。（默认 **seed** 为 1）

bsearch

功能

bsearch 函数进行二进制检索。

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** ... 指向进行检索的关键字的指针 **base** ... 指针指向包含检索信息的数组 **nmemb** ... 数组元素的个数 **size** ... 数组长度 **compare** ... 指针指向用来比较两个关键字的函数	• 若数组包含关键字，返回的指针指向第一个和 “key” 匹配的成员 • 若数组中不包含关键字，返回空指针

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** 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，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。
  - 若两参数相等，返回 0
  - 若第一参数大于第二参数，返回正整数。

qsort

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	**base** ... 指向要排序数组的指针 **nmemb** ... 数组中成员的个数 **size** ... 数组成员的长度 **compare** ... 指向用来比较两个关键字的函数的指针	无

说明

- qsort 函数对 **base** 指向的数组成员进行升序排序。
  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** 指向的函数取出两个参数（数组元素 1 和 2）进行比较，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。

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	**ends** ... 要设置的中断值	通常 ... 0 若无法改变中断值 ... –1

说明

- 将 **endds** 给出的值设置为中断值（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。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 strbrk_m 和 strbrk_f 函数都不可用。

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** ... 中断值的增大/减小量	通常... 原中断值 当无法增大/减小中断值时 ... –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。
- 由于 CC78K0R 分配的区域存在于内部 RAM 中，参数 ptr 总是 near 型指针，于是 strsbrk_m 和 strsbrk_f 函数都不可用。

stritoa，strltoa，strultoa

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** ... 要转换的字符串 **string** ... 指向转换结果的指针 **radix** ... 指定的基数	正常... 指向转换后字符串的指针 其它情况 ... 空指针

说明

- 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**，返回的指针会指向转换后字符串。
- **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。

10.10字符串/存储器函数

有如下字符串/存储器函数可用

- memcpy, memmove
- strcpy, strncpy
- strcat, strncat
- memcmp
- strcmp, strncmp
- memchr
- strchr, strrchr
- strspn, strcspn
- strpbrk
- strstr
- strtok
- memset
- strerror
- - strlen
- strcoll
- strxfrm

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** ... 指向数据待拷贝的目的对象的指针 **s2** ... 指向包含要拷贝数据的源对象的指针 **n** ... 要拷贝的字符数量	**s1** 的值

说明

memcpy

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

memmove

- **memmove** 函数也从 **s2** 指向的对象中将 **n** 个连续字节拷贝到 **s1** 指向的对象中。
- 即使 **s1** 和 **s2** 重叠，该函数也能正确进行拷贝。

strcpy，strncpy

功能

字符串函数 **strcpy** 用来将一个字符串的内容拷贝到另一个字符串中。

字符串函数 **strncpy** 用来将不超过指定数量的字符从一个字符串拷贝到另一个字符串中。

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, **strncpy**	**s1**... 指向拷贝目的数组的指针 **s2** ... 指向拷贝源数组的指针 **n** ... 要拷贝的字符数量	**s1** 的值

说明

strcpy

- **strcpy** 函数将 **s2** 指向的字符串内容拷贝到 **s1** 指向的数组中（包括终止字符）。
- 若 **s2 < s1** ≤ （**s2 +** 要拷贝的字符数量），则无法保证 **strcpy** 的行为（由于拷贝是从头开始，而不是从指定的字符串按顺序开始进行）。

strncpy

- **strncpy** 函数从 **s2** 指向的字符串中将不超过 **n** 数量的字符拷贝到 **s1** 指向的数组中。
- 若 **s2 < s1** ≤ （**s2 +** 要拷贝的字符长度或 **s2** 的最小值**+ n – 1**），则无法保证 **strncpy** 的行为（由于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**，那么就会在 **s1** 的末尾添加空字符直到拷贝了 **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。

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**... 此指针指向目的字符串，另一个字符串（S2）将会连接到该字符串 **s2** ... 此指针指向源字符串，将被拷贝并连接到另一个字符串（S1）去。	**s1** 的值
strncat	**s1**... 此指针指向目的字符串，另一个字符串（S2）将会连接到该字符串 **s2** ... 此指针指向源字符串，将被拷贝并连接到另一个字符串（S1）去。 **n** ... 要连接的字符数量	**s1** 的值

说明

strcat

- **strcat** 函数将 **s2** 指向的字符串拷贝到（包括空终结符）**s1** 指向的字符串中。原先在 **s1** 末尾的空终结符被 **s2** 的第一个字符覆盖。
- 当拷贝对象相互重叠时，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。
- 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空终结符。如果字符数大于 n 指定的值，则从顶部开始拷贝 n 个字符。
- 空终结符一定会添加。

- 当拷贝对象相互重叠时，操作无保证。

memcmp

功能

存储器函数 **memcmp** 将两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** ... 指向要比较的两个数据对象的指针 **n** ... 比较的字符数	• 若 **s1** 和 **s2** 相等，返回 0 • 若 **s1** 大于 **s2**，返回正值；若 **s1** 小于 **s2**，返回负值 (**s1 – s2**)

说明

- **memcmp** 函数就 **n** 指定的字节数，将 **s1** 指向的数据对象和 **s2** 指向的数据对象进行比较。
- 如果两个对象的 n 个字符都相等，则函数返回 0。
- 如果两个对象的 n 个字符不相等，memcmp 函数返回值为整数差值（s1 字母 – s2 字母），若对象 **s1** 大于对象 **s2**，函数返回正值，若 **s1** 小于 **s2**，返回负值。

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**... 指向待比较的第一个字符串的指针 **s2** ... 指向另一个待比较的字符串的指针	• 若 **s1** 等于 **s2**，返回 0 • 若 **s1** 小于或大于 **s2**，则返回首个差异字符的整数值 (**s1 – s2**)
strncmp	**s1**... 指向待比较的第一个字符串的指针 **s2** ... 指向另一个待比较的字符串的指针 **n** ... 待比较的字符数量	• 若在 **n** 指定数量的字符范围内 **s1** 等于 **s2**，返回 0 • 若在 **n** 指定数量的字符范围内 **s1** 小于或大于 **s2**，则返回首个差异字符的整数值 (**s1 – s2**)

说明

strcmp

- **strcmp** 函数对 **s1** 和 **s2** 分别指向的两个以空字符结尾的字符串进行比较。
- 若 **s1** 等于 **s2**，函数返回 0。如果 **s1** 小于或大于 **s2**，则函数返回小于 0 的整数（负数）或大于 0 的整数（正数）（**s1 – s2**），该数值是首个差异字符转换而来的整数值。

strncmp

- **strncmp** 函数对 **s1** 和 **s2** 分别指向的两个以空字符结尾的字符串中的不超过 **n** 个字符进行比较。
- 若在指定字符范围内 **s1** 等于 **s2**，函数返回 0。如果在指定字符范围内 **s1** 小于或大于 **s2**，则函数返回小于 0 的整数（负数）或大于 0 的整数（正数）（**s1 – s2**），该数值是首个差异字符转换而来的整数值。

memchr

功能

存储器函数 **memchr** 将指定字符转换为 **unsigned char**，在对象给定长度中进行搜索，并返回指针指向首次出现的该字符。

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** ... 指针指向待搜索的存储器对象 **c** ... 待搜索的字符 **n** ... 待搜索的字节数	• 若找到 **c**，返回指向首次出现的 **c** 的指针 • 若未找到 **c**， 返回空指针

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**），然后返回一个指针，指向 **s** 指针所指的对象中从开头起 **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。

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**... 指向待搜索的字符串的指针 **c** ... 指定进行搜索的字符	• 若在 **s** 中找到 **c**，返回一个指针，指示字符串 **s** 中最先或最后出现的 **c** • 若在 **s** 中未找到 **c**，返回空指针

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。
- 如果在字符串中未找到匹配项，则函数返回空指针。

strspn，strcspn

功能

字符串函数 **strspn** 返回一个字符串中的初始子串长度，该字符串仅由该子串组成，子串中的这些字符包含在另一个字符串中。
字符串函数 **strcspn** 返回一个字符串中的初始子串长度，该字符串仅由该子串组成，子串中的这些字符不包含在另一个字符串中。

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 *2**);

函数	参数	返回值
strspn	**s1**... 指向待搜索的字符串的指针 **s2** ... 指针指向的字符串，其内容用来匹配	字符串 **s1** 中的子串的长度，该子串仅由 **s2** 字符串中的字符构成
strcspn		字符串 **s1** 中的子串的长度，该子串仅由 **s2** 字符串中不包含的字符构成

说明

strspn

- **strspn** 函数返回 **s1** 指针所指向的字符串内包含的子串长度，该子串仅由 **s2** 指针所指向的字符串中的字符组成。换句话说，如果 s1 字符串中有字符和 s2 中指定的任何一个字符都不符合，该函数返回字符串 **s1** 中该不匹配字符的第一个索引位置。
- **s2** 的空终结符不计入 **s2** 的长度。

strcspn

- **strcspn** 函数返回 **s1** 指针所指向的字符串内包含的子串长度，该子串仅由 **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 字符串中有字符和 s2 中任何一个字符相等的话，该函数返回字符串 **s1** 中第一个匹配字符的索引位置。
- **s2** 的空终结符不计入 **s2** 的长度。

strpbrk

功能

字符串函数 **strpbrk** 返回一个指针，如果待搜索的字符串的内容和指定字符串中任意字符相匹配，该指针指向待搜索的字符串中匹配的第一个字符。

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**... 指向待搜索的字符串的指针 **s2** ... 指针指向的字符串其内容用来匹配	• 若发现匹配，则返回一个指针，指向字符串 **s1** 中第一个和 **s2** 字符串中任意字符内容匹配的字符 • 若未发现匹配，则返回空指针

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 **s2** 字符串中任意字符匹配的字符。
- 如果在 **s1** 字符串中未发现 **s2** 字符串中的任何字符，则函数返回空指针。

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**... 指向待搜索的字符串的指针 **s2** ... 指针指向给定的字符串	• 若在 **s1** 中找到 **s2**，则返回的指针指向字符串 **s1** 中首次出现的 **s2** 字符串 • 若未在 **s1** 中找到 **s2**，则返回空指针 • 若 **s2** 为空字符串，则返回 **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** 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，则函数返回 **s1** 的值。

strtok

功能

字符串函数 strtok 返回的指针指向字符串中取出的记号（将其反汇编为字符串，该字符串中不包含分界符）。

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**... 指向字符串的指针，从该字符串中获得记号；或为空指针 **s2** ... 指向包含记号分界符的字符串的指针	• 若找到记号，返回的指针指向记号第一个字符 • 若无记号返回，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。
- **strtok** 函数在 **s1** 指针所指向的字符串中搜索 **s2** 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，则该字符被空字符覆盖，且指向下个字符的指针将会被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。

memset

功能

存储器函数 **memset** 用指定字符对存储器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** ... 指向存储器中待初始化的对象的指针 **c** ... 该字符的值将赋给各个字节 **n** ... 待初始化的字节数量	**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** 指针所指向的对象开始位置起连续赋给 **n** 个字节。

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** ... 错误编号	• 若与错误编号相关的信息是存在的，则返回的指针指向描述错误信息的字符串 • 若没有与错误编号相关的信息，返回空指针

说明

- **strerror** 函数返回一个指针，指向与 **errnum** 值相关的下列字符串之一。

错误编号	返回值
0	指针指向字母串 “错误 0”
1 (EDOM)	指针指向字母串 “参数过大”
2 (ERANGE)	指针指向字母串 “结果过大”
3 (ENOMEM)	指针指向字母串 “存储空间不够”
其它	空指针

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长度。

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** 相等时 ... 0 当字符串 **s1** 和 **s2** 不等时 ... 首个不同的字符转换为整数后的差值（**s1** 的字符 **–** **s2** 的字符）

说明

- CC78K0R 编译器不支持文化领域（特殊字符）的特定操作。
  操作与 **strcmp** 相同。

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** ... **s1** 中的最大字符数量	返回转换得到的字符串的长度（不包含指示结尾的字符串） 若返回值为 n（或更多），则 **s1** 所指示的数组内容无定义。

说明

- CC78K0R 编译器不支持文化领域（特殊字符）的特定操作。
  操作与下列函数相同。
  strncpy (**s1**, **s2**, **c**) ;
  return (**strlen** (**s2**)) ;

10.11 数学函数

有如下数学函数可用

- memcpy, memmove
- acos
- asin
- atan
- atan2
- cos
- sin
- tan
- cosh
- sinh
- tanh
- exp
- frexp
- ldexp
- log
- log10
- modf
- pow
- sqrt
- ceil
- fabs
- floor
- fmod
- matherr
- acosf
- asinf
- atanf
- atan2f
- cosf
- sinf
- tanf

- coshf
- sinhf
- tanhf
- expf
- frexpf
- ldexpf
- logf
- log10f
- modff
- powf
- sqrtf
- ceilf
- fabsf
- floorf
- fmodf

acos

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** ... 进行操作的数值	当 –1 ≤ **x** ≤ 1 时 ... 返回 **x** 的 **acos** 当 **x** < –1 或 1 < **x** 时，, **x** = NaN 时 ... **NaN**

说明

- 计算 **x** 的 **acos** （从 0 到 π 范围内）。
- 当出现 **x** < –1、1 < **x** 的定义域错误时，返回 **NaN** ，设置 **EDOM** 。
- 若 **x** 非数值，返回 **NaN** 。

asin

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** ... 进行操作的数值	当 –1 ≤ **x** ≤ 1 时 ... **x** 的 **asin** 当 x < –1，1 < x，x = NaN ... **NaN** 当 **x** = –0 时 ... –0 当出现下溢时 ... 非标准数

说明

- 计算 **x** 的 **asin** 值（在–π/2 和+π/2 之间）。
- 当出现 **x** < –1、1 < **x** 的定义域错误时，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。
- 若转换后产生下溢，则返回非标准数。

atan

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** ... 进行操作的数值	通常... **x** 的 **atan** 当 **x =** NaN 时... **NaN** 当 **x =** –0 时... –0 当出现下溢时 ... 非标准数

说明

- 计算 **x** 的 **atan** 值（在–π/2 到+π/2 范围内）。
- 当 **x** 非数值时，返回 **NaN**。
- 当 **x** 为–0 时，返回–0。
- 若转换后产生下溢，则返回非标准数。

atan2

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** ... 进行操作的数值 **y** ... 进行操作的数值	通常 ... **y/x** 的 **atan** 当 **x** 和 **y** 均为 0 或 **y/x** 的值无法表示时，或者当 **x** 或 **y** 为 NaN 及 **x** 和 **y** 均为± ∞时 ... **NaN** 当出现下溢时 ... 非标准数

说明

- 计算 **y/x** 的 **atan** （在–π到+π范围内）。
- 如果 **x** 和 **y** 均为 0 或 **y/x** 的值无法表示，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。
- 若操作结果产生下溢，则返回非标准数。

cos

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** ... 进行操作的数值	通常 ... **x** 的 **cos** 当 **x** = NaN、**x** =± ∞时 ... **NaN**

说明

- 计算 **x** 的 **cos** 。
- 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。

sin

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** ... 进行操作的数值	通常 ... **x** 的 **sin** 当 **x** = NaN、**x** =± ∞ 时 ... **NaN** 当出现下溢时 ... 非标准数

说明

- 计算 **x** 的 **sin**。
- 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**。
- 若操作结果产生下溢，则返回非标准数。
- 若 **x** 的绝对值极大，则操作结果的值几乎无意义。

tan

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** ... 进行操作的数值	通常 ... **x** 的 **tan** 值 当 **x** = NaN、**x** = ± ∞时 ... **NaN** 当出现下溢时 ... 非标准数

说明

- 计算的 **tan(x)** 的值。
- 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**。
- 若操作结果产生下溢，则返回非标准数。
- 若 **x** 的绝对值极大，则操作结果的值几乎无意义。

cosh

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** ... 进行操作的数值	通常 ... **x** 的 **cosh** 当 **x =** ±∞时 ... + ∞ **x** = NaN ... **NaN** 当出现上溢时 ... **HUGE_VAL** （带正号）

说明

- 计算 **x** 的 **cosh** 。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为无穷大，返回正无穷大的值。
- 如果由于操作产生了上溢，则返回带正号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。

sinh

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** ... 进行操作的数值	通常 ... **x** 的 **sinh** 当 **x** = NaN 时 ... **NaN** 当 **x** = ± ∞ 时... ± ∞ 当出现下溢时 ... **HUGE_VAL** （带溢出值的符号） 当出现下溢时... 0

说明

- 计算 **x** 的 **sinh** 。
- 若 **x** 非数值，返回 **NaN**。
- 若 **x** 为± ∞，返回± ∞。
- 如果由于操作产生了上溢，则返回带有溢出值符号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若由于操作产生了下溢，返回±0。

tanh

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** ... 进行操作的数值	通常 ...**x** 的 **tanh** 当 **x** = NaN 时 ... **NaN** 当 **x** = ± ∞ 时... ± 1 当出现下溢时 ... ± 0

说明

- 计算 **x** 的 tanh 。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为± ∞，返回±1。
- 若由于操作产生了下溢，返回±0。

exp

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** ... 进行操作的数值	通常 ... **x** 的指数函数 当 **x** = NaN 时... **NaN** 当 **x** =+∞ 时 ... + ∞ 当 **x** = —∞ 时 ... + 0 当出现下溢时 ... 非标准数 当由于下溢出现有效位丢失的情况时 +0 当出现上溢时 ... **HUGE_VQAL**（带正号）

说明

- 计算 **x** 的指数函数。
- 若 **x** 为非数值，返回 NaN。
- 若 **x** 为+∞，返回+ ∞。
- 若 **x** 为 —∞，返回+ 0。
- 若操作结果产生下溢，则返回非标准数。
- 若由于操作产生的下溢导致有效位丢失，则返回+0。
- 如果由于操作产生了上溢，则返回带正号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。

frexp

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** ... 进行操作的数值 **exp** ... 存储指数部分的指针	通常 ... **x** 的尾数 当 **x** = NaN、**x** = ± ∞ 时 ... **NaN** 当 **x** =± 0 时... ± 0

说明

- 将浮点数 **x** 分解为尾数 **m** 和指数 **n**，形如 **x** = m*2^n，返回尾数 **m**。
- 指数 **n** 存储的内容就是指针 **exp** 所指示的。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** 的值为 0。
- 若 **x** 为无穷大，则返回 **NaN**，将 **EDOM** 设置给 **errno** ，且***exp** 的值为 0。
- 若 **x** 为**±0**，则返回**±0** 且***exp** 的值为 0。

ldexp

功能

ldexp 计算 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
exp	**x** ... 进行操作的数值 **exp** ... 指数	通常 ... **x***2 ^ **exp** 当 **x** = NaN 时... **NaN** 当 **x** = ± ∞ 时 ... ± ∞ 当 **x** = ± 0 时... ± 0 当出现上溢时 ... **HUGE_VAL**（带上溢值的符号） 当出现下溢时 ... 非标准数 当由于下溢出现有效位的丢失时... ± 0

说明

- 计算 **x*2^exp**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 x 为±∞，返回±∞。
- 若 **x** 为±0，返回±0。
- 若操作结果出现上溢，则返回带上溢值符号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若操作结果产生下溢，则返回非标准数。
- 若由于操作引起的下溢导致有效位的丢失，则返回±0。

log

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** ... 进行操作的数值	通常 ... **x** 的自然对数 当 **x** < 0 时... **NaN** 当 **x** = 0 时... - ∞ 当 **x** 为非数值时 ... **NaN** 当 **x** 为无穷大时 ... +∞

说明

- 计算 **x** 的自然对数。
- 若 **x** ≤ 0，返回 **NaN**，且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回-∞，且将 **ERANGE** 设置给 **errno**。
- 当 x 为非数值时，返回 **NaN**
- 若 **x** 为+∞，返回+∞。

log10

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** ... 进行操作的数值	通常 ... **x** 的 10 底对数 当 **x**< 0 时 ... **NaN** 当 **x** = 0 时 ... - ∞ 当 **x** 为非数值时 ... **NaN** 当 **x** 为无穷大时 ... +∞

说明

- 计算 **x** 的 10 底对数。
- 当出现 **x** < 0 的定义域错误时，返回 **NaN**，并将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回—∞，并将 **ERANGE** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为+∞，返回+∞。

modf

功能

modf 计算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** ... 进行操作的数值 **iptr** ... 指向整数部分的指针	通常 ... **x** 的小数部分 当 **x** 为非数值或无穷大时... **NaN** 当 **x** 为±0 时... ±0

说明

- 将浮点数 **x** 分成小数部分和整数部分
- 返回与 **x** 符号相同的小数部分，将整数部分存储在指针 **iptr** 指示的位置。
- 若 **x** 为非数值，则返回 **NaN** 并将其存储在指针 **iptr** 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，将 **EDOM** 设置给 **errno**。
- 若 x =± 0，则将±0 存储在指针 iptr 指示的位置。

pow

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** ... 进行操作的数值 **y** ... 倍数	通常 ... **x^y** 当 **x** = NaN 或 **y** = NaN 时， 当 **x** = +∞ 且 **y** = 0 **x** < 0 且 **y** 不是整数, **x** < 0 且 **y** =±∞ **x** = 0 且 **y** ≤ 0 时 ... **NaN** 当出现上溢时... **HUGE_VAL** （带上溢值的符号） 当出现下溢时 ... 非标准数 由于下溢导致有效位丢失时 ... ±0

说明

- 计算 **x^y**。
- 当 **x = NaN** 或 **y = NaN** 时，返回 **NaN** 。
- 当 **x** = +∞ 且 **y** = 0 时，或 **x** < 0 且 **y** 不是整数时，或 **x** < 0 且 **y** =±∞，或 **x** = 0 且 **y** ≤ 0 时，返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若操作结果导致上溢，则返回带上溢值的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若出现下溢，返回非标准数。
- 若由于下溢导致有效位丢失，返回±0。

sqrt

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** ... 进行操作的数值	当 **x** ≥ 0 时 ... **x** 的平方根 当 **x** < 0 时 ... 0 当 **x = NaN** 时 ... **NaN** 当 **x** = ±0 时 ... ±0

说明

- 计算 **x** 的平方根。
- 在 **x** < 0 定义域错误的情况下，返回 0 且将 **EDOM** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为±0，返回±0。

ceil

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** ... 进行操作的数值	通常 ... 不小于 **x** 的最小整数 当 **x** 为非数值或 **x** 为无穷大时 ... **NaN** 当 **x** = –0 时 ... +0 当不小于 x 的最小整数的无法表达时 ... **x**

说明

- 计算得到不小于 **x** 的最小整数。
- 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** 为–0，返回+0。
- 当不小于 **x** 的最小整数的无法表达时，返回 **x**。

fabs

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** ... 用来计算绝对值的数值	通常 ... **x** 的绝对值 当 **x** 为非数值时... **NaN** 当 **x** = –0 时 ... +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。

floor

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** ...进行操作的数值	通常 ... 不大于 **x** 的最大整数 当 **x** 为非数值或 x 为无穷大时... **NaN** 当 **x** = –0 时... +0 当不大于 **x** 的最大整数无法表达时 ... **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** 为无穷大，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若 **x** 为–0，返回+0。
- 若不大于 **x** 的最大整数无法表达，则返回 **x**。

fmod

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** ... 进行操作的数值 **y** ... 进行操作的数值	通常 ... **x/y** 的余数 当 **x** 为非数值或 **y** 为非数值时，当 **y** 为±0 时，当 **x** 为±∞时... **NaN** 当 **x** 不等于∞且 **y** =±∞ 时 ... **x**

说明

- 计算 **x/y** 的余数并表达为 **x – i*y**。i 为整数。
- 若 **y** 不等于 0，则返回值的符号与 **x** 相同，且其绝对值小于 **y** 的绝对值。
- 若 **x** 为非数值或 **y** 为非数值，返回 **NaN**。
- 若 **y** 为±0 或 **x** =±∞，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若 **y** 为无穷大，则返回 **x**（除非 **x** 也为无穷大）。

matherr

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 ..... 指示算术异常的数值编号
	name ... 函数名
		无

说明

- 出现异常时，在标准库和运行时刻库中自动调用 matherr，用来处理浮点数。
- 当从标准库调用时，将 **EDOM** 和 **ERANGE** 设置给 **errno**。
  如下所示为算术异常类型和 **errno** 之间的关系。

类型	算术异常	设置给 **errno** 的值
1	下溢	**ERANGE**
2	丢失	**ERANGE**
3	上溢	**ERANGE**
4	除以零	**EDOM**
5	无法操作	**EDOM**

通过更改或创建 matherr，可以进行原始错误处理。

acosf

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** ... 进行操作的数值	当 –1 ≤ **x** ≤ 1 时 ... **x** 的反余弦 当 **x** < –1、1 < **x**、**x** 位非数值时 ... **NaN**

说明

- 计算 **x** 的反余弦（在 0 到 π 范围内）。
- 在出现 **x** < –1、1 < **x** 的定义域错误情况下，返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。

asinf

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** ... 进行操作的数值	当 –1 ≤ **x** ≤ 1 时 ... **x** 的反正弦 当 **x** < –1、1 < **x**、**x** = NaN 时 ... **NaN** x = –0 ... –0 当出现下溢时 ... 非标准数

说明

- 计算 **x** 的反正弦（在–*π*/2 到+*π*/2 范围内）。
- 在出现 **x** <–1、1 <**x** 的定义域错误情况下，返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** = –0，返回–0。
- 若操作结果产生下溢，则返回非标准数。

atanf

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** ... 进行操作的数值	通常 ... **x** 的反正切 当 **x = NaN** 时 ... **NaN** 当 **x =** –0 时 ... –0 当出现下溢时 ... 非标准数

说明

- 计算 **x** 的反正切（在–π/2 到+π/2 范围内）。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x =** –0，返回–0。
- 若操作结果产生下溢，则返回非标准数。

atan2f

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** ... 进行操作的数值 **y** ... 进行操作的数值	通常 ...**y/x** 的反正切 当 **x** 和 **y** 均为 0 或 **y/x** 的值无法表示时，或者当 **x** 或 **y** 为 **NaN** 及 **x** 和 **y** 均为无穷大时 ... **NaN** 当出现下溢时 ... 非标准数

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**。
- 若操作结果产生下溢，则返回非标准数。

cosf

功能

cosf 计算余弦。

头文件

math.h

函数原型

float cost (**float x**) ;

函数	参数	返回值
cosf	**x** ... 进行操作的数值	通常 ... **x** 的余弦 当 **x** = NaN、**x** 为无穷大时 ... **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。

sinf

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** ... 进行操作的数值	通常 ... **x** 的正弦 当 **x** = NaN、**x** 为无穷大时 ... **NaN** 当出现下溢时 ... 非标准数

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**。
- 若操作结果产生下溢，则返回非标准数。
- 若 **x** 的绝对值极大，则操作结果的值几乎无意义。

tanf

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** ... 进行操作的数值	通常 ... **x** 的正切 当 **x** = NaN、**x** 为无穷大时 ... **NaN** 当出现下溢时 ... 非标准数

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**。
- 若操作结果产生下溢，则返回非标准数。
- 若 **x** 的绝对值极大，则操作结果的值几乎无意义。

coshf

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** ... 进行操作的数值	通常 ... **x** 的双曲余弦 **x = NaN** ... **NaN** **x** 为无穷大时... ±∞ 时 当出现上溢、... **HUGE_VAL**（带正号）

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。
- 如果由于操作产生了上溢，则返回带正号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。

sinhf

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** ... 进行操作的数值	通常 ... **x** 的双曲正弦 **x** = NaN ... **NaN** 当 **x** =±∞ 时... ±∞ 当出现上溢时...**HUGE_VAL**（带上溢值的符号） 当出现下溢时 ... ±0

说明

- 计算 **x** 的双曲正弦。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为±∞，返回±∞。
- 如果由于操作产生了上溢，则返回带上溢值符号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若由于操作产生了下溢，返回±0。

tanhf

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** ... 进行操作的数值	通常... **x** 的双曲正切 **x** = **NaN** ... **NaN** 当 **x** = ±∞ 时... ±1 当出现下溢时 ... ±0

说明

- 计算 **x** 的双曲正切。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为±∞，返回±1。
- 若由于操作产生了下溢，返回±0。

expf

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** ... 进行操作的数值	通常... **x** 的指数函数 **x** = **NaN** ... **NaN** 当 **x** =+∞ 时... +∞ 当 **x** = -∞ 时... +0 当出现上溢时 ...**HUGE_VAL**（带正号） 当出现下溢时 ... 非标准数 由于下溢导致有效位丢失时 ... +0

说明

- 计算 **x** 的指数函数。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为+∞，返回+∞。
- 若 **x** 为-∞，返回+0。
- 如果由于操作产生了上溢，则返回带正号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若操作结果产生下溢，则返回非标准数。
- 若由于操作引起的下溢导致有效位的丢失，则返回+0。

frexpf

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** ... 进行操作的数值 **exp** ... 存储指数部分的指针	通常... **x** 的尾数 当 **x** = NaN、**x** =±∞ 时... **NaN** 当 **x** =±0 时... ±0

说明

- 将浮点数 **x** 分成尾数 **m** 和指数 **n**，如 **x = m***2^**n**，返回尾数 **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。
- 若 **x** 为±0，则返回±0 且***exp** 的值为 0。

ldexpf

功能

ldexpf 计算 **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** ... 进行操作的数值 **exp** ... 指数	通常... **x***2^**exp** 当 **x** = **NaN** 时... **NaN** 当 **x** =±∞ 时... ±∞ 当 **x** =±0 时... ±0 当出现上溢时 ...**HUGE_VAL**（带上溢值的符号） 当出现下溢时...非标准数 由于下溢导致有效位丢失时...±0

说明

- 计算 **x***2^**exp**。
- 若 **x** 为非数值，返回 **NaN**。若 **x** 为±∞，返回±∞。 若 x 为±0，返回±0。
- 如果由于操作产生了上溢，则返回带上溢值符号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若操作结果产生下溢，则返回非标准数。
- 若由于操作引起的下溢导致有效位的丢失，则返回±0。

logf

功能

logf 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** ... 进行操作的数值	通常... **x** 的自然对数 当 **x** < 0 时... **NaN** 当 **x** = 0 时... - ∞ 当 **x** 为非数值时... **NaN** 当 **x** 为无穷大时... +∞

说明

- 计算 **x** 的自然对数。
- 在 **x** < 0 的域错误情况下，返回 **NaN**，且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回-∞，且将 **ERANGE** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为+∞，返回+∞。

log10f

功能

log10f 计算 10 底对数。

头文件

math.h

函数原型

float log10f (**float x**) ;

函数	参数	返回值
log10f	**x** ... 进行操作的数值	通常... **x** 的 10 底对数 当 **x** < 0 时... **NaN** 当 **x** = 0 时... -∞ 当 **x** 为非数值时... **NaN** 当 **x** = +∞时... +∞

说明

- 计算 **x** 的以 10 为底的对数值。
- 在 **x** < 0 的域错误情况下，返回 **NaN**，且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回-∞，且将 **ERANGE** 设置给 **errno**。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为+∞，返回+∞。

modff

功能

modff 计算分数部分和整数部分。

头文件

math.h

函数原型

float modff (**float x**, **float *iptr**) ;

函数	参数	返回值
modff	**x** ... 进行操作的数值 **iptr** ... 整数部分的指针	通常... **x** 的分数部分 当 **x** 为非数值或无穷大时 ... **NaN** 当 **x** =±0 时... ±0

说明

- 将浮点数 **x** 分成分数部分和整数部分。
- 返回与 **x** 符号相同的分数部分，并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**。
- 若 **x** = ±0，则返回±0 并将其存储在指针 **iptr** 指示的位置。

powf

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** ... 进行操作的数值 **y** ... 乘数	通常... **x^y** 当如下情况 **x** = NaN 或 **y** = NaN 时 **x** = +∞ 且 **y** = 0 时 **x** < 0 且 **y** 不为整数时 **x** < 0 且 **y** =±∞ 时 **x** = 0 且 **y** ≤ 0 时 ... **NaN** 当出现下溢时 ... 非标准数 当出现上溢时...**HUGE_VAL** （带上溢值的符号） 由于下溢导致有效位丢失时 ...0

说明

- 计算 **x^y**。
- 当 **x = NaN** 或 **y = NaN** 时，返回 **NaN**。
- 当 **x = +∞** 且 **y =** 0、或 **x** < 0 且 **y** 不为整数时、或 **x** < 0 且 **y** =±∞ 时 、或 **x** = 0 且 **y** ≤ 0 时，返回 **NaN** 且将 EDOM 设置给 errno。
- 如果由于操作产生了上溢，则返回带上溢值符号的 **HUGE_VAL**，且将 **ERANGE** 设置给 **errno**。
- 若出现下溢，返回非标准数。
- 若由于下溢导致有效位丢失，返回±0。

sqrtf

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** ... 进行操作的数值	当 **x** ≥ 0 时... **x** 的平方根 当 **x** < 0 时... 0 当 **x = NaN** 时... **NaN** 当 **x** = ±0 时... ±0

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**。
- 若 **x** 为±0，返回±0。

ceilf

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** ... 进行操作的数值	通常... 不小于 **x** 的最小整数 当 x 为非数值或 x 为无穷大时 ... **NaN** 当 **x** = –0 时... +0 当不小于 **x** 的最小整数的无法表达时 ... **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**。

fabsf

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** ... 用来计算绝对值的数值	通常... **x** 的绝对值 当 **x** 为非数值时... **NaN** 当 **x** = –0 时... +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。

floorf

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** ... 进行操作的数值	通常... 不大于 **x** 的最大整数 当 **x** 为非数值或无穷大时 ... **NaN** 当 **x** = –0 时... +0 当不大于 **x** 的最大整数无法表达时 ... **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** 为无穷大，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若 **x** 为–0，返回+0。
- 若不大于 **x** 的最大整数无法表达，返回 **x**。

fmodf

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** ... 进行操作的数值 **y** ... 进行操作的数值	通常... **x/y** 的余数 当 **x** 为非数值或 **y** 为非数值时，当 **y** 为±0 时，当 **x** 为±∞时 ... **NaN** 当 **x** 不等于±∞ 且 **y** =±∞ 时 ... **x**

说明

- 计算 **x/y** 的余数，表达为 **x – i*y**，i 为整数。
- 若 **y** 不等于 0，则返回值的符号与 **x** 相同，且其绝对值小于 **y** 的绝对值。
- 若 **y** 为±0 或 **x** =±∞，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 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）。

10.12诊断函数

有如下诊断函数可用

- __assertfail

_ _assertfail

功能

_ _assertfail 支持 **assert** 宏。

头文件

math.h

函数原型

int _ _assertfail (char*_ _msg, char*_ _cond, char*_ _file, int_ _line) ;

函数	参数	返回值
_ _assertfail	**_ _msg** ... 指针指向的字符串指示将被传递给 **printf** 函数的输出转换规格 **_ _cond** ... **assert** 宏的实际参数 **_ _file** ... 源文件名 **_ _line** ... 源行号	未定义

说明

- **__assertfail** 函数接收来自 **assert** 宏的信息（参见 **10.2 头文件 (13) assert.h**），调用 **printf** 函数，输出信息，再调用 **abort** 函数。
- **assert** 宏给程序添加诊断功能。
  当执行 **assert** 宏时，若 **p** 为假（等于 0），则 **assert** 宏会将导致 **p** 值为假的特定调用相关的信息（信息中包括实际参数文本、源文件名、源行号，其余两个是宏_ **_FILE_** _和_ **_LINE_** _的值）传递给_ **_assertfail** 函数。

10.13 更新启动例程及库函数所需的批处理文件

CC78K0R 编译器具有用来更新部分标准库函数和启动例程的批处理文件。BAT 目录下的批处理文件如下表所示。

表 10-4 更新库函数的批处理文件

批处理文件	应用
mkstup.bat	更新启动例程（cstart*.asm）。 在更改启动例程时用批处理文件进行汇编。
reprom.bat	更新固件 ROM 终止例程（rom.asm）。 在更改 rom.asm 时，用批处理文件对库进行更新。
repgetc.bat	更新 **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。
repputc.bat	更新 **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 SM+ for 78K0R 软件。 当需要用 SM+ for 78K0R 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）的必须部分。当指定了**-QR** 选项时，用本批处理文件对库进行更新。
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）的必须部分。当未指定**-QR** 选项时，用本批处理文件对库进行更新。
repvect.bat	更新地址值，设置中断向量表的跳转表处理，该表分配位置在 flash 区域(vect*.asm). 默认将 flash 区域跳转表的最高地址设置在 2000H。如果必须要对其修改，修改 vect.inc 文件中 ITBLTOP 的 EQU 值，并使用该批处理文件对库进行更新。

10.13.1 使用批处理文件

使用 BAT 子目录下的批处理文件。

因为这些文件是用来激活汇编器和库管理程序的批处理文件，所以需要的环境必须能够运行汇编程序包 RA78K0R Ver.1.30 或更高版本。在使用批处理文件前，应用环境变量 PATH 设置包含 RA78K0R 执行格式文件的目录。

创建一个与批处理文件 BAT 同级的子目录（LIB），将汇编后文件放在此子目录下。如果 C 启动例程或库被安装在 BAT 同级的子目录 LIB 下时，这些文件就会被覆盖。

要使用批处理文件，应将当前目录移动到子目录 BAT 下并执行各个批处理文件。这时需要下列参数。

产品类型 = 芯片类型 （目标芯片的分类）

F1166A0 ··· UPD78F1166_A0 等

如下所示为使用各批处理文件方法。

各处使用的批处理文件：

(1) 启动例程

mkstup 芯片类型

<例>

```
mkstup  F1166A0
```

(2) 固件 ROM 例程的更新

reprom 芯片类型

<例>

```
reprom  F1166A0
```

(3) getchar 函数的更新

repgetc 芯片类型

<例>

```
repgetc  F1166A0
```

(4) putchar 函数的更新

repputc 芯片类型

<例>

```
repputc  F1166A0
```

(5) putchar 函数（支持 SM78K0R）的更新

repputcs 芯片类型

<例>

```
        repputcs  F1166A0
```

(6) setjmp/longjmp 函数的更新（带恢复/保存处理）

repselo 芯片类型

<例>

```
        repselo  F1166A0
```

(7) setjmp/longjmp 函数的更新（不带恢复/保存处理）

repselon 芯片类型

<例>

```
        repselon  F1166A0
```

(8) 中断向量表的更新

repvect 芯片类型

<例>

```
        repvect  F1166A0
```

第 11 章 扩展函数

本章介绍了该 C 编译器特有的扩展函数，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。

该 C 编译器的扩展函数用于生成代码，可以帮助用户高效使用 78K0R 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"有效利用编译器"**。

通过使用 C 编译器的这些扩展函数创建的 C 源程序可以更好的利用 K0R 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 78K0R 系列实时操作系统。

11.1 宏名称

该 C 编译器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过程中的选项指定，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。在以下示例中，指定了 **__K0R__** 和 **__F1166A0__**。

如需有关宏名称的详细信息，请参阅 **9.8 编译器保留定义的宏名称**。

[示例]

```
编译选项
        >CC78K0R  -CF1166A0  prime.c ...

指定设备类型：
        #pragma pc（F1166A0）
```

11.2 关键字

以下的标记已经被作为关键字添加到该 C 编译器中用来实现扩展函数。与 ANSI-C 关键字一样，这些标记不能用作标签或变量名称。所有关键字必须以小写字母形式出现，因为 C 编译器不会把对应的大写字母解释为关键字。

以下展示了被添加到该编译器的关键字列表。对于这些关键字，可以通过指定严格意义的 ANSI-C 语言规格选项(**-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. 添加的关键字列表

关键字		用法
总是可用	只有指定了-ZA 选项时可用	
_ _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** 型/_ _**boolean** 类型变量
	bit	**bit** 型变量
_ _interrupt		硬件中断
_ _interrupt_brk		软件中断
__banked, _non_banked		Bank 接口
__BANK0-15		常量地址上的 Bank 函数
_ _asm		汇编语句
_ _rtos_interrupt		为 RTOS 分配的句柄
_ _pascal		Pascal 函数
__flash		固件 ROM 函数
__flashf		**__flashf** 函数
_ _directmap		绝对地址分配
_ _temp		临时变量
__near, __far		存储器分配区域规格
__mxcall		__mxcall 函数

（1） 函数

关键字 **callt**，**__callt**，**norec**，**__leaf**，**__interrupt**，**__interrupt_brk**，**__rtos_interrupt, __flash, __flashf** 都是属性修饰词。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）;
```

属性修饰词规格仅限于以下所列各项。将 callt 和_ _callt，norec 和_ _leaf 看作相同规格。然而，即使在指定了**-ZA** 选项时，也可以继续使用以'_ _'作为名称开始的属性修饰词。

- **callt**
- **norec**
- **callt norec**
- **norec callt**
- **__interrupt**
- **_ _interrupt_brk**
- **__rtos_interrupt**
- **_ _flash**
- **_ _flashf**

注意　　如果出现 callf,__callf, noauto, __pascal 和__mxcall 的描述，会输出警告信息，描述语句被忽略。

(2）变量

- 适用于 **sreg** 或_ _**sreg** 规格的规则对 C 语言中**寄存器**同样有效（如需详细信息，请参阅 **11.5 如何使用 saddr 区域(sreg/__sreg)**）。
- 适用于 **bit**、**boolean** 或_ _**boolean** 规格的规则对 C 语言中的 **char** 或 **int** 型说明符同样有效。
  然而，这些类型只能用来指定函数外部定义的变量（外部变量）。
- 适用于**__directmap** 规格的规则对 C 语言中有关类型修饰词同样有效（如需详细信息，请参阅 **11.5 绝对地址分配规格(__directmap)**）。
- 适用于**__near** 和**__far** 规格的规则对 C 语言中有关类型修饰词同样有效（如需详细信息，请参阅 **11.5 near/far 区域规格**）。

注意　如果出现__temp 的描述，会输出警告信息，描述语句被忽略。

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

有如下几种类型的存储模式可用。

存储器模式	解释说明
小型模式 （指定了-ms 选项时选择该模式）	该存储器模式包括 64KB(max)的代码部分和 64KB 的数据部分，总共 128KB
中等模式 （指定了-mm 选项时选择该模式）	该存储器模式包括 1 MB(max)的代码部分和 64KB(max)的数据部分
紧凑模式 （指定了-mc 选项时选择该模式）	该存储器模式包括 64KB(max)的代码部分和 1 MB(max)的数据部分
大型模式 （指定了-ml 选项时选择该模式）	该存储器模式包括 1 MB(max)的代码部分和 1 MB (max)的数据部分

(2）寄存器组

- 启动时寄存器组被设置为“RB0”（在 CC78K0R 编译器的启动例程中进行相关设定）。因为这个设定，默认一直使用寄存器组 0，除非寄存器组被改变。
- 指定的寄存器组可以在中断函数的开始处设置，用来说明寄存器组的改变。

(3）存储空间

CC78K0R 编译器使用存储空间的描述如下。

图 11-1 存储空间的利用

地址		用途		大小（字节）
00	080 至 0BFH	CALLT 表		64
FF	E20 至 EB3H	**sreg** 变量，**boolean** 型变量		148
FF	EB4 至 EC3H	寄存器变量		16
FF	EC4 至 ECBH	**norec** 函数的参数		8
FF	ECC 至 ED3H	**norec** 函数的自动变量		8
FF	ED4 至 ED7H	区段信息(segment infomation)		4
FF	ED8 至 EDFH	运行时刻库的参数		8
FF	EE0 至 EF7H	RB3-RB1	工作寄存器[注]	24
	EF8 至 EFFH	RB0	工作寄存器	8
FF	F00 至 FFFH	**sfr** 变量		256
F0	000 至 7FFH	第 2**sfr** 变量		最大 2048 [注 2]

注 1 当指定了某个寄存器组时使用。

注 2 根据所使用设备不同而不同。

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，通知编译器使用它自己的方法翻译。

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，则如果 C 源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 源程序中的该关键字，或用**#ifdef** 指令对其分类处理。

CC78K0R 编译器支持以下**#pragma** 指令，借助这些指令可以实现扩展函数。

#pragma 之后指定的关键字用大写或小写字母均可。

有关使用**#pragma** 指令的扩展函数，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	描述 C 源程序中的 SFR 名称→ **11.5 如何使用 sfr 区域**
#pragma vect **#pragma interrupt**	描述C源程序中处理的中断 → **11.5 中断函数（#pragma vect /#pragma interrupt）**
#pragma di **#pragma ei**	描述 C 源程序中 DI/EI 指令 → **11.5 中断函数（#pragma DI,，#pragma EI）**
#pragma halt **#pragma stop** **#pragma brk** **#pragma nop**	描述 C 源程序中的 CPU 控制指令 → **11.5（12） CPU 控制指令（#pragma HALT/STOP/BRK/NOP）**
#pragma section	改变编译器输出区段的名称，并指定区段的位置 → **11.5 改变编译器输出部分名称（#pragma section …）**
#pragma name	改变模块名称 → **11.5 模块名称改变函数（#pragma name）**
#pragma rot	使用移位函数 → **11.5 移位函数（#pragma rot）**
#pragma mul	使用乘法函数→ **11.5 乘法函数（#pragma mul）**
#pragma div	使用除法函数 → **11.5 除法函数（#pragma div）**
#pragma opc	使用数据插入函数 → **11.5 数据插入函数（#pragma opc）**

#pragma 指令	应用
#pragma rtos_interrupt	使用实时操作系统 RX78K0R（RTOS）的中断处理程序 → **11.5 实时操作系统（RTOS）的中断处理程序（#pragma rtos_interrupt ...）**
#pragma rtos_task	使用实时操作系统 RX78K0R（RTOS）的任务函数 → **11.5 实时操作系统（RTOS）的任务函数（#pragma rtos_task）**
#pragma ext_table	指定 flash 区域跳转表的首地址 → **11.5 flash 区域跳转表(#pragma ext_table)**
#pragma ext_func	从 boot 区域到 flash 区域的函数调用 → **11.5 从 boot 区域到 flash 区域的函数调用（#pragma ext_func）**
#pragma inline	扩展标准库 memcpy 和 memset 内联函数 → **11.5 存储器处理功能（#pragma inline）**

11.5 如何使用扩展函数

本节按以下形式介绍了每一种扩展函数。

(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) norec函数(norec)

(6) 位变量与布尔型变量(bit / boolean / __boolean)

(7) ASM 语句 (#asm #endasm / __asm)

(8) Kanji(2-字节字符)（/*kanji*/, //kanji）

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

(10) 中断函数修饰词(__interrupt, __interrupt_brk)

(11) 中断函数(#pragma DI, #pragma EI)

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

(13) 位域声明

(14) 更改编译器输出区名称的函数(#pragma section ...)

(15) 二进制常量描述函数(Binary constant 0bxxx)

(16) 模块名更改函数(#pragma name)

(17) 循环移位函数(#pragma rot)

(18) 乘法函数(#pragma mul)

(19) 除法函数(#pragma div)

(20) 数据插入函数(#pragma opc)

(21) 实时操作系统（RTOS）的中断处理程序(RTOS) (#pragma rtos_interrupt ...)

(22) 实时操作系统（RTOS）的中断处理程序修饰词(RTOS) (__rtos_interrupt)

(23) 实时操作系统（RTOS）的任务函数(#pragma rtos_task)

(24) Flash区域分配方法(-ZF)

(25) Flash区域跳转表(#pragma ext_table)

(26) 从boot区域到flash区域的函数调用(#pragma ext_func)

(27) 固件ROM函数(__flash)

(28) 参数/返回值的 int 扩展限制方法(-ZB)

(29) 内存操控函数(#pragma inline)

(30) 绝对地址分配规格(__directmap)

(31) near/far区域规格

(32) 存储器模式规格

本节按照如下格式对每个扩展函数进行描述说明。

功能：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效果。

用法：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：介绍了扩展函数使用的限制（如果存在的话）。

说明：对以上应用示例进行详细说明。

兼容性：介绍了该编译器的兼容性，对由其他 C 编译器开发的 C 源程序是否可以通过该 C 编译器的编译。

callt 函数（**callt/_ _callt**）

功能

- **callt** 指令会将被调用函数的地址存储在称作 **callt** 表的区域[80H 至 BFH]中，于是使用更短的代码来调用该函数，比直接调用该函数使用的代码更小。
- 要调用由 **callt**（或**__ callt**）（称作 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

给要调用的函数（开始时描述）添加 **callt/_ _callt** 属性，方法如下所示。

```
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

- 不论使用何种存储器模式，callt 函数都会被分配到[C0H 至 0FFFFH]区域内。
- 每个有 **callt/_ _ callt** 关键字声明的函数，其地址将会在链接目标模块时被分配到 **callt** 表中。因此，在汇编源代码模块中使用这个 **callt** 表时，要创建的例程必须用符号说明为“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 80H 至 BFH。
- 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 属性数量和链接模块中允许的总数量如下表所示。

选项	-QL1	-QL2
callt 属性函数数量	32	30

- 其中不使用**-QL** 选项的默认选项如下所示。

callt 函数	限定值
每个加载模块中允许的数量	最大值 32
链接的模块中允许的总数量	最大值 32

示例

```
(C 源程序)
    ============ ca1.c ============  ============ ca2.c ============
    _ _callt extern int tsub ( ) ;

    void main ( )                            _ _callt int tsub ( )
    {                                        {
            int ret_val;                             int val;
            ret_val = tsub ( ) ;                     return val;
    }                                        }
```

<编译器的输出目标>

```
ca1 module
        EXTRN       ?tsub                   ; 声明
        callt       [?tsub]                 ; 调用

ca2 module
        PUBLIC      _tsub                   ; 声明
        PUBLIC      ?tsub                   ;
@@CALT  CSEG  CALLT0                        ; 分配至区块
?tsub:  DW          _tsub
@@BASE  CSEG        BASE
_tsub:                                      ; 函数定义
                    :
                                            ;函数体
                    :
```

说明

- 函数 **tsub()** 被添加了 **callt** 属性，使其可以储存在 **callt** 表中。

兼容性

<从另一种 C 编译器至该 C 编译器>

- 如果未使用关键字 **callt/_ _ callt**，则不需要修改 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。

<从该 C 编译器至另一种 C 编译器>

- 必须使用**#define**。如需详细信息，请参阅 **11.6 C 源程序的修改**。

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** 区域（**_@KREG00** 至**_@KREG15**）。在声明寄存器的模块的预处理/后处理期间对寄存器或 **saddr** 区域进行保存和恢复。
- 如需寄存器变量分配的详细信息，请参阅 **11.7** 函数调用接口。
- 寄存器变量根据引用次数被分配到寄存器 **HL** 或 **saddr** 区域[FFEB4H 至 FFEC3H]。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。

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 存储类型符声明变量。

```
register  类型-名称  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。
- **char** 占用的空间只有其他类型（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 **near** 指针情况下，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起就被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 **near** 指针来说，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起就被分配到普通存储空间。

示例

< C 源代码>

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

[寄存器变量分配到寄存器 HL 和 saddr 区域的示例]

下列标签在启动例程中声明（请参阅附录 A saddr 区域的标签列表）。

<编译器的输出目标>

```
        EXTRN _@KREG00              ; 对将要使用的 saddr 区域进行引用
@@CODEL  CSEG
_main:
        push    hl                  ; 在函数开始处保存寄存器的内容
        movw    ax, _@KREG00        ; 在函数开始处保存 saddr 区域的内容
        push    ax                  ;
;line 3 :       register int i, j;
;line 4 :       i = 0; j = 1;
        movw    hl, #00H            ; 以下代码在函数中间输出
        onew    ax
        movw    _@KREG00 , ax       ; j
; line 5 : i += j;
        addw    ax , hl
        movw    hl , ax
; line 6 :
        pop     ax                  ; 在函数结束处恢复 saddr 区域的内容
        movw    _@KREG00 , ax
        pop     hl                  ; 在函数结束处恢复寄存器的内容
        ret
        END
```

兼容性

<从另一种 C 编译器至 CC78K0R 编译器>

- 如果另一种 C 编译器支持 **register** 声明，则无需修改 C 源程序。
- 要改为 **register** 变量，在程序中添加变量的 **register** 声明。

<从 CC78K0R 编译器至另一种 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。

如何使用**saddr**区域（**sreg/_ _sreg**）

（**a**）**sreg** 声明的使用方法

功能

- 以关键字 **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[FFE20H 至 FFEB3H]，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** 变量的处理方式和 C 源代码中的普通变量处理方式相同。
- 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 FEF1FH]。该范围[FFEB4H 至 FFEDFH]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并可以提高程序执行速度。

使用方法

- 在模块中或函数内部用关键字 **sreg** 和_ _**sreg** 对变量进行声明。函数内部只有静态存储类型的变量可以成为 **sreg** 变量。

```
sreg 类型-名称 变量-名称 / sreg static 类型-名称 变量-名称
_ _sreg 类型-名称 变量-名称 / _ _sreg static 类型-名称 变量-名称
```

- 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-名称 变量-名称 / extern _ _sreg 类型-名称 变量-名称
```

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** 占用的空间只有其他类型（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。
- 当指定**-ZA** 时，仅启用_ _**sreg** 同时禁用 **sreg** 标识。
- 在 **int/short** 和 **near** 指针和指针情况下，每个加载模块最多可使用 74 个变量（当使用 **saddr** 区域[FFE20H 至 FFEB3H]时）。
  注意在使用 **bit** 型和 **boolean** 型变量、寄存器变量或 **noauto** 函数时可使用的变量数量会减少。

示例

< C 源代码>

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

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

以下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，C 编译器会输出以下代码。在此情况下， **ORG** 伪指令将无法输出。

<汇编源程序>

```
            PUBLIC  _hsmm0          ; 声明
            PUBLIC  _hsmm1          ;
            PUBLIC  _hsptr          ;

@@DATS      DSEG    SADDRP          ; 分配到段
            ORG     0FFE20H         ;
_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
subw    ax,_hsmm1
movw    _hsmm0,ax
```

兼容性

<从另一种 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另一种 C 编译器>

- 通过**#define** 进行修改。如需详细信息，请参阅 **11.6 C** 源代码的修改。这些修改使 **sreg** 变量的处理方法和普通变量的处理方法相同。

（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

- 不管是否有 **sreg** 声明，外部变量/外部 **static** 变量（除 **const** 型）自动分配到 **saddr** 区域。
- 取决于 n 的值以及是否指定 M，外部变量和外部 **static** 变量的分配可以如下表所示。

n 的值	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 near 指针型变量
4	当 n = 2 时的变量，另外加上 **long**，**unsigned long**，**float**，**double** 和 **long double** 型以及 far 指针型变量
M	结构体、共用体以及数组
当忽略时	所有变量

- 不管以上规格如何，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在（a）中描述。

指定方法

指定**-RD [n][M]**（**n**：1、2 或 4）选项。

限制

- 在**-RD [n][M]**选项中，指定不同 **n, M** 值的模块之间不能彼此连接。

（c） 内部静态变量 **saddr** 自动分配选项的使用方法（**-RS**）

功能

- 将内部静态变量（除 **const** 型）自动分配到 **saddr** 区域，而不管是否作出 **sreg** 声明。
- 取决于 n 的值以及是否指定 M，具体内部静态变量的分配情况如下所示。

n 的值	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 near 指针型变量
4	当 n = 2 时的变量，另外加上 **long**，**unsigned long**，**float**，**double** 和 **long double** 型以及 far 指针型变量
M	结构体、共用体以及数组
忽略时	所有变量（包括结构体、共用体以及只有这种情况下才可以使用的数组）

- 不管以上规格如何，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** 区域的变量其处理方法和 **sreg** 变量相同。这些变量的函数和限制如（a）所述。

规格方法

指定**-RS [n][M]**（**n**：1，2 或 4）选项。

备注　　在**-RS [n][M]**选项中，指定不同 **n,M** 值的模块之间可以彼此连接。

如何使用sfr区域（sfr）

功能

- **sfr** 指的是一组特殊函数寄存器，比如 78K0R 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** 变量进行写入检查。
- 对只写 **sfr** 变量进行读取检查。
- 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** 名称都分配在地址[FFF00H 至 FFFFFH，F0000H 至 F07FFH]组成的范围内。

  注　　根据所使用的设备不同而有所差异。

效果

- 对 **sfr 区域**的操作可以在 C 源代码中直接描述。
- 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并能提高程序执行的速度。

使用方法

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

```
#pragma  sfr
```

- **#pragma sfr** 指令必须在 C 源代码行的开始处说明。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，也没有对变量或函数的引用。
- 在 C 源程序中，对 **sfr** 名称的描述按照设备原有的名称进行（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 _ _K0R_ _
#pragma sfr
#endif

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

不输出有关声明的代码，且会有以下代码在函数中输出。

<编译器的输出目标>

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

兼容性

<从另一种 C 编译器至 CC78K0R 编译器>

- C 源程序中，和设备或编译器无关的部分无需修改。

<从 CC78K0R 编译器至另一种 C 编译器>

- 删除**#pragma sfr** 声明或通过**#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 _ _K0R_ _
#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，必须创建专用库来访问该区。

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）代码。
- **norec** 函数的参数分配到寄存器和 **saddr** 区域（FFEC4H 至 FFECBH）。
- 如果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（FFEC4H 至 FFECBH）中，并调用 **norec** 函数。
- 自动变量分配到 **saddr** 区域（FFEC4H 至 FFECBH），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** 或 **far** 型指针之外的参数，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。
  如果使用 **long/float/double/long double** 或 **far** 型指针作为参数，第一个和随后的参数都按照声明顺序存储在 saddr 区域。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，如果第一个参数类型是 **char**，**signed char**，**unsigned char**，它将被存放在寄存器 A 中。
- 如果在 **norec** 函数开始处，寄存器 **DE** 中没有存储 **norec** 传递的参数，则将存储在寄存器 **AX** 或 **A** 中的参数复制到寄存器 **DE**。如果寄存器 **DE** 中已经存储了参数，则存储在 **AX** 中的参数被复制到**_@RTARG6** 和 **_@RTARG 7**。
- 如果使用了 **long/float/double/long double** 或 **far** 型指针型之外的自动变量，则分配之后剩余的参数按引用频率存储到 **DE**，**_@RTARG6** 和**_@RTARG 7**，**_@NRARG0**，**_@NRARG 1**...
  如果使用 **long/float/double/long double** 或 **far** 型指针型的自动变量，则分配之后剩余的参数按引用频率存储到 **_@NRARG0**，**1**...
  关于 **DE**，**_@RTARG6** 和**_@RTARG 7**，**_@NRARG0**，**_@NRARG 1**，请参阅附录 A saddr 区域标签列表。

效果

- 可以缩短目标代码并改善程序执行的速度。

使用方法

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-名称 函数-名称
```

- **_ _ leaf** 可以代替 **norec** 来描述。

限制

- 从 **norec** 函数不可以调用其他函数。
- 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** 且启用_ _**leaf**。
- 编译时对参数和自动变量的限制进行检查，如果不满足则会出现错误。
- 如果以寄存器声明参数和自动变量，则忽略寄存器声明。
- 以下展示可以在 **norec** 函数中使用的参数和自动变量的类型。
  如果类型为 **char/signed char/unsigned char**，则 **norec** 函数连续分配到 **saddr** 区域，然而如果使用了其他类型，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/far** 指针型，则在 **norec** 函数中可以使用的参数数量为 2 个变量。参数不能用 **long/float/double/long double/ far** 指针型。
- 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 **long/float/double/long double/ far** 指针型之外的类型，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/ far** 指针型。

(当指定**-QR** 选项时）

- 如果使用 **long/float/double/long double/ far** 指针型之外的类型，则参数数量为 6 个变量，且如果使用 **long/float/double/long double/ far** 指针型，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/ far** 指针型之外的类型，则自动变量可以使用最多 20 个字节，如果使用 **long/float/double/long double/ far** 指针型，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条件，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        ; 声明引用后续将要使用的 saddr 区域
        EXTRN       _@NRARG6        ;
        EXTRN       _@NRARG1        ;
            :
        _@NRARG0    ← m             ; 将参数存储在 saddr 区域
            :
        de          ← 1             ; 将参数存储至 DE
            :
        ax          ← k             ; 将参数存储在 AX
        call        !_rout          ; 调用 norec 函数

_rout:
        movw        _@RTARG6，ax    ; 从 saddr 区域接收参数
        shlw        ax，2
        addw        ax，_@NRARG1    ; 使用 saddr 区域的自动变量
        movw        bc，ax          ;
        ret
```

说明

在以上示例中，在 **rout** 函数的定义时添加了 **norec** 属性，以声明该函数为 **norec** 函数。

兼容性

<从另一种 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另一种 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**。

bit型变量，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。

```
MOV1，AND1，OR1，XOR1，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级的程序，且能够以位为单位访问 **saddr** 和 **sfr** 区域。

使用方法

- 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** 还可以替代 **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** 变量可以自动当作 **bit** 型变量。

```
变量-名称.n（其中 n = 0 至 31）
```

限制

- 使用 CY（溢出）标志位来操作两个 **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。
- 仅使用 **bit** 型变量，每个加载模块最多可以使用 1184 个变量（当使用 **saddr** 区域[FFE20H 至 FFEB3H]时）。
- 位变量声明时不能赋初值。
- 如果连同 **const** 关键字一起声明变量，则忽略 **const** 声明。
- 如下表所示，仅用 0 和 1 可以进行操作符和常数的操作。

分类	操作符	分类	操作符
Assignment	=	Logical AND	&&
Bitwise AND	&，&=	Logical OR	||
Bitwise OR	|，|=	Equal	==
Bitwise XOR	^，^=	Not Equal	!=

- *，&（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**。
- 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      0FFE20H
_data1      DBIT
_data2      DBIT
```

函数中输出以下代码。

<编译器的输出目标>

```
set1        _data1                  ; 初始化
clr1        _data2                  ; 初始化
bf          _data1，$?L0001         ; 判断
mov1        CY，_data2              ; 赋值
mov1        _data1，CY              ; 赋值
bf          _data1，$?L0005         ; 逻辑与表达式
bf          _data2，$?L0005         ; 逻辑与表达式
```

兼容性

<从另一种 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另一种 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** 型变量按普通变量进行处理。）。

ASM 语句（**#asm**，**#endasm/ _ _asm**）

功能

#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 CC78K0R 编译器输出的汇编源代码文件中。
- **#asm** 和**#endasm** 行不作输出。

_ _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）。

使用方法

#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。在**#asm** 与**#endasm** 之间进行汇编源代码的描述。

```
#asm
    :           /*汇编源程序*/
#endasm
```

_ _asm

- **ASM** 声明在 C 源代码中以下列格式说明。

```
_ _asm（字符串文字）;
```

- 字符串文字的说明方法符合 ANSI 规格，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

- **#asm** 指令不允许嵌套。
- 如果使用 **ASM** 语句，则不创建目标模块文件。而是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** 或**@@CODEL** 的 **CSEG**。

示例

#asm - #endasm

<C 源代码>

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

<编译器的输出对象>

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

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

_ _**asm**

<C 源代码>

```
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。

Kanji（2-字节字符）（/* kanji */, //kanji）

功能

- Kanji 代码可以在 C 源文件的注释语句中出现。
- 注释中的 Kanji 代码被当作注释的一部分处理，所以该代码不会被编译。
- 注释中使用的 Kanji 代码可以由选项或环境变量来指定。
  如果没有指定选项，环境变量 LANG78K 中的代码集被设置为 kanji 代码。
- 如果由选项和环境变量 LANG78K 同时指定 Kanji 代码，则选项优先指定。
- 如果在环境变量 LANG78K 中设置“SJIS”，注释中的 Kanji 类型被解释为 shift JIS 代码。
- 如果在环境变量 LANG78K 中设置“EUC”，编译器将注释中的 Kanji 类型被解释为 EUC 代码。
- 如果在环境变量 LANG78K 中设置“NONE”，编译器认为注释中的不包含有 Kanji 代码。
- 默认指定为 SJIS 代码。

效果

- 使用 kanji 代码允许使用日文的程序员写出更加容易理解的注释，这样可以改善 C 源程序管理的工作

使用方法

- 通过编译选项或环境变量来设置 kanji 代码（如果使用默认参数，则无需另外设置）。

（1） 通过编译选项设置，可以设置的选项如下表所示。

选项	说明
-zs	SJIS（shift JIS 代码）
-ze	EUC（EUC 代码）
-zn	NONE（不使用 kanji 代码）

（2） 通过环境变量 LANG78K 来设置。

（a） 设置“SJIS”，“EUC”或“NONE”（如果需要，可以在相应文件中描述，比如 autoexec.bat）

（b） 指定 SJIS，EUC 或 NONE 时，大小写都可以。

（c） Kanji 字符出现在 C 源文件注释中的情况和 LANG78K 指定的设置要一致。

```
SET    LANG78K = SJIS （shift JIS 代码）
SET    LANG78K = EUC （EUC 代码）
SET    LANG78K = NONE（不使用 kanji 代码）
```

限制

- 在注释语句中只能出现 shift JIS 代码和 EUC 代码。
  在注释之外的位置，只能使用 0x7F 以内的 ASCII 码。全角字符及半角片假名（包括半角标点符号）只能出现在注释内。
  如果出现了任何上述字符，则不会输出期望的代码。

示例

<C 源代码>

```
//main 函数
void main（ ）
{
        /* 注释语句 */
}
```

Kanji 类型信息被输出到汇编源文件中。

<编译器的输出对象>

```
$KANJICODE    SJIS
```

当 C 源文件内容被输出到汇编源文件时，注释中的 kanji 字符同样被输出。

```
        ; line     1: //main 函数
        ; line     2: void main（ ）
        ; line     3: {
@@CODEL  CSEG
_main :
        ; line     4:         /* 注释语句 */
        ; line     5: }
```

说明

- Kanji 代码只能出现在 C 源文件的注释语句中。
- 当使用格式为“//注释”时，请先指定编译选项-zp。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果在注释区域之外（不是在“/* */”或“//新行”之内）出现了 kanji 代码，则必须对 C 源程序进行修改。
- 如果代码和 CC78K0R 编译器指定的类型不同，kanji 代码必须先进行转换。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 如果目标 C 编译器支持在注释中描述 kanji 字符，则无需修改 C 源程序。
- 如果目标 C 编译器不支持在注释中描述 kanji 字符，则 C 源程序中的 kanji 字符必须删除。

中断函数（#pragma vect/#pragma interrupt）

功能

- 所描述的函数名称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) 寄存器变量所在的 **saddr** 区域
(3) **norec** 函数参数/自动变量所在的 **saddr** 区域（不管使用参数还是变量）
(4) 运行时刻库所在的 **saddr** 区域（仅正常模式）
(5) 保存区段（segment）信息所在的 **saddr** 区域
(6) ES 和 CS 寄存器

请注意，这些具体内容是否执行取决于中断函数的规格或状态，保存/恢复分别进行，如下所示。

- 如果指定"不改变"，则不管是否使用这些代码，寄存器组切换的代码，以及对寄存器内容和 **saddr** 区域内容进行保存/恢复的代码不作输出。
- 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。

- 如果编译时未指定**-QR** 选项，则不使用 **norec** 函数中寄存器变量和参数/自动变量所需要的 **saddr** 区域；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的大小，则输出恢复代码。

- 下表总结了以上内容并列表展示了和保存/恢复区域。

保存/恢复区域	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**	
		堆栈	RBn	堆栈	RBn	堆栈	RBn	堆栈	RBn
使用的寄存器	NG	NG	NG	NG	NG	OK	NG	OK	NG
所有寄存器	NG	OK	NG	OK	NG	NG	NG	NG	NG
已使用的运行时刻库所需的 **saddr** 区域，ES, CS 寄存器，存储区段信息所使用的 **saddr** 区域	NG	NG	NG	NG	NG	OK	OK	OK	OK
所有运行时刻库所需的 **saddr** 区域，ES, CS 寄存器，存储区段信息所使用的 **saddr** 区域	NG	OK	OK	OK	OK	NG	NG	NG	NG
使用的寄存器变量需要的 **saddr** 区域	NG	NG	NG	OK	OK	NG	NG	OK	OK
norec 函数的参数/自动变量相关的使用的所有 **saddr** 区域	NG	NG	NG	OK	OK	NG	NG	NG	NG

堆栈：指定使用堆栈

RBn：指定寄存器组

OK： 保存

NG： 不保存

效果

- 可以用 C 源代码进行中断函数的描述。
- 因为寄存器组可以改变，则不再输出保存寄存器的代码；于是，可以缩小目标代码并改善执行速度。
- 无需知道向量表的地址，就可以识别中断请求名称。

使用方法

- 使用**#pragma** 指令来指定中断请求名称、函数名称、堆栈开关、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** 。
- 说明**#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) 既未定义又未引用变量或函数的预处理指令

```
#pragmaΔvect（或 interrupt）Δ中断请求名称Δ函数名称Δ

                [堆栈改变规格] Δ    ⎧⎧  堆栈使用规格  ⎫⎫
                                   ⎨⎨  不改变规格    ⎬⎬
                                   ⎩⎩  寄存器组规格  ⎭⎭
```

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）。 如果是软件中断 BRK，使用 BRK_I。
函数名称：	描述中断过程的函数名称。
堆栈变化规格：	SP = 数组名称 [+ 偏移位置]（示例：SP = buff + 10）。 通过 **unsigned short** 定义数组（例：unsigned short buff [5];）。 在 buff 数组范围内指定偶数值作为偏移位置。（例：在 unsigned short buff [5]时，buff 的大小是 10 个字节，所以应该指定为 10 或者更小一个偶数地址。）
堆栈使用规格：	STACK（默认）
不改变规格：	NOBANK
寄存器组规格：	RB0/ RB1/ RB2/ RB3
Δ：	空格

注意 CC78K0R 编译器在启动例程中初始化指定为第 0 组寄存器，请确认改变寄存器组时，指定为寄存器组 1-3。

限制

- 未指定**-zf** 选项时，不论存储器模式如何，中断函数的分配区域为[C0H 至 0FFFFH]。
  当指定**-zf** 选项时，中断函数的分配区域会根据存储器模式而变化，并且可以用__near 或__far 来指定分配区域。
- 在 near 区域之外的数组不能被指定为堆栈的改变基准，如果这样指定，则发生错误。
- 非偶数不能指定为偏移地址，如果这样指定，则发生错误。
- 和其它系列不同，unsigned short 类型数组为改变堆栈指针而保留。
- 中断请求名称必须以大写字母表示。
- 在一个模块内将对中断请求名称进行双重检验。
- 在如下三个条件满足时，寄存器的内容也可能会发生变化，但是编译器无法检验这个错误。
  如果指定要改变寄存器组，自行设置寄存器组保证它们不会相互重叠。如果寄存器组发生重叠，控制它们的中断来防止重叠。
  当指定 NOBANK（“不改变”规格）时，寄存器不做保存。于是，控制寄存器保证他们的内容不会丢失。
  (i) 如果发生两个或更多的中断。
  (ii) 如果发生的两个或更多的中断还同时使用了相同的 BANK。
  (iii) 如果在#pragma interrupt 语句的描述中指定了 NOBANK 或寄存器组。
- **callt/__callt /norec/__leaf /__rtos_interrupt/__flash/__flashf** 函数不能被指定为中断函数。
- 只有当指定了**-zf** 选项时，才能指定__far 属性。
- 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。
- 如果指定 **leafwork** 为 **1** 至 **16**，则输出警告，且忽略该项指定。
- 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 short** 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。可以通过**-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/寄存器组说明之前说明。如果在堆栈使用/寄存器组说明之后说明堆栈改变，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** 规格的保存目的地，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

[指定了寄存器组时]

< C 源代码 1>

```
#pragma interrupt  NMI inter rb1
void inter()
{
    /* 处理 NMI 引脚引起的中断 */
}
```

<编译器输出对象>

```
@@VECT02        CSEG  AT  0002H        ; NMI
_@vect02 :
        DW          _inter
@@BASE          CSEG  BASE
_inter :
        ; 切换寄存器组的代码
        ; 保存 ES 和 CS 寄存器
        ; 保存编译器所用的 saddr 区域的代码
        ; 处理 NMI 引脚引起的中断（函数体）
        ; 恢复编译器所用的 saddr 区域的代码
        ; 恢复 ES 和 CS 寄存器
        reti
```

[指定堆栈和寄存器组的改变]

< C 源代码>

```
#pragma interrupt           INTP0  inter  sp = buff + 10  rb2
unsigned short buff [ 10 ] ;
void func ( ) ;
void inter ( )
{
func ( ) ;
}
```

<编译器输出对象>

```
@@BASE  CSEG        BASE
_inter :
        sel         RB2                 ; 切换寄存器组
        movw        ax , sp             ; 改变堆栈指针
        movw        sp , #_buff + 10    ; "
        push        ax                  ; "
        movw        c , #0CH            ; 保存编译器所用的 saddr 区域
        dec         c                   ; "
        dec         c                   ; "
        movw        ax , _@SEGAX [ c ] ; "
        push        ax                  ; "
        bnz         $$ - 6              ; "
        mov         a , ES              ; 保存 ES 和 CS 寄存器
        mov         x , a               ; "
        mov         a , CS              ; "
        push        ax                  ; "
        call        !!_func
        pop         ax                  ; 恢复 ES 和 CS 寄存器
        mov         CS , a              ; "
        mov         a , x               ; "
        mov         CS , a              ; "
        movw        de , #_@SEGAX       ; 恢复编译器所用的 saddr 区域的代码
        mov         c , #06H            ; "
        pop         ax                  ; "
        movw        [ de ] , ax         ; "
        incw        de                  ; "
        incw        de                  ; "
        dec         c                   ; "
        bnz         $$ - 5              ; "
        pop         ax                  ; 将堆栈指针恢复到原始位置
        movw        sp , ax             ; "
        reti
@@VECT06 CSEG       AT 0006H
_@vect06 :
        DW          _inter
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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，根据以上**使用方法**中描述的过程来修改程序。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除**#pragma vect** 或**#pragma interrupt** 指令，中断函数可以用作普通函数。
- 当普通函数要用作中断函数时，根据每个编译器的具体规格来改变程序。

中断函数修饰词（__ interrupt, __ interrupt_brk）

功能

- 以_ _ **interrupt** 修饰词声明的函数被视为硬件中断函数，且不可屏蔽/可屏蔽中断函数的返回指令 RETI 可以返回。
- 以**__interrupt_brk** 修饰词声明的被视为软件中断函数，软件中断函数可以用返回指令 RETB 返回。
- 以该修饰词声明的被函数视为（不可屏蔽/可屏蔽/软件）中断函数，并将以下（1）到（6）项的寄存器和变量区保存/恢复，其用作编译器的工作区。
  但是如果在此函数中出现函数调用，则所有变量区保存到堆栈。

(1）寄存器 (2）寄存器变量所用的 saddr 区域 (3）norec 函数的参数/自动变量的 saddr 区域（不管是否使用） (4）运行时刻库的 saddr 区域 (5）保存区段（segment）信息的 **saddr** 区域 (6）ES 和 CS 寄存器

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

效果

- 通过此修饰词来声明函数，向量表和中断函数定义的设置可以不放在同一个文件中说明。

使用方法

- 用_ _ **interrupt** 或_ _ **interrupt_brk** 作为描述中断函数的修饰词。

<对于不可屏蔽/可屏蔽中断函数>

```
_ _ interrupt  void  func() {processing}
```

<对于软件中断函数>

```
_ _ interrupt_brk  void  func() {processing}
```

限制

- 未指定**-zf** 选项时，不论存储器模式如何，中断函数的分配区域为[C0H 至 0FFFFH]。
  当指定**-zf** 选项时，中断函数的分配区域会根据存储器模式而变化，并且可以用__near 或__far 来指定分配区域。
- **callt/__callt/ norec/__leaf/__rtos_interrupt/__flash/__flashf** 函数不能被指定为中断函数。

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，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**）规格，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“中断函数”的用法。

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 RB1                     /* 中断请求名称*/
#pragma interrupt   BRK_I inter_b RB2                   /* 软件中断使用“BRK_I” */
__interrupt         void inter ( ) ;                    /*原型声明*/
__interrupt_brk     void inter_b ( ) ;                  /*原型声明*/
__interrupt         void inter ( ) { processing } ;     /*函数体*/
__interrupt_brk     void inter_b ( ) { processing } ;   /*函数体*/
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另外某个 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格修改程序。

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。
- 如果忘记了**#pragma** 指令，则 **DI( )**和 **EI( )**被视为普通函数。
- 如果"**DI( );**"在函数的开始处说明（除自动变量、注释和预处理指令的声明），则在函数预处理之前输出 **DI** 对应的代码（函数名称标签之后立即进行）。
- 要函数预处理之后输出 **DI** 对应的代码，请在说明"**DI( );**"之前开辟新块（以'**{**'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（以'**{**'对此块划分界限）。

效果

- 创建的函数可以禁止中断。

使用方法

- 在 C 源代码开始处说明**#pragma DI** 和**#pragma EI** 指令。但是以下各项可以放在**#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令
- 在源代码中说明 **DI( );**或 **EI( );** 的方式和函数调用相同。
- **DI** 和 **EI** 可以在**#pragma** 之后用大写字母或小写字母表示均可。

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，则将其按普通函数进行处理。

示例

```
#ifdef      _ _K0R_ _
#pragma   DI
#pragma   EI
#endif
```

< C 源代码 **1>**

```
#pragma     DI
#pragma     EI
void  main（ ）
{
            DI（ ）;
            ; 函数体
            EI（ ）;
}
```

<编译器的输出对象**>**

```
  _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>

```
#pragma  DI
#pragma  EI
void main（ ）
{
        {
                DI（ ）;
                ; 函数体
                EI（ ）;
        }
}
```

<编译器的输出对象>

```
_main:
        ; 预处理过程
        di
        ; 函数体
        ei
        ; 后处理过程
        ret
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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**中描述的过程对程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除**#pragma DI** 和**#pragma EI** 指令或以**#ifdef** 进行删除，**DI** 和 **EI** 可以用作普通函数名称（示例：**#ifdef_ _K0R_ _ ... #endif**）。
- 要将普通函数用作中断函数，请根据每个编译器的具体规格修改程序。

CPU控制指令（#pragma HALT/STOP/BRK/NOP）

功能

- 以下代码输出到该对象以创建一个目标文件。

```
（1） HALT 操作指令（HALT）
（2） STOP 操作指令（STOP）
（3） BRK 指令
（4） NOP 指令
```

效果

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

使用方法

- 在 C 源代码开始处说明**#pragma HALT**，**#pragma STOP#pragma NOP** 和**#pragma BRK** 指令。
- 以下各项可以在**#pragma** 指令之前说明。
  (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，和函数调用的格式相同，如下表示。

```
（1） HALT（ ）;
（2） STOP（ ）;
（3） BRK（ ）;
（4） NOP（ ）;
```

限制

- 当使用此函数时，**HALT( )**，**STOP( )**，**BRK( )**和 **NOP( )**不能用作函数名称。
- HALT，STOP，BRK 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  BRK
#pragma  NOP
Void     main（ ）
{
         HALT（ ）;
         STOP（ ）;
         BRK（ ）;
         NOP（ ）;
}
```

<编译器的输出对象>

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另外某个 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。
- 要将这些指令用作 CPU 控制指令，请根据每个编译器的具体规格修改程序。

位域声明

（a） 类型说明符的扩展

功能

- **unsigned char** 型位域不可跨字节界限进行分配。
- **unsigned int** 和 **signed int** 型位域不可跨字界限进行分配，而可以跨字节界限进行分配。
- 相同类型的位域分配在同一个字节单元（或字单元）中。如果类型不同，则位域分配在不同字节单元（或字单元）中。

效果

- 可以节省内存容量，可以缩短目标代码且可以提高运行速度。

使用方法

- 作为位域类型说明符，除 **unsigned int** 型之外可以指定 **signed int**，**unsigned char** 和 **signed char** 型。

声明格式如下。

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

示例

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另外某个 C 编译器>

- 如果不用 **unsigned char**，**signed char** 和 **signed int** 作为类型说明符，则无需修改源程序。
- 如果用作类型说明符，则将 **unsigned char**，**signed char** 和 **signed int** 改为 **unsigned int**。

（**b**） 位域的分配方向

功能

- 改变要分配的位域的方向，当指定**-RB** 选项时从 MSB 端开始分配位域。
- 如果未指定**-RB** 选项，则从 LSB 端分配位域。

使用方法

- 在编译时指定的**-RB** 选项可以从 MSB 端开始分配位域。
- 不指定该选项以从 LSB 端开始分配位域。

示例 **1**

<位域声明>

```
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;
};
```

因为 A 到 H 是 8 位或更少，所以可以其分配在同一个字节单元中。

MSB							LSB
A	B	C	D	E	F	G	H

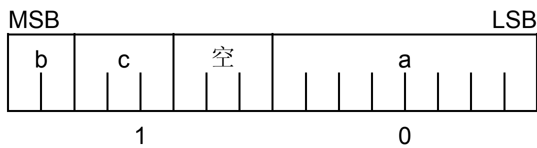
MSB							LSB
H	G	F	E	D	C	B	A

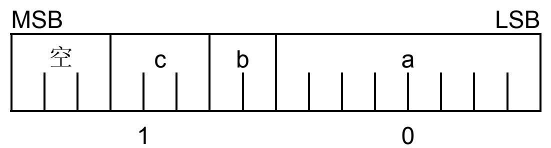
示例 2

<位域声明>

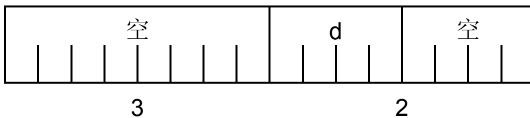
```
struct t {
    char            a;
    unsigned char   b:2;
    unsigned char   c:3;
    unsigned char   d:4;
    int             e;
    unsigned char   f:5;
    unsigned char   g:6;
    unsigned char   h:2;
    unsigned int    i:2;
};
```

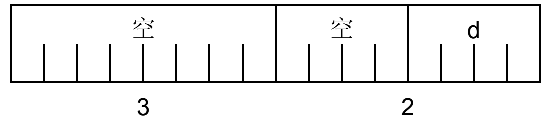
从 MSB 端进行位域分配 当指定**-RB** 选项时	从 LSB 端进行位域分配 当未指定**-RB** 选项时

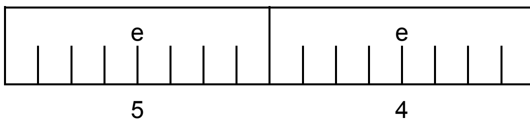


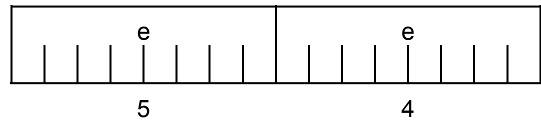


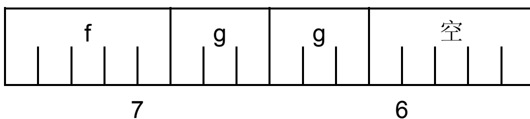
char 型的成员 a 分配到第一字节单元，成员 b 和 c 分配到随后第二个的字节单元中。如果 a 字节单元没有足够空间来容纳 **char** 类型成员，则该成员将分配到后面的字节单元。在此情况下，如果在第二字节单元中仅存在 3 位空间且成员 d 有四位，则其将分配到第三个字节单元。

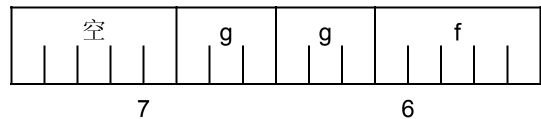


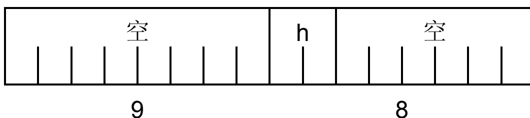


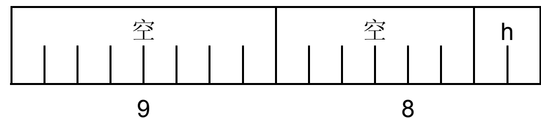






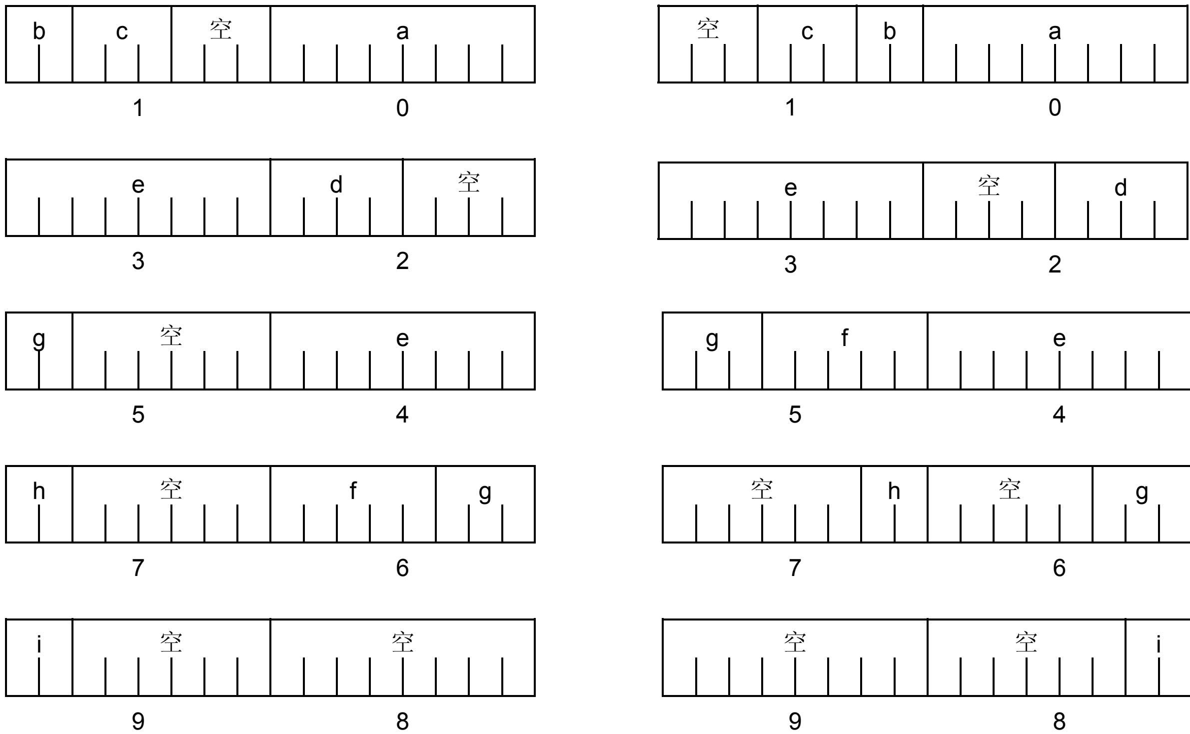






因为成员 g 为 **unsigned int** 型的位域，所以可以跨字节界限进行分配。因为 h 为 **unsigned char** 型的位域，所以其不能和 **unsigned int** 型的 g 位域分配在相同的字节单元中，而是分配在下一字节单元。

因为 i 为 **unsigned int** 型的位域，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），以上位域成为以下形式。



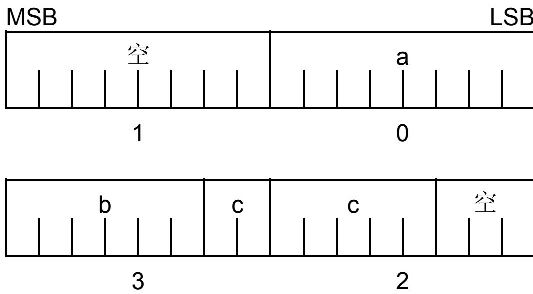
备注　　分配图下方的数字表示从该结构开始的字节偏移值。

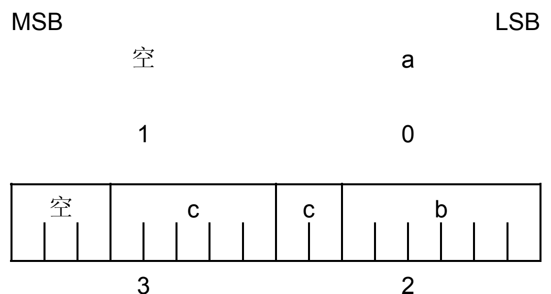
示例 3

<位域声明>

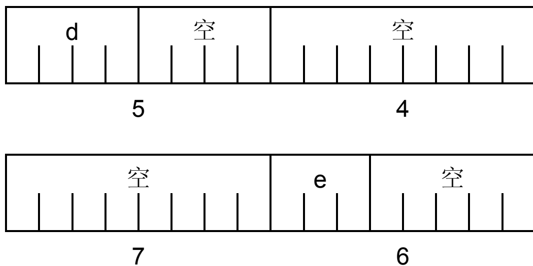
```
struct t {
    char              a;
    unsigned int      b:6;
    unsigned int      c:7;
    unsigned int      d:4;
    unsigned char     e:3;
    unsigned char     f:10;
    unsigned char     g:2;
    unsigned char     h:5;
    unsigned int      i:6;
};
```

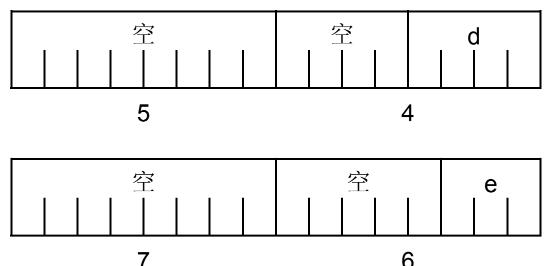
从 MSB 端进行位域分配 当指定**-RB** 选项时	从 LSB 端进行位域分配 当未指定**-RB** 选项时



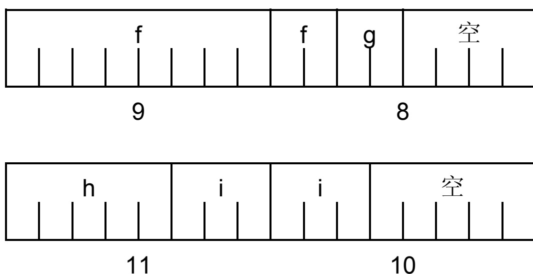


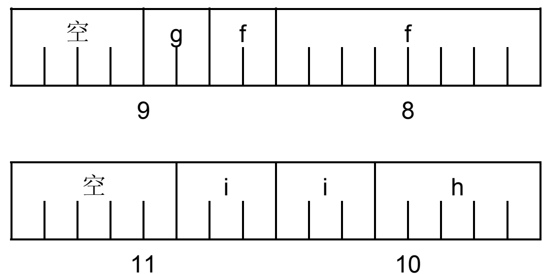
因为 b 和 c 为 **unsigned int** 型的位域，所以其从下一字单元分配。
因为 d 也是 **unsigned int** 型的位域，所以其从下一字单元分配。





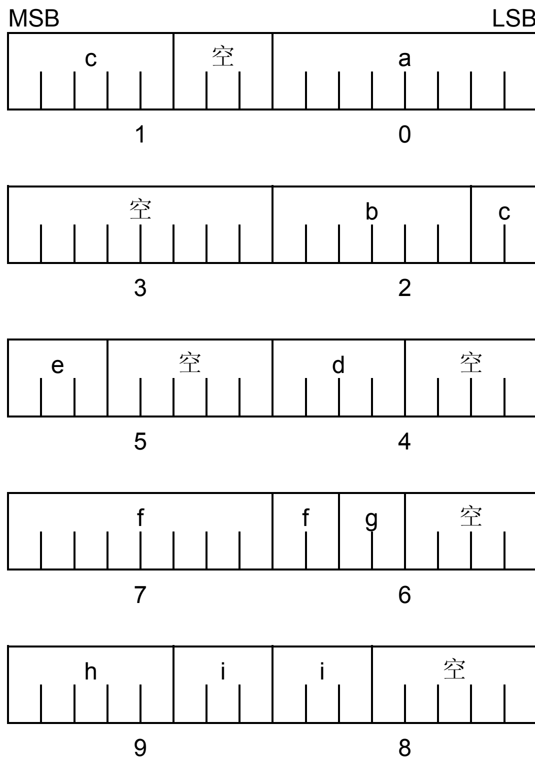
因为 e 为 **unsigned char** 型的位域，所以其分配到下一字节单元。

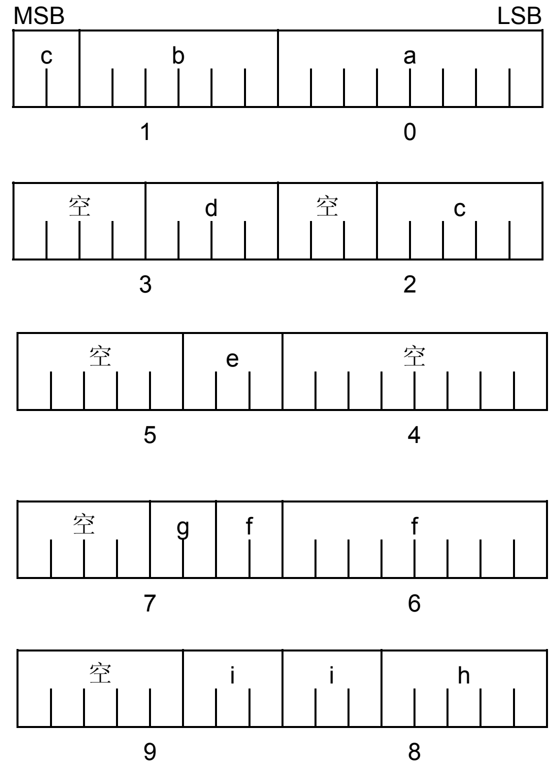




f 和 g、和 h 和 i 每一者分配到单独的字单元。

当指定**-RC** 选项时（对结构成员进行封装），以上位域变为如下形式。





备注　　分配图下方的数字表示从该结构开始的字节偏移值。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 无需修改源程序。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 如果使用了 **-RB** 选项，且代码和位域分配序列配合进行，则必须修改源程序。

改变编译器输出区段名称（#pragma section…）

功能

- 改变编译器输出区段名称并指定起始地址。如果省略起始地址，则按照默认办法分配。有关编译器输出区段名称和默认位置的详细信息，请参阅附录 B 字段名称表。
  此外，这些区段的位置可以通过省略起始地址，在连接时使用连接指令文件指定。如需有关连接指令的详细信息，请参阅 **RA78K0R** 汇编包用户操作手册。
- 要以指定的 **AT** 起始地址改变该区段名称**@@CALT**，**callt** 函数必须在源文件中的另一函数之前或之后说明。
- 如果在**#pragma** 指令之后进行数据说明，则该数据位于改变后的区段中。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区段，可以使每一区段的位置相互独立，以使得数据可以分配到所希望的数据单元中。

使用方法

- 通过使用如下的**#pragma** 指令去指定要改变的区段名称，新区段名称和该区段的起始地址。
- 在 C 源代码开始处说明此**#pragma** 指令。
- 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) 既未定义又未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。要返回到原始区段名称，在改变的区段中说明编译器输出区段名称。

在文件开始处如下声明。

```
#pragma  section  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段名称。**Section** 和 **AT** 可以用大写或小写字母或其组合来表示。

- 其中要说明新区段名称的格式必须符合汇编程序规格（字段名称最多可以使用八个字母）。
- 仅可以使用 C 语言的十六进制数和汇编程序的十六进制数来说明起始地址。

[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，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（请参阅“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 C 文本说明之后执行此**#pragma** 指令。如果有此情况发生，则导致错误。（请参阅“编码错误示例 1”）。
- 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（请参阅“编码错误示例 1”）。

示例 **1**

区段名称**@@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
```

示例 **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 "//"之后的区段。

```
#pragma section   @@DATA        ??DATA
      int         a1;                        // ??DATA
      sreg int    b1;                        // @@DATS
      int         c1 = 1;                    // @@INIT 和@@R_INIT
      const int   d1 = 1;                    // @@CNST
#pragma section   @@DATS        ??DATS
      int         a2;                        // ??DATA
      sreg int    b2;                        // ??DATS
      int         c2 = 1;                    // @@INIT 和@@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 和@@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 和??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
```

示例 **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          // 文件包含主程序 C 代码，随后又#pragma
指令

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 语句在主程序 C 代码之后的 c3.h 中指定
                                         // 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 语句在主程序 C 代码之后的 c3.h 中指定
                                         // 且此后不能指定#pragma 指令。
 int i;
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不支持区段名称改变函数，则无需修改源程序。
- 要改变区段名称，请根据以上使用方法中描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 删除**#pragma section**...或以**#ifdef** 将其隔离。
- 要改变区段名称，请根据每个编译器的具体规格修改程序。

限制

- 无法改变向量表相关的字段（例如，**@@VECT02** 等）区段名称。
- 如果在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。
- 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 FFFE20H 至 FFFEB3H 内。为 **@@CALT** 指定地址的范围限制在 0x80 至 0xbf 内。为 **@@CODE** 和 **@@BASE** 指定地址的范围限制在 0x0 至 0xFFEFF 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，用户必须自行检验以查看汇编输出汇编表是否复制区段名称。
- 如果使用 **#pragma section** 改变了 ROMization 相关的区段名称[注]，则用户必须自行改变启动例程，且职责自负。

当指定 **-ZF** 选项时，每个区段名称都被改变，第二个" @"符号会被"E"替换。

注　ROMization-相关区段名称

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

要改变 ROMization 相关的区段，需要使用启动例程，下文将介绍一个改变结束例程的示例。

[通过启动例程改变 ROMization 相关区段名称的示例]

下文为改变 ROMization 相关的区段名称，并需要相应改变来改变启动例程（**cstart.asm** 或 **cstartn.asm**）和结束例程（**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：新区段第一个标志声明和拷贝初值的部分，并将结束例程标签的声明部分添加到结束模块。

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** 的示例

<i> 添加标签的声明，该标签指示名称改变后的区段的结束

```
#pragma section    @@R_INIT    RTT1
#pragma section    @@INIT      TT1
EXTRN    RTT1_E,TT1_E          ; 添加 RTT1_E 和 TT1_E 的 EXTRN 声明
```

<ii> 添加一段将初值从已改变名称的 RTT1 区段复制到 TT1 区段的程序代码

```
                :
        LDATS1:
                MOVW        AX,HL
                CMPW        AX,#LOW _?DATS
                BZ          $LDATS2
                MOV         [HL+0], #0
                INCW        HL
                BR          $LDATS1
        LDATS2:
                MOV         ES, #HIGH   RTT1_S
                MOV         HL, #LOWW  RTT1_S
                MOV         DE, #LOWW  TT1_S
        LTT1:
                MOVW        AX,HL
                CMPW        AX, #LOWW  TT1_E
                BZ          $LTT2                ；添加的代码，用来将初值从已改变名称
                MOV         A,ES:[HL]            ；的 RTT1 区段复制到 TT1 区段
                MOV         [DE],A
                INCW        HL
                INCW        DE
                BR          $LTT1
        LTT2:
        ;
                CALL        !!_main             ; main（）;
                CLRW        AX
                CALL        !!_exit             ; exit（0）;
                BR          $$
        ;
```

<iii>　设置已改变名称的区段开始标签

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

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

@@CODEL       CSEG
@@CALT        CSEG   CALLT0
@@CNST        CSEG   MIRRORP
@@BITS        BSEG

    END
```

（**b**）改变结束例程 **rom.asm** 的示例

<i> 添加标签的声明，该标签指示名称改变后的区段的结束

```
    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   UNIT64KP
    _?R_INIT:
    @@R_INIS    CSEG   UNIT64KP
    _?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   UNIT64KP    ; 添加标签的设置指示 RTT1 区段结束。
    RTT1_E:                        ; 添加标签设置

    TT1         DSEG   BASEP       ; 添加标签的设置指示 TT1 区段结束。
    TT1_E:                         ; 添加标签设置

    ;
        END
```

二进制常量（二进制常量 0bxxx）

功能

- 在可以用整数常量描述的位置使用二进制常量。

效果

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

使用方法

- 说明 C 源代码中的二进制常量。以下介绍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：‘0’或‘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。

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

示例

<C 源代码>

```
unsigned        i;
i = 0b11100101;
```

编译器的输出对象与下列语句的效果相同。

```
unsigned        i;
i = 0xE5;
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 无需修改。

<从 CC78K0R 编译器至另外某个 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格。
- 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、十进制或十六进制。

模块名称改变函数（**#pragma name**）

功能

- 将指定模块名称的前 254 个字母输出到目标模块文件的符号信息表。
- 当指定了**-G2** 选项，将指定的模块名称的前 254 个字母输出到汇编表文件作为符号信息（**MOD_NAM**），且当指定**-NG** 选项时作为 **NAME** 伪指令。
- 如果模块名称指定了 255 个或 255 个以上字母，则输出警告消息。
- 如果描述中出现未经认可的字母，则出现错误且处理程序异常终止。
- 如果存在一个以上的这种**#pragma** 指令，则不管稍后描述了哪条指令被启用，都会输出警告消息。

效果

- 对象的模块名称可以改为任何名称。

使用方法

- 以下展示说明方法。

```
#pragma name          模块名称
```

模块名称必须由 OS 授权为文件名称的字符组成，除‘（‘,‘）’和 kanji（2-字节字符）之外。
大写字母和小写字母区分对待。

示例

```
#pragma name module1
:
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

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

<从 CC78K0R 编译器至另外某个 C 编译器>

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

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** 语句来实现。

使用方法

- 在源文件中描述的方法和函数调用的格式相同。
  循环移位函数可使用以下四个函数名称。

```
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。

- 通过模块的**#pragma rot** 指令声明循环移位函数的使用方法。
  但是，以下各项可以在**#pragma rot** 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
- **#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     x , !_b
        mov     a , !_a
L0003 :
        ror     a , 1
        dec     x
        bnz     $L0003
```

限制

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

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

<从 CC78K0R 编译器至另外某个 C 编译器>

- 删除**#pragma rot** 声明或以**#ifdef** 将其屏蔽。
- 要用作循环移位函数，请根据每个编译器的具体规格（**#asm**，**#endasm** 或 **asm()**; 等）修改程序。

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

- 生成的代码会自动对应乘法指令输入输出的数据类型大小。因此，生成的代码执行速度会比较快，而且代码量小于普通乘法表达式表达式生成的代码。

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) 既未定义又未引用变量或函数的预处理指令
- **#pragma** 之后的关键字可以用大写或小写字母表示。

限制

- 为乘法专用函数名称不能作为函数名称（已经使用**#pragma mul** 进行声明）
- 乘法函数必须用小写字母描述，如果使用大写字母描述，会被当做普通函数处理。

示例

<C 源代码>

```
#pragma mul
unsigned char a = 0x11 ;
unsigned char b = 2 ;
unsigned int i ;
void main（）
{
        i = mulu（a，b）;
}
```

<编译器的输出对象>

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除**#pragma mul** 声明或以**#ifdef** 将其屏蔽，乘法函数名称可以用作函数名称。
- 要用作乘法函数，请根据每个编译器的具体规格修改程序（**#asm**，**#endasm** 或 **asm()**;等）。

除法函数（#pragma div）

功能

- 将计算除法表达式值的代码输出到目标。
- 如果没有对应的**#pragma** 指令，则乘法函数被视为普通函数。

效果

- 生成的代码和 CC78K0 相兼容，会自动对应除法指令输入输出的数据类型大小。因此，生成的代码执行速度会比较快，而且代码量小于普通乘法表达式表达式生成的代码。

使用方法

- 在源文件中描述的方法和函数调用的格式相同。以下两个函数可用于除法。

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。

- 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) 既未定义又未引用变量或函数的预处理指令
- **#pragma** 之后的关键字可以用大写或小写字母表示。

限制

- 除法函数不是通过内联实现，而是通过由库的调用实现。
- 为除法专用函数名称不能作为函数名称（已经使用**#pragma mul** 进行声明）
- 除法函数必须用小写字母描述，如果使用大写字母描述，会被当做普通函数处理。

示例

<C 源代码>

```
#pragma div
unsigned int a = 0x1234 ;
unsigned char b = 0x12 ;
unsigned char c ;
unsigned int i ;
void main（）{
        i = divuw（a，b）;
        c = moduw（a，b）;
}
```

<编译器的输出对象>

```
mov     c , !_b
movw    ax , !_a
callt   [ @@divuw ]
movw    !_i , ax
mov     c , !_b
movw    ax , !_a
callt   [ @@divuw ]
mov     a , c
mov     !_c , a
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

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

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除**#pragma div** 声明或以**#ifdef** 将其屏蔽，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格修改程序（**#asm**，**#endasm** 或 **asm()** ; 等）。

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，不通过汇编器就无法获得对象，但是如果使用数据插入函数，则对象可以在不通过汇编器的情况下获得。

使用方法

- 在源文件中使用大写字母描述，其描述方法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) 既未定义又未引用变量或函数的预处理指令
- **#pragma** 之后的关键字可以用大写或小写字母表示。

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。如果其以小写字母表示，则将其作为普通函数进行处理。

示例

<C 源代码>

```
#pragma opc
void main（ ）{
        _ _OPC（0xA7）;
        _ _OPC（0x51，0x12）;
        _ _OPC（0x30，0x34，0x12）;
}
```

<编译器的输出对象>

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果编译器不使用数据插入函数，则无需修改。
- 要改为数据插入函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除 **#pragma opc** 声明或以 **#ifdef** 将其屏蔽，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，请根据每个 C 编译器的具体规格修改程序（**#asm**，**#endasm** 或 **asm()**; 等）。

实时操作系统（**RTOS**）的中断函数（**#pragma rtos_interrupt ...**）

功能

- 将**#pragma rtos_interrupt** 指令指定的函数名称当作 78K0R 系列 RTOS（实时操作系统）RX78K0R 的中断函数。
- 将所描述的函数名称的地址注册到中断向量表中作为指定的中断请求名称。
- RTOS 的中断函数按照如下顺序产生代码。

(1) 保存所有的寄存器
(2) 保存编译器使用的 **saddr** 区域
(3) 保存 ES 和 CS 寄存器
(4) 使用“call !!addr20”指令来调用 kernel 标号 __kernel_int_entry
(5) 保留本地变量区域(只有存在本地变量时)
(6) 函数体
(7) 释放本地变量区域(只有存在本地变量时)
(8) 使用“br !!addr20”指令无条件跳转到标签 _ret_int

不产生结尾程序。

效果

- 可以用 C 源文件来描述 RTOS 的中断函数。
- 因为中断请求名称已经被识别，中断向量表的地址就可以不出现。

使用方法

- 用**#pragma** 指令来指定中断请求名称、函数名称以及堆栈的改变。
- **#pragma** 指令在 C 源文件的开头处描述。
  当**#pragma PC**（**设备类型**）描述之后，在**#pragma PC** 之后描述主要的**#pragma** 指令。
  以下各项可以在**#pragma** 指令之前说明。
  (i) 注释
  (ii) 既未定义又未引用变量或函数的预处理指令

```
#pragma △ rtos_interrupt [△中断请求名称△函数名称]
```

- **#pragma** 之后描述的关键字，中断请求名称必须大写。其他的关键字可以使用大写或小写均可。

限制

- 未指定**-zf** 选项时，不论存储器模式如何，RTOS 中断函数的分配区域为[C0H 至 0FFFFH]。
  当指定**-zf** 选项时，中断函数的分配区域会根据存储器模式而变化，并且可以用**__**near 或**__**far 来指定分配区域。
- 中断请求名称必须用大写字母来描述。
- 不能为软件中断和不可屏蔽中断指定中断请求名称（已经被固定），如果有这样的指定，则会发生错误。
- 一个模块单元中的中断请求要进行双重检验。
- **callt/__callt/norec/__leaf/__interrupt/__interrupt_brk/__flash/__flashf** 不能被指定为 RTOS 的中断函数。
- 只有当指定了-zf 选项时，才能指定**__**far 属性。
- RTOS 系统调用函数名称 **ret_int/_kernel_int_entry** 不能被用作函数名称。

示例

[没有指定堆栈改变]

<C 源代码>

```
#pragma  rtos_interrupt  INTP0  intp

int i ;

void intp ( ) {

        int a[3] ;

        a[0] = 1 ;

}
```

<编译器的输出对象>

```
@@BASE      CSEG     BASE
_intp :
            push     ax                  ; 保存寄存器
            push     bc ;
            push     de ;
            push     hl ;
            movw     ax , _@RTARG0       ; 保存编译器所使用的 saddr 区域
            push     ax ;
            movw     ax , _@RTARG2 ;
            push     ax ;
            movw     ax , _@RTARG4 ;
            push     ax ;
            movw     ax , _@RTARG6 ;
            push     ax ;
            movw     ax , _@SEGAX ;
            push     ax ;
            movw     ax , _@SEGDE ;
            push     ax ;
            mov      a , ES ;
            mov      x , a ;
            mov      a , CS ;
            push     ax ;
            call     !!__kernel_int_entry
            subw     sp , #06H
            movw     hl , sp
; line 5 : int a [ 3 ] ;
; line 6 : a [ 0 ] = 1;
            onew     ax
            movw     [ hl ] , ax         ; a
; line 7 :}
            addw     sp , #06H           ; 释放本地变量区域
            br       !!_ret_int
@@VECT06    CSEG AT 0006H
_@vect06 :
            DW       _intp
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果编译器不支持 RTOS 中断函数，则无需修改。
- 要改为 RTOS 中断函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 如果删除**#pragma rtos_interrupt** 声明，该函数名称可以当作普通函数。
- 要用作 RTOS 中断函数使用，请根据每个 C 编译器的具体规格修改程序。

实时操作系统（RTOS）的中断函数修饰词（__rtos_interrupt ...）

功能

- 使用**__rtos_interrupt** 修饰词声明的函数被视为 RTOS（实时操作系统）的中断函数。
  关于 RTOS 中断函数所使用的寄存器和对 saddr 保存和恢复的具体细节，请参阅“实时操作系统（**RTOS**）的中断函数（**#pragma rtos_interrupt ...**）”。

效果

- 中断向量表的设置和 RTOS 的中断函数的定义可以在不同的文件中描述。

使用方法

- 在 RTOS 的中断函数时添加**__rtos_interrupt** 修饰词。

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

限制

- 未指定**-zf** 选项时，不论存储器模式如何，RTOS 中断函数的分配区域为[C0H 至 0FFFFH]。
  当指定**-zf** 选项时，中断函数的分配区域会根据存储器模式而变化，并且可以用__near 或__far 来指定分配区域。
- **callt/__callt/norec/__leaf/__interrupt/__interrupt_brk/__flash/__flashf** 不能被指定为 RTOS 的中断函数。
- 只有当指定了**-zf** 选项时，才能指定__far 属性。
- RTOS 系统调用函数名称 **ret_int/_kernel_int_entry** 不能被用作函数名称。

注意事项

- 向量地址不能只靠声明该修饰词来设置。
  向量地址的设置必须通过**#pragma** 语句或相应的汇编语句来进行。
- 如果在同一个文件中出现了**#pragma rtos_interrupt ...**语句，也有 RTOS 中断函数定义，使用**#pragma rtos_interrupt** 指定的函数名称被判定为 RTOS 的中断函数，即使没有修饰词也同样判定。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果编译器不支持 RTOS 中断函数，则无需修改。
- 要改为 RTOS 中断函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 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（如需详细信息，请参阅 **11.6 C 源代码 C 源程序的修改**）。
- 要用作 RTOS 中断函数使用，请根据每个 C 编译器的具体规格修改程序。

实时操作系统（RTOS）的任务函数（#pragma rtos_task）

功能

- 使用**#pragma rtos_task** 指定的函数被视为 RTOS（实时操作系统）的任务函数。
- 如果指定了函数名称，但是在同一个文件中没有发现实体定义，则会有错误发生。
- 对 RTOS（实时操作系统）的任务函数的预处理不会为帧指针/寄存器变量保存相关寄存器，后处理不做输出。
- 通过“call !!addr20”指令来调用 RTOS 系统调用函数 ext_tsk，如果 **ext_tsk** 在函数之后出现，后处理不做输出。
- 如果在 RTOS 的任务函数中没有 **ext_tsk**，并且指定了**-W2** 报警等级，则会有警告信息发生。

效果

- 可以用 C 源文件来描述 RTOS 的任务函数。
- 帧指针/寄存器变量相关的寄存器保存和后处理不做输出，所以代码效率可以提升。

使用方法

- 使用下列**#pragma** 指令来指定函数名称

```
#pragma △rtos_task [△任务函数名称 ]
```

- **#pragma** 指令在 C 源文件的起始位置开始描述。

但是，以下各项可以在**#pragma** 指令之前编码。

(i) 注释

(ii) 既未定义又未引用变量或函数的预处理指令

- **#pragma** 之后的关键字可以用大写或小写字母说明。

限制

- **callt/__callt/norec/__leaf/__interrupt/__interrupt_brk/__flash/__flashf** 不能被指定为 RTOS 的任务函数。
- 只有当指定了**-zf** 选项时，才能指定**__far** 属性。
- 不能使用普通函数的调用方式来调用 RTOS 的任务函数。

示例

<C 源代码>

```
#pragma rtos_task func
int    i ;
void main ( )
{
        regiser int    a ;
        int            x ;
        x = 1 ;
        r = 2 ;
}
void func ( )
{
        register int   r ;
        int            x ;
        x = 1 ;
        r = 2 ;
        ext_tsk ( ) ;
}
```

<编译器的输出对象>

```
@@CODEL CSEG
_main :
        push    hl                  ; 保存帧指针
        subw    sp , #04H
        movw    hl , sp
        onew    ax
        movw    [ hl ] , ax         ; x
        incw    ax
        movw    [ hl + 2 ] , ax     ; r
        addw    sp , #04H           ; 结尾程序不做输出
        pop hl
        ret
_func :
        subw    sp , #04H           ; 不保存帧指针
        movw    hl , sp
        onew    ax
        movw    [ hl ] , ax         ; x
        incw    ax
        movw    [ hl + 2 ] , ax     ; r
        call    !!_ext_tsk          ; 结尾程序不做输出
```

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果编译器不支持 RTOS 任务函数，则无需修改。
- 要改为 RTOS 任务函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 通过删除 **#pragma rtos_task** 声明，RTOS 任务函数当作普通函数处理。
- 要用作 RTOS 任务函数使用，请根据每个 C 编译器的具体规格修改程序。

Flash区域分配方法（-ZF）

注意　　本函数用来使能设备写入 flash 存储器的功能。

功能

- 生成的代码分配在 flash 区域内。
- flash 区域中的外部变量不能在 boot 区域中被引用。
- boot 区域中的外部变量不能在 flash 区域中被引用。
- 在 boot 区域中的程序和 flash 区域中的程序不能定义同名的外部变量，也不能定义同名的全局函数。

效果

- 可以把程序分配在 flash 区域。
- 不使用-ZF 选项就可以使能函数和 boot 区域目标连接。

使用方法

- 编译时指定**-ZF** 选项。

限制

- 对 flash 区域使用启动例程或库。

Flash区域跳转表（#pragma ext_table）

注意　本函数用来使能设备写入 flash 存储器的功能。

功能

- 决定启动例程跳转表的首地址，以及中断函数跳转表的首地址，或者从 boot 区域调用 flash 区域的函数调用跳转表的首地址。
- 跳转表首地址开始的前 64 个地址专用于中断函数（包括启动和例程函数），每个占用 4 字节空间。普通函数的跳转表一般分配的位置在"跳转表首地址 + 4*64"之后。跳转表中每项占用 4 字节空间。

效果

- 启动例程和中断函数可以分配在 flash 区域。
- 可以从 boot 区域调用 flash 区域的函数。

使用方法

- 下列**#pragma** 指令指定 flash 区域跳转表的首地址。

```
#pragma ext_table 跳转表首地址
```

#pragma 指令在 C 源文件的开始处描述。

- 然而，以下各项可以在**#pragma** 指令之前说明。
  (i)　注释
  (ii)　除**#pragma ext_func, #pragma vect**（同时指定了**-ZF** 选项）**, #pragma interrupt** 或**#pragma rtos_interrupt** 之外的其他**#pragma** 指令。
  (iii)　既未定义又未引用变量或函数的预处理指令

限制

- 跳转表被分配到 flash 区域的首地址
- 如果**#pragma ext_table** 之前出现了**#pragma ext_func, #pragma vect**（同时指定了**-ZF** 选项）**, #pragma interrupt** 或**#pragma rtos_interrupt**，则会发生错误。
- 跳转表的首地址应该位于 80H 到 0FF80H 之间。
- 为了配合指定的跳转表首地址，必须为中断向量（_**@vect00** 到_**@vect7e**）重新配置库。中断向量库中默认是 2000H，为了指定 2000H 之外的地址，按照下列描述来对库进行重新配置。

(i) 在\Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\src 目录下的 vect.inc 文件中，有 ITBLTOP EQU 2000H 语句，在该语句中改变 H 寄存器的位置。

(ii) 在 DOS 提示符下运行\Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\bat\目录下的 repvect.bat 文件，并且通过汇编程序对库进行更新。拷贝更新后的库\Program Files\NEC Electronics Tools\CC78K0R\Vx.xx\src\cc78k0r\lib 用于连接。

备注 根据安装方法的不同，上述的目录可能有所差异。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不使用**#pragma ext_table**，则无需修改。
- 要指定 flash 区域跳转表的首地址，请根据以上使用方法中描述的过程来改变地址。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 删除**#pragma ext_table**，或用**#ifdef** 将其屏蔽。
- 要指定 flash 区域跳转表的首地址，必须进行如下改变。

示例

[要在地址 2000H 之后生成跳转表，并放置中断函数]

<C 源程序>

```
#pragma    ext_table    0x2000
#pragma    interrupt    INTP0  intp
void intp ( )
{
}
```

（a） 将中断函数放置在 boot 区域（未指定**-ZF** 选项）

<输出代码>

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

- 在中断向量表中设置中断函数的首地址。

（b） 将中断函数放置在 flash 区域（指定了**-ZF** 选项）

<输出代码>

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

@EVECT06            CSEG        AT  02009H
                    br          !_intp
```

- 在跳转表中设置中断函数的首地址。
- 跳转表的地址值是 2000H + 4*（0006H/2），由于跳转表的首地址是 200CH，并且中断向量地址（2 字节）是 0006H。
- 中断向量库会在中断向量表中将地址设置为 20009H。

<中断向量 06 的库>

```
                    PUBLIC      _@vect06
@@VECT06            CSEG        AT  0006H
_@vect06 :
                    DW          2009H
```

从boot区域到flash区域的函数调用（#pragma ext_func）

注意　　本函数用来使能设备写入 flash 存储器的功能。

功能

- 从 boot 区域到 flash 区域的函数调用是通过 flash 区域的跳转表来执行的。
- 对 flash 区域中来说，可以直接调用 boot 区域内的函数。

效果

- 可以从 boot 区域调用 flash 区域中的函数。

使用方法

- 下列**#pragma** 指令指定函数名称以及 boot 区域调用的 flash 区域 ID 值。

```
#pragma  ext_func  函数名称  ID 值
```

#pragma 指令在 C 源文件的开始处描述。

但是，以下各项可以在**#pragma** 指令之前说明。

(i)　　注释

(ii)　　既未定义又未引用变量或函数的预处理指令

限制

- ID 值得范围是 0 到 255（0xFF）。
- 如果**#pragma ext_table** 之前出现了**#pragma ext_func**，则会发生错误。
- 如果相同的函数有了不同的 ID 值，或者不同的函数有了相同的 ID 值，则会发生错误。下面所列的（a）和（b）都是错误的。

(a) **#pragma ext_func f1 3**
#pragma ext_func f1 4

(b) **#pragma ext_func f1 3**
#pragma ext_func f2 3

- 如果从 boot 区域对 flash 区域进行了函数调用，在 flash 中没有对应的函数定义，则连接器无法执行检查。这属于用户的责任。
- **callt** 函数只能存放在 boot 区域。如果在 flash 区域出现 **callt** 函数定义（指定了**-ZF** 选项），则导致错误。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不使用**#pragma ext_func**，则无需修改。
- 要执行从 boot 区域到 flash 区域的函数调用，请根据以上**使用方法**中描述的过程来对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 删除**#pragma ext_ func**，或用**#ifdef** 将其屏蔽。
- 要执行从 boot 区域到 flash 区域的函数调用，必须进行如下改变。

示例

要在地址 2000H 之后生成跳转表，从 boot 区域调用 flash 区域的函数 f1 和 f2。

<C 源文件>

```
(1) Boot 区域方面

#pragma    ext_table    0x2000
#pragma    ext_func     f1          3
#pragma    ext_func     f2          4
extern     void         f1 ( void ) ;
extern     void         f2 ( void ) ;

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

(2) Flash 区域方面

#pragma    ext_table    0x2000
#pragma    ext_func     f1          3
#pragma    ext_func     f2          4

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

备注 **1. #pragma ext_func f1 3** 意味着要跳转的目标是函数 **f1**，位于跳转表中的地址是 **2000H + 4*64 + 4*3**。

备注 **2. #pragma ext_func f2 4** 意味着要跳转的目标是函数 **f2**，位于跳转表中的地址是 **2000H + 4*64 + 4*4**。

备注 **3.** 跳转表开始的 **4*64** 个字节专用于中断函数（包括启动例程）。

<C 源文件：设备支持 bank 函数>

```
(1) Boot 区域方面

@@CODEL             CSEG
_func :
           call     !0210CH
           call     !02110H
ret

(2) Flash 区域方面

@ECODEL             CSEG
_f1 :
           ret
_f2 :
           ret
@EXT03     CSEG     AT   0210CH
br         !!_f1
br         !!_f2
```

固件ROM函数（__flash）

注意　本函数用来使能设备写入 flash 存储器的功能。

功能

- 本函数调用固件 ROM 函数，通过位于固件 ROM 函数和 C 语言函数之间的接口库来自写入 flash 存储器。
- 在接口库调用接口中，第一个参数被传递到寄存器，第二个及以后的参数被传递到堆栈。第一个参数所用的寄存器如下所示。

1-2 字节	数据 AX
4 字节 数据	AX（低端），BC（高端）

- 必须设置接口库来返回计算值，按照返回值的类型大小存放在对应的寄存器中

1-2 字节 数据	BC
near 指针	BC
4 字节 数据，far 指针	BC（低端），DE（高端）

效果

- 可以用 C 源程序来描述固件 ROM 函数的相关操作。

使用方法

- 在声明接口库原形时，在顶端添加**__flash** 属性。

限制

- 不支持通过函数指针进行的函数调用。
- 当函数定义时带有**__flash**，则导致错误。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不使用**__flash** 保留字，则无需修改。
- 要改变固件 ROM 函数，请根据以上**使用方法**中描述的过程来对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 可能使用**#define**（如需详细信息，请参阅 **11.6 C 源程序的修改**）。
- 如果 CPU 支持固件 ROM 函数或替代函数，用户必须创建一个专用库来读写这个区域。

参数/返回值的int展开限制方法（-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，可以缩短目标代码并提高运行速度。

使用方法

- 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]

<编译器的输出对象>

```
_main :
; line 5 :              c = func1 ( d , e ) ;
            mov         x , !_e
            push        ax
            mov         x , !_d                   ; 不执行 int 扩展
            call        !_func1
            pop         ax
            mov         a , c
            mov         !_c , a
; line 6 :              c = func2 ( d , e ) ;
            mov         x , !_e
            clrb        a                         ; 因为没有原形声明，执行 int 扩展
            push        ax
            mov         x , !_d
            mov         x , #00H
            xch         a , x                     ; 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 : unsigned char func1（ unsigned char x , unsigned char y ）
; line 9 : {
_func1 :
            push        hl
            push        ax
            movw        ax , sp
            movw        hl , ax
            mov         a , [ hl ]
            mov         x , a
            mov         a , [ hl + 6 ]
            movw        hl , ax
; line 10 : return x + y ;
            mov         a , l
            add         a , h
            mov         c , a                     ; 不执行 int 扩展
; line 11 : }
            pop         ax
            pop         hl
            ret
```

限制

- 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

<从另外某个 C 编译器至 CC78K0R 编译器>

- 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 CC78K0R 编译器至另外某个 C 编译器>

- 无需修改。

内存操控函数（**#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，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。

使用方法

- 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) 既未定义又未引用变量或函数的预处理指令

示例

<C 源代码>

```
#pragma  inline
char           ary1[100]，ary2[100];
void           main()
{
               memset（ary1，‘A’，50）;
               memcpy（ary1，ary2，50）;
}
```

<编译器的输出对象>

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

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不使用内存运算函数，则无需修改。
- 当改变内存操作函数时，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 删除**#pragma inline** 指令，或用**#ifdef** 对其限制。

绝对地址分配规格（__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_ **_directma** 变量按常规变量进行处理。
- 因为初值视为分配地址，所以不能定义初值，保留一个未定义的值。
- 下文给出了可指定的地址规格范围、保留区域范围是通过模块链接的为特定地址保留的范围，变量双重检验范围。

各项	范围	
	当指定小型模式或中等模式时	当指定大型模式或紧凑模式时
地址规格范围	0xf0000 至 0xfffff	0x00000 至 0xfffff
保留区域范围	0xffd00 至 0xffeff	0xffd00 至 0xffeff
双重检验范围	设备内部 RAM 的起始地址 至 结束地址	设备内部 RAM 的起始地址 至 结束地址

- 如果指定的地址在地址规格范围之外，则发生错误。
- 如果由_ **_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 **W0762** 警告消息并显示有重复的变量名称。
- 如果地址规格范围在 **saddr** 区域之内，则自动更正_ **_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** 必须和 **__directmap** 连同使用。如果不指定，则引用时会出现错误。
- 如果指定地址范围在 near 区域内，变量被认为在 near 区域内可以访问。
- 如果指定地址范围既不是 saddr 区域也不是 near 区域，变量被认为在 far 区域内可以访问。
- 如果没有指定__near 或 __far 修饰词，变量的访问情况由存储器模式规格来决定。
- 如果指定了类型修饰符，变量的访问情况由规格来决定。如果指定的地址范围和类型修饰符相矛盾，则发生错误。

  下表列出了地址规格范围，存储器模式和类型修饰符之间的关系。

地址规格范围		类型修饰符					
		__near __sreg	__far __sreg	__sreg	__near	__far	无指定
saddr 区域内	访问方法	sreg	sreg	sreg	sreg	sreg	sreg
	指针长度	2 字节	4 字节	小型：2 字节 中等：2 字节 紧凑：4 字节 大型：4 字节	2 字节	4 字节	小型：2 字节 中等：2 字节 紧凑：4 字节 大型：4 字节
near 区域内	访问方法	错误	错误	错误	near	far	小型：near 中等：near 紧凑：far 大型：far
	指针长度				2 字节	4 字节	小型：2 字节 中等：2 字节 紧凑：4 字节 大型：4 字节
far 区域内	访问方法	错误	错误	错误	错误	far	小型：错误 中等：错误 紧凑：far 大型：far
	指针长度					4 字节	小型：错误 中等：错误 紧凑：4 字节 大型：4 字节

效果

- 一个或一个以上的变量可以分配到相同属性的地址。

使用方法

- 在模块中声明 **__directmap**，其中将定义要分配在绝对地址中的变量的

```
_ _directmap                 类型名称    变量名称              = 分配地址规格;
_ _directmap     static      类型名称    变量名称              = 分配地址规格;
_ _directmap     _ _sreg     类型名称    变量名称              = 分配地址规格;
_ _directmap     _ _sreg     static      类型名称    变量名称  = 分配地址规格;
```

- 如果为结构体/共用体/数组声明_ _**directmap**，则将地址放入括号{}中。

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  _main
_c      EQU     0FFE00H         ; 由 EQU 定义__directmap 声明的变量的地址
_d      EQU     0FFE20H
_e      EQU     0FFE21H         ;
_xx     EQU     0FFE30H         ;
        EXTRN   __mmfe00        ; 链接保留区域模块的 EXTRN 输出
        EXTRN   __mmfe20        ;
        EXTRN   __mmfe21        ;
        EXTRN   __mmfe30        ;
        EXTRN   __mmfe31        ;
@@CODEL         CSEG
_main :
; line 10 :
        oneb    !loww ( _c )
; line 11 :
        mov     _d , #012H      ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line 12 :
        set1    _e.5            ; 因为使用了__sreg，可以进行位处理
; line 13 :
        Mov     _xx , #05H      ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line 14 :
        mov     _xx + 1 , #0AH  ; 因为地址指定在 saddr 区域内，输出 saddr 指令
; line 15 :
        ret
        END
```

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。
- 如果指定的地址位于保留区域范围之外，则变量区域不作保留，有必要指定指令文件(directive file)或为保留该区域创建单独的模块。
- **__directmap** 变量不能用 extern 声明，因为它和静态变量的处理方法相同。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果不使用关键字_ **_directmap**，则无需修改。
- 要改为_ **_directmap** 变量，请根据以上**使用方法**描述的过程对源程序进行修改。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 兼容性可以使用**#define** 实现（如需详细信息，请参阅 **11.6 C** 源代码的修改）。
- 要将_ **_directmap** 用作绝对地址分配规格，请根据每个编译器的具体规格来修改程序。

near/far区域规格

功能

- 函数的分配位置或用__near 或 __far 类型修饰符特意指定变量。

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

- 指向 near 区域的指针长度应该为 2 字节，指向 far 区域的指针长度应该为 4 字节。
- 如果在声明同一个变量或函数时使用了__near 和 __far 类型修饰符，则会发生错误。
- 语法上来说，__near 和 __far 类型修饰符都被当作类型修饰符处理。
- 如果和__callt, __interrupt, __rtos_interrupt, __interrupt_brk, __sreg 或 __boolean 同时使用，则__near 或 __far 类型修饰符被忽略。
- 同时指定__near 和 __far 类型修饰符的话，或发生错误。
- 如果是指定自动变量，参数或寄存器变量， __near 和 __far 类型修饰符被忽略。
- 无需使用 ES 寄存器就可以访问 near 区域内的变量。指针长度为 2 字节。
- 访问 far 区域内的变量需要设置 ES 寄存器。指针长度为 4 字节。
- near 区域内的函数可以通过“!addr16”来调用，far 区域内的函数通过“!!addr20”来调用。
- 因为所有指令都需要引用 CS 寄存器调用函数指针，调用函数指针时请确保设置 CS 寄存器。
- near 区域内函数的函数指针输出代码会将 CS 寄存器清 0。
- far 指针的最高字节总是未定义。
- 从 near 指针或 int 转换为 far 指针，或从 near 指针转换为 long 会导致如下操作。

  （1） “0xf”会加到变量指针的高位字节（0 是例外，并进行零扩展）

  （2） 函数指针是零扩展。
- 涉及 far 指针的加减法只使用低位 2 字节，高位字节不变。
- ptrdiff_t 总是整数类型。
- 涉及 far 指针的等式运算会使用低位的 3 个字节。

- 涉及 far 指针的关系运算会使用低位的 2 个字节。为了比较并没有指向同一个目标的指针，指针必需被转换为 unsigned long 型。如果指定了-za 选项，低 3 位字节都用于比较。
- 根据指定的存储器模式，字符串常数可能存放在 far 区域也可能在 near 区域。

存储器模式	存放位置
小型模式	near 区域
中等模式	near 区域
紧凑模式	far 区域
大型模式	far 区域

- 当使用大型模式或紧凑模式时，指向自动变量，参数和 sreg 变量的指针都式 4 字节长度。

效果

- __far 类型修饰符的规格可以让函数和变量分配在 far 区域，也可以在 far 区域引用。
- __near 类型修饰符的规格可以让函数和变量分配在 near 区域，也可以在 near 区域引用。
  分配到 near 区域的函数和变量都可以用短指令类调用或引用。

使用方法

- 在声明函数或变量时添加__near 或__far 类型修饰符。

示例

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

- i1 是整数类型，被分配到 near 区域。
- i2 是整数类型，被分配到 far 区域。
- p1 是 4 字节类型变量，指向“一个 far 区域中的整数类型”。变量自身被分配在 near 区域。
- p2 是 2 字节类型变量，指向一个 near 区域中的 4 字节类型，它指向“一个 far 区域中的整数类型”，变量自身被分配在 far 区域。
- fun1 是个返回“整数类型”的函数，该函数自身分配在 far 区域。
- fun2 是个返回 4 字节类型的函数，该返回值指向“一个 far 区域中的整数类型”，该函数自身分配在 near 区域。
- fp1 是个 2 字节变量，指向“一个 near 区域中的函数，该函数返回整数类型”。这个变量自身分配在 far 区域内。
- fp2 是个 2 字节变量，指向一个 near 区域中的函数，该函数返回 4 字节类型指针，此指针指向“far 区域中一个整数类型”。这个变量自身分配在 near 区域内。
- fp3 是个 4 字节类型变量，指向一个 far 区域中的函数，该函数返回 2 字节类型指针，此指针指向“near 区域中一个整数类型”。这个变量自身分配在 near 区域内。
- fp4 是个 2 字节变量，指向一个 near 区域中的函数，该函数返回 2 字节类型指针，此指针指向“near 区域中一个整数类型”。这个变量自身分配在 far 区域内。

限制

- 即使指定了__far 类型修饰符，数据也不能被分配在超出 64KB 边界之外的区域。
  函数可以被分配在超出 64KB 边界之外的区域。

兼容性

<从另外某个 C 编译器至 CC78K0R 编译器>

- 如果未使用保留关键字__near 或 __far，则无需修改代码。

<从 CC78K0R 编译器至另外某个 C 编译器>

- 如果未使用__near 或 __far 类型修饰符，则无需修改代码。
- 如果使用了__near 或 __far 类型修饰符，#define 可以被用来规格 near/far 区域。

注意事项

- 如果低位 2 个字节被用于关系运算，数据无法被分配在 64KB 边界区域的最后一个字节。如果这样分配了，连接器或编译器会输出一个错误。
  因为关系运算所进行的是 ANSI 编译操作[注]，该运算使用的指针指向数组范围之外。
  注　ANSI 规定的关系运算的限制
  如果表达式 P 指向数组对象的某个元素，表达式 Q 指向该数组对象的最后一个元素，指针表达式 Q+1 比 P 大。
- far 区域指针的大小为 4 字节，但是计算对象是低位 3 个字节，所以最高字节总是未定义。

示例

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

有值被写入 un.ptr 然后 un.ldata 被引用；最高字节变为未定义。为了保证 un.ldata 的最高字节为 0，共用体必需首先用 0 对其初始化。

- 连接器检查区块的数据分配，检查如下区段类型组合和重定位属性。
  DSEG UNIT64KP
  DSEG PAGE64KP
  DSEG PAGE64KP

- 如果使用#pragma section 或连接命令文件改变以上某个重定位属性，连接器不做检查。

存储器模式规格

功能

- 指函数或变量的分配如下表。

存储器模式	数据	函数
小型模式	near 区域	near 区域
中等模式	near 区域	far 区域
紧凑模式	far 区域	near 区域
大型模式	far 区域	far 区域

- 如果使用了__near 和 __far 类型修饰符，则指定的__near 和 __far 类型修饰符优先。
- 小型模式

  包括 64KB 的数据部分和 64KB 的代码部分，总计 128KB。

  数据 ROM 分配在 00000H 至 0FFFFH 或 10000H 至 1FFFFH，并且镜像映射于 FxxxxH。

  代码分配在 00000H 至 0FFFFH。

  由于指定__far 类型修饰符时，CS 寄存器值可能会被改变。所以在调用函数指针时，请确保设置 CS 寄存器。

- 中等模式

  变量分配在 near 区域，函数分配在 far 区域。包括 64KB 的数据部分和 1 MB 的代码部分。

  数据 ROM 分配在 00000H 至 0FFFFH 或 10000H 至 1FFFFH，并且镜像映射于 FxxxxH。

  代码分配无限制。

- 紧凑模式

  变量分配在 far 区域，函数分配在 near 区域。包括 64KB 的数据部分和 1 MB 的代码部分。

  数据分配无限制，代码分配在 000000H 至 00FFFFH。

- 大型模式

  变量和函数都分配在 far 区域。包括 1 MB 的数据部分和 1 MB 的代码部分。

  数据分配无限制，代码分配在 000000H 至 00FFFFH。

用法

编译时指定-ms, -mm, -mc 或-ml 选项

选项存储器模式	解释说明
-ms	小型模式
-mm	中等模式
-mc	紧凑模式
-ml	大型模式

示例

<C 源文件>

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

<编译器输出对象：当使用小型模式时>

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

<编译器输出对象：当使用中等模式时>

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

<编译器输出对象：当使用紧凑模式时>

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

<编译器输出对象：当使用大型模式时>

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

限制

- 即使制定了紧凑模式或大型模式，数据不能被分配到大于 64KB 边界的扩展区域中。
- 指定了不同存储器模式的模块不能相互连接。
- 每个装载模块文件分配在 far 区域内带有/不带有初始值的变量大小为（64K-1）字节。（注：如果指定了-za 选项则为 64KB）
  这个容量可以通过改变区块名称来增加，可以将某个特定文件中带有/不带有初始值的变量包含到另外一个输出区块名称，使用函数“改变编译器输出区块名称（#pragma section...）”。
  在这个情况下，必须修改启动例程和结束例程（参见“改变编译器输出区块名称（#pragma section...）”一节中的**[在启动例程中改变和 ROMization 有关的区块名称]**）。
  每个输出区块名称的最大容量都不会改变。
- 如果未指定-za 选项，数据不能被分配在 64KB 边界区域的最后一个字节（参见“near/far 区域规格”注意事项）。

11.6 C源代码的修改

通过使用 CC78K0R 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 78K0R 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 CC78K0R 编译器以及相反操作。

<从另外某个 C 编译器至 CC78K0R 编译器>

- **#pragma** 注
  如果另外的 C 编译器支持**#pragma** 预处理指令，则必须修改 C 源代码。修改 C 源代码的方法和修改的工作量取决于另外该 C 编译器的规格。
- 扩展规格
  如果另外的 C 编译器已扩展了规格，诸如添加了新的关键字，则必须修改 C 源代码。修改 C 源代码的方法和程度取决于另外该 C 编译器的规格。

  注　#pragma 为 ANSI 支持的预处理指令。#pragma 之后的字符串对编译器的识别为指令。如果编译器不支持此指令，则忽略#pragma 指令，且继续进行编译直到其完全结束。

<从 CC78K0R 编译器至另外的 C 编译器>

- 因为此 C 编译器添加了关键字作为扩展函数，所以必须删除这些关键字或用**#ifdef** 进行屏蔽使 C 源代码可以移植到另外的 C 编译器。

示例

<1>　要使关键字无效（同样应用于 **callf**，**sreg** 和 **norec** 等）

```
#ifndef _ _K0R_ _
#define        callt                    /* 将 callt 作为普通函数 */
#endif
```

<2>　要从一种类型变为另一种

```
#ifndef _ _K0R_ _
#define bit    char                     /*将 bit 型改为 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） 传输参数
   （b） 存储参数和自动变量的位置

3. norec 函数调用接口
   （a） 传输参数
   （b） 存储参数的位置
   （c）存储自动变量的位置

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，返回值存储的位置如下表所示。

类型	存放位置
1-字节整数	BC
2-字节整数	
4-字节整数	BC（低），DE（高）
指针（__near 属性）	BC
指针（__far 属性）	BC（低），DE（高）
结构体，共用体（小型模式，中等模式）	BC
结构体，共用体（紧凑模式，大型模式）	BC（低），DE（高）
1 位	CY（进位标志）
浮点数（**float** 型）	BC（低），DE（高）
浮点数（**double** 型）	BC（低），DE（高）

11.7.2 普通函数调用接口

（1） 传输参数

- 当有函数被调用时，第二个及以后的参数通过堆栈传输到函数定义方。
- 第一个参数通过寄存器或堆栈传输到函数定义方。
  第一个参数传输到的位置如下表所示。

表 11-3 类型调整细节（从整形和短整形改变为 char 型）

类型	存放位置
1-字节数据[注] [注]	**AX**
3-字节数据[注] 4-字节数据[注]	**AX，BC**
浮点数	**AX，BC**
其他	通过堆栈传输

注　　1 至 4 字节数据可以包括结构体、共用体和指针。

（2） 参数存储的位置和顺序

- 用 **register** 声明的参数/自动变量，或当指定**-QV** 选项时，在函数的开始位置就被分配到寄存器。其他参数和自动变量被分配到堆栈。
- 如果一个参数没有被指定寄存器，传输的位址就是它被指定的位置。该参数从函数调用方通过堆栈传递而来。
- 参数和自动变量被指定到 HL 寄存器，除非没有堆栈帧。
  如果指定了**-QR** 选项，参数和自动变量同样可以被指定到_**@KREGxx**。关于_**@KREGxx** 的信息请参阅“附录 A 用于 saddr 区域的标签列表”
- 参数和自动变量按照引用频率的顺序被指定到寄存器。
  很少被引用的参数和自动变量可能不用被指定到寄存器，即使指定了-qv 选项或者声明该参数或自动变量时使用了 register 关键词。
- 被指定存放参数和自动变量的寄存器由函数定义方保存和恢复。

示例

<C 源文件 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 ;
}
```

[当指定-qr 选项时]

<输出代码>

```
_main :
; line 4 :              func0 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H        ; 22136
        push    ax                          ; 第二个及以后的参数通过堆栈传输
        movw    ax , #01234H        ; 4660  ; 第一个参数通过寄存器传输
        call    !!_func0                    ; 函数调用
        pop     ax                          ; 释放堆栈，函数调用时在其中保存数据
; line 5 : }
        ret
; line 6 : void func0（ register int p1 , int p2 ）
; line 7 : {
_func0 :
        push    hl
        movw    de , _@KREG14
        push    de                          ; 保存存放寄存器变量的 saddr 区域
        movw    de , _@KREG12
        push    de                          ; 保存存放寄存器变量的 saddr 区域
        movw    _@KREG14 , ax               ; 将寄存器参数 p1 存放到 saddr
        push    ax                          ; 为自动变量保留区域
        movw    hl , sp
; line 8 : register int r ;
; line 9 : int          a ;
; line 10 : r = p2 ;
        movw    ax , [ hl + 12 ]    ; p2    ; 参数 p2
        movw    _@KREG12 , ax       ; r     ; 自动变量 r
; line 11 : a = p1 ;
        movw    ax , _@KREG14       ; p1    ; 参数 p1
        movw     [ hl ] , ax        ; a     ; 自动变量 a
; line 12 : }
        pop     ax                          ; 为自动变量保留区域
        pop     ax
        movw    _@KREG12 , ax               ; 为寄存器参数保留 saddr 区域
        pop     ax
        movw    _@KREG14 , ax               ; 恢复为寄存器参数保留 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 ;
}
```

[当指定-qr 选项时]

<输出代码>

```
_main :
; line 4 :  func0 ( 0x1234 , 0x5678 ) ;
        movw    ax , #05678H        ; 22136
        push    ax                          ; 第二个及以后的参数通过堆栈传输
        movw    ax , #01234H        ; 4660  ; 第一个参数通过寄存器传输
        call    !!_func1                    ; 函数调用
        pop     ax                          ; 释放堆栈，函数调用时在其中保存数据
; line 5 : }
        ret
; line 6 : void func0 ( int p1 , register int p2 )
; line 7 : {
_func0 :
        push    hl
        push    ax                          ; 将第一个参数 p1 装载到堆栈
        movw    de , _@KREG14
        push    de                          ; 保存存放寄存器变量的 saddr 区域
        movw    de , _@KREG12
        push    de                          ; 保存存放寄存器变量的 saddr 区域
        movw    ax , [ sp + 12 ]
        movw    _@KREG12 , ax               ; 将参数 p2 存放到 saddr
        push    ax                          ; 为自动变量 a 保留区域
        movw    hl , sp
; line 8 : register int r ;
; line 9 : int a ;
; line 10 : r = p2 ;
        movw    ax , _@KREG12       ; p2    ; 参数 p2
        movw    _@KREG14 , ax       ; r     ; 自动变量 r
; line 11 : a = p1 ;
        movw    ax , [ hl + 6 ]     ; p1    ; 参数 p1
        movw    [ hl ] , ax         ; a     ; 自动变量 a
; line 12 : }
        pop     ax                          ; 释放自动变量 a 占用的区域
        pop     ax
        movw    _@KREG12 , ax               ; 恢复寄存器参数占用的 saddr 区域
        pop     ax
        movw    _@KREG14 , ax               ; 恢复寄存器参数占用的 saddr 区域
        pop     ax                          ; 恢复第一个参数 p1 占用的区域
        pop     hl
        ret
```

11.7.3 norec函数调用接口

（1） 传输参数

- 在函数调用时，参数通过寄存器或_@NRARGx 传递到函数定义方。
- 只有当指定了-qr 选项时，参数可以通过_@NRARGx 传递。
  关于_@NRARGx，请参阅附录“附录 **saddr** 区域标签列表”。
- 下表列出了传递参数所有的位置。

类型	存放位置
1-字节数据[注]，2-字节数据[注]	从第一个参数开始，按照 AX，DE 和_@NRARGx 的顺序
3-字节数据[注]，4-字节数据[注]	_@NRARGx
浮点数	_@NRARGx

注　1-4 字节数据包括指针。
　　即使 1 字节数据有多个参数，只有一个参数可以被指定到寄存器对。
　　第一个参数如果是 1 字节数据，则被指定到寄存器 A。

（2） 存储参数的位置

- 在函数的开始处，参数被分配到 **DE** 寄存器，**_@RTARG6**，**_@RTARG7** 或_**@NRARGx**。
- 仅当指定了**-QR** 选项时，参数才会分配到_**@NRARGx**。
  关于_**@RTARGx** 和_**@NRARGx**，请参阅“附录 **A saddr** 区域标签列表”。
- 如果有参数无法分配，则发生错误。
- 如果 DE 不用于传递参数，通过寄存器 AX 或 A 进行传递的参数被指定到 DE 或 E。否则，参数被指定到 **_@RTARG6** 或_**@RTARG7**。
  对于其他参数，用来传递的位置就是被指定的位置。

（3） 存储自动变量的位置

- 如果没有参数被指定，则自动变量被分配到 **DE** 寄存器，**_@RTARG6**，**_@RTARG7** 或_**@NRARGx**，然后指定到_**@NRATxx**。
- 仅当指定了**-QR** 选项时，自动变量才会分配到_**@NRARGx** 或_**@NRATxx**。
  关于_**@RTARGx** 和_**@NRATxx**，请参阅“附录 **A saddr** 区域标签列表”。
- 如果有自动变量无法分配，则发生错误。

第 12 章 汇编程序的引用

本章介绍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连接器进行连接。本章就是介绍在 C 源程序中调用其他编程语言程序的过程步骤，以及在其他语言程序中调用 C 语言程序的过程步骤。

本章将按如下顺序来介绍如何使用 RA78K0R 汇编程序包和 C 编译器来实现 C 语言与另外一种编程语言的接口：

（1） 由 C 语言调用汇编语言程序函数。
（2） 由汇编语言调用 C 语言程序函数。
（3） 访问 C 语言中定义的变量。
（4） 在 C 语言程序中访问由汇编语言定义的变量。
（5） 注意事项

12.1 访问参数/自动变量

关于参数和自动变量的赋值细节，敬请参阅“11.7.2 普通函数调用接口”和“11.7.3 norec 函数调用接口”。

访问存储在堆栈中的参数和自动变量时，HL 寄存器被用作基址指针。

12.2 返回值的存储

敬请参阅“11.7.1 返回值”。

12.3 在C语言程序中调用汇编语言程序

本节显示的是默认过程的示例。

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 C 语言程序调用汇编语言程序的示例。

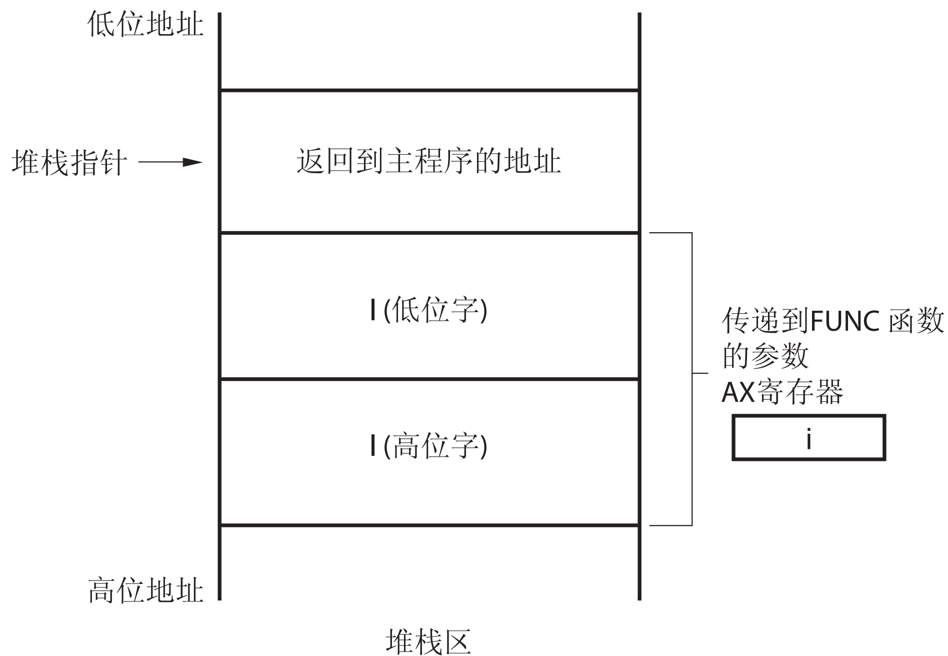
```
extern int FUNC(int, long);                /* 函数原型 */

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， 程序接口及程序执行的流程如下所述。

（i） 将 **main** 函数传给 **FUNC** 函数的第一个参数放置在寄存器中，将第二个及后续参数都放置在堆栈中。

（ii） 使用 **CALL** 指令将控制权移交给 **FUNC** 函数。

下图所显示的是上例中在控制权移交给 **FUNC** 函数后的即时堆栈情况。



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，让我们通过汇编语言程序示例来进行解释。

```
$PROCESSOR ( F1166A0 )
        PUBLIC   _FUNC
        PUBLIC   _DT1
        PUBLIC   _DT2
@@DATA DSEG BASEP
_DT1 :   DS      ( 2 )
_DT2 :  DS       ( 4 )

@@CODE CSEG
_FUNC :
        PUSH     HL                      ; 保存基址指针          （1）
        PUSH     AX
        MOVW     HL , SP                 ; 拷贝堆栈指针          （2）
        MOVW     AX , [ HL ]             ; arg1
        MOVW     !_DT1 , AX              ; 移动第一个参数(i)
        MOVW     AX , [ HL + 10 ]        ; arg2
        MOVW     !_DT2 + 2 , AX
        MOVW     AX , [ HL + 8 ]         ; arg2
        MOVW     !_DT2 , AX              ; 移动第二个参数 (l)
        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为寄存器变量保存寄存器。因此，如果被调函数使用的工作寄存器发生改变时，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，则无需对其进行保存。

（2） 将基址指针（HL）拷贝至堆栈指针（SP）

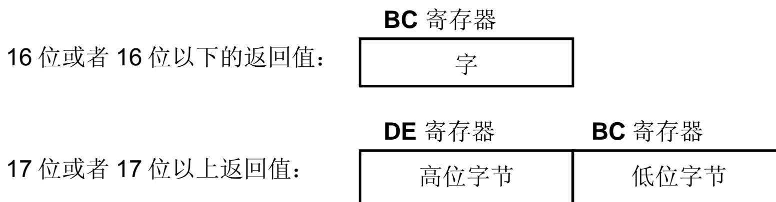
由于函数内部的‘PUSH，POP’指令改变了堆栈指针（**SP**）的值，因此要将堆栈指针拷贝至‘HL’寄存器，用做参数的基址指针。

（3） FUNC 函数基本处理过程

第（1）和（2）两步处理过程执行后，进行被调函数的基本处理过程。

（4） 设置返回值

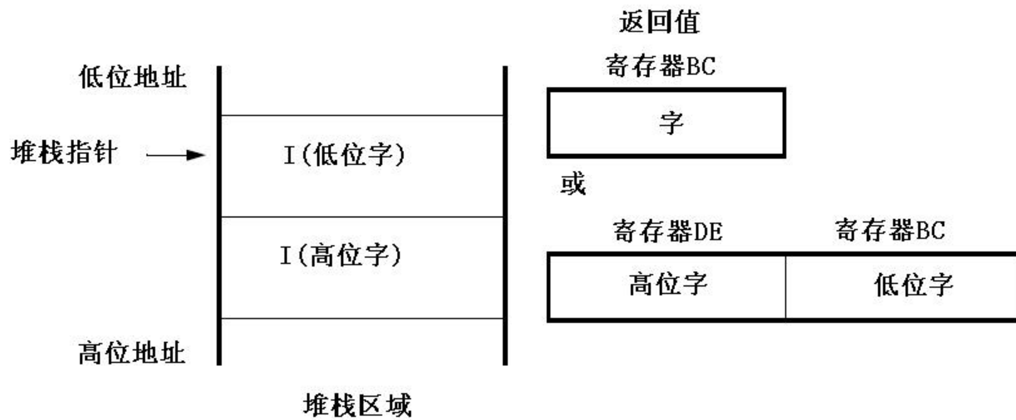
如果有返回值，则将其存储在‘BC’和‘DE’寄存器中，如果没有返回值，就无需进行存储。



（5） 恢复寄存器值

恢复保存的基址指针和工作寄存器。

（6） 返回至主函数



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 语言函数的过程为以下几步：
（1） 保存 C 工作寄存器（**AX**, **BC**, 及 **DE**）
（2） 将参数保存到堆栈。
（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 (F1166A0)

        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)"函数
        pop     ax                  ;
        ret
        END
```

（1） 保存工作寄存器（AX，BC 和 DE）

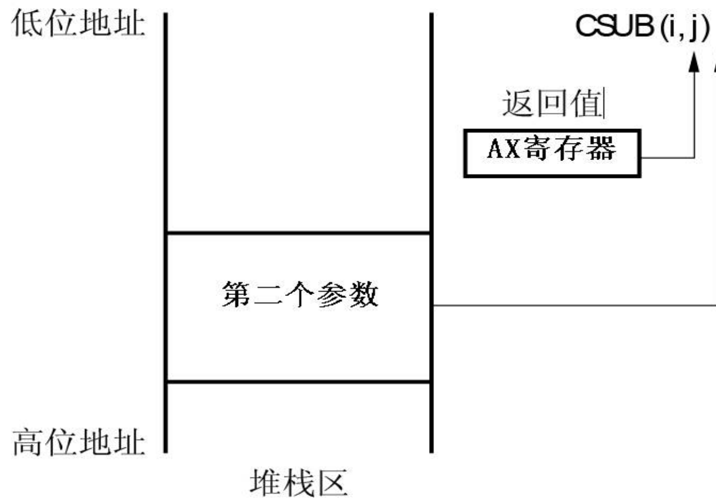
在 C 语言中使用三对寄存器 **AX**, **BC** 及 **DE**，在调用返回时并不恢复这些寄存器的值，因此，如果需要使用这些寄存器中的值，就要在函数调用方将这些值保存起来。

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，该寄存器始终需要在 C 语言函数方进行保存。

（2） 参数入栈

任何参数均放置在堆栈中。图 12-4 显示了参数传递情况。



（3） 调用 C 语言函数

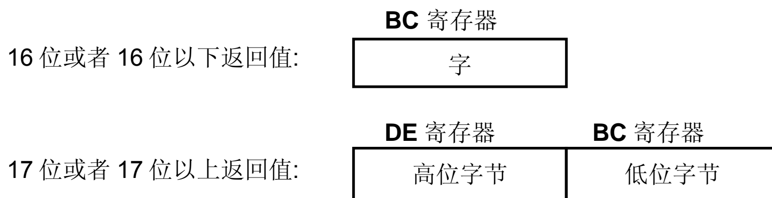
用 **CALL** 指令调用 C 语言函数。如果该 C 语言函数为 **callt** 函数，那么就由 callt 指令执行函数调用。

（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。



12.5 引用其它语言定义的变量

12.5.1 引用其它语言中定义的变量

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 RA78K0 汇编器下的汇编程序。

```
$PROCESSOR (F1166A0)

          PUBLIC    _subf
          EXTRN     _c
          EXTRN     _i

@@CODE CSEG
_subf:
          MOV       !_c , #04H
          MOVW      AX , #07H
          MOVW      !_i , AX
          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 RA78K0 汇编器下的汇编程序。

```
          NAME ASMSUB

          PUBLIC    _c
          PUBLIC    _i

ABC       CSEG
_c:       DB        0
_i:       DW        0

          END
```

12.6 注意事项

（1）‘_’（下划线）

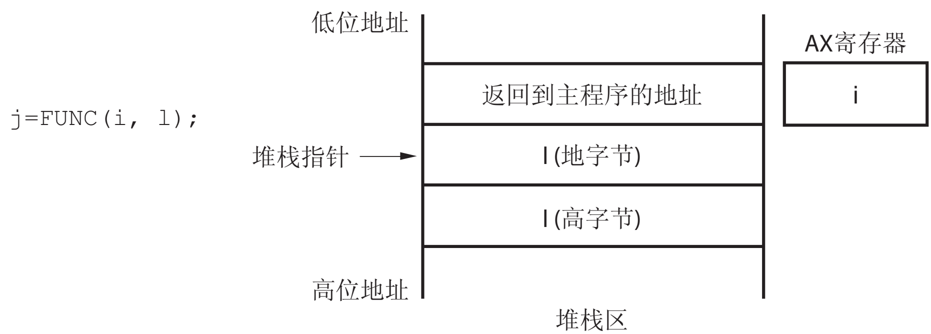
该 C 编译器对外部定义及输出的目标模块中的引用名称前加一个下划线‘_’（ASCII 码 ‘5FH’）。在下面 C 程序示例中，“j = FUNC（i, l）;” 就被认为是对外部函数名称_FUNC 的一个引用。

```
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 */
}
```

RA78K0 中的程序名称为‘_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。



第 13 章 编译器的有效应用

本章介绍如何有效地使用该 C 编译器。

13.1 高效编码

在使用 78K0R 系列 MCU 开发应用产品时，通过使用 **saddr** 区域或者使用设备的 **callt** 区域，该 C 编译器生成高效的目标代码。

- 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（使用自动变量 并且 **saddr** 区域可用）
    - └ 声明为 register 型

（1） 使用外部变量

如果 **saddr** 区域可用，则定义外部变量时，将变量定义为 **sreg/_ _sreg** 变量。**sreg/_ _sreg** 变量的指令代码长度要比存储器指令代码长度短，这有助于减小目标代码长度从而提高程序执行速度。（不使用 **sreg** 变量，而通过指定 **-RD** 选项也可以获得相同效果。）

定义 sreg/_ _sreg 变量:	extern sreg int 变量名称； extern_ _sreg int 变量名称；

备注 请参阅“**11.5 如何使用 saddr 区域**”。

（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，于是便缩短了目标代码，也提高了程序执行速度。

声明 bit/boolean 型变量:	bit 变量称； boolean 变量称； _ _boolean 变量称；

备注 请参阅“**11.5 位变量与布尔型变量（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使用 **callt** 表区域，那么这些函数就应该定义为 **callt** 函数。

由于对 **callt** 函数的调用是使用设备的 **callt** 区进行的，所以，对 **callt** 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。

```
callt 函数的定义: callt          int tsub () {
                                  :
}
```

备注 请参阅“**11.5 callt 函数**”及“**11.5 norec 函数**”。

除了使用 **saddr** 区域之外，通过使用优化选项编译来生成一些无需修改的 C 源程序目标代码。如果用户需要了解每个**-Q** 子选项详情，敬请参阅 **CC78K0R C 编译器操作篇**。

（4） 使用扩展描述

- 函数定义

- 函数没有被递归调用

  关于被反复调用的函数，那些没有使用递归算法的函数应该声明为_ _**leaf/norec** 函数。
  norec 函数没有预处理和后处理过程（堆栈帧）。
  因此与普通函数相比，可以缩短目标代码并且提高程序执行速度。

  备注 有关 **norec** 函数定义 （**norec int rout ( )**...），请参阅 **11.5 norec 函数**和 **11.7.3 norec 函数调用接口**。

- 使用自动变量的函数

  对于那些不使用自动变量的函数，如果 **saddr** 区域可用，用 **register** 存储类型限定符来进行函数声明。通过该 **register** 声明，声明的目标将被分配至寄存器中。
  使用寄存器操作数的程序执行速度要比使用存储器操作数的程序执行速度快，而且目标代码也同样会缩小。

  备注 用于 **register** 变量（register int i; ...）定义的，请参阅 **11.5 寄存器变量（register）**

- 使用内部静态变量的函数

  如果 **saddr** 区域可以用于那些使用内部静态的函数，则用__**sreg** 声明函数或者指定**-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 请参阅 **11.5 如何使用 saddr 区域**。

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

- 对那些只有位成员的位域，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 ;
```

- 在中断处理中使用寄存器组切换。

  #pragma interrupt INTP0 inter RB1

- 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 用于saddr区域的标签列表

在 CC78K0R 编译器中，可以用下列标签名来引用 **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) 寄存器变量

标签名	地址
_@KREG00	0FFEB4H
_@KREG01	0FFEB5H
_@KREG02	0FFEB6H
_@KREG03	0FFEB7H
_@KREG04	0FFEB8H
_@KREG05	0FFEB9H
_@KREG06	0FFEBAH
_@KREG07	0FFEBBH
_@KREG08	0FFEBCH
_@KREG09	0FFEBDH
_@KREG10	0FFEBEH
_@KREG11	0FFEBFH
_@KREG12	0FFEC0H [注]
_@KREG13	0FFEC1H [注]
_@KREG14	0FFEC2H [注]
_@KREG15	0FFEC3H [注]

注　当函数参数由 **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** 区域域。

(b) norec 函数的参数

标签名	地址
_@NRARG0	0FFEC4H
_@NRARG1	0FFEC6H
_@NRARG2	0FFEC8H
_@NRARG3	0FFECAH

(c) norec 函数的自动变量

标签名	地址
_@NRAT00	0FFECCH
_@NRAT01	0FFECDH
_@NRAT02	0FFECEH
_@NRAT03	0FFECFH
_@NRAT04	0FFED0H
_@NRAT05	0FFED1H
_@NRAT06	0FFED2H
_@NRAT07	0FFED3H

(d) 保存区段信息

标签名	地址
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

(e) 运行时刻的参数

标签名	地址
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

附录B 区段名称列表

这一章介绍编译器输出的所有段及这些段的位置情况。

（1） 和 （2） 显示了以下各表中使用的选项及重定位属性。

本节描述编译器输出的所有的段。

（1） CSEG 重定位段属性

CALLT0：	将指定段的起始地址定位在 80H 至 BFH 范围内的一个偶地址处。
AT 绝对表达式：	将指定段定位至某个绝对地址处（地址范围为： 00000H 至 FFEFFH）.
UNITP：	将指定段的起始地址定位在范围内任意一个偶地址处。（地址范围为：C0H 至 EFFFEH）.

（2） DSEG 重定位属性

SADDRP：	将指定段起始地址定位在 saddr 区域的 FFFE20H 至 FFEFFH 范围内某一个偶地址处。
UNITP：	将指定段起始地址定位在范围内任意一个偶地址处。（默认范围在 RAM 区内）。

B.1 区段名称列表

B.1.1 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	BASE	存放代码的区段（分配到 near 区域）
@@CODEL	CSEG		存放代码的区段（分配到 far 区域）
@@LCODE	CSEG	BASE	存放库代码的区段（分配到 near 区域）
@@LCODEL	CSEG		存放代码的区段（分配到 far 区域）
@@CNST	CSEG	MIRRORP	存放 **const** 常量的区段（分配到 near 区域）
@@CNSTL	CSEG	PAGE64KP	存放 **const** 常量的区段（分配到 far 区域）
@@R_INIT	CSEG	UNIT64KP	存放 near 初始数据的区段（具有初始值）
@@RLINIT	CSEG	UNIT64KP	存放 far 初始数据的区段（具有初始值）
@@R_INIS	CSEG	UNIT64KP	存放初始数据的区段（具有初始值的 **sreg** 变量）
@@CALT	CSEG	CALLT0	存放 **callt** 函数表的区段
@@VECTnn	CSEG	AT 00mmH	存放中断矢量表的区段[注]
@@BASE	CSEG	BASE	存放 callt 函数和中断函数的区段
@@LBASE	CSEG	BASE	存放库和 callt 函数的区段
@@INIT	DSEG	BASEP	数据区段（具有初始值，分配在 near 区域）
@@INITL	DSEG	UNIT64KP	数据区段（具有初始值，分配在 far 区域）
@@DATA	DSEG	BASEP	数据区段（没有初始值，分配在 near 区域）
@@DATAL	DSEG	UNIT64KP	数据区段（具有初始值，分配在 far 区域）
@@INIS	DSEG	SADDRP	数据区段（具有初始值的 **sreg** 变量）
@@DATS	DSEG	SADDRP	数据区段（没有初始值的 **sreg** 变量）
@@BITS	BSEG		存放 **boolean** 型（布尔型）及 **bit** 型（位型）变量的区段

注　　nn 和 mm 的具体数值由中断类型决定。

B.1.2 flash存储器区域

区段名称	区段类型	重定位属性	说明
@@ECODE	CSEG	BASE	存放代码的区段（分配到 near 区域）
@@ECODEL	CSEG		存放代码的区段（分配到 far 区域）
@@LECODE	CSEG	BASE	存放库代码的区段（分配到 near 区域）
@@LECODEL	CSEG		存放库代码的区段（分配到 far 区域）
@@ECNST	CSEG	MIRRORP	存放 **const** 常量的区段（分配到 near 区域）
@@ECNSTL	CSEG	PAGE64KP	存放 **const** 常量的区段（分配到 far 区域）
@@ER_INIT	CSEG	UNIT64KP	存放 near 初始数据的区段（具有初始值）
@@ERLINIT	CSEG	UNIT64KP	存放 far 初始数据的区段（具有初始值）
@@ER_INIS	CSEG	UNIT64KP	存放初始数据的区段（具有初始值的 **sreg** 变量）
@@EVECTnn	CSEG	AT mmmmH	存放中断矢量表的区段[注 1]
@@EXTxx	CSEG	AT yyyyH	存放 Flash 区域跳转表的区段[注 2]
@@EINIT	DSEG	BASEP	数据区段（具有初始值，分配在 near 区域）
@@EINITL	DSEG	UNIT64KP	数据区段（具有初始值，分配在 far 区域）
@@EDATA	DSEG	BASEP	数据区段（没有初始值，分配在 near 区域）
@@EDATAL	DSEG	UNIT64KP	数据区段（没有初始值，分配在 far 区域）
@@EINIS	DSEG	SADDRP	数据区段（具有初始值的 **sreg** 变量）
@@EDATS	DSEG	SADDRP	数据区段（没有初始值的 **sreg** 变量）
@@EBITS	BSEG		存放 **boolean** 型（布尔型）及 **bit** 型（位型）变量的区段

注 1 nn 和 mm 的具体数值由中断类型决定。

注 2 xx 和 yyyy 的具体数值由 flash 区域函数的 ID 值决定。

B.2 区段位置

表 **B-1** 区段位置

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  rb1               /* 中断矢量 */

Void          inter (void) ;                                 /* 中断函数原型声明*/
const         int  i_cnst = 1 ;                              /* const 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 ;                                   /* 具有初始值的外部变量*/
int           i_data ;                                       /*不具有初始值的外部变量*/
__sreg int    sr_inis = 3 ;                                  /* 具有初始值的 sreg 变量*/
__sreg int    sr_dats ;                                      /*不具有初始值的 sreg 变量*/

void main （）                                                /* 函数定义*/
{
              int           i ;
              i = 100 ;
}

void          inter （）                                      /* 中断函数定义*/
{
}

callt         void f_clt （）                                 /* callt 函数定义 */
{
}
```

B.4 输出汇编模块示例

汇编语言源程序中的伪指令及指令集随着目标设备的不同而有所不同。

详情请参阅 RA78K0R 用户手册在线帮助。

[当指定为小型模式时]

```
; 78K0R 系列 C 编译器 Vx.xx 汇编源程序
;                                                        日期:xx xxx xxxx 时间:xx:xx:xx

; 命令              : -cxxx sample.c –ms –sa –ng
; 输入文件          : sample.c
; 产生的汇编文件    : sample.asm
; 参数文件          :

$PROCESSOR(xxx)
$NODEBUG
$NODEBUGA
$KANJICODE              SJIS
$TOL_INF                03FH, 0330H, 00H, 020H,00H

            PUBLIC      _inter
            PUBLIC      _main
            PUBLIC      _i_cnst
            PUBLIC      ?f_clt
            PUBLIC      _b_bit
            PUBLIC      _l_init
            PUBLIC      _i_data
            PUBLIC      _sr_inis
            PUBLIC      _sr_dats
            PUBLIC      _f_clt
            PUBLIC      _@vect06

@@BITS      BSEG                            ; 存放 boolean 型（布尔型）及 bit 型（位型）变量的区
段
_b_bit      DBIT
@@CNST      CSEG        MIRRORP             ; 存放 const 常量的区段
_i_cnst :   DW          01H         ; 1
@@R_INIT    CSEG        UNIT64KP            ; 存放初始数据的区段（具有初始值的外部变量）
            DW          00002H , 00000H ; 2
@@INIT      DSEG        BASEP               ; 存放临时数据的区段（具有初始值）
_l_init :   DS          ( 4 )
@@DATA      DSEG        BASEP               ; 存放临时数据的区段（不具有初始值）
_i_data :   DS          ( 2 )
@@R_INIS    CSEG        UNIT64KP            ; 存放初始数据的区段（具有初始值的 sreg 变量）
            DW          03H         ; 3
@@INIS      DSEG        SADDRP              ; 存放临时数据的区段（具有初始值的 sreg 变量）
_sr_inis :  DS          ( 2 )
@@DATS      DSEG        SADDRP              ; 存放临时数据的区段（不具有初始值的 sreg 变量）
_sr_dats :  DS          ( 2 )
@@CALT      CSEG        CALLT0              ; 存放 callt 函数表的区段
?f_clt :    DW          _f_clt
```

```
; line 1 :      #pragma INTERRUPT        INTP0 inter rb1          /*中断矢量*/
; line 2 :
; line 3 :      void          main ( void ) ;                     /*中断函数原型声明*/
; line 4 :      const         int i_cnst = 1 ;                    /* const 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 ;                        /*具有初始值的外部变量*/
; line 8 :      int           i_data ;                            /*不具有初始值的外部变量*/
; line 9 :      __sreg int    sr_inis = 3 ;                       /*具有初始值的 sreg 变量*/
; line 10 :     __sreg int    sr_dats ;                           /*不具有初始值的 sreg 变量*/
; line 11 :
; line 12 :     void          main ( )                            /* 函数定义 */
; line 13 : {
@@CODE          CSEG          BASE                                ; 存放代码部分的区段
_main :
                push          hl
; line 14 :      int i ;
; line 15 :     i = 100 ;
                movw hl , #064H ; 100
; line 16 : }
                pop           hl
                ret
; line 17 :
; line 18 : void inter ( )                                         /* 中断函数定义 */
; line 19 : {
@@BASE          CSEG          BASE                                ; 存放 callt 和中断函数的区段
_inter :
; line 20 : }
                reti
; line 21 :
; line 22 :     callt void    f_clt ( )                           /* callt 函数定义 */
; line 23 : {
_f_clt:
; line 24 : }
                ret
@@VECT06        CSEG          AT 0006H                            ; 存放中断向量表的区段
_@vect06 :
                DW            _inter
                END
; Target chip : uPDxxxx
; Device file : xx.xxx
```

[当指定为中等模式时]

```
; 78K0R 系列 C 编译器 Vx.xx 汇编源程序
;                                                    日期:xx xxx xxxx 时间:xx:xx:xx

; 命令                : -cxxx sample.c –mm –sa –ng
; 输入文件            : sample.c
; 产生的汇编文件      : sample.asm
; 参数文件            :

$PROCESSOR(xxx)
$NODEBUG
$NODEBUGA
$KANJICODE              SJIS
$TOL_INF                03FH, 0330H, 00H, 020H,00H

            PUBLIC      _inter
            PUBLIC      _main
            PUBLIC      _i_cnst
            PUBLIC      ?f_clt
            PUBLIC      _b_bit
            PUBLIC      _l_init
            PUBLIC      _i_data
            PUBLIC      _sr_inis
            PUBLIC      _sr_dats
            PUBLIC      _f_clt
            PUBLIC      _@vect06

@@BITS      BSEG                                ; 存放 boolean 型（布尔型）及 bit 型（位型）变量的区
段
_b_bit      DBIT
@@CNST      CSEG        MIRRORP                 ; 存放 const 常量的区段
_i_cnst :   DW          01H             ; 1
@@R_INIT    CSEG        UNIT64KP                ; 存放初始数据的区段（具有初始值的外部变量）
            DW          00002H , 00000H ; 2
@@INIT      DSEG        BASEP                   ; 存放临时数据的区段（具有初始值）
_l_init :   DS          ( 4 )
@@DATA      DSEG        BASEP                   ; 存放临时数据的区段（不具有初始值）
_i_data :   DS          ( 2 )
@@R_INIS    CSEG        UNIT64KP                ; 存放初始数据的区段（具有初始值的 sreg 变量）
            DW          03H             ; 3
@@INIS      DSEG        SADDRP                  ; 存放临时数据的区段（具有初始值的 sreg 变量）
_sr_inis :  DS          ( 2 )
@@DATS      DSEG        SADDRP                  ; 存放临时数据的区段（不具有初始值的 sreg 变量）
_sr_dats :  DS          ( 2 )
@@CALT      CSEG        CALLT0                  ; 存放 callt 函数表的区段
?f_clt :    DW          _f_clt
```

```
; line 1 :      #pragma INTERRUPT        INTP0  inter  rb1          /*中断矢量*/
; line 2 :
; line 3 :      void         main  ( void ) ;                       /*中断函数原型声明*/
; line 4 :      const        int i_cnst = 1 ;                       /* const 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 ;                           /*具有初始值的外部变量*/
; line 8 :      int          i_data ;                               /*不具有初始值的外部变量*/
; line 9 :      __sreg int   sr_inis = 3 ;                          /*具有初始值的 sreg 变量*/
; line 10 :     __sreg int   sr_dats ;                              /*不具有初始值的 sreg 变量*/
; line 11 :
; line 12 :     void         main ( )                               /* 函数定义 */
; line 13 : {
@@CODEL   CSEG                                                      ; 存放代码部分的区段
_main :
          push         hl
; line 14 :  int         i ;
; line 15 :     i = 100 ;
          movw         hl , #064H ; 100
; line 16 : }
          pop           hl
          ret
; line 17 :
; line 18 : void           inter ( )                                 /* 中断函数定义 */
; line 19 : {
@@BASE    CSEG         BASE                                         ; 存放 callt 和中断函数的区段
_inter :
; line 20 : }
          reti
; line 21 :
; line 22 :     callt void   f_clt ( )                              /* callt 函数定义 */
; line 23 : {
_f_clt:
; line 24 : }
          ret
@@VECT06  CSEG         AT 0006H                                     ; 存放中断向量表的区段
_@vect06 :
          DW           _inter
          END

; Target chip : uPDxxxx
; Device file : xx.xxx
```

[当指定为紧凑模式时]

```
; 78K0R 系列 C 编译器 Vx.xx 汇编源程序
;                                                        日期:xx xxx xxxx 时间:xx:xx:xx

; 命令                  : -cxxx sample.c –mc –sa –ng
; 输入文件              : sample.c
; 产生的汇编文件        : sample.asm
; 参数文件              :

$PROCESSOR(xxx)
$NODEBUG
$NODEBUGA
$KANJICODE              SJIS
$TOL_INF                03FH, 0330H, 00H, 020H,00H

          PUBLIC        _inter
          PUBLIC        _main
          PUBLIC        _i_cnst
          PUBLIC        ?f_clt
          PUBLIC        _b_bit
          PUBLIC        _l_init
          PUBLIC        _i_data
          PUBLIC        _sr_inis
          PUBLIC        _sr_dats
          PUBLIC        _f_clt
          PUBLIC        _@vect06

@@BITS    BSEG                                  ; 存放 boolean 型（布尔型）及 bit 型（位型）变量的区
段
_b_bit    DBIT
@@R_INIS  CSEG          UNIT64KP                ; 存放初始数据的区段（具有初始值的外部变量）
          DW            03H             ; 3
@@INIS    DSEG          SADDRP                  ; 存放临时数据的区段（具有初始值的 sreg 变量）
_sr_inis :  DS          ( 2 )
@@DATS    DSEG          SADDRP                  ; 存放临时数据的区段（不具有初始值的 sreg 变量）
_sr_dats :  DS          ( 2 )
@@CNSTL   CSE           G PAGE64KP              ; 存放 const 常量的区段
_i_cnst : DW            01H             ; 1
@@RLINIT  CSEG          UNIT64KP                ; 存放初始数据的区段（具有初始值）
          DW            00002H , 00000H ; 2
@@INITL   DSEG          UNIT64KP                ; 存放临时数据的区段（具有初始值）
_l_init : DS            ( 4 )
@@DATAL   DSEG          UNIT64KP                ; 存放临时数据的区段（不具有初始值）
_i_data : DS            ( 2 )
@@CALT    CSEG          CALLT0                  ; 存放 callt 函数表的区段
?f_clt :  DW            _f_clt
```

```
; line 1 :      #pragma INTERRUPT          INTP0  inter  rb1          /*中断矢量*/
; line 2 :
; line 3 :      void           main  ( void ) ;                       /*中断函数原型声明*/
; line 4 :      const          int i_cnst = 1 ;                       /* const 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 ;                           /*具有初始值的外部变量*/
; line 8 :      int            i_data ;                               /*不具有初始值的外部变量*/
; line 9 :      __sreg int     sr_inis = 3 ;                          /*具有初始值的 sreg 变量*/
; line 10 :     __sreg int     sr_dats ;                              /*不具有初始值的 sreg 变量*/
; line 11 :
; line 12 :     void           main ( )                               /* 函数定义 */
; line 13 : {
@@CODE          CSEG           BASE                                   ; 存放代码部分的区段
_main :
                push           hl
; line 14 :      int           i ;
; line 15 :     i = 100 ;
                movw           hl , #064H ; 100
; line 16 : }
                pop             hl
                ret
; line 17 :
; line 18 : void               inter ( )                               /* 中断函数定义 */
; line 19 : {
@@BASE          CSEG           BASE                                   ; 存放 callt 和中断函数的区段
_inter :
; line 20 : }
                reti
; line 21 :
; line 22 :     callt void     f_clt ( )                              /* callt 函数定义 */
; line 23 : {
_f_clt:
; line 24 : }
                ret
@@VECT06        CSEG           AT 0006H                               ; 存放中断向量表的区段
_@vect06 :
                DW             _inter
                END

; Target chip : uPDxxxx
; Device file : xx.xxx
```

[当指定为大型模式时]

```
; 78K0R 系列 C 编译器 Vx.xx 汇编源程序
;                                                      日期:xx xxx xxxx 时间:xx:xx:xx

; 命令              : -cxxx sample.c –ml –sa –ng
; 输入文件          : sample.c
; 产生的汇编文件    : sample.asm
; 参数文件          :

$PROCESSOR(xxx)
$NODEBUG
$NODEBUGA
$KANJICODE              SJIS
$TOL_INF                03FH, 0330H, 00H, 020H,00H

            PUBLIC      _inter
            PUBLIC      _main
            PUBLIC      _i_cnst
            PUBLIC      ?f_clt
            PUBLIC      _b_bit
            PUBLIC      _l_init
            PUBLIC      _i_data
            PUBLIC      _sr_inis
            PUBLIC      _sr_dats
            PUBLIC      _f_clt
            PUBLIC      _@vect06

@@BITS      BSEG                            ; 存放 boolean 型（布尔型）及 bit 型（位型）变量的区
段
_b_bit      DBIT
@@R_INIS    CSEG        UNIT64KP            ; 存放初始数据的区段（具有初始值的外部变量）
            DW          03H     ; 3
@@INIS      DSEG        SADDRP              ; 存放临时数据的区段（具有初始值的 sreg 变量）
_sr_inis :  DS          ( 2 )
@@DATS      DSEG        SADDRP              ; 存放临时数据的区段（不具有初始值的 sreg 变量）
_sr_dats :  DS          ( 2 )
@@CNSTL     CSEG        PAGE64KP            ; 存放 const 常量的区段
_i_cnst :   DW          01H     ; 1
@@RLINIT    CSEG        UNIT64KP            ; 存放初始数据的区段（具有初始值）
            DW          00002H , 00000H ; 2
@@INITL     DSEG        UNIT64KP            ; 存放临时数据的区段（具有初始值）
_l_init :   DS          ( 4 )
@@DATAL     DSEG        UNIT64KP            ; 存放临时数据的区段（不具有初始值）
_i_data :   DS          ( 2 )
@@CALT      CSEG        CALLT0              ; 存放 callt 函数表的区段
?f_clt :    DW          _f_clt
```

```
; line 1 :      #pragma INTERRUPT          INTP0 inter rb1            /*中断矢量*/
; line 2 :
; line 3 :      void          main ( void ) ;                         /*中断函数原型声明*/

; line 4 :      const         int i_cnst = 1 ;                        /* const 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 ;                            /*具有初始值的外部变量*/
; line 8 :      int           i_data ;                                /*不具有初始值的外部变量*/
; line 9 :      sreg int      sr_inis = 3 ;                           /*具有初始值的 sreg 变量*/
; line 10 :     sreg int      sr_dats ;                               /*不具有初始值的 sreg 变量*/
; line 11 :
; line 12 :     void          main ( )                                /* 函数定义 */
; line 13 : {
@@CODEL         CSEG                                                  ; 存放代码部分的区段
_main :
                push          hl
; line 14 :      int          i ;
; line 15 :     i = 100 ;
                movw          hl , #064H ; 100
; line 16 : }
                pop           hl
                ret
; line 17 :
; line 18 : void              inter ( )                               /* 中断函数定义 */
; line 19 : {
@@BASE          CSEG          BASE                                    ; 存放 callt 和中断函数的区段
_inter :
; line 20 : }
                reti
; line 21 :
; line 22 :     callt void    f_clt ( )                               /* callt 函数定义 */
; line 23 : {
_f_clt:
; line 24 : }
                ret
@@VECT06        CSEG          AT 0006H                                ; 存放中断向量表的区段
_@vect06 :
                DW            _inter
                END
; Target chip : uPDxxxx
; Device file : xx.xxx
```

附录C 运行时刻库列表

下表显示的是运行时刻库列表。

调用这些操作指令时，需要指定的格式，比如在函数名称以@@ 等符号开始。但是，对 **cstart**, **cstarte** **,cprep** 及 **cdisp** 指令的调用，格式是在函数名前加上_@ 。

运行时刻库不支持的操作指令未列于表 C-1 之中。

编译器执行内联（inline）展开。

long 型加法及减法运算，与/或/非(**and/or/xor**)及移位操作也可进行行内联展开。

表 **C-1** 运行时刻库列表

分类	函数名称	函数
加 1	lsinc	有符号长整型数据加 1
	luinc	无符号长整型型数据加 1
	finc	浮点型数据加 1
减 1	lsdec	有符号长整型数据减 1
	ludec	无符号长整型数据减 1
	fdec	浮点型数据减 1
符号 取反	lsrev	有符号长整型数据的符号取反
	lurev	无符号长整型数据的符号取反
	frev	浮点型数据的符号取反
1 的补码	lscom	得到有符号长整型数据的 1 的补码
	lucom	得到无符号长整型数据的 1 的补码
逻辑 非	lsnot	有符号长整型数据求反
	lunot	无符号长整型数据求反
	fnot	浮点型数据符号求反
乘法	csmul	在有符号字符型数据之间执行乘法运算
	cumul	在无符号字符型数据之间执行乘法运算
	ismul	在有符号整型数据之间执行乘法运算
	iumul	在无符号整型数据之间执行乘法运算
	lsmul	在有符号长整型数据之间执行乘法运算
	lumul	在无符号长整型数据之间执行乘法运算
	fmul	在浮点型数据之间执行乘法运算

表 C-1. 运行时刻库列表

分类	函数名称	函数
除法	csdiv	在有符号字符型数据之间执行除法运算
	cudiv	在无符号字符型数据之间执行除法运算
	isdiv	在有符号整型数据之间执行除法运算
	iudiv	在无符号整型数据之间执行除法运算
	lsdiv	在有符号长整型数据之间执行除法运算
	ludiv	在无符号长整型数据之间执行除法运算
	fdiv	在浮点型数据之间执行除法运算
取余	csrem	在有符号字符型数据之间执行除法运算后，获取余数
	curem	在无符号字符型数据之间执行除法运算后，获取余数
	isrem	在有符号整型数据之间执行除法运算后，获取余数
	iurem	在无符号整型数据之间执行除法运算后，获取余数
	lsrem	在有符号长整型数据之间执行除法运算后，获取余数
	lurem	在无符号长整型数据之间执行除法运算后，获取余数
加法	lsadd	在有符号长整型数据之间执行加法运算
	luadd	在无符号长整型数据之间执行加法运算
	fadd	在浮点数据之间执行加法运算
减法	lssub	在有符号长整型数据之间执行减法运算
	lusub	在无符号长整型数据之间执行减法运算
	fsub	在浮点型数据之间执行加碱运算
左移	lslsh	将有符号长整型数据左移
	lulsh	将无符号长整型数据左移
右移	lsrsh	将有符号长整型数据右移
	lursh	将无符号长整型数据右移
比较	cscmp	有符号字符型数据比较
	iscmp	有符号整型数据比较
	lscmp	有符号长整型数据比较
	lucmp	无符号长整型数据比较
	fcmp	浮点型数据比较
按位与	lsband	在有符号长整型数据之间执行与（AND）运算
	luband	在无符号长整型数据之间执行与（AND）运算
按位或	lsbor	在有符号长整型数据之间执行或（OR）运算
	lubor	在无符号长整型数据之间执行或（OR）运算
按位异或	lsbxor	在有符号长整型数据之间执行异或（XOR）运算
	lubxor	在无符号长整型数据之间执行异或（XOR）运算

表 C-1 运行时刻库列表

分类	函数名称	函数
浮点数的转换	ftols	将浮点数转换成有符号长整型数
	ftolu	将浮点数转换成无符号长整型数
转换为浮点数	lstof	将有符号长整型数转换成浮点数
	lutof	将无符号长整型数转换成浮点数
位数据的转换	btol	将位型数据转换成长整型数据
启动例程	cstart	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。
函数的前处理和后处理	cprep	函数的预处理
	cdisp	函数的后处理
	cprep2	函数预处理 （包括用于寄存器变量的 **saddr** 区域）
	cdisp2	函数后处理（包括用于寄存器变量的 **saddr** 区域）
	cprep3	函数预处理 （包括用于寄存器变量的 **saddr** 区域）
	cdisp3	函数后处理（包括用于寄存器变量的 **saddr** 区域）
	hdwinit	在 CPU 复位后，立即执行对外围设备(sfr)的初始化处理

表 C-1 运行时刻库列表

分类	函数名称	函数
辅助函数	mulu	兼容 **mulu** 指令
	mulue	兼容 **mulu** 指令
	divuw	兼容 **divuw** 指令
	divuwe	兼容 **divuw** 指令
	cmpa1	用来替代固定类型指令模式
	cmpax1	
	ctoi	
	incde	
	decde	
	inchl	
	dechl	
	dellab	
	dell03	
	della4	
	delsab	
	dels03	
	hlllab	
	hlll03	
	hllla4	
	hllsab	
	hlls03	
	apinch	
	apdech	
	incwhl	
	decwhl	
	swap4	
	tableh	
	uctoi	

附录D 库占用的堆栈列表

下表显示了标准库所占用的堆栈数量。

表 **D-1** 标准库占用堆栈列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
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	0
	va_start	0	0
	va_starttop	0	0
	va_end	0	0

表 D-1 标准库堆栈占用空间列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
stdio.h	sprintf	52 (102)[注 1]	52 (112)[注 1]
	sscanf	290 (330)[注 1]	290 (3384)[注 1]
	printf	60 (100)[注 1]	64 (104)[注 1]
	scanf	302 (328)[注 1]	306 (336)[注 1]
	vprintf	60 (100)[注 1]	66 (110)[注 1]
	vsprintf	52 (100)[注 1]	52 (110)[注 1]
	getchar	0	0
	gets	8	14
	putchar	0	0
	puts	6	10
	__putc	4	4

表 D-1 标准库堆栈占用空间列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
stdlib.h	atoi	4	4
	atol	10	10
	strtol	20	20
	strtoul	20	20
	calloc	12	12
	free	8	8
	malloc	6	6
	realloc	12	12
	abort	0	0
	atexit	0	0
	exit	6 + n 注 2	6 + n 注 2
	abs	0	0
	div	6	6
	labs	0	0
	ldiv	16	16
	brk	0	0
	sbrk	2	2
	atof	18	18
	strtod	18（30）注 6	20（34）注 6
	itoa	10	10
	ltoa	16	16
	ultoa	16	16
	rand	18（14）注 3	18（14）注 3
	srand	0	0
	bsearch	40 + n 注 4	44 + n 注 4
	qsort	16 + n 注 5	18 + n 注 5
	strbrk	0	0
	strsbrk	2	2
	stritoa	10	10
	strltoa	16	16
	strultoa	16	16

表 D-1 标准库堆栈占用空间列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
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 标准库堆栈占用空间列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
math.h	acos	34	34
	asin	34	34
	atan	34	34
	atan2	38	38
	cos	32	32
	sin	32	32
	tan	38	38
	cosh	38	38
	sinh	38	38
	tanh	44	44
	exp	34	34
	frexp	8 (20)[注 6]	12 (24)[注 6]
	ldexp	6 (20)[注 6]	6 (22)[注 6]
	log	34	34
	log10	34	34
	modf	8 (20)[注 6]	12 (24)[注 6]
	pow	38	38
	sqrt	26	26
	ceil	6(18)[注 6]	6(20)[注 6]
	fabs	4	4
	floor	6(18)[注 6]	6(20)[注 6]
	fmod	10 (22)[注 6]	10 (24)[注 6]
	matherr	0	0
	acosf	34	34
	asinf	34	34
	atanf	34	34
	atan2f	38	38
	cosf	32	32
	sinf	32	32
	tanf	38	38
	coshf	38	38
	sinhf	38	38

表 D-1 标准库堆栈占用空间列表

分类	函数名称	由小型模式和中等模式分享	由紧凑模式和大型模式分享
math.h	tanhf	44	44
	expf	34	34
	frexpf	8 (20)注 6	12 (24)注 6
	ldexpf	6 (20)注 6	6 (22)注 6
	logf	34	34
	log10f	34	34
	modff	8 (20)注 6	12 (24)注 6
	powf	38	38
	sqrtf	26	26
	ceilf	6 (18)注 6	6 (20)注 6
	fabsf	4	4
	floorf	6 (18)注 6	6(20)注 6
	fmodf	10 (22)注 6	10 (24)注 6
assert.h	_ _assertfail	72 (112)注 7	82 (122)注 7

注

1. 括号内数值适用于使用支持浮点数的版本。
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。
3. 括号中的数值是为乘法**/** 除法时使用的。
4. **n** 为 bsearch 函数调用的外部函数占用堆栈空间。
5. **n** 等于（**20 +** qsort 函数调用的的外部函数的堆栈占用空间）×（**1 +** 递归调用次数）。
   当使用由小型模式和中等模式分享的库 **X=38**
   当使用由紧凑模式和大型模式分享的库 **X=40**
6. 括号内的数值是指有操作异常发生时的数值。
7. 括号内的数值是指使用支持浮点数的 printf 版本时的数值。

表 D-2 显示了运行时刻库所占用的堆栈数目。

表 D-2 运行时刻库堆栈占用空间列表

分类	函数名称	堆栈消耗
加 1	lsinc	0
	luinc	0
	finc	20
减 1	lsdec	0
	ludec	0
	fdec	20
符号取反	lsrev	2
	lurev	2
	frev	4
二进制反码	lscom	0
	lucom	0
逻辑非	lsnot	0
	lunot	0
	fnot	4
乘法	csmul	0
	cumul	0
	ismul	4（2）注 2
	iumul	4（2）注 2
	lsmul	8（4）注 2
	lumul	8（4）注 2
	fmul	12 (28，30)注 1
除法	csdiv	8
	cudiv	2
	isdiv	10
	iudiv	4
	lsdiv	12
	ludiv	6
	fdiv	12 (28，30)注 1

表 D-2 运行时刻库堆栈占用空间列表

分类	函数名称	堆栈消耗
取余	csrem	8
	curem	2
	isrem	12
	iurem	6
	lsrem	12
	lurem	6
加法	lsadd	0
	luadd	0
	fadd	12 (28，30)注 1
减法	lssub	2
	lusub	2
	fsub	12 (28，30)注 1
左移	lslsh	4
	lulsh	4
右移	lsrsh	4
	lursh	4
比较	cscmp	0
	iscmp	0
	lscmp	2
	lucmp	2
	fcmp	8 (26，28)注 1
按位与	lsband	0
	luband	0
按位或	lsbor	0
	lubor	0
按位异或	lsbxor	0
	lubxor	0
浮点数的转换	ftols	10
	ftolu	10
转换为浮点数	lstof	10
	lutof	10
位数据的转换	btol	0

表 D-2 运行时刻库堆栈占用空间列表

分类	函数名称	堆栈消耗
启动例程	cstart	4
函数的前处理和后处理	cprep	2 + n [注 2]
	cdisp	0
	cprep2	基址指针+第一参数+寄存器变量+自动变量的大小
	cdisp2	0
	cprep3	基址指针+第一参数+寄存器变量+自动变量的大小
	cdisp3	0
	hdwinit	0

表 D-2 运行时刻库堆栈占用空间列表

分类	函数名称	普通模式
辅助函数	mulu	0
	mulue	0
	divuw	6
	divuwe	6
	cmpa1	0
	cmpax1	0
	ctoi	0
	incde	0
	decde	0
	inchl	0
	dechl	0
	dellab	0
	dell03	0
	della4	0
	delsab	0
	dels03	0
	hlllab	0
	hlll03	0
	hllla4	0
	hllsab	0
	hlls03	0
	apinch	0
	apdech	0
	incwhl	0
	decwhl	0
	swap4	0
	tableh	0
	uctoi	0

注 1. 括号内的数值适用于有操作异常发生时(当使用了编译器自带的 matherr 函数时)。

（**A**，**B**）

A：当使用由小型模式和中等模式分享的库 时

B：当使用由紧凑模式和大型模式分享的库 时

2. 括号中的数值是为乘法器时使用的。

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

在库执行的某段时间内，中断时被禁止的，此时使用乘法/除法以保证操作的内容不会在中断时被破坏。
下表显示了在使用乘法/除法时，为库禁用中断的最长时间。
在未使用乘法/除法时，不会出现这种为库禁用中断的周期时间。

表 E-1 为库禁止中断的最长时间（clock 数量）

分类	函数名称	最长中断禁止时间	备注
乘法	@@ismul	12	执行两个有符号型整数数据的乘法
	@@iumul	12	执行无符号型整数数据的乘法
	@@lsmul	24	执行两个有符号型长整数数据的乘法
	@@lumul	24	执行两个无符号型长整数数据的乘法
stdlib.h	rand	24	使用@@lumul
	qsort	12	使用@@iumul

符号
?? 29
operator 170
operator 170
#asm - #endasm 353
#define directive 172
#include 51
#include directive 167
#pragma BRK 370
#pragma DI 367
#pragma directive 329
#pragma div 396
#pragma EI 367
#pragma ext_func 411
#pragma ext_table 408
#pragma HALT 370
#pragma inline 417
#pragma interrupt 358
#pragma mul 394
#pragma name 391
#pragma NOP 370
#pragma rot 392
#pragma rtos_interrupt 400
#pragma rtos_task 405
#pragma section 379
#pragma sfr 344
#pragma STOP 370
#pragma vect 358
\a 29
\b 29
\f 29
\n 29
\r 29
\t 29
\v 29

A
abort 240
abs 242
绝对地址访问函数....26, 419
acos 274
acosf 297
聚合类型....41
ANSI 324
算术运算符....95
数组 149
数组声明符....63
数组类型 41
asin 275
asinf 298
__asm 353
ASM 语句 24, 353
汇编语言....14
assert 191
__assertfail 320
赋值运算符....120
atan 276
atan2 277
atan2f 300
atanf 299
atexit 192, 241
atof 192, 245
atoi 232
atol 232
auto 54

B
二进制常数....25, 389
位域....372
位域声明....24, 372
位变量....349
位变量，布尔型变量....24
按位与运算符 112
按位或运算符 114
按位异或运算符....113
块范围....33
__boolean 349
布尔型变量....349
分支语句....127
break 语句....145
BRK 370
brk 192, 244
bsearch 249

C
C 语言....14
calloc 236
callt 函数 23, 334
callt/__callt 334
cast 运算符....93
ceil 292
ceilf 315
改变编译器输出的段名....25, 379
char 类型 37
Character 常量 47
Character 类型 41
逗号运算符....123
注释 52
兼容类型....42
合成类型 43
复合赋值....122
复合语句或块 127
条件控制语句 127
const 61
Constant 44
常量表达式....125
Continue 语句 144
cos 278
cosf 301
cosh 281
coshf 304
CPU 控制指令....24, 370
ctype 185

D

数据插入函数 25, 398
__DATE__ 179
十进制常量 45
分隔符 50
DI 367
__directmap 419
div 192, 243
除法函数. 25, 396
Do 语句 140

E

EI 367
枚举型常量 46
枚举型标识符 59
Enumeration 类型 37
errno 188
error 188
转义序列 29
exit 192, 241
exp 284
expf 307
表达式语句及 Null 语句 127
extern 54
外部定义 154
外部链接 34
外部目标定义 157

F

fabsf 316
__FILE__ 179
文件范围 33
固件 ROM 函数 25, 414
__flash 414
flash 区域分配方法 25, 407
flash 区域分支表 25, 408
float 190
浮点常量 44
浮点类型 38
floor 294
fmod 295
fmodf 318
for 语句 141
free 237
frexp 285
frexpf 308
函数 18
函数声明符 63
函数定义 155
从 boot 区域到 flash 区域的函数调用 25, 411
函数原型范围 33
函数范围 33
函数类型 42

G

一般整型提升 73
getchar 226
gets 227
goto 语句 143

H

HALT 370
Header 名称 51
十六进制常量 45
如何使用 saddr 区域 24, 339
如何使用 sfr 区域 344

I

标识符 34
if ... else 语句 136
if 语句 136
非完整类型 41
整型常量 45
整数类型 37
内部链接 34
__interrupt 365
中断函数 24, 358, 367
中断函数修饰词 24, 365
RTOS 的中断函数 25, 400
RTOS 的中断函数修饰词 25, 403
__interrupt_brk 365
isalnum 200
isalpha 200
isascii 200
iscntrl 200
isdigit 200
isgraph 200
islower 200
isprint 200
ispunct 200
isspace 200
isupper 200
isxdigit 200
itoa 247

K

Kanji (2 字节字符) 24
keyword 33

L

标签语句 127
labs 242
ldexp 286
ldexpf 309
ldiv 192, 243
limits 188
__LINE__ 179
log 287
log10 288
log10f 311
logf 310
逻辑 与 运算符 116
逻辑 或 运算符 117
longjmp 192, 205
Looping Statements 127
ltoa 247

M

机器语言 14
Macro 名称 179

Macro 替换 170
malloc 238
math 189
matherr 296
memchr 261
memcmp 259
memcpy 255
memmove 255
存储器操作函数 26, 417
存储器模式规格 26, 427
存储空间 328
memset 267
参数/返回值的 int 扩展限制方法 25, 415
modf 289
modff 312
模块名称更改函数 25
模块名称改变函数 391
多字节字符 28
乘法函数 25, 394

N

near/far 区域规格 26
No linkage 34
NOP 370
norec 函数 24, 346

O

对象类型 36
八进制常量 45
__OPC 398

P

指针 149
指针声明符 62
后缀操作符函数 79
pow 290
powf 313
预处理指令 158
printf 192, 222
__putc 230
putchar 228
puts 229

Q

qsort 250

R

rand 192, 248
realloc 239
重入 192
register 54, 337
寄存器组 327
指定寄存器组 358
寄存器变量 24, 337
关系运算符 104
return 语句 146
rolb 392
rolw 392
rorb 392
rorw 392
移位函数 25, 392
RTOS 324
__rtos_interrupt 修饰词 403

S

sbrk 192, 244
标量类型 42
scanf 192, 223
ROMization 相关的区段名称 385, 430
setjmp 185, 192, 205
sfr 区域 24
sfr 变量 344
移位运算符 101
有符号整数类型 37
简化赋值 121
sin 279
sinf 302
sinh 282
sinhf 305
sprintf 192, 211
sqrt 291
sqrtf 314
srand 192, 248
sreg 声明 339
sscanf 192, 217
堆栈更改规格 360
启动例程 321, 385
static 54
stdarg 186
__STDC__ 179
stddef 189
stdlib 187
STOP 370
存储类型标识符 54
strbrk 251
strcat 257
strchr 262
strcmp 260
strcoll 270
strcpy 256
strcspn 263
string 187
字符串文字 48
stritoa 253
strlen 269
strltoa 253
strncat 257
strncmp 260
strncpy 256
strpbrk 264
strrchr 262
strsbrk 252
strspn 263
strstr 265
strtod 192, 245
strtok 192, 266
strtol 234
strtoul 234
struct 147
结构 147
结构指针 149

结构标识符 57
结构类型 41
结构变量 147
strultoa 253
strxfrm 271
switch 语句.. 137

T

Tag 60
tan 280
tanf 303
tanh 283
tanhf 306
Task 405
RTOS 的任务函数 25, 405
__TIME__ 179
toascii 202
tolow 203
_tolower 203
tolower 201
toup 203
_toupper 203
toupper 201
三字符序列... 29
类型名称 64
Type specifier 55
typedef 54

U

ultoa 247
单目运算符 86
Union 151
共用体类型 41
无符号整数类型 37

V

va_arg 207
va_end 207
va_start 207
va_starttop 207
void 75
空指针 75
volatile 61
vprintf 192, 224
vsprintf 192, 225

W

while 语句.. 139

Z

-ZB 415
-ZF 407

区域信息

本文档中的某些信息可能因国家不同而有所差异。用户在使用任何一种 **NEC** 产品之前，请与当地的 **NEC** 办事处联系，以获取权威的代理商和发行商信息。请验证以下内容：

- 设备的可用性
- 定货信息
- 产品发布进度表
- 相关技术资料的可用性
- 开发环境要求（例如：要求第三方工具和组件，主计算机，电源插头，**AC** 供电电源等）
- 网络要求

此外，对于商标、注册商标、出口限制条款和其他法律规定，不同的国家也有不同的要求。

详细信息请联系：

（中国区）

网址：
http://www.cn.necel.com/
http://www.necel.com/

[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**

[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**

[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**

[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