

To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
 - “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
 - “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
 - “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

用户手册



CC78K0 Ver3.70

C 编译器

操作篇

目标设备

78K0 系列

文档编号 U17200CA1V0UM00 (第 1 版)

出版日期 2007 年 9 月 CP(K)

© NEC Electronics Corporation 2003

日本印刷

[备忘录]

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

- 本档所刊登的内容有效期截至 2007 年 9 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。

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

[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

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

前言

CC78K0 C 编译器（下文简称为该编译器）开发的基础，是美国信息系统国家标准草案中的**第 2 章 环境**和**第 3 章 语言 — C 语言编程**（1998 年 12 月 7 日）。因此，使用本 C 编译器可以对符合 ANSI 标准的 C 语言源程序进行编译，可以开发 78K/0 系列应用产品。

CC78K0C 编译器语言篇（本手册）是为了让那些使用本 C 编译器进行软件开发的用户能够正确了解本 C 编译器的基本功能及语言规范而编写的。

本手册不介绍如何操作本 C 编译器。因此，在您掌握了本手册的内容后，请阅读 **CC78K0 C 编译器 操作篇 (U17201E)**。

关于 78K/0 系列的体系结构，敬请参阅 78K/0 系列各个产品的用户手册。

[目标设备]

可以使用本 C 编译器开发 78K/0 系列微控制器的软件。

请注意，开发时需要安装与目标设备对应的设备文件。

[读者]

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

[组织结构]

这本手册的结构组织如下描述。

第 1 章 概述

概括介绍该 C 编译器的一般功能、性能指标及特色。

第 2 章 C 语言的结构

介绍 C 源程序模块文件的构成要素。

第 3 章 数据类型与存储类的声明

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

第 4 章 类型转换

介绍本 C 编译器自动执行的数据类型转换。

第 5 章 运算符与表达式

介绍 C 中使用的运算符和表达式，以及运算符的优先级。

第 6 章 C 语言的控制结构

介绍 C 的程序控制结构及在 C 中执行的语句。

第 7 章 结构体与共用体

关于结构体与共用体的概念，以及如何引用结构体与共用体成员。

第 8 章 外部定义

介绍外部定义的类型，以及如何使用外部定义。

第 9 章 预处理指令

详细介绍预处理指令的类型，以及如何使用预处理指令。

第 10 章 库函数

详细介绍 C 库函数的类型，以及如何使用各个库函数。

第 11 章 扩展函数

介绍该 C 编译器的扩展函数，以使用户最大限度地发挥目标设备的功能。

第 12 章 汇编程序与 C 程序的引用和兼容性

介绍将 C 源程序与汇编源程序连接的方法。

第 13 章 编译器的高效使用

概括介绍如何更有效地使用本 C 编译器。

附录

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

[如何阅读这本手册]

- 对于不熟悉 C 编译器或 C 语言的读者：
从第 1 章开始阅读，因为本手册涵盖了从 C 的程序控制结构到本 C 编译器的扩展函数内容。在第 1 章中，使用一个 C 源程序示例来介绍本手册中的引用部分。
- 对于熟悉 C 编译器或 C 语言的读者：
本 C 编译器的语言规范符合 **ANSI 标准 C**。因此，您可以从第 11 章开始，该章介绍了本 C 编译器特有的扩展函数。在阅读第 11 章时，如有必要，还可以参考 78K/0 系列中的目标设备附带的用户手册。

[相关文档]

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

开发工具的相关文档（用户手册）

资料名		资料编号
CC78K0 C 编译器 V 3.70 版本	操作篇	U17201E
	语言篇	本文档
RA78K0 汇编程序包 V3.80 版本	操作篇	U17199E
	汇编语言	U17198E
	结构化汇编语言	U17197E
SM+ 系统模拟器	操作篇	U17246E
	用户开放接口	U17247E
SM78K0 系列 V2.52 版本 系统模拟器	操作篇	U16768E
PM plus V5.20 版本		U16934E
ID78K0-NS 集成调试器 V2.52 版本	操作篇	U16488E
ID78K0-QB 集成调试器 V2.81 版本	操作篇	U16996E
78K/0 系列	指令	U12326E

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

[参考文献]

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

[术语]

RTOS = 78K/0 系列实时操作系统 RX78K0

[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

第 1 章 概述	16
1.1 C语言与汇编语言	16
1.2 使用C编译器的程序开发过程	18
1.3 C源程序的基本结构.....	20
1.3.1 程序格式	20
1.4 该C编译器的最佳性能特性	23
1.5 该C编译器的特色	25
第 2 章 C语言的结构	30
2.1 字符集	31
2.1.1 字符集.....	31
2.1.2 转义字符序列	31
2.1.3 三字符序列.....	32
2.2 关键字	33
2.2.1 ANSI-C关键字.....	33
2.2.2 CC78K0 增加的关键字.....	33
2.3 标识符	34
2.3.1 标识符的作用范围	34
2.3.2 标识符的连接	36
2.3.3 标识符名字空间	36
2.3.4 对象的生存期	37
2.4 数据类型	38
2.4.1 基本类型	40
2.4.2 字符类型	43
2.4.3 不完全类型.....	44
2.4.4 派生类型	44
2.4.5 标量类型	45
2.4.6 兼容类型	45
2.4.7 复合类型	45
2.5 常量.....	46
2.5.1 浮点型常量.....	46
2.5.2 整型常量	46
2.5.3 枚举常量	47
2.5.4 字符常量	48
2.6 字符串文字	49
2.7 运算符	50
2.8 定界符	51
2.9 头文件名	52
2.10 注释.....	53
第 3 章 数据类型与存储类的声明	54
3.1 存储类声明符	55
3.2 类型声明符	56
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.....	62
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	68
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	79
5.2	后缀运算符	80
5.2.1	下标运算符	81
5.2.2	函数调用运算符	82
5.2.3	结构体与共用体成员	83
5.2.4	后缀自增/自减运算符	85
5.3	单目运算符	86
5.3.1	前缀自增/自减运算符	87
5.3.2	地址和间接运算符	88
5.3.3	单目算术运算符 (+ - ~ !)	89
5.3.4	sizeof 运算符	90
5.4	类型转换运算符	91
5.4.1	类型转换运算符 (类型名称)	92
5.5	算术运算符	93
5.5.1	乘法运算符 (* / %)	94
5.5.2	加法运算符	95
5.6	移位运算符	96
5.7	关系运算符	98
5.7.1	关系运算符 (< > <= >=)	99
5.7.2	等式运算符 (== !=)	100
5.8	按位逻辑运算符	101
5.8.1	按位与运算符 (&)	102
5.8.2	按位异或运算符 (^)	103
5.8.3	按位或运算符 ()	104
5.9	逻辑运算符	105
5.9.1	逻辑与运算符 (&&)	106
5.9.2	逻辑或运算符 ()	107
5.10	条件运算符	108
5.10.1	条件运算符 (? :)	109
5.11	赋值运算符	110
5.11.1	简单赋值运算符 (=)	111
5.11.2	复合赋值运算符 (*= /= %= += -= <<= >>= &= ^= =)	112
5.12	逗号运算符	113
5.12.1	逗号运算符 (,)	114
5.13	常量表达式	115
第 6 章	C 语言的控制结构	117
6.1	带标签的语句	119
6.1.1	case 标签	120
6.1.2	default 标签	122
6.2	复合语句或块	123
6.3	表达式语句和空语句	124
6.4	条件控制语句	125
6.4.1	if 和 if ... else 语句	126
6.4.2	switch 语句	127
6.5	循环语句	128
6.5.1	while 语句	129
6.5.2	do 语句	130
6.5.3	for 语句	131
6.6	分支语句	132
6.6.1	goto 语句	133
6.6.2	continue 语句	134
6.6.3	break 语句	135
6.6.4	return 语句	136

第 7 章 结构体和共用体	137
7.1 结构体	137
7.2 共用体	140
第 8 章 外部定义	143
8.1 函数定义	144
8.2 外部对象定义	146
第 9 章 预处理指令（编译器指令）	147
9.1 条件编译	147
9.1.1 #if 指令	149
9.1.2 #elif 指令	150
9.1.3 #ifdef 指令	151
9.1.4 #ifndef 指令	152
9.1.5 #else 指令	153
9.1.6 #endif 指令	154
9.2 源文件包含指令	155
9.2.1 #include < > 指令	156
9.2.2 #include " " 指令	157
9.2.3 #include 预处理记号字符串指令	158
9.3 宏替换	159
9.3.1 #define 指令	161
9.3.2 #define () 指令	162
9.3.3 #undef 指令	163
9.4 行控制	164
9.5 #error 预处理指令	165
9.6 #pragma 指令	166
9.7 空指令	167
9.8 预定义的宏名	168
第 10 章 库函数	170
10.1 函数之间的接口	170
10.1.1 参数	170
10.1.2 返回值	171
10.1.3 保存单独库（Individual Libraries）所用的寄存器	172
10.1.4 对bank区域的支持	177
10.2 头文件	178
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 字符和字符串函数	198
10.4.2 程序控制函数	202
10.4.3 特殊函数	203
10.4.4 I/O函数	205
10.4.5 应用函数	221
10.4.6 字符串/存储器函数	242
10.4.7 数学函数	258
10.4.8 诊断函数	303
10.5 更新启动例程及库函数的批处理文件	304
10.5.1 使用批处理文件	305
第 11 章 扩展函数	307
11.1 宏名称	307
11.2 关键字	308
11.3 存储器	311
11.4 #pragma 指令	313
11.5 如何使用扩展函数	315
11.6 C源代码的修改	480
11.7 函数调用接口	481
11.7.1 返回值	481
11.7.2 普通函数调用接口	482

11.7.3	noauto 函数调用接口（仅在正常模式可用）	489
11.7.4	norec函数调用接口（正常模式）	491
11.7.5	静态模式函数调用接口	493
11.7.6	Pascal 函数调用接口	499
第 12 章	汇编程序的引用	502
12.1	访问参数/自动变量	503
12.1.1	普通模式	503
12.1.2	静态模式	506
12.2	返回值的存储	508
12.3	在C语言程序中调用汇编语言程序	509
12.3.1	修改函数信息文件	509
12.3.2	C 语言函数调用过程	510
12.3.3	汇编语言程序的数据保存和调用返回	512
12.4	由汇编语言程序调用C语言程序	514
12.4.1	由汇编语言程序调用C语言函数	514
12.5	引用其它语言定义的变量	518
12.5.1	引用C语言定义的变量	518
12.5.2	由C语言程序引用汇编语言定义的变量	519
12.6	注意事项	520
第 13 章	编译器的有效应用	521
13.1	高效编码	521
附录A	用于saddr区域的标签列表	525
A.1	正常模式	525
A.2	静态模式	527
附录B	区段名称列表	528
B.1	区段名称列表	528
B.1.1	程序区域和数据区域	528
B.1.2	flash存储器区域	529
B.2	区段位置	530
B.3	C 源程序示例	531
B.4	输出汇编模块示例	532
附录C	运行时库列表	536
附录D	库占用的堆栈列表	544
附录E	库响应中断的最长时间列表	555
附录F	索引	错误！未定义书签。

插图列表

插图编号	插图标题	页码
图 1-1	编译流程	17
图 1-2	使用该C编译器进行程序开发的过程	19
图 2-1	类型的划分	39
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	125
图 6-2	循环语句的控制流程	128
图 6-3	分支语句的控制流程	132
图 10-1	函数被调用时的栈区（未指定-ZR）	173
图 10-2	format命令的语法	208
图 10-3	输入format命令的语法	212
图 11-1	存储空间的利用（正常模式）	311
图 11-2	存储空间的利用（静态模式）	312
图 11-3	通过位域声明的位分配（示例 1）	368
图 11-4	通过位域声明进行位分配（示例 2）	369
图 11-5	位域声明决定的位分配（指定了-RC选项）	370
图 11-6	根据位域声明分配位（示例 3）	371
图 11-7	根据位域声明分配位（示例 3）（指定了-RC选项）	372
图 12-1	调用后的堆栈区	510
图 12-2	调用后的堆栈区	511
图 12-3	返回后的堆栈区	513
图 12-4	参数入栈	515
图 12-5	参数入栈	517
图 12-6	参数在堆栈中的位置	520

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	23
表 2-1	字符集中可以使用的字符列表	31
表 2-2	转义字符序列列表	31
表 2-3	三字符序列列表	32
表 2-4	ANSI-C关键字列表	33
表 2-5	CC78K0追加的关键词列表	33
表 2-6	标识符列表	34
表 2-7	标识符可以使用的数字和字符	34
表 2-8	基本数据类型列表	41
表 2-9	指数关系	42
表 2-10	运算异常列表	43
表 2-11	整型常量和可表示的类型	47
表 2-12	运算符列表	50
表 3-1	数据类型与存储类的声明示例	54
表 4-1	类型转换列表	71
表 4-2	从有符号整型向无符号整型之间的转换	73
表 5-1	运算符计算优先级	78
表 5-2	除号/求余运算结果符号	93
表 5-3	移位运算	96
表 5-4	按位与运算符	102
表 5-5	按位异或运算符	103
表 5-6	按位或运算符	104
表 5-7	逻辑与运算符	106
表 5-8	逻辑或运算符	107
表 10-1	第一参数传递列表（正常模式）	171
表 10-2	参数传递列表（静态模式）	171
表 10-3	返回值存储列表（正常模式）	172
表 10-4	返回值存储列表（静态模式）	172
表 10-5	ctype.h的内容	178
表 10-6	setjmp.h的内容	179
表 10-7	stdarg.h的内容	180
表 10-8	stdio.h的内容	180
表 10-9	stdlib.h的内容	181
表 10-10	string.h的内容	183
表 10-11	math.h的内容	187
表 10-12	assert.h的内容	191
表 10-13	标准库函数列表	193
表 10-14	sprintf的标志	206
表 10-15	sprintf的格式代码	207
表 10-16	sprintf的精度代码	207
表 10-17	sscanf的转换指定符	210
表 10-18	更新库函数的批处理文件	304
表 11-1	添加的关键字列表	308

表 11-2 #pragma指令列表.....	313
表 11-3 指定-QL选项时可以使用的callt属性函数的数量.....	318
表 11-4 callt函数用法的限制.....	318
表 11-5 使用寄存器变量的限制.....	321
表 11-6 使用sreg变量的限制.....	325
表 11-7 通过-RD选项分配到saddr区域的变量.....	327
表 11-8 通过-RS选项分配到saddr区域的变量.....	328
表 11-9 通过-RK选项分配到saddr区域的变量.....	329
表 11-10 仅使用常数 0 或 1 的操作符（通过Bit型变量）.....	341
表 11-11 当使用中断函数时保存/恢复区域.....	347
表 11-12 当使用中断函数时保存/恢复区域（静态模式）.....	348
表 11-13. 类型调整的详细信息（从int和short型改为char型）.....	423
表 11-14. 类型调整的详细内容（从long型改为int型）.....	424
表 11-15 中断函数存储目标.....	462
表 11-16 存储返回值的位置.....	481
表 11-17. 第一个参数传入的位置（函数调用方）.....	483
表 11-18 静态模式下传输参数的区.....	494
表 12-1. 参数传递（函数调用方）.....	503
表 12-2. 参数/自动变量的存储（被调用函数内）.....	504
表 12-3. 参数传递（函数调用方）.....	506
表 12-4. 参数/自动变量存储（被调用函数内）.....	506
表 12-5 返回值存储位置.....	508
表A-1 寄存器变量（正常模式）.....	525
表A-2 norec 函数的参数（正常模式）.....	526
表A-3 norec 函数的自动变量.....	526
表A-4 运行时刻的参数.....	526
表A-5 共享区域（静态模式）.....	527
表A-6 用于参数，自动变量及工作区.....	527
表B-1 区段名称列表（程序区域和数据区域）.....	528
表B-2 区段名称列表（flash存储器区域）.....	529
表B-3 区段位置.....	530
表C-1 运行时刻库列表.....	536
表 D-1 标准库占用堆栈列表.....	544
表 D-2 运行时刻库堆栈占用空间列表.....	549
表 E-1 为库禁止中断的最长时间（clock数量）.....	555

第 1 章 概述

本章介绍 CC78K0 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。

CC78K0 系列 C 编译器是一款语言处理程序，不论编写的 C 语言源程序符合 78K/0 系列规范或是符合 ANSI-C 规范，CC78K0 系列 C 编译器都会将 C 语言转换为机器语言。使用 CC78K0 系列 C 编译器，还将获得适用于 78K/0 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。

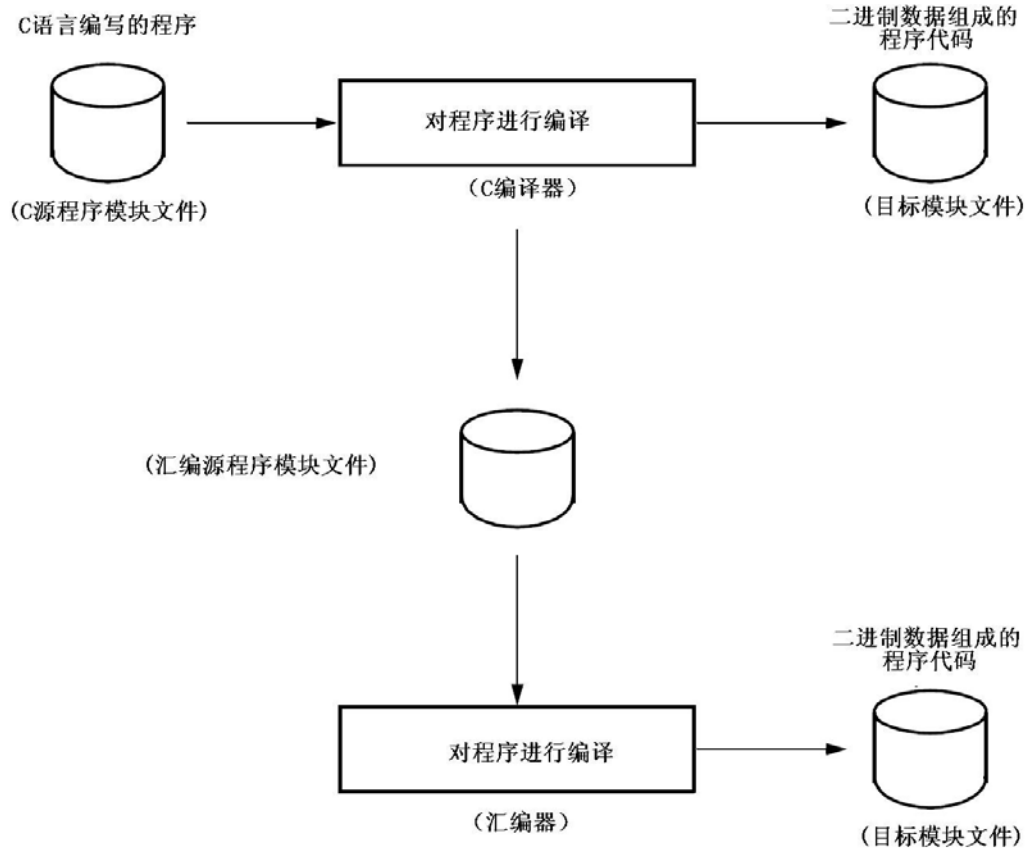
所以，人们开发了高级语言来代替这种汇编语言。C 语言是高级语言的其中一种，它使得程序员在编程时无需考虑计算机的体系结构。与汇编语言程序相比，可以认为 C 语言编写开发的程序逻辑结构更加易于理解。

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。C 编译器的转换流程如图 1-1 所示。

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）。

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
- RA78K0S 汇编程序包

结构化汇编预处理:	为了通过汇编语言得到结构化程序
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

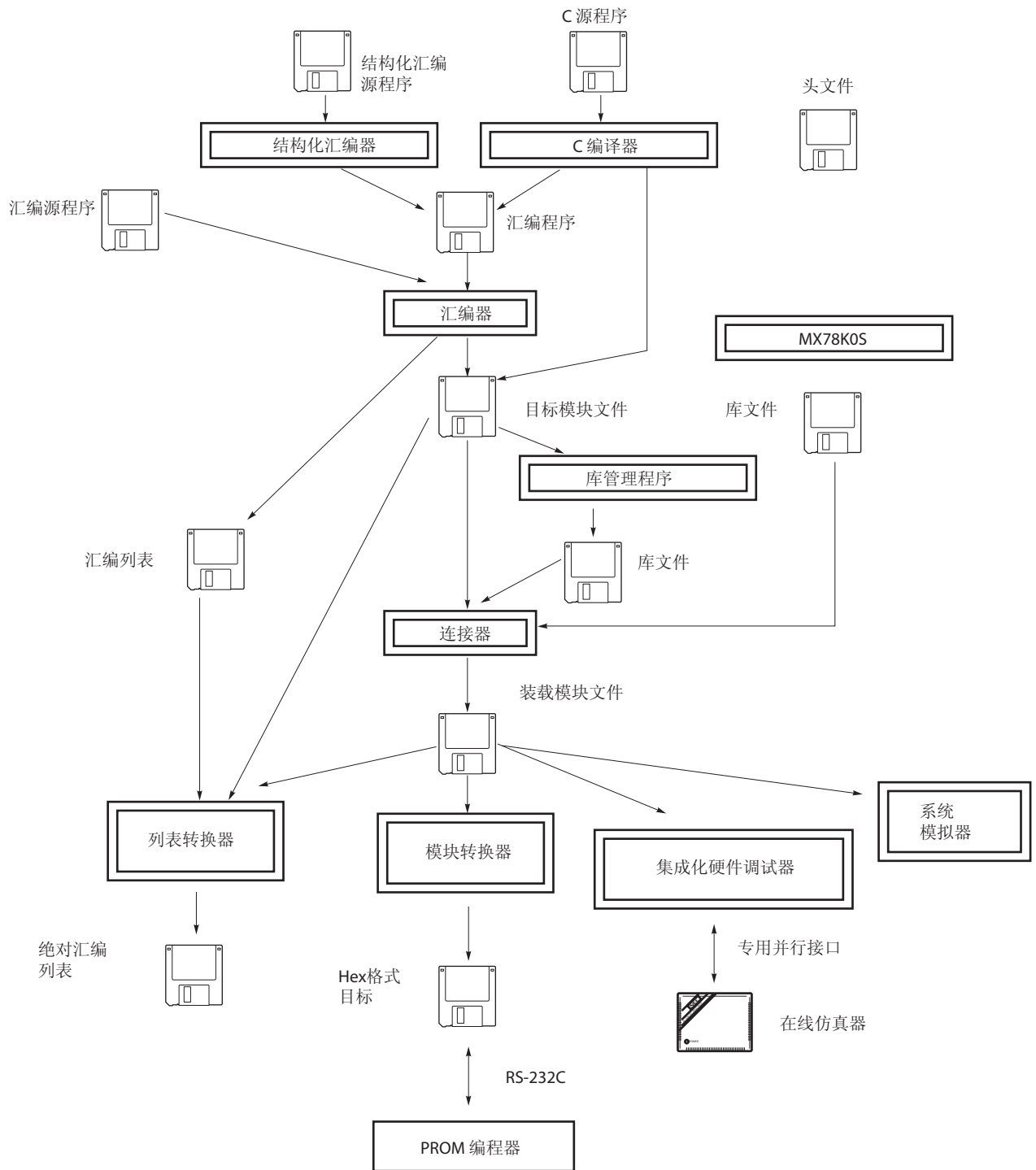
- 调试器（用于 78K/0） 用于调试 C 源程序模块文件

使用 C 编译器进行程序开发的步骤如下。

- <1>将程序划分成若干个功能块。
- <2>创建每个功能块对应的 C 源程序模块。
- <3>对每个 C 源程序模块进行转换。
- <4>将常用的模块注册到库中。
- <5>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 使用该 C 编译器进行程序开发的过程

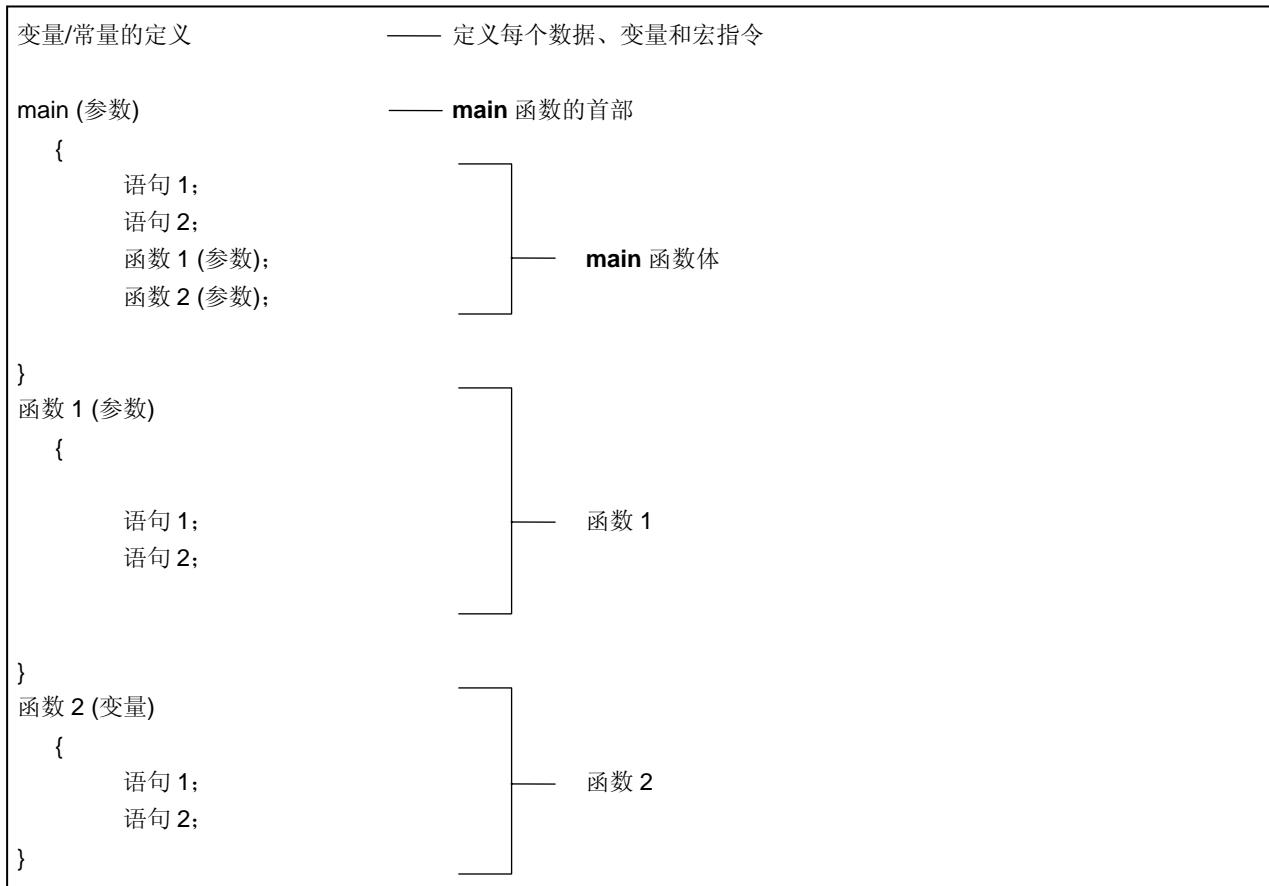


1.3 C 源程序的基本结构

1.3.1 程序格式

C 程序是函数的集合。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 xxx xxx	预处理程序指令（定义宏） <6>
#define FALSE 0			
#define SIZE 200			
void printf(char *,int);	}	xxx xxxx (xxx, xxx)	函数原型声明符 <7>
void putchar(char);			
char mark[SIZE+1];	}	char xxx	类型声明符 <1> 外部定义 <5>
main()			xx [xx]
{			
int i,prime,k,count;	—	int xxx	类型声明符 <1>
count=0;	—	xx = xx	操作符 <2>
for(i=0;i<=SIZE;i++)	}	for (xx;xx;xx) xxx ;	控制结构 <3>
mark[i]=TRUE;			
for(i=0;i<=SIZE;i++){			
if(mark[i]){			
prime=i+i+3;	—	xxx = xxx + xxx + xxx	操作符 <2>
printf("%6d",prime);	—	xxx (xxx) ;	操作符 <2>
count++;			
if((count%8)==0) putchar('\n');	—	if (xxx) xxx ;	控制结构 <3>
for(k=i+prime;k<=SIZE;k+=prime)			
mark[k]=FALSE;			
}			
}			
printf("\n%d primes found.",count);	—	xxx (xxx) ;	操作符 <2>
}			
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

这是语句用来规定程序流程。C 语言有多个指令用于控制结构，如条件控制、迭代和分支跳转。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用相同的内存时，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 ^{注 2}
每个函数定义或函数调用中的参数数量	39 个参数
每个宏定义或宏调用中的参数数量	31 个参数
每个源代码逻辑行的字符数量	2048 字符
连接后一个字符串内的字符数量	509 字符
一个数据目标的大小	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	65535 字节
每个结构体或共用体的成员数量	256 个成员
每个枚举类型中 enum 常量的数量	255 个常量
一个结构体或共用体中包含结构体或共用体的嵌套层数	15 层
初始化时元素的嵌套层数	15 层
每个源程序模块文件中定义的函数数量	1,000

表 1-1 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
-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. 该值包括 C 编译器保留的宏定义。

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 78K/0 系列的特殊功能寄存器（SFR）能在 C 语言中进行描述，从而缩短了目标代码，并改善了程序执行速度。

下面概括介绍这些用于缩短目标代码并改善执行速度的扩展函数。

- 函数可以使用 **callt** 表区域进行调用. **callt / __callt** 函数
- 变量可以分配到寄存器中 寄存器变量
- 变量可以分配到 **saddr** 区域中 **sreg / __sreg**
- 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 **noauto** 函数,
- **norec / __leaf** 函数
- 在 C 源程序中可以进行汇编语言程序的描述 **ASM** 语句
- 可以按位访问 **saddr** 或 **sfr** 区域. **bit** 型变量,
- **boolean / __boolean** 型变量
- 函数体可以存储在 **callf** 区域. **callf / __callf** 函数
- 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
- 可以访问内存空间中的特定地址 绝对地址函数
- 特定的数据和指令可以直接嵌入到代码区域中 插入数据函数
- 使用的栈在被调用函数方进行校正 **__pascal** 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](#)。

(1) callt 函数（callt / __callt）

函数可以使用 **callt** 表区域进行调用。每个待调用的函数（该函数称为 **callt** 函数）地址存储在 **callt** 表中，供以后调用。这使得目标代码比使用通常的调用指令 **call** 的目标代码要短。

(2) 寄存器变量（register）

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

(3) 如何使用 **saddr** 区域（sreg / __sreg）

使用关键字 **sreg** 声明的变量可以分配到 **saddr** 区域。**sreg** 变量的相关指令比分配到内存的那些变量的指令在代码长度上要更短。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对应类型的变量分配到 **saddr** 区域。

(4) **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。

(5) **noauto** 函数 (**noauto**)

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

(6) **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。

(7) 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。

(8) 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。

(9) 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。

(10) 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。

(11) 中断函数 (**#pragma DI, #pragma EI**)

将中断禁止指令和中断使能指令嵌入到目标代码中。

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

(13) **callf** 函数 (**callf / __callf**)

callf 指令将函数体存储在 **callf** 入口区域，允许使用比 **call** 指令更短的代码来调用该函数。这样可以改善执行速度并减少目标代码量。

(14) 绝对地址访问函数 (**#pragma access**)

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

(15) 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。

(16) 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。

(17) 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。

(18) 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。

(19) 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。

(20) 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。

(21) 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。

(22) BCD 操作函数 (**#pragma bcd**)

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

(23) Bank 函数

Bank 函数在 **bank** 区域中存储函数体，通过 **bank** 函数调用库来进行函数的调用。于是，可以支持带有 **bank** 函数功能的设备。

(24) 常量地址上的 Bank 函数

在常量地址上调用 **bank** 函数，只在支持 **bank** 函数的设备中可用。

(25) 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。

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

可以描述 **RX78K0** 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)。

- (27) 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 RX78K0 在另外的文件中设置向量。
- (28) 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 RX78K0 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。
- (29) 静态模式
在编译过程中指定 `-SM` 选项, 可以缩短目标代码, 改进执行速度, 实现高速中断处理, 并节省内存空间。
- (30) 类型更改 (`-ZI`)
通过指定 `-ZI` 选项和 `-ZL` 选项, `int/short` 型将被视为 `char` 型, `long` 型将被视为 `int` 型。
- (31) Pascal 函数 (`__pascal`)
用于在函数调用时放置参数的堆栈调整工作在被调用函数方执行, 而不是在函数调用方执行。当此类函数调用大量出现时, 能够缩短目标代码。
- (32) 函数调用接口的自动 pascal 函数化 (`-ZR`)
在编译过程中指定 `-ZR` 选项, 除了 `norec`、`__interrupt` 及参数长度可变的函数之外, 其他函数全都会增加 `__pascal` 属性。
- (33) 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。
- (34) 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
- (35) 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。
- (36) 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。
- (37) 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。
- (38) 数组偏移量计算简化方法 (`-QW2 / -QW3`)
编译过程中指定 `-QW2`、`-QW3`、`-QW4` 及 `-QW5` 选项, 可以简化偏移量计算代码, 缩短目标代码, 并提高执行速度。

(39) 寄存器直接引用函数 (**#pragma realregister**)

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

(40) [HL+B]基址变址寻址使用方法 (**-QE**)

编译时指定 **-QE** 选项，可以缩短目标代码，并改善程序执行速度。

(41) 片上固件自编程序子程序直接调用函数 (**#pragma hromcall**)

通过 C 规范说明可以很方便的进行寄存器调用，在源程序中以函数调用的格式指定这个函数，或者在模块中使用 **#pragma hromcall** 指令声明片上固件自编程序子程序直接调用函数。

(42) **__flashf** 函数 (**__flashf**)

函数声明时添加 **__flashf** 属性，在 **__flashf** 函数中描述了片上固件自编程序子程序直接调用函数，不产生 bank 保存/恢复和切换寄存器组 3 的代码。

(43) 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。

(44) 绝对地址分配规范 (**__directmap**)

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

(45) 静态模式展开规范 (**-ZM**)

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

(46) 临时变量 (**__temp**)

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

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

(47) 支持序言/尾声的库 (**-ZD**)

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

第 2 章 C 语言的结构

本章介绍 C 源程序模块文件的构成要素。

一个 C 源程序模块文件由以下标记（字符序列中可以辨别出来的单元）构成。

关键字	标识符	常量
字符串文字	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;		
extern bit data2;	data1, data2	/*标识符*/
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。

以下字符可以同时源字符集和执行字符集中同时使用：

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 十进制数	
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。（请参阅下文中的转义字符序列。）	

备注 在字符常量中，字符串文字、注释语句中，可能会使用上述字符以外的字符。

2.1.2 转义字符序列

转义字符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

2.1.3 三字符序列

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

表 2-3 三字符序列列表

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

2.2 关键字

2.2.1 ANSI-C 关键字

以下标记在本 C 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 CC78K0 增加的关键字

在本 C 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** 语言规范。

callf, __callf, __banked 1 ~ 15、__rtos_interrupt 及 __interrupt_brk 是用作关键字的标记，以便与 CC78K0 兼容。

表 2-5 CC78K0 追加的关键词列表

__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}
__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
__directmap	绝对地址分配规范
__temp	临时变量
__mxcall	__mxcall 函数 ^{注2}

注 1. 这是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. 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。

表 2-7 标识符可以使用的数字和字符

_ (下划线)	a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z	
A	B	C	D	E	F	G	H	I	J	K	L	M	
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
0	1	2	3	4	5	6	7	8	9				

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

2.3.1 标识符的作用范围

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，多个标识符可以被视为同一个标识符。可以使用以下三种不同的方法将标识符连接起来：外部连接、内部连接和无连接。

(1) 外部连接

外部连接是指作为翻译（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：

- 已经声明但未指定存储类型的函数标识符
- 已经声明为 **extern** 但未指定存储类型的对象或函数的标识符
- 具有文件作用域但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 标识符名字空间

所有的标识符可以归入以下的“名字空间”。

- 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

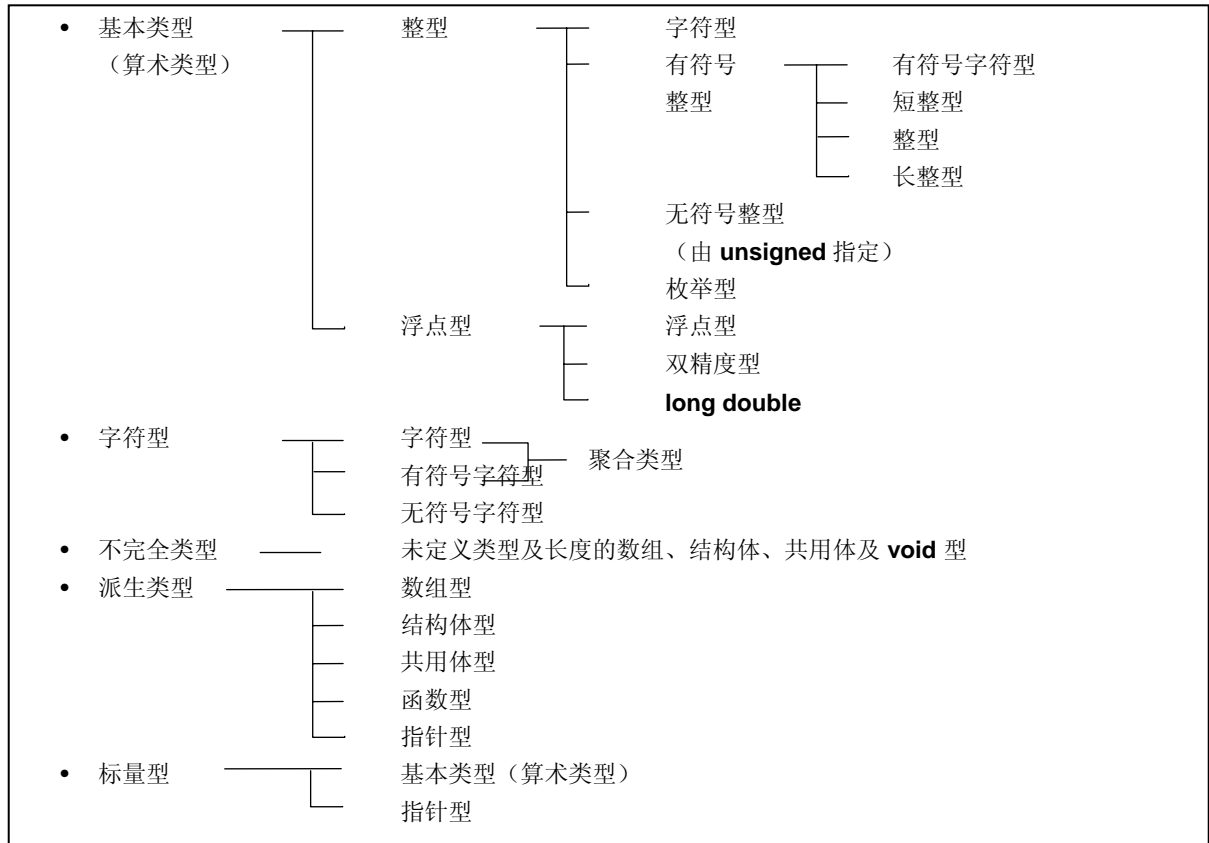
2.4 数据类型

数据类型决定存储在各个对象中的值的含义，分为以下三类。

- 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。

表 2-8 基本数据类型列表

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

- **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。
- **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。
- **unsigned short int** 数据和 **unsigned int** 数据将作为值范围相同、但具有不同类型的数据来处理。
- **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。
- **float**、**double** 和 **long double** 类型的值范围都是绝对的。

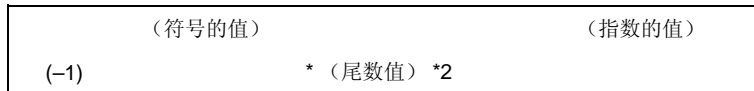
(a) 浮点数 (float 型) 规范

• 格式

浮点数的格式如下所示。



这种格式的数值如下。

**s: 符号 (1 位)**

0 表示正数, 1 表示负数。

e: 指数 (8 位)

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 2-9 所示。

表 2-9 指数关系

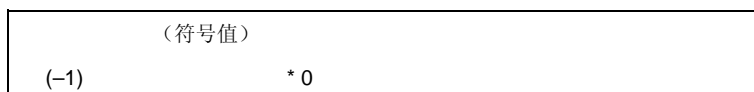
指数 (十六进制)	指数的值
FE	127
⋮	⋮
81	2
80	1
7F	0
7E	-1
⋮	⋮
01	-126

m: 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 (即值 1) 始终是 1, 并在此格式下忽略表示。

• 0 的表示

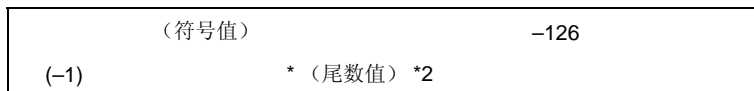
当指数为 0, 尾数为 0 时, ± 0 的表示如下。



- 无穷大的表示
当指数为 FFH，尾数为 0 时， $\pm\infty$ 表示如下。



- 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。

- 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。
- 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）。
- 运算异常
有 5 种运算异常，如下表所示。

表 2-10 运算异常列表

异常	返回值
下溢	非标准化数
不准确	± 0
上溢	$\pm\infty$
除 0	$\pm\infty$
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) 聚合类型

聚合类型又细分为两类。

数组类型和结构体类型。聚合类型数据是一组成员对象的集合，成员对象将被顺序提取。

(a) 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。

(b) 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。

(2) 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。

(3) 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。

(4) 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)
 I/L (long double)
 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有效数字。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。

共有以下三种整型常量。

- 十进制常量: 以一个非 0 数字开始的十进制数
十进制数字 = 123456789
- 八进制常量: 整型前缀 0 + 八进制数
八进制数 = 01234567
- 十六进制常量: 整型前缀 0x 或 0X + 十六进制数字
十六进制数字 = 0123456789
abcdef ABCDEF

无符号后缀 u U 长整型后缀 l L

(1) 十进制常量

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

(2) 八进制常量

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

(3) 十六进制常量

十六进制常量是一个以 16 为底数（基数）的整数，该数必须以 0x 或 0X 开始，其后跟随的数字可以是 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）。

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 2-11 整型常量和可表示的类型

（整型常量）	（可表示的类型）
无后缀十进制数	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
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 2-12 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。

<p>示例: /* 注释语句 */ //注释语句</p>

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。

表 3-1 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。

- **typedef**
- **extern**
- **static**
- **auto**
- **register**

(1) **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](#)。

(2) **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）。

(3) **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。

(4) **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。

(5) **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本 C 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。

• 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 78K/0 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。
 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）。

下面是一个位域示例。

```
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 ;
    /*...*/ (*cp!=burgundy) /*...*/
    .
    .
    .
```

3.2.3 标记

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

以下声明中的标识符是一个标记名。

```

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

```

标记包含结构体/共用体或枚举的内容，其中已经定义了成员变量。在后续的声明中，结构体、共用体或枚举型的结构变得与标记的列表结构相同。在具有相同作用域的后续声明中，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。

- `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

- 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。

- 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

转换后 转换前		（有符号）char		（有符号）short int		（有符号）int		（有符号）long int		float	double	long double
		（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			
（有符号）char	+		○	○	○	○	○	○	○	○	○	○
	-		N	○	N	○	N	○	N	○	○	○
无符号 char		△		○	○	○	○	○	○	○	○	○
（有符号）short int	+				○		○	○	○	○	○	○
	-				N		N	○	N	○	○	○
无符号 short int				△		△		○	○	○	○	○
（有符号）int	+				○		○	○	○	○	○	○
	-				N		N	○	N	○	○	○
无符号 int				△		△		○	○	○	○	○
（有符号）long int	+								○	○	○	○
	-								N	○	○	○
无符号 long int								△		○	○	○
float											○	○
double												
long double												

备注

1. **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。
 2. 约定
 - : 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。

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，这样才能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 类型转换列表。

下面的表 4-2 列出了从有符号整型向无符号整型之间的转换运算。

表 4-2 从有符号整型向无符号整型之间的转换

		无符号	
		值范围更小	值范围更大
有符号	+	/	○
	-	/	+

○: 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）。

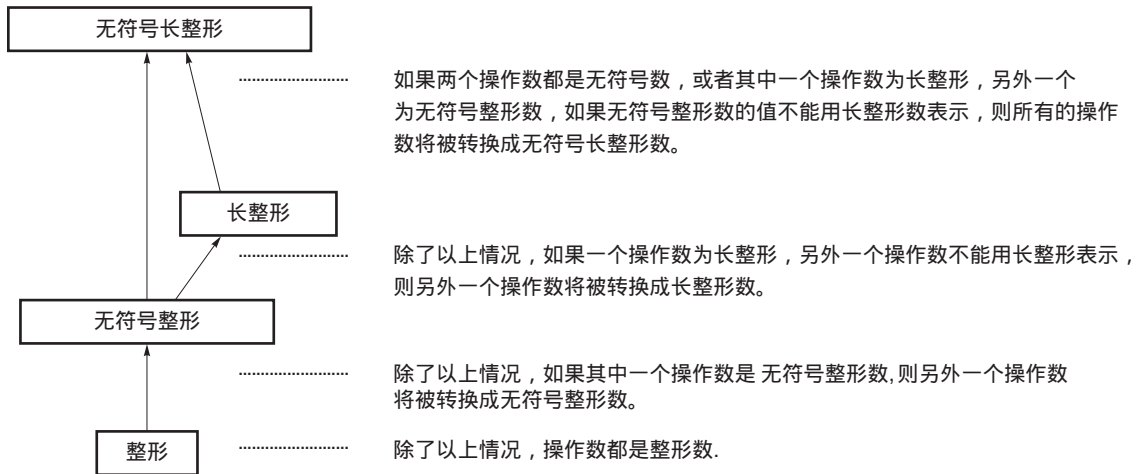
(3) 常见算术类型转换

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，请参阅 **CC78K0 C 编译器操作篇 (U17200E) 第 5 章 编译选项**）。

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 语言支持大量用于算术、逻辑和其他运算的运算符。其丰富的运算符集合还包括那些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

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

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
- 后缀自增/自减运算符

5.2.1 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。

5.2.2 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。

5.2.3 结构体与共用体成员

<1>.（点）运算符

语法

后缀表达式 . 标识符

功能

.（点）运算符（也称为成员运算符）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。

<2>->（箭头）运算符

语法

后缀表达式 -> 标识符

功能

->（箭头）运算符（也称为间接成员运算符）指定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 intrnode;
    }ni;
    struct {
        int type;
        struct{
            long longnode;
        }*nl_p;
    }nl;
}u;

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

5.2.4 后缀自增/自减运算符

<1> 后缀自增运算符

语法

后缀表达式 ++

功能

后缀自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。

<2> 后缀自减运算符

语法

后缀表达式 --

功能

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

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。

- 前缀自增和自减运算符 `++ --`
- 地址和间接运算符 `& *`
- 单目算术运算符 `+ - ~ !`
- `sizeof` 运算符 `sizeof`

单目运算符将在随后内容中详细介绍。

5.3.1 前缀自增/自减运算符

单目运算符

++ --

<1> 前缀自增运算符

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
```

语法

```
++前缀表达式
```

<2> 前缀自减运算符

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
```

语法

```
--单目表达式
```

5.3.2 地址和间接运算符

单目运算符

& *

<1> 单目 &运算符

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）。

语法

& 运算数

<2> 单目 * 运算符

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）。

语法

* 运算数

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有一个指针类型。

5.3.3 单目算术运算符 (+ - ~ !)

单目运算符

+ - ~ !

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。

语法

+ 运算数

- 运算数

~ 运算数

! 运算数

5.3.4 sizeof 运算符

单目运算符

sizeof 运算符

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。

语法

```
sizeof 单目表达式  
sizeof (类型名)
```

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。

(类型名) 表达式

5.4.1 类型转换运算符（类型名称）

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。

语法

```
(类型名) 表达式
```

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

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。

- | | | | |
|---------|---|---|---|
| • 乘法运算符 | * | / | % |
| • 加法运算符 | + | - | |

表 5-2 除号/求余运算结果符号

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

a % 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。

5.5.1 乘法运算符 (* / %)

<1> * 运算符

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。

语法

$E1 * E2$

<2> / 运算符

功能

/ 运算符对两个运算数执行通常的除法，并返回商。

语法

$E1 / E2$

<3> % 运算符

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。

语法

$E1 \% E2$

5.5.2 加法运算符

<1> + 运算符

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。

语法

$E1 + E2$

<2> - 运算符

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）。

语法

$E1 - E2$

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 <<>>

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。

5.6.1 移位运算符 (<< >>)

<1> 左移运算符 (<<)

功能

<< 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。

语法

E1 << E2

<2> 右移运算符 (>>)

功能

>> 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，或 E1 是带符号型的且是一个非负值，则结果的值就是 E1 除以 2 的 E2 次幂。

语法

E1 >> E2

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。

- | | |
|---------|-----------|
| • 关系运算符 | < > <= >= |
| • 等式运算符 | = = != |

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。

在本编译器中，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。

5.7.1 关系运算符 (< > <= >=)

<1> < (小于) 运算符

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。

语法

$E1 < E2$

<2> > 大于运算符

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。

语法

$E1 > E2$

<3> <= (小于或等于) 运算符

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。

语法

$E1 <= E2$

<4> >= (大于或等于) 运算符

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。

语法

$E1 >= E2$

5.7.2 等式运算符 (== !=)

<1> == (等于) 运算符

功能

如果两个运算数相等，== 运算符返回 1；否则，返回 0。

语法

$E1 == E2$

<2> != (不等于) 运算符

功能

如果两个运算数不相等，!= 运算符返回 1；否则，返回 0。

语法

$E1 != E2$

5.8 按位逻辑运算符

按位逻辑运算符以位为单位，对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。

5.8.1 按位与运算符 (&)

功能

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

按位与运算符必须使用“&”运算符来说明。

表 5-4 按位与运算符

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

语法

E1 & E2

5.8.2 按位异或运算符 (^)

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。

表 5-5 按位异或运算符

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

语法

E1 ^ E2

5.8.3 按位或运算符 (|)

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。

表 5-6 按位或运算符

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

语法

E1 E2

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。

5.9.1 逻辑与运算符 (&&)

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。

表 5-7 逻辑与运算符

		左操作数的值	
		0	非 0
右操作数的值	0	0	0
	非 0	0	1

语法

E1 && E2

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。

5.9.2 逻辑或运算符 (||)

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。

表 5-8 逻辑或运算符

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

语法

E1 E2

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? :

5.10.1 条件运算符 (?:)

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。

语法

```
第一运算数 ? 第二运算数 : 第三运算数
```

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。

5.11.1 简单赋值运算符 (=)

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

语法

```
E1 = E2
```

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

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 表示一个适当的二元运算符）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;

语法

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

5.12 逗号运算符

逗号运算符的类型如下所示。

逗号运算符，

5.12.1 逗号运算符 (,)

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);  
}
```

语法

```
E1 , E2
```

5.13 常量表达式

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

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

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

(1) 一般整型常量表达式

一般整型常量表达式的类型一般都是整型。可以使用以下的运算数。

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

(2) 算术常量表达式

算术常量表达式为整型。可以使用以下的运算数。

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

(3) 地址常量表达式

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

- 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 包括表达式和分号的语句
- 选择语句 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。

[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。

<p>case 标签 default 标签</p>

6.1.1 case 标签

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。

语法

```
case 常量表达式 : 语句
```

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。

6.1.2 default 标签

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。

语法

```
default: 语句
```

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式和分号。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

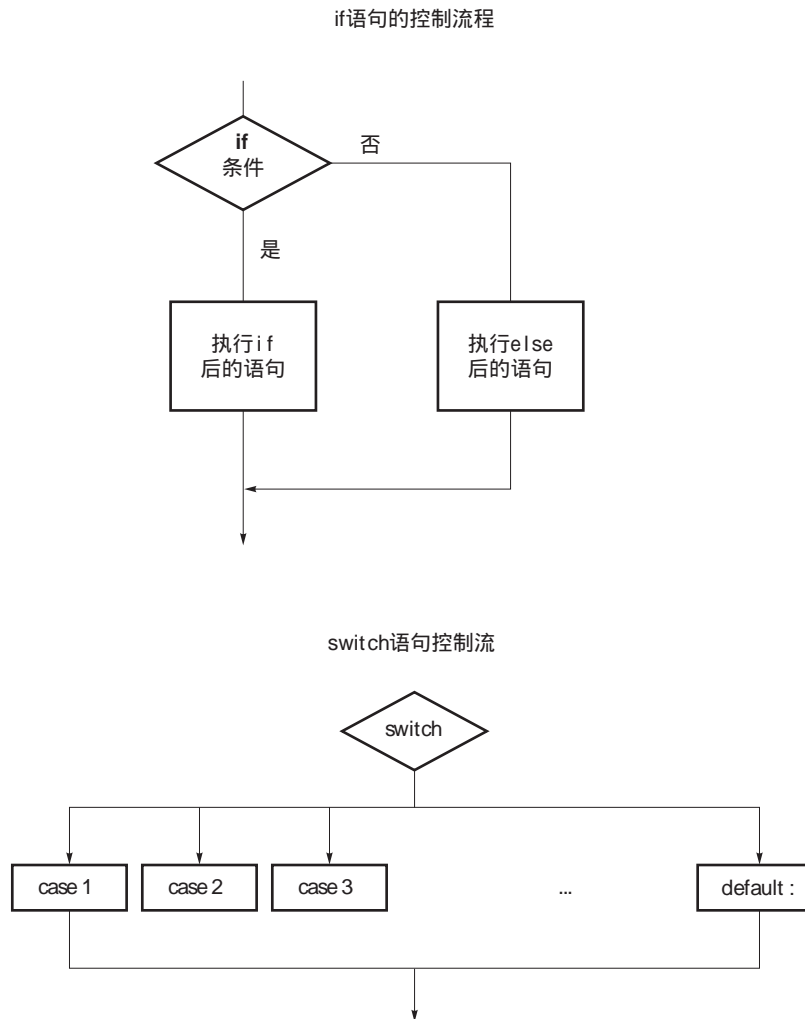
条件控制（选择）语句包括 **if** 和 **switch** 语句。根据括号中的控制表达式的值，**if** 或 **switch** 语句让程序能够在几组备选语句中选择其中一组执行。

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



6.4.1 if 和 if ... else 语句

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。

语法

```
if (表达式) 语句  
if (表达式) 语句-1else 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。

6.4.2 switch 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。如果不存在控制表达式值的对应 **case** 标签，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。

语法

```
switch (表达式) 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。

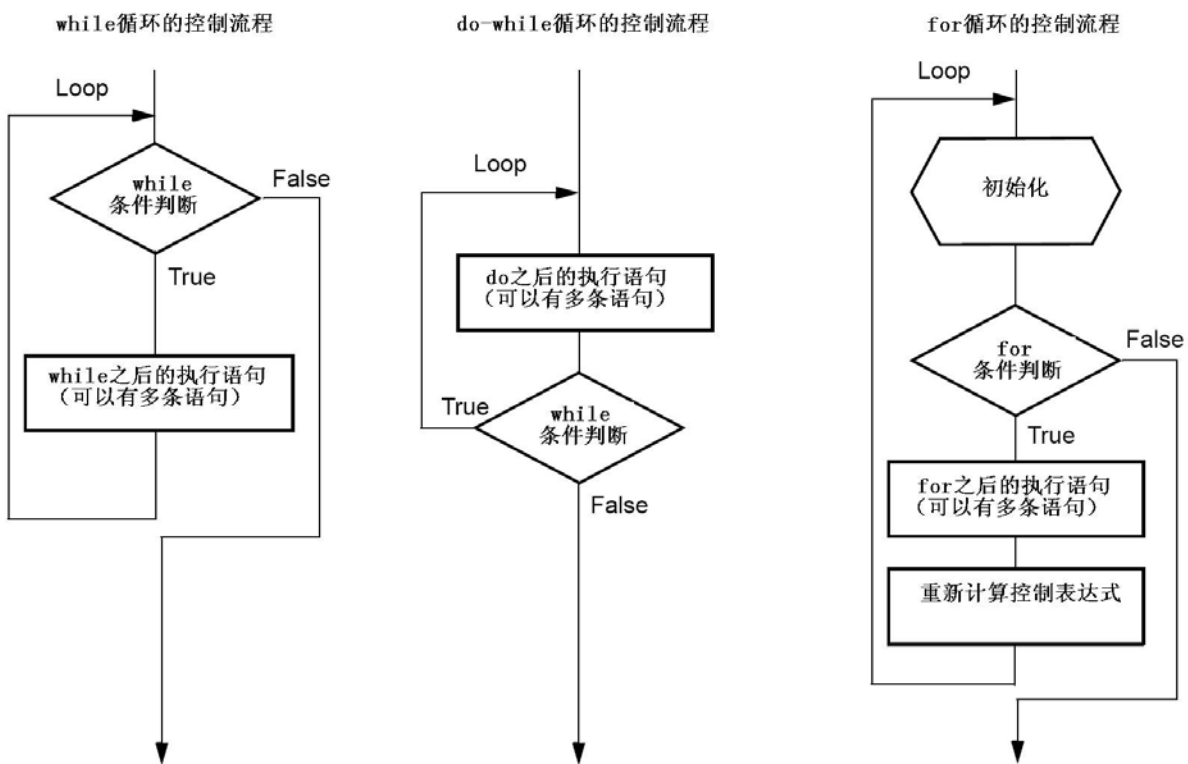
6.5 循环语句

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 语言规范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



6.5.1 while 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。

语法

```
while (表达式) 语句
```

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。

6.5.2 do 语句

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。

语法

```
do 语句 while (表达式);
```

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。

6.5.3 for 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。

语法

```
for (第一表达式; 第二表达式; 第三表达式) 语句
```

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。

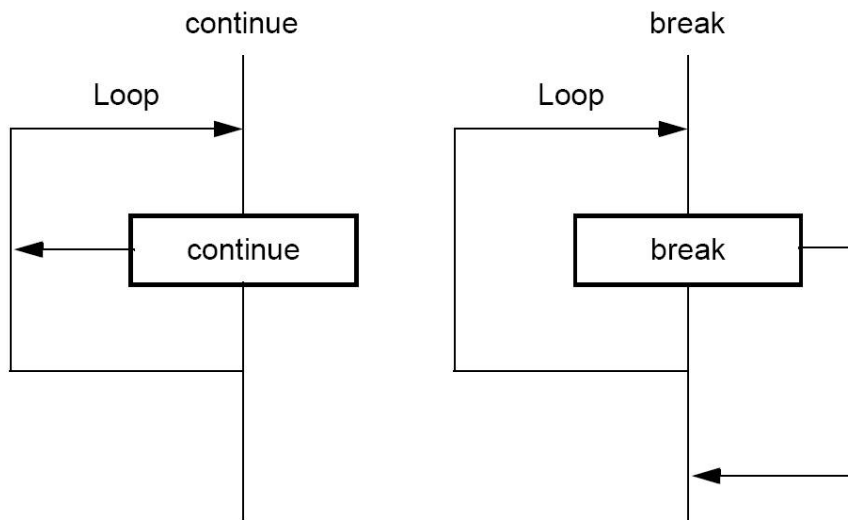
6.6 分支语句

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



6.6.1 goto 语句

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。

语法

```
goto 标识符;
```

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。

6.6.2 continue 语句

功能

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

语法

```
continue ;
```

例

```
while(/*...*/){  
    /*...*/  
    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。

6.6.3 break 语句

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。

语法

```
break ;
```

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。

6.6.4 return 语句

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。

语法

```
return 表达式 ;
```

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放入任何标记名。

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

[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** 的次序分配存储区。

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;
    unsigned int c:1;
}bit_field;
```

} 位域

(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。

[外部对象定义示例]

<code>int i1=1;</code>	用外部连接定义
<code>static int i2=2;</code>	用内部连接定义
<code>extern int i3=3;</code>	用外部连接定义
<code>int i4;</code>	用外部连接进行临时定义
<code>static int i5;</code>	用内部连接进行临时定义
<code>int i1;</code>	引用先前声明的有效临时定义
<code>int i2;</code>	违反连接规则
<code>int i3;</code>	引用先前声明的有效临时定义
<code>int i4;</code>	引用先前声明的有效临时定义
<code>int i5;</code>	违反连接规则
<code>extern int i1;</code>	引用先前的具有外部连接的声明
<code>extern int i2;</code>	引用先前的具有内部连接的声明
<code>extern int i3;</code>	引用先前的具有外部连接的声明
<code>extern int i4;</code>	引用先前的具有外部连接的声明
<code>extern int i5;</code>	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  
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
```

9.1.1 #if 指令

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。

语法

```
#if 常量表达式 换行 [组]
```

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。

9.1.2 #elif 指令

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）。

语法

```
#elif 常量表达式, 换行 [组]
```

例

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

说明

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

9.1.3 #ifdef 指令

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。

语法

```
#ifdef 标识符 换行 [组]
```

例

```
#define ON
#ifdef ON
.
.
.
#endif
```

说明

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

9.1.4 #ifndef 指令

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。

语法

```
#ifndef 标识符, 换行 [组]
```

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。

9.1.5 #else 指令

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。

语法

```
#else 换行 [组]
```

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。

9.1.6 #endif 指令

功能

#endif 指令指示**#ifdef** 块的结尾。

语法

```
#endif 换行
```

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** <文件名>
- **#include** “文件名”
- **#include** 预处理记号字符串

#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 该 C 编译器的最佳性能特性”。

备注 预处理记号字符串：由**#define** 指令定义的字符串

9.2.1 #include < > 指令

功能

若指令格式如 **#include < >**，则 C 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、**INC78K** 环境变量指定的目录和注册表中注册的目录 `\NECTools32\INC78K0S` 目录，用指定文件的全部内容来替换 **#include** 指令行语句。

语法

```
#include <文件名> 换行
```

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在注册表中注册的目录 `\NECTools32\INC78K0S` 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。

9.2.2 #include “ ”指令

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注册库中注册的目录**\NECTools32\INC78K0S** 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。

语法

```
#include "文件名" 换行
```

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、注册库中注册的目录**\NECTools32\INC78K0S** 中搜索双引号中指定的文件 **myprog.h**，用指定文件 **myprog.h** 的全部内容来替换指令行 **#include “myprog.h”**。

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。

9.2.3 #include 预处理记号字符串指令

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。

语法

```
#include 预处理记号字符串 换行
```

例

```
#define INCFIL "myprog.h"  
#define INCFIL
```

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、注册库中注册的目录 `\NECTools32\INC78K0S` 中搜索指定文件。若用“文件名”替换记号字符串，则首先搜索当前工作目录。如果未能找到指定文件，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注册库中注册的目录 `\NECTools32\INC78K0S` 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：

- 对象格式
 #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"1"=%d,x"2"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。

9.3.1 #define 指令

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。

语法

```
#define 标识符 替换列表 换行
```

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”。

9.3.2 #define ()指令

功能

函数形式的**#define** 指令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。

语法

```
#define 标识符 ( [标识符表] ) 替换列表 换行
```

例

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

说明

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

9.3.3 #undef 指令

功能

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

语法

```
#undef 标识符 换行
```

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。

(1) 更改行号

要想更改行号，应进行如下的指定。0 和大于 32767 的数都无法指定。

```
#line 数字字符串 换行
```

[例]

```
#line 10
```

(2) 更改行号和文件名

要想更改行号和文件名，应进行如下指定。

```
#line 数字字符串 "字符串" 换行
```

[例]

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

(3) 用预处理程序记号字符串进行更改

除上述指定外，还可以进行如下的指定。在这种情况下，指定的预处理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】

在本例中使用了指示本编译器设备系列的宏名称__K0S__。若设备为 78K/0 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 78K0S（不适用 78K0S）”。

```
#if __K0S__
.
.
.
#else
#error "not for 78K0S"
.
.
.
#endif
```

9.6 #pragma 指令

#pragma 指令是使用编译器定义方法来指示编译器进行操作的指令。在本编译器中有若干个用来使产生的代码适应 78K/0 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

在本 C 编译器中已经定义了下列宏名。

<code>__LINE__</code>	当前源程序行的行号（十进制常量）
<code>__FILE__</code>	源文件名（字符串字面量（字符文字））
<code>__DATE__</code>	源文件编译日期（字符串文字，形如“Mmm dd yyyy”）
<code>__TIME__</code>	源文件编译时间（字符串文字，形如“hh:mm:ss”）
<code>__STDC__</code>	十进制常量“1”，表示与 ANSI [*] 规范兼容

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

```
'__KOS__'
```

- 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‘_’
英语字符为大写

```
(例) __054__    __054Y__
```

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。

C 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

```
#define __STATIC_MODEL__ 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)

```
cc78ks -c054Y 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)'

(Example)

```
#pragma PC(054Y)
```

```
·  
·  
·
```

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，C 语言中必定会含有进行 I/O 操作的库函数。

该 C 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。本章介绍该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，如果没有在正常模式中指定老版本函数接口支持选项（**-ZO**）时，第一参数也会尽可能的通过寄存器传递。此外，在静态模式中，所有的参数都会被寄存器传递。

关于**-ZO**选项，请参见 **CC78K0 C 编译器操作用户手册（U17200E）“第五章 编译选项”**。

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。另外，当指定了函数调用接口自动 **pascal** 函数选项**-ZR** 时，如果参数是通过堆栈进行传递的，那么从堆栈中移除参数的操作是由被调用方完成的。

如果参数通过堆栈传递，那么参数在堆栈中的存放是自底至顶逐个降序进行的。

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

对于静态模式来说，所有参数都通过寄存器传递。

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

参数传递的列表如下所示。在正常模式中，第二个及其后的参数通过堆栈进行传递。

标准库的函数接口（参数传递及返回值存储）与普通函数相同。

表 10-1 第一参数传递列表（正常模式）

第一参数的类型	传递方法
1 字节、2 字节整型	AX
3 字节整型	AX, BC
4 字节整型	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。

表 10-2 参数传递列表（静态模式）

参数的类型	第一参数	第二参数	第三参数
1 字节整型	A	B	H
2 字节整型	AX	BC	HL

备注 如果参数共有 4 字节，那么部分参数会分配给 AX 和 BC，其余分配给 HL 或 H。
1 至 4 字节整型不包括结构体和共用体。

10.1.2 返回值

函数的返回值从它的最低有效位（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 中。当返回指针时，结构体的首地址储存在寄存器 BC 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。

(1) 正常模式

表 10-3 返回值存储列表（正常模式）

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

(2) 静态模式

表 10-4 返回值存储列表（静态模式）

返回值的类型	存储方法
1 位	CY
1 字节整型	A
2 字节整型	AX
4 字节整型	AX（低）、BC（高）
指针	AX

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

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

使用 **saddr 区域** 的库，把使用的 **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。

(1) 未指定-ZR 选项

传递参数和返回值的步骤如下所示。

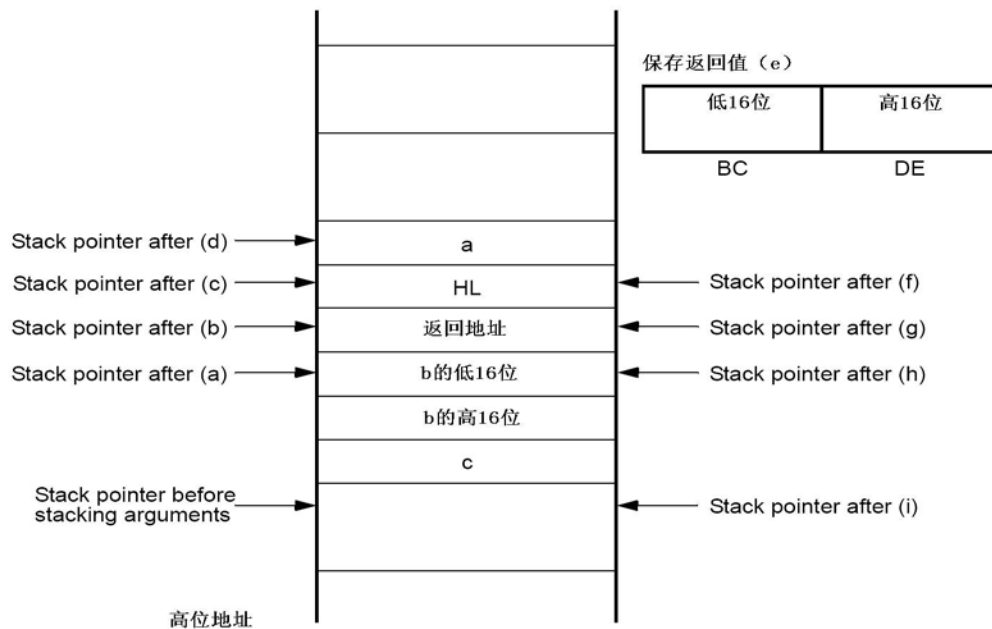
```
被调用函数 "long func(int a, long b, char *c);"
```

<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。

 （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。在图 10-1 所示的例子当中, 添加了 6。

图 10-1 函数被调用时的栈区 (未指定-ZR)

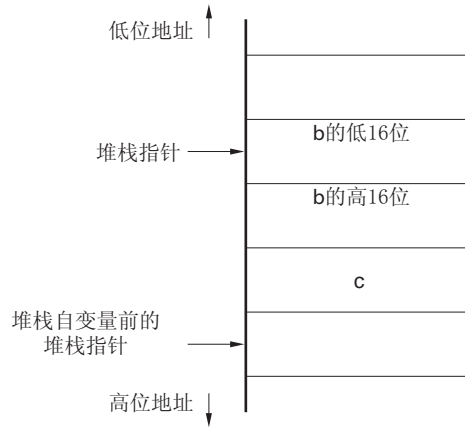


(2) 已指定-ZR 选项

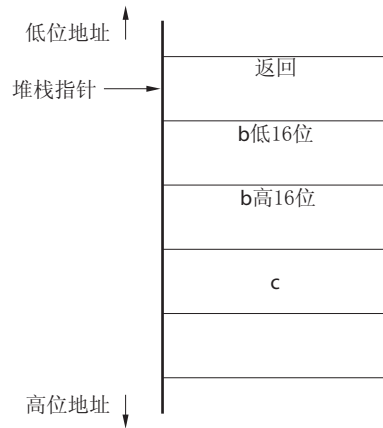
下面的例子展示了在指定 **ZR** 选项时传递参数和返回值的步骤。

```
被调用函数 "long func(int a, long b, char *c);"
```

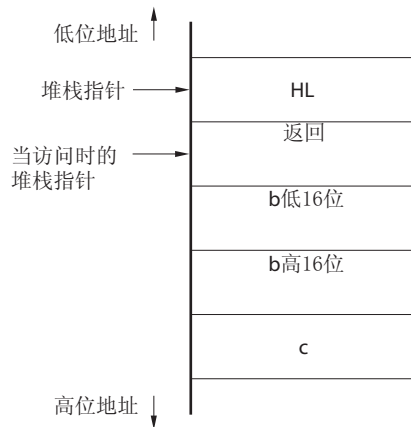
- <1> (由调用方) 把参数放入栈中
参数“c”和“b”的高 16 位及参数“b”的低 16 位以该顺序放入栈中。参数“a”由 AX 寄存器传递。



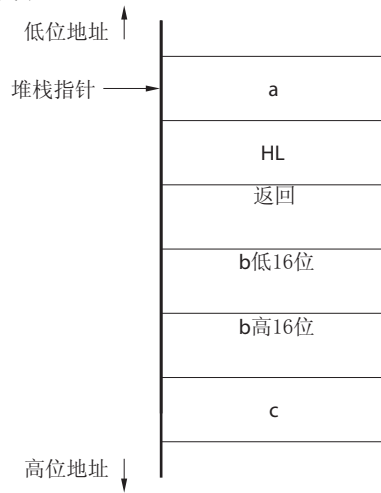
<2> (由调用方) 通过 **call** 指令调用 **func**
 当堆栈处于如下所示的状态时，控制权转移到函数 **func**。



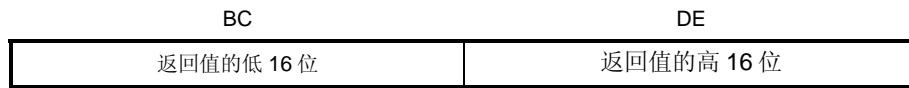
<3> (由被调用方) 保存使用的寄存器



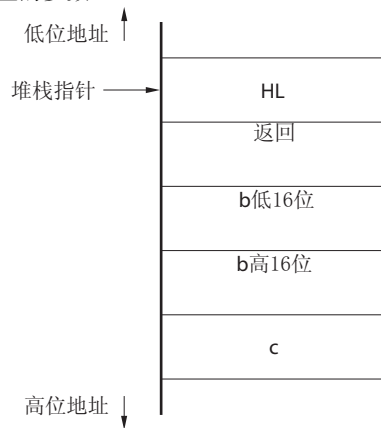
<4> 由寄存器调用的第一参数放入栈中。



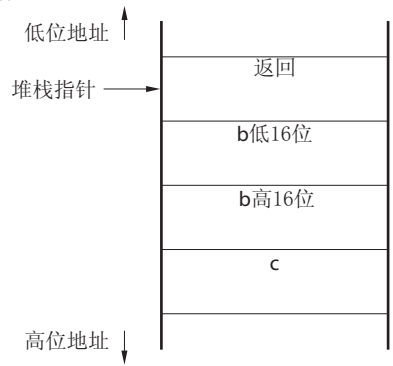
<5> (由被调用方) 进行对函数 **func** 的处理, 把返回值存储在寄存器中。
返回值的低 16 位存储在 BC 中, 高 16 位存储在 DE 中。



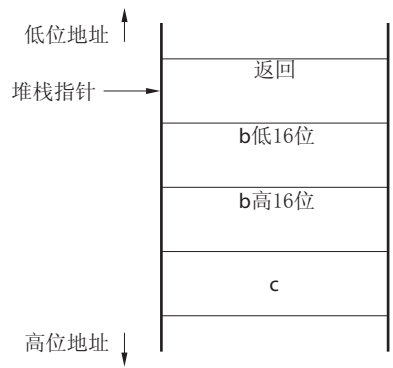
<6> (由被调用方) 恢复第一个放置的参数



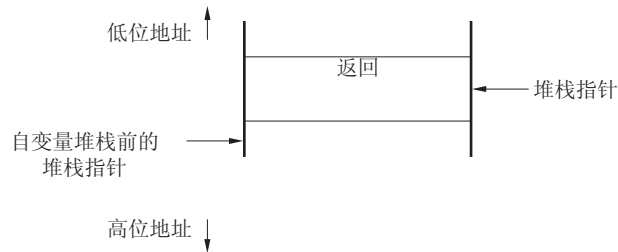
<7> (由被调用方) 恢复保存的寄存器



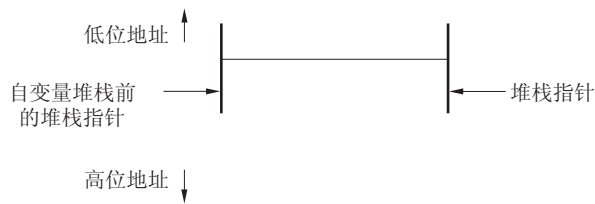
<8> 把返回地址存储在寄存器中, 并将堆栈指针移动到放置参数之前的位置, 则可以从堆栈中移除参数。



<9> (由被调用方) 恢复存储在寄存器中的返回地址。



<10> (由被调用方) 通过 **ret** 指令将控制权返回给调用方的函数



10.1.4 对 bank 区域的支持

当使用 **bank** 函数 (-MF) 时，函数指针的大小是 4 个字节。于是在处理函数指针和地址时有如下限制：

- (1) 需要处理为指针的函数

sprintf, sscanf, printf, scanf, vprintf, vsprintf

当函数指针被用作参数时，操作结果无法保证。

- (2) 需要处理为地址的函数

setjmp

在选定 **bank** 中的信息无法保存。

对于分配在 **bank** 区域的函数不要使用 **setjmp**

由于在使用 **longjmp** 时 **bank** 信息无法恢复，操作结果无法保证。

- (3) 把函数指针用作参数的函数

bsearch, qsort, atexit

无法使用四个字节的功能指针。

在使用 **bank** 函数 (-MF) 的同时，不要使用这些函数。

10.2 头文件

本 C 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。本 C 编译器的头文件如下所示。

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

注意事项 支持的函数随存储模式（正常模式和静态模式）的不同而有所差异。而且，在普通模式中函数的操作会随-ZI和-ZL选项而有所不同。对于因-ZI和-ZL选项而无法正常工作的函数，会输出报警信息“未进行原型声明”。

(1) ctype.h

此头文件用来定义字符函数和字符串函数。在此标准头文件中定义了下列库函数。

但是，在指定了编译选项-ZA（此选项禁止使用不符合 ANSI 规范的函数，允许使用符合 ANSI 规范的一部分函数）时，`_toupper` 和 `_tolower` 将不会被定义，定义了 `tolower` 和 `toupper` 作为替代。当未指定-ZA 时，`tolower` 和 `toupper` 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。

表 10-5 ctype.h 的内容

函数名	对-ZI或-ZL的指定	正常模式				静态模式			
		无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
isalnum		OK	OK	OK	OK	OK	—	OK	—
isalpha		OK	OK	OK	OK	OK	—	OK	—
iscntrl		OK	OK	OK	OK	OK	—	OK	—
isdigit		OK	OK	OK	OK	OK	—	OK	—
isgraph		OK	OK	OK	OK	OK	—	OK	—
islower		OK	OK	OK	OK	OK	—	OK	—
isprint		OK	OK	OK	OK	OK	—	OK	—
ispunct		OK	OK	OK	OK	OK	—	OK	—
isspace		OK	OK	OK	OK	OK	—	OK	—
isupper		OK	OK	OK	OK	OK	—	OK	—
isxdigit		OK	OK	OK	OK	OK	—	OK	—

表 10-5 ctype.h 的内容

函数名 \ 对-ZI 或-ZL 的指定	正常模式				静态模式			
	无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
tolower	OK	OK	OK	OK	OK	—	OK	—
toupper	OK	OK	OK	OK	OK	—	OK	—
isascii	OK	OK	OK	OK	OK	—	OK	—
toascii	OK	OK	OK	OK	OK	—	OK	—
_toupper	OK	OK	OK	OK	OK	—	OK	—
_tolower	OK	OK	OK	OK	OK	—	OK	—
tolow	OK	OK	OK	OK	OK	—	OK	—
toup	OK	OK	OK	OK	OK	—	OK	—

OK: 支持

—: 不支持

(2) setjmp.h

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

表 10-6 setjmp.h 的内容

函数名 \ 对-ZI 或-ZL 的指定	正常模式				静态模式			
	无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
setjmp	OK	OK	OK	OK	OK	—	OK	—
longjmp	OK	OK	OK	OK	OK	—	OK	—

OK: 支持

—: 不支持

在头文件 **setjmp.h** 中定义了下列对象。

[对 **int** 数组类型 **jmp_buf** 的声明]

- 正常模式

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

- 静态模式

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

(3) stdarg.h (仅正常模式)

此头文件用来定义特殊函数。在此头文件中定义了下列三个函数。

表 10-7 stdarg.h 的内容

函数名 \ 对-ZI或-ZL的指定	正常模式			
	无	ZI	ZL	ZI ZL
va_arg	OK	OK	OK	OK
va_start	Δ	Δ	Δ	Δ
va_end	OK	OK	OK	OK

OK: 支持

Δ: 操作可以保证, 但有某些限制

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。

表 10-8 stdio.h 的内容

函数名 \ 对-ZI或-ZL的指定	正常模式				静态模式			
	无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
sprintf	OK	NG	OK	NG	—	—	—	—
sscanf	OK	NG	OK	NG	—	—	—	—
printf	OK	NG	OK	NG	—	—	—	—
scanf	OK	NG	OK	NG	—	—	—	—
vprintf	OK	NG	OK	NG	—	—	—	—
vsprintf	OK	NG	OK	NG	—	—	—	—
getchar	OK	OK	OK	OK	OK	—	OK	—
gets	OK	OK	OK	OK	OK	OK	OK	OK
putchar	OK	OK	OK	OK	OK	—	OK	—
puts	OK	OK	OK	OK	OK	—	OK	—

OK: 支持

NG: 操作无法保证

—: 不支持

声明了下列宏名称。

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

(5) `stdlib.h`

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

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

表 10-9 `stdlib.h` 的内容

函数名	对-ZI 或-ZL 的指定	正常模式				静态模式			
		无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
<code>atoi</code>		OK	NG	OK	NG	OK	—	OK	—
<code>atol</code>		OK	OK	NG	NG	—	—	—	—
<code>strtol</code>		OK	OK	NG	NG	—	—	—	—

表 10-9 stdlib.h 的内容

函数名	对-ZI或-ZL 的指定	正常模式				静态模式			
		无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
strtoul		OK	OK	NG	NG	—	—	—	—
calloc		OK	OK	OK	OK	OK	—	OK	—
free		OK	OK	OK	OK	OK	—	OK	—
malloc		OK	OK	OK	OK	OK	—	OK	—
realloc		OK	OK	OK	OK	OK	—	OK	—
abort		OK	OK	OK	OK	OK	OK	OK	OK
atexit		OK	OK	OK	OK	OK	—	OK	—
exit		OK	OK	OK	OK	OK	—	OK	—
abs		OK	OK	OK	OK	OK	—	OK	—
div		OK	—	OK	—	—	—	—	—
labs		OK	OK	NG	NG	—	—	—	—
ldiv		OK	OK	—	—	—	—	—	—
brk		OK	OK	OK	OK	OK	—	OK	—
sbrk		OK	OK	OK	OK	OK	—	OK	—
atof		OK	OK	OK	OK	—	—	—	—
strtod		OK	OK	OK	OK	—	—	—	—
itoa		OK	OK	OK	OK	OK	—	OK	—
ltoa		OK	OK	—	—	—	—	—	—
ultoa		OK	OK	—	—	—	—	—	—
rand		OK	NG	OK	NG	—	—	—	—
srand		OK	OK	OK	OK	—	—	—	—
bsearch		OK	OK	OK	OK	—	—	—	—
qsort		OK	OK	OK	OK	—	—	—	—
strbrk		OK	OK	OK	OK	OK	—	OK	—
strsbrk		OK	OK	OK	OK	OK	—	OK	—
strtoa		OK	OK	OK	OK	OK	—	OK	—
strltoa		OK	OK	—	—	—	—	—	—
strultoa		OK	OK	—	—	—	—	—	—

OK: 支持

NG: 操作无法保证

—: 不支持

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（静态模式除外）]

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

[对结构体类型 `ldiv_t` 进行声明，其成员 `quot` 和 `rem` 为 `long int` 类型（在静态模式和正常模式中指定了 `-ZL` 时除外）]

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

[对宏名称 `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。

表 10-10 `string.h` 的内容

函数名 \ 对-ZI 或-ZL 的指定	正常模式				静态模式			
	无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
<code>memcpy</code>	OK	OK	OK	OK	OK	—	OK	—
<code>memmove</code>	OK	OK	OK	OK	OK	—	OK	—
<code>strcpy</code>	OK	OK	OK	OK	OK	OK	OK	OK
<code>strncpy</code>	OK	OK	OK	OK	OK	—	OK	—
<code>strcat</code>	OK	OK	OK	OK	OK	OK	OK	OK
<code>strncat</code>	OK	OK	OK	OK	OK	—	OK	—
<code>memcmp</code>	OK	NG	OK	NG	OK	—	OK	—
<code>strcmp</code>	OK	NG	OK	NG	OK	—	OK	—

函数名 \ 对-ZI 或-ZL 的指定	正常模式				静态模式			
	无	ZI	ZL	ZI ZL	无	ZI	ZL	ZI ZL
strncmp	OK	NG	OK	NG	OK	—	OK	—
memchr	OK	OK	OK	OK	OK	—	OK	—
strchr	OK	OK	OK	OK	OK	—	OK	—
strcspn	OK	NG	OK	NG	OK	—	OK	—
strpbrk	OK	OK	OK	OK	OK	OK	OK	OK
strrchr	OK	OK	OK	OK	OK	—	OK	—
strspn	OK	NG	OK	NG	OK	—	OK	—
strstr	OK	OK	OK	OK	OK	OK	OK	OK
strtok	OK	OK	OK	OK	OK	OK	OK	OK
memset	OK	OK	OK	OK	OK	—	OK	—
strerror	OK	OK	OK	OK	OK	—	OK	—
strlen	OK	NG	OK	NG	OK	—	OK	—
strcoll	OK	NG	OK	NG	OK	—	OK	—
strxfrm	OK	NG	OK	NG	OK	—	OK	—

OK: 支持

NG: 操作无法保证

—: 不支持

(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)
```

当编译选项中指定了**-ZI**选项（**int** 和 **short** 类型被当作 **char** 类型，**unsigned int** 和 **unsigned short** 类型被当作 **unsigned char** 类型）时，通过编译器声明的宏 **__FROM_INT_TO_CHAR__**，可以对 **INT_MAX**、**INT_MIN**、**SHRT_MAX**、**SHRT_MIN**、**SINT_MAX**、**SINT_MIN**、**SSHRT_MAX**、**SSHRT_MIN**、**UINT_MAX** 和 **USHRT_MAX** 进行如下声明。

```
#define INT_MAXCHAR_MAX
#define INT_MINCHAR_MIN
#define SHRT_MAX CHAR_MAX
#define SHRT_MIN CHAR_MIN
#define SINT_MAX SCHAR_MAX
#define SINT_MIN SCHAR_MIN
#define SSHRT_MAX SCHAR_MAX
#define SSHRT_MIN SCHAR_MIN
#define UINT_MAX UCHAR_MAX
#define USHRT_MAX UCHAR_MIN
```

当编译选项中指定了 **-ZL** 选项（**long** 类型被当作 **int** 类型，**unsigned long** 类型被当作 **unsigned int** 类型）时，通过编译器声明的宏 **__FROM_LONG_TO_INT__**，可以对 **LONG_MAX**、**LONG_MIN** 和 **ULONG_MAX** 进行如下的声明。

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

(10) **stddef.h**

在此头文件中声明和定义了下列对象。

[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))
```

• **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 (仅正常模式)

math.h 定义了下列函数。

表 10-11 math.h 的内容

函数名	对-ZI 或-ZL 的指定		正常模式	
	无	ZI	ZL	ZI ZL
acos	OK	OK	OK	OK
asin	OK	OK	OK	OK
atan	OK	OK	OK	OK
atan2	OK	OK	OK	OK
cos	OK	OK	OK	OK
sin	OK	OK	OK	OK
tan	OK	OK	OK	OK
cosh	OK	OK	OK	OK
sinh	OK	OK	OK	OK
tanh	OK	OK	OK	OK
exp	OK	OK	OK	OK
frexp	OK	OK	OK	OK
ldexp	OK	OK	OK	OK
log	OK	OK	OK	OK
log10	OK	OK	OK	OK
modf	OK	OK	OK	OK
pow	OK	OK	OK	OK
sqrt	OK	OK	OK	OK
ceil	OK	OK	OK	OK
fabs	OK	OK	OK	OK
floor	OK	OK	OK	OK
fmod	OK	OK	OK	OK
matherr	OK	—	OK	—
acosf	OK	OK	OK	OK
asinf	OK	OK	OK	OK
atanf	OK	OK	OK	OK
atan2f	OK	OK	OK	OK
cosf	OK	OK	OK	OK
sinf	OK	OK	OK	OK
tanf	OK	OK	OK	OK
coshf	OK	OK	OK	OK
sinhf	OK	OK	OK	OK
tanhf	OK	OK	OK	OK
expf	OK	OK	OK	OK
frexpf	OK	OK	OK	OK
ldexpf	OK	OK	OK	OK
logf	OK	OK	OK	OK
log10f	OK	OK	OK	OK
modff	OK	OK	OK	OK

OK: 支持

—: 不支持

表 10-10 math.h 的内容

函数名 \ 对-ZI 或-ZL 的指定	正常模式			
	无	ZI	ZL	ZI ZL
Powf	OK	OK	OK	OK
Sqrtf	OK	OK	OK	OK
Ceulf	OK	OK	OK	OK
Fabsf	OK	OK	OK	OK
Floorf	OK	OK	OK	OK
Fmodf	OK	OK	OK	OK

OK: 支持

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` (仅正常模式)表 10-12 `assert.h` 的内容

函数名	对-ZI 或-ZL 的指定	正常模式			
		无	ZI	ZL	ZI ZL
<code>__assertfail</code>		OK	OK	OK	OK

OK: 支持

`assert.h` 定义了下列对象。

```

#ifndef 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，为未定义 `__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。

本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。

(1) 不可重入的函数

setjmp、longjmp、atexit、exit

(2) 下列函数所使用的区域在启动例程中被特意保留

div、ldiv、brk、sbrk、rand、srand、strtok

(3) 处理浮点数的函数

sprintf、sscanf、printf、scanf、vprintf、vsprintf^{*}、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本 C 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 10-13 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)	istoupper
	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 (normal model only)
	va_starttop (normal model only)
	va_start_banked (normal model only)
	va_starttop_banked (normal model only)
	va_arg (normal model only)
	va_end (normal model only)
I/O Functions (I/O 函数)	sprintf (normal model only)
	sscanf (normal model only)
	printf (normal model only)
	scanf (normal model only)
	vprintf (normal model only)
	vsprintf (normal model only)
	getchar
	gets
	putchar
	puts

表 10-13 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 (normal model only)	
	ldiv (normal model only)	
	brk, sbrk	
	atof, strtod	
	Utility Functions (应用函数)	itoa, ltoa (normal model only)
		ultoa (normal model only)
rand, srand		
bsearch (normal model only)		
qsort (normal model only)		
strbrk		
strsbrk		
strtoa, strttoa (normal model only)		
struttoa (normal model only)		

表 10-13 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, strchr
	strspn, strcspn
	strpbrk
	strstr
	strtok
	memset
	strerror
	strlen
	strcoll
strxfrm	
Mathematical Functions (数学函数)	acos (normal model only)
	asin (normal model only)
	atan (normal model only)
	atan2 (normal model only)
	cos (normal model only)
	sin (normal model only)
	tan (normal model only)
	cosh (normal model only)
sinh (normal model only)	

表 10-13 标准库函数列表

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

表 10-13 标准库函数列表

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

10.4.1 字符和字符串函数

(1) is-

功能

is- 判断字符的类型。

头文件

`ctype.h`, 用于所有字符函数

函数原型

```
int is-(int c);
```

函数	参数	返回值
is-	c.. 进行判断的字符	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

(2) toupper, tolower

功能

字符函数 **toupper** 和 **tolower** 的作用都是将一种类型的字符转换成另一种类型。

若 **c** 为小写字母，则 **toupper** 函数会返回对应 **c** 的大写字母。

若 **c** 为大写字母，则 **tolower** 函数会返回对应 **c** 的小写字母。

头文件

ctype.h

函数原型

int to-(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。 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。

(3) 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。

(4) `_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 _to-(int c);`

函数	参数	返回值
<code>_toupper</code> <code>toup</code>	c.. 要转换的字符	从“c”中减去“a”再加上“A”得到的值
<code>_tolower</code> <code>tolow</code>		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。

10.4.2 程序控制函数

(1) **setjmp, longjmp**

功能

程序控制函数 **setjmp** 在被调用时会保存环境信息（程序的当前状态）。

程序控制函数 **longjmp** 恢复由 **setjmp** 保存的环境信息。

头文件

setjmp. h

函数原型

```
int setjmp(jmp_buf env);
```

```
void longjmp(jmp_buf env,int val);
```

函数	参数	返回值
setjmp	env ... 保存环境信息的数组	<ul style="list-style-type: none"> 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）。

10.4.3 特殊函数

- (1) **va_start** (仅正常模式), **va_starttop** (仅正常模式)
va_start_banked (仅正常模式), **va_starttop_banked** (仅正常模式)
va_arg (仅正常模式), **va_end** (仅正常模式)

功能

va_start 函数 (宏) 用来启动变量参数列表。

va_starttop 函数 (宏) 用来设置可变参数的数量处理。

va_start_banked 函数 (宏) 用来设置可变参数的数量处理。

va_starttop_banked 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);
void va_starttop (va_list ap, parmN);
void va_start_banked (va_list ap, parmN);
void va_starttop_banked (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 va_start_banked va_starttop_banked	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, va_start_banked

- 在 **va_start** 和 **va_start_banked** 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** 或 **va_start_banked**。
- **va_start_banked** 可以用于那些通过 **bank** 函数调用例程来发起调用的函数。

va_start_banked

- **va_start** 不能用于那些通过 **bank** 函数调用例程来发起调用的函数，这种函数请使用 **va_start_banked**。

va_starttop, va_starttop_banked

- 不能指定 **va_starttop** 和 **va_starttop_banked** 函数的第一个参数，因为第一个参数通过寄存器传递。
- 使用下列宏
 - (i) 当指定第一个参数时使用 **va_starttop** 或 **va_starttop_banked** 宏。
 - (ii) 当指定第二个及以后的参数时，使用 **va_starttop** 或 **va_starttop_banked** 宏。

va_starttop_banked

- **va_starttop** 不能用于那些通过 **bank** 函数调用例程来发起调用的函数，针对这种函数请使用 **va_starttop_banked**。

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。
此处所指的相应位置就是紧跟在 **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）。

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。

10.4.4 I/O 函数

(1) `sprintf` (仅正常模式)

功能

`sprintf` 函数根据格式将数据写入字符串（数组）中。

头文件

`stdio.h`

函数原型

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

函数	参数	返回值
<code>sprintf</code>	<p>s ... 指向将输出写入内容的字符串的指针</p> <p>format ... 该指针指向的字符串用来说明格式命令的规范</p> <p>... .. 零个或多个要转换的参数</p>	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。由此精度说明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：

表 10-14 **sprintf** 的标志

标志	内容
-	转换结果在字段内左对齐。
+	带符号转换的结果总是用+或-符号开头。
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。

表 10-15 printf 的格式代码

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。

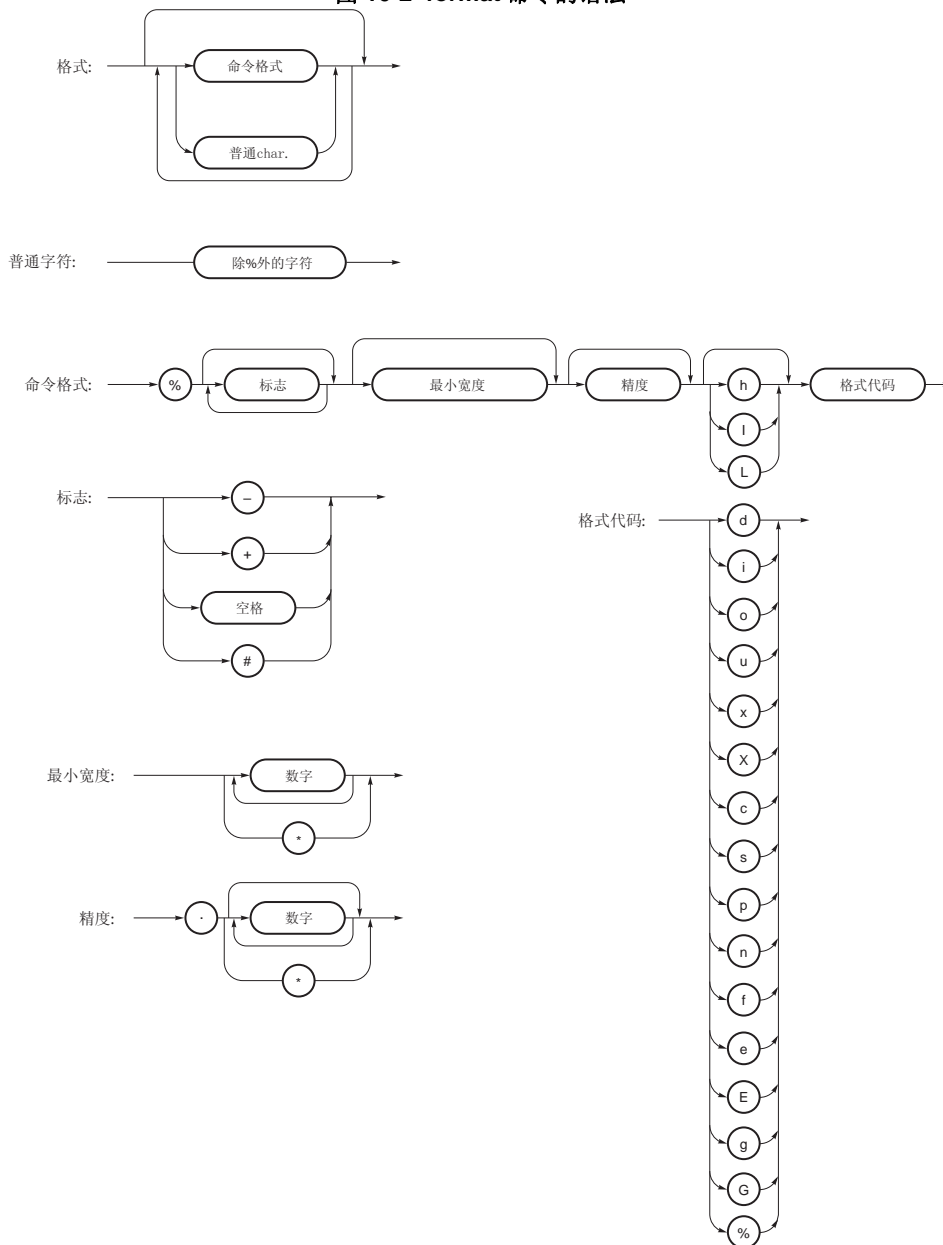
表 10-16 printf 的精度代码

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 命令的语法如图 10-2 所示。

图 10-2 format 命令的语法



(2) **sscanf** (仅正常模式)

功能

sscanf 函数根据格式从字符串（数组）中读取数据。

头文件

stdio.h

函数原型

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

函数	参数	返回值
sscanf	s ... 指向输入字符串的指针 format ... 该指针指向的字符串用来说明输入格式命令的规范 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出。
- 由 **format** 指向的控制字符串由零个（或更多）格式命令组成，可分为下列三类。
 - (1) 空白字符（使 **isspace** 为真的一个或更多个字符）
 - (2) 非空白字符（% 除外）
 - (3) 格式说明符
- 各格式说明符都以 % 字符开头，后面跟着下列内容：
 - (i) 可选的 * 字符，会限制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。

- 跳过输入的空白字符（由 **isspace** 指定），当类型说明符为 **[、c** 或 **n** 时除外。

- 从字符串“**s**”中读取输入数据项，当类型说明符为 **n** 时除外。输入数据项的被定义为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。

- 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：

表 10-17 sscanf 的转换指定符

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 $\pm \infty$ 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。

表 10-17 sscanf 的转换指定符

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。
[<p>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。</p> <p>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。</p> <p>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。</p>
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。

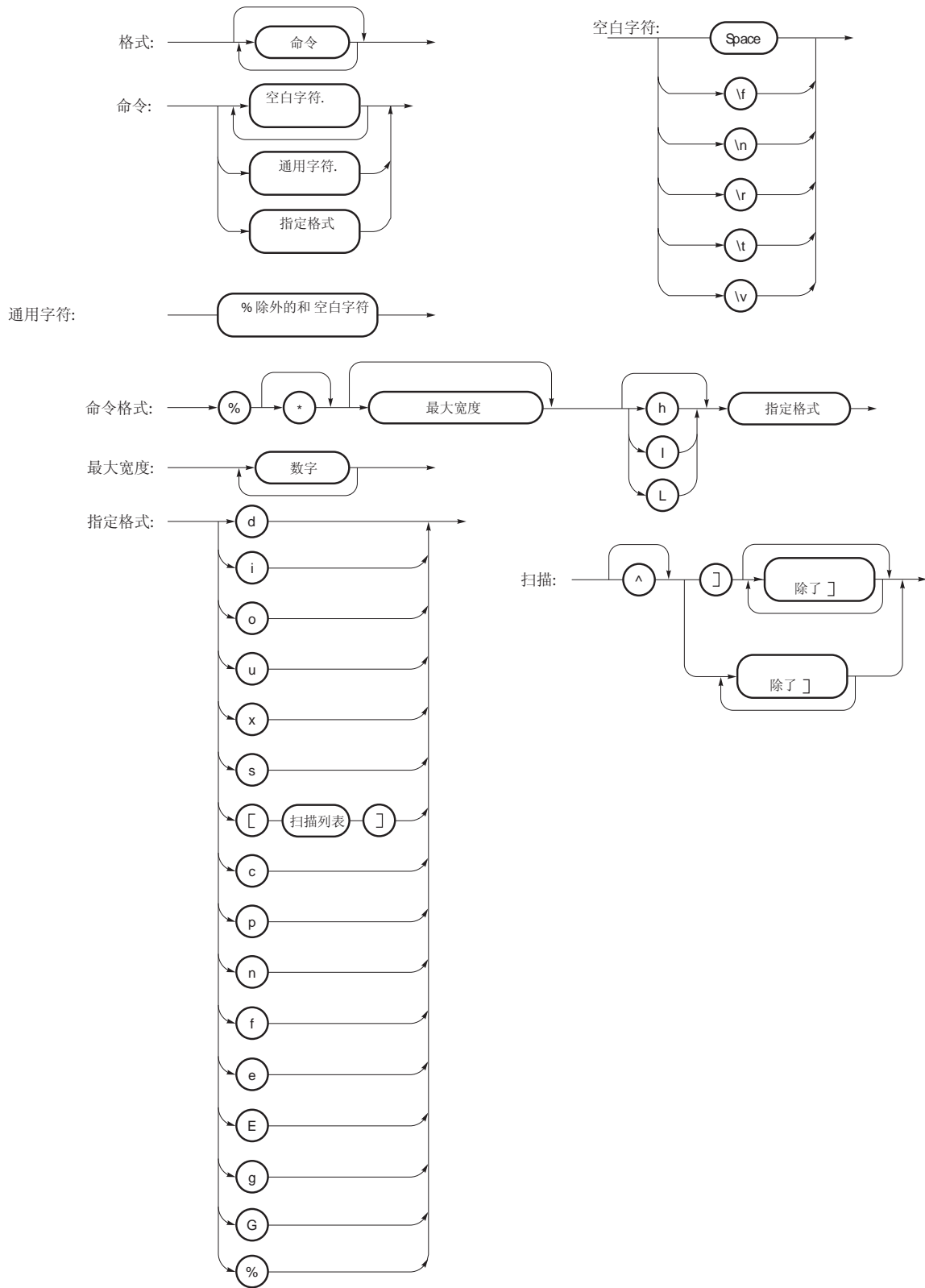
若格式说明符无效，则格式命令执行失败。

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。

图 10-3 输入 format 命令的语法



(3) **printf** (仅正常模式)

功能

printf 根据格式输出数据到 **SFR** 中。

头文件

stdio.h

函数原型

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

函数	参数	返回值
printf	format ... 该指针指向的字符串用来说明输出转换的规范 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。

(4) **scanf** (仅正常模式)

功能

scanf 按照格式从 **SFR** 中读取数据。

头文件

stdio.h

函数原型

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

函数	参数	返回值
scanf	format ... 该指针指向的字符串用来说明输入转换的规范 指针指向用来分配转换值的对象 (0 个或多个) 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。当参数个数过剩时，会对表达式评估但不会有实际输入。
- **format** 由 0 个或多个指令组成。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。

(5) **vprintf** (仅正常模式)

功能

vprintf 根据格式输出数据到 **SFR** 中。

头文件

stdio.h

函数原型

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

函数	参数	返回值
vprintf	format ... 该指针指向的字符串 用来说明输出转换的规范 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

- 按照格式规范中的输出转换说明，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。

(6) **vsprintf** (仅正常模式)

功能

vsprintf 按照格式将数据写入字符串中。

头文件

stdio.h

函数原型

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

函数	参数	返回值
vsprintf	s ... 指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范 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。

(7) getchar**功能**

getchar 从 SFR 中读取一个字符

头文件

stdio.h

函数原型

int getchar(void);

函数	参数	返回值
getchar	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。

(8) **gets**

功能

gets 读取一个字符串。

头文件

stdio.h

函数原型

char *gets(char *s);

函数	参数	返回值
gets	s ... 指向输入字符串的指针	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。

(9) putchar**功能**

putchar 输出一个字符到 **SFR** 中。

头文件

stdio.h

函数原型

int putchar(int c);

函数	参数	返回值
putchar	c ... 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。

(10) puts

功能

puts 输出一个字符串。

头文件

stdio.h

函数原型

int puts(const char *s);

函数	参数	返回值
puts	s ... 指向输出字符串的指针	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。

10.4.5 应用函数

(1) atoi, atol

功能

字符串函数 **atoi** 将十进制整数字符串的内容转换为 **int** 值。

字符串函数 **atol** 将十进制整数字符串的内容转换为 **long int** 值。

头文件

stdlib.h

函数原型

int atoi(const char *nptr);

long int atol(const char *nptr);

函数	参数	返回值
atoi	nptr... 要转换的字符串	<ul style="list-style-type: none"> • 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		<ul style="list-style-type: none"> • 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)。

(2) **strtol, strtoul**

功能

字符串函数 **strtol** 将字符串转换为 **long** 整型。

字符串函数 **strtoul** 将字符串转换为 **unsigned long** 整型。

头文件

stdlib.h

函数原型

long int strtol(const char *nptr, char **endptr, int base);

unsigned long int strtoul(const char *nptr, char **endptr, int base);

函数	参数	返回值
strtol	nptr... 要转换的字符串 endptr ... 存储指向不可识别段的指针的指针 base ... 指定的基数	<ul style="list-style-type: none"> • 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		<ul style="list-style-type: none"> • 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）

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**。

(3) 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	<ul style="list-style-type: none"> • 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。存储器函数 **brk** 的间断值设置见 [10.4.5 应用函数 \(11\) brk, sbrk](#)。

(4) free**功能**

存储器函数 **free** 会释放已分配的存储块。

头文件

stdlib.h

函数原型

void free(void *ptr);

函数	参数	返回值
free	ptr ... 指针指向将要释放的存储块开头位置	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。）

(5) **malloc**

功能

存储器函数 **malloc** 分配存储块。

头文件

stdlib.h

函数原型

```
void *malloc(size_t size);
```

函数	参数	返回值
malloc	size ... 要分配的存储块的大小	<ul style="list-style-type: none"> • 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。
- 存储器的分配从中断值处开始，紧邻所分配空间的下一个地址将成为新的中断值。存储器函数 **brk** 的中断值设置见 [10.4.5 应用函数 \(11\) brk, sbrk](#)。

(6) realloc

功能

存储器函数 **realloc** 重新分配存储块（即，改变已分配存储区的大小）。

头文件

stdlib.h

函数原型

```
void *realloc(void *ptr, size_t size);
```

函数	参数	返回值
realloc	<p>ptr ... 指针指向先前分配的存储块开头</p> <p>size ... 要给此存储块设定的新长度</p>	<ul style="list-style-type: none"> • 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中断值。

(7) **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。

(8) 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	<ul style="list-style-type: none"> • 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	无返回。

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。

(9) **abs, labs**

功能

数学函数 **abs** 返回其 **int** 类型参数的绝对值。

数学函数 **labs** 返回其 **long** 类型参数的绝对值。

头文件

stdlib.h

函数原型

int abs(int j);

long int labs(long int j);

函数	参数	返回值
abs	j ... 要得到的绝对值	<ul style="list-style-type: none"> 若 j 在下列范围内，则返回 j 的绝对值 $-32767 \leq j \leq 32767$ 若 j 为 -32768，则返回 -32768 (0x8000)
labs		<ul style="list-style-type: none"> 若 j 在下列范围内，则返回 j 的绝对值 $-2147483647 \leq j \leq 2147483647$ 若 j 的值为 -2147483648，则返回 -2147483648 (0x80000000)

说明

abs

- **abs** 返回其 **int** 类型参数的绝对值。
- 若 **j** 为 -32768 ，函数返回 -32768 。

labs

- **labs** 返回其 **long** 类型参数的绝对值。
- 若 **j** 的值为 -2147483648 ，则函数返回 -2147483648 。

(10) **div** (仅正常模式), **ldiv** (仅正常模式)

功能

数学函数 **div** 进行用分子除以分母的整数除法。

数学函数 **ldiv** 进行用分子除以分母的长整数除法。

头文件

stdlib.h

函数原型

div_t **div**(int numer,int denom);

ldiv_t **ldiv**(long int numer,long int denom);

函数	参数	返回值
div	numer ... 除法的分子 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 乘以商**) 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。

(11) **brk, sbrk**

功能

存储器函数 **brk** 设置中断值。

存储器函数 **sbrk** 增加或减小设置的中断值。

头文件

stdlib.h

函数原型

int brk(char *endds);

char *sbrk(int incr);

函数	参数	返回值
brk	endds ... 用来设置存储块释放位置的中断值	<ul style="list-style-type: none"> • 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）	<ul style="list-style-type: none"> • 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)**。

sbrk

- **sbrk** 函数用 **incr** 指定的字节数增加或减小设置的中断值。（增加还是减小，由 **incr** 的正负号决定。）
- 若增加或减小后的中断值处于许可地址范围之外，则函数不改变原中断值，并将 **errno** 设置给 **ENOMEM (3)**。

(12) `atof`, `strtod`

功能

字符串函数 `atof` 将十进制整数字符串的内容转换为 `double` 值。

字符串函数 `strtod` 将字符串的内容转换为 `double` 值。

头文件

`stdlib.h`

函数原型

`double atof(const char *nptr);`

`double strtod(const char *nptr, char **endptr);`

函数	参数	返回值
<code>atof</code>	<code>nptr</code> ... 要转换的字符串	<ul style="list-style-type: none"> 若正确转换，则返回转换后的值 若出现正上溢，则返回 <code>HUGE_VAL</code>（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
<code>strtod</code>	<code>nptr</code> ... 要转换的字符串 <code>endptr</code> ... 存储指向不可识别块的指针的指针	<ul style="list-style-type: none"> 若正确转换，则返回转换后的值 若出现正上溢，则返回 <code>HUGE_VAL</code>（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。

(13) **itoa**, **ltoa** (仅正常模式), **ultoa** (仅正常模式)

功能

itoa 字符串函数将 **int** 整数转换为对应的字符串。

字符串函数 **ltoa** 将 **long int** 整数转换为对应的字符串。

字符串函数 **ultoa** 将 **unsigned long** 整数转换为对应的字符串。

头文件

stdlib.h

函数原型

char *itoa(int value,char *string,int radix);

char *ltoa(long value,char *string,int radix);

char *ultoa(unsigned long value,char *string,int radix);

函数	参数	返回值
itoa , ltoa , ultoa	value ... 整数要转换成的字符串 string ... 指向转换结果的指针 radix ... 输出字符串的基	<ul style="list-style-type: none"> • 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。

(14) rand, srand

功能

数学函数 **rand** 生成伪随机数序列。

数学函数 **srand** 对由 **rand** 生成的序列进行初始值 (**seed**) 设置。

头文件

stdlib.h

函数原型

int rand(void);

void srand(unsigned int seed);

函数	参数	返回值
rand	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）

(15) **bsearch** (仅正常模式)

功能

bsearch 函数进行二分检索。

头文件

stdlib.h

函数原型

```
void *bsearch(const void *key,const void *base,size_t nmemb,
             size_t size,int (*compare)(const void *,const void *));
```

函数	参数	返回值
bsearch	<p>key ... 指向进行检索的关键字的指针</p> <p>base ... 指针指向包含检索信息的数组</p> <p>nmemb ... 数组元素的个数</p> <p>size ... 数组长度</p> <p>compare ... 指针指向用来比较两个关键字的函数</p>	<ul style="list-style-type: none"> • 若数组包含关键字，返回指向第一个匹配“key”的成员的指针； • 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。
- 当指定了 **-ZR** 选项时，被转递给 **bsearch** 函数参数的函数类型必须是 **pascal** 函数。

(16) **qsort** (仅正常模式)

功能

qsort 函数用 **quicksort** 算法对指定数组的成员进行排序。

头文件

stdlib.h

函数原型

```
void qsort(void *base,size_t nmemb,size_t size,
           int (*compare)(const void *,const void *));
```

函数	参数	返回值
qsort	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。
- 当指定了 **-ZR** 选项时，被转递给 **qsort** 函数参数的函数类型必须为 **pascal** 函数。

(17) **strbrk**

功能

strbrk 设置中断值。

头文件

stdlib.h

函数原型

int strbrk(char *endds);

函数	参数	返回值
strbrk	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。

(18) **strsbrk**

功能

strsbrk 增大/减小中断值。

头文件

stdlib.h

函数原型

char *strsbrk(int incr);

函数	参数	返回值
strsbrk	incr ... 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。

(19) **strtoa**, **strltoa** (仅正常模式), **strultoa** (仅正常模式)

功能

strtoa 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 *strtoa(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
strtoa 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

strtoa、**strltoa**、**strultoa**

- 将指定的数值 **value** 转换为以空字符串结束符结尾的字符串，并将结果存储在由字符串 **string** 指定的区域中。进行转换时，使用指定的 **radix**，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。

10.4.6 字符串/存储器函数

(20) `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
<code>memcpy</code> <code>memmove</code>	<p>s1 ... 指向数据要拷贝的目的对象的指针</p> <p>s2 ... 指向包含要拷贝数据的源对象的指针</p> <p>n ... 要拷贝的字符数量</p>	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。

(21) 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 $s2 < s1 \leq (s2 + \text{要拷贝的字符数量})$ ，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 $s2 < s1 \leq (s2 + \text{要拷贝的字符长度或 } s2 \text{ 的最小值} + n - 1)$ ，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。

(22) **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 strncat	<p>s1... 此指针指向连接的目的字符串</p> <p>s2 ... 此指针指向源字符串，将 S2 所指的字符串拷贝到该字符串 (S1) 中。</p> <p>n ... 要连接的字符数量</p>	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。

(23) 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	<ul style="list-style-type: none"> • 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。
- 如果两个对象相等，则函数返回 0。
- 若对象 **s1** 大于对象 **s2**，函数返回正值，若 **s1** 小于 **s2**，返回负值。

(24) `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
<code>strcmp</code>	<p>s1... 指向待比较的第一个字符串的指针</p> <p>s2 ... 指向待比较的另一个字符串的指针</p>	<ul style="list-style-type: none"> 若 s1 等于 s2, 返回 0 若 s1 小于或大于 s2, 则返回小于 0 或大于 0 的整数 (s1 - s2)
<code>strncmp</code>	<p>s1... 指向待比较的第一个字符串的指针</p> <p>s2 ... 指向待比较的另一个字符串的指针</p> <p>n ... 待比较的字符数量</p>	<ul style="list-style-type: none"> 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, 则返回小于 0 或大于 0 的整数(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**）。

`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**）。

(25) 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	<ul style="list-style-type: none"> • 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。

(26) **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	<ul style="list-style-type: none"> 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。

(27) **strspn, strcspn**

功能

字符串函数 **strspn** 返回一个字符串的初始子串的长度，在另一个字符串中包含的字符仅由该子串组成。
字符串函数 **strcspn** 返回一个字符串的初始子串的长度，在另一个字符串中包含的字符仅由该子串组成。

头文件

string.h

函数原型

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

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

函数	参数	返回值
strspn	s1... 指向待搜索的字符串的指针	字符串 s1 中仅由 s2 字符串中构成的子串的长度
strcspn	s2 ... 指针指向的字符串，其内容用来匹配	字符串 s1 中仅由 s2 字符串构成的子串的长度

说明

strspn

- **strspn** 函数返回 **s1** 指针所指向的字符串内包含的子串长度，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** 字符串和 **s2** 中指定的任何一个字符都不符合，该函数返回字符串 **s1** 中第一个字符的索引位置。
- **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** 字符串中的字符和 **s2** 中指定的任何一个字符相符的话，该函数返回字符串 **s1** 中第一个匹配字符的索引位置。
- **s2** 的空终结符不计入 **s2** 的长度。

(28) **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	<p>s1... 指向待搜索的字符串的指针</p> <p>s2 ... 指针指向的字符串其内容用来匹配</p>	<ul style="list-style-type: none"> • 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。

(29) 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	<p>s1... 指向待搜索的字符串的指针</p> <p>s2 ... 指针指向给定的字符串</p>	<ul style="list-style-type: none"> • 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。

(30) 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	<p>s1... 指向字符串的指针，从该字符串中获得记号；或为空指针</p> <p>s2 ... 指向包含记号分界符的字符串的指针</p>	<ul style="list-style-type: none"> • 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。

(31) **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 ... 指向存储器中要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。

(32) **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	<ul style="list-style-type: none"> • 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。

- 0..... “错误 0”
- 1 (EDOM)..... “参数过大”
- 2 (ERANGE) “结果过大”
- 3 (ENOMEM) ... “存储空间不够”

其它情况下，函数返回空指针。

(33) 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。

(34) 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 int 后的差值 (s1 的字符 - s2 的 字符)

说明

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

(35) `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
<code>strxfrm</code>	<p><code>s1</code> ... 指向比较字符串的指针</p> <p><code>s2</code> ... 指向比较字符串的指针</p> <p><code>n</code> ... <code>s1</code> 中的最大字符数量</p>	<p>返回转换得到的字符串的长度（不包含指示结尾的字符串）</p> <p>若返回值为 <code>n</code>（或更多），则 <code>s1</code> 所指示的数组内容无定义。</p>

说明

- 本编译器不支持文化领域（特殊字符）的特定操作。与下列函数操作相同。

`strncpy (s1, s2, c);`

`return (strlen (s2));`

10.4.7 数学函数

(1) **acos** (仅正常模式)

功能

acos 计算反余弦。

头文件

math.h

函数原型

double acos (double x);

函数	参数	返回值
acos	x ... 进行操作的数值	当 $-1 \leq x \leq 1$ 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)。
- 若 **x** 非数值, 返回 **NaN**。
- 当出现 $x < -1$ 、 $1 < x$ 的定义域错误时, 返回 **NaN**, 设置 **EDOM**。

(2) `asin` (仅正常模式)

功能

`asin` 计算反正弦。

头文件

`math.h`

函数原型

`double asin (double x);`

函数	参数	返回值
<code>asin</code>	<code>x</code> ... 进行操作的数值	当 $-1 \leq x \leq 1$ 时 ... <code>x</code> 的 <code>asin</code> 当 $x < -1$, $1 < x$, $x = \text{NaN}$... <code>NaN</code> 当 <code>x = -0</code> 时 ... <code>-0</code> 当出现下溢时 ... 非标准数

说明

- 计算 `x` 的 `asin` 值 (在 $-\pi/2$ 和 $+\pi/2$ 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。

(3) **atan** (仅正常模式)

功能

atan 计算反正切。

头文件

math.h

函数原型

double atan (double x) ;

函数	参数	返回值
atan	x ... 进行操作的数值	通常... x 的 atan 当 x = NaN 时... NaN 当 x = -0 时... -0

说明

- 计算 **x** 的 **atan** 值 (在 $-\pi/2$ 到 $+\pi/2$ 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。

(4) `atan2` (仅正常模式)

功能

`atan2` 计算 y/x 的反正切。

头文件

`math.h`

函数原型

`double atan2 (double y, double x);`

函数	参数	返回值
<code>atan2</code>	<p><code>x</code> ... 进行操作的数值</p> <p><code>y</code> ... 进行操作的数值</p>	<p>通常 ... y/x 的 <code>atan</code></p> <p>当 <code>x</code> 和 <code>y</code> 均为 0 或 y/x 的值无法表示时, 或者当 <code>x</code> 或 <code>y</code> 为 <code>NaN</code> 及 <code>x</code> 和 <code>y</code> 均为 $\pm \infty$ 时 ...</p> <p>NaN</p> <p>非标准数 ... 当出现下溢时</p>

说明

- 计算 y/x 的 `atan` (在 $-\pi$ 到 $+\pi$ 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。

(5) `cos` (仅正常模式)

功能

`cos` 计算余弦值。

头文件

`math.h`

函数原型

`double cos (double x);`

函数	参数	返回值
<code>cos</code>	<code>x ...</code> 进行操作的数值	通常 ... <code>x</code> 的 <code>cos</code> 当 <code>x = NaN</code> 、 <code>x = ±∞</code> 时 ... <code>NaN</code>

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。

(6) **sin** (仅正常模式)

功能

sin 计算正弦值。

头文件

math.h

函数原型

double sin (double x) ;

函数	参数	返回值
sin	x ... 进行操作的数值	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。

(7) **tan** (仅正常模式)

功能

tan 计算正切值。

头文件

math.h

函数原型

double tan (double x);

函数	参数	返回值
tan	x ... 进行操作的数值	通常 ... x 的 tan 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。

(8) `cosh` (仅正常模式)

功能

`cosh` 计算双曲余弦。

头文件

`math.h`

函数原型

`double cosh (double x);`

函数	参数	返回值
<code>cosh</code>	<code>x ...</code> 进行操作的数值	通常 ... <code>x</code> 的 <code>cosh</code> 当出现上溢、 <code>x = NaN</code> 或 <code>x = ±∞</code> 时 ... <code>HUGE_VAL</code> (带正号) <code>x = NaN</code> ... <code>NaN</code>

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`。

(9) **sinh** (仅正常模式)

功能

sinh 计算双曲正弦。

头文件

math.h

函数原型

double sinh (double x) ;

函数	参数	返回值
sinh	x ... 进行操作的数值	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**。

(10) tanh (仅正常模式)

功能

tanh 计算双曲正切。

头文件

math.h

函数原型

double tanh (double x) ;

函数	参数	返回值
tanh	x ... 进行操作的数值	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 $\pm\infty$ ，返回 ± 1 。
- 若由于操作产生了下溢，返回 ± 0 。

(11) `exp` (仅正常模式)

功能

`exp` 计算指数函数。

头文件

`math.h`

函数原型

`double exp (double x);`

函数	参数	返回值
<code>exp</code>	<code>x</code> ... 进行操作的数值	通常 ... <code>x</code> 的指数函数 当 <code>x = NaN</code> 时... <code>NaN</code> 当 <code>x = ±∞</code> 时 ... <code>±∞</code> 当出现上溢时 ... <code>HUGE_VAL</code> (带正号) 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 <code>+0</code>

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`。

(12) frexp (仅正常模式)

功能

frexp 计算尾数和指数部分。

头文件

math.h

函数原型

double frexp (double x, int *exp) ;

函数	参数	返回值
frexp	x ... 进行操作的数值 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 $x = m \cdot 2^n$ ，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。

(13) `ldexp` (仅正常模式)

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
<code>exp</code>	<code>x</code> ... 进行操作的数值 <code>exp</code> ... 指数	通常 ... $x*2^{exp}$ 当 <code>x = NaN</code> 时... <code>NaN</code> 当 <code>x = ±∞</code> 时 ... $±∞$ 当 <code>x = ±0</code> 时... $±0$ 当出现上溢时 ... <code>HUGE_VAL</code> (带上溢值的符号) 当出现下溢时 返回 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 $±0$ 。

(14) log (仅正常模式)

功能

log 计算自然对数。

头文件

math.h

函数原型

double log (double x) ;

函数	参数	返回值
log	x ... 进行操作的数值	通常 ... x 的自然对数 当 x ≤ 0 时... HUGE_VAL (带负号) 当 x 为非数值时 ... NaN 当 x 为无穷大时 ... $+\infty$

说明

- 计算 **x** 的自然对数。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为 $+\infty$ ，返回 $+\infty$ 。
- 出现 **x** < 0 的定义域错误时，返回带负号的 **HUGE_VAL** 且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回带负号的 **HUGE_VAL** 且将 **ERANGE** 设置给 **errno**。

(15) **log10** (仅正常模式)

功能

log10 计算以 10 为底的对数。

头文件

math.h

函数原型

double log10 (double x) ;

函数	参数	返回值
log10	x ... 进行操作的数值	通常 ... x 的 10 底对数 当 x ≤ 0 时 ... HUGE_VAL (带负号) 当 x 为非数值时 ... NaN 当 x 为无穷大时 ... $+\infty$

说明

- 计算 **x** 的 10 底对数。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为 $+\infty$ ，返回 $+\infty$ 。
- 当出现 **x** < 0 的定义域错误时，返回带负号的 **HUGE_VAL** 且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回带负号的 **HUGE_VAL** 且将 **ERANGE** 设置给 **errno**。

(16) `modf` (仅正常模式)

功能

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

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。

(17) pow (仅正常模式)

功能

pow 计算 x 的 y 次幂。

头文件

math.h

函数原型

double pow (double x, double y) ;

函数	参数	返回值
pow	x ... 进行操作的数值 y ... 倍数	通常 ... x^y 当 x = NaN 或 y = NaN 时, 当 x = $+\infty$ 且 y = 0 x < 0 且 y 不是整数, x < 0 且 y = $-\infty$ 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 。
- 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**。
- 当 **x** = NaN 或 **y** = NaN 时，返回 NaN。
- 当 **x** = $+\infty$ 且 **y** = 0 时，或 **x** < 0 且 **y** 不是整数时，或 **x** < 0 且 **y** = $-\infty$ ，或 **x** = 0 且 **y** ≤ 0 时，返回 NaN 且将 **EDOM** 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。

(18) sqrt (仅正常模式)

功能

sqrt 计算平方根。

头文件

math.h

函数原型

double sqrt (double x);

函数	参数	返回值
sqrt	x ... 进行操作的数值	当 x ≥ 0 时 ... x 的平方根 当 x = 0 时 ... 0 当 x < 0 时 ... NaN

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。

(19) `ceil` (仅正常模式)

功能

`ceil` 计算不小于 x 的最小整数。

头文件

`math.h`

函数原型

`double ceil (double x);`

函数	参数	返回值
<code>ceil</code>	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 $x = \infty$ 时 ... NaN 当 $x = -0$ 时 ... <code>+0</code> 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 `-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 的最小整数的无法表达时，返回 x 。

(20) fabs (仅正常模式)

功能

fabs 返回浮点数 **x** 的绝对值。

头文件

math.h

函数原型

double fabs (double x) ;

函数	参数	返回值
fabs	x ... 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**。

(21) floor (仅正常模式)

功能

floor 计算不大于 **x** 的最大整数。

头文件

math.h

函数原型

double floor (double x);

函数	参数	返回值
floor	x ...进行操作的数值	通常 ... 不大于 x 的最大整数 当 x 为非数值或 $x = \infty$ 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-0，返回+0。
- 若 **x** 为无穷大，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若不大于 **x** 的最大整数无法表达，则返回 **x**。

(22) fmod (仅正常模式)

功能

fmod 计算 x/y 的余数。

头文件

math.h

函数原型

double fmod (double x, double y);

函数	参数	返回值
fmod	x ... 进行操作的数值 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 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）。

(23) `matherr` (仅正常模式)

功能

`matherr` 对浮点数操作库进行异常处理。

头文件

`math.h`

函数原型

```
void matherr (struct exception *x);
```

函数	参数	返回值
<code>matherr</code>	<pre>struct exception { int type; char *name; } type..... 指示算术异常的数值编号 name... 函数名</pre>	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 <code>errno</code> 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。

(24) `acosf` (仅正常模式)

功能

`acosf` 计算反余弦。

头文件

`math.h`

函数原型

`float acosf (float x);`

函数	参数	返回值
<code>acosf</code>	<code>x</code> ... 进行操作的数值	当 $-1 \leq x \leq 1$ 时 ... <code>x</code> 的反余弦 当 $x \leq -1$ 、 $1 < x$ 、 <code>x</code> 非数值 时 ... <code>NaN</code>

说明

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

(25) `asinf` (仅正常模式)

功能

`asinf` 计算反正弦。

头文件

`math.h`

函数原型

`float asinf (float x);`

函数	参数	返回值
<code>asinf</code>	<code>x</code> ... 进行操作的数值	当 $-1 \leq x \leq 1$ 时 ... <code>x</code> 的反正弦 当 $x \leq -1$ 、 $1 < x$ 、 $x = \text{NaN}$ 时 ... NaN <code>x = -0</code> ... <code>-0</code> 当出现下溢时 ... 非标准数

说明

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

(26) `atanf` (仅正常模式)

功能

`atanf` 计算反正切。

头文件

`math.h`

函数原型

`float atanf (float x);`

函数	参数	返回值
<code>atanf</code>	<code>x</code> ... 进行操作的数值	通常 ... <code>x</code> 的反正切 当 <code>x = NaN</code> 时 ... <code>NaN</code> 当 <code>x = -0</code> 时 ... <code>-0</code>

说明

- 计算 `x` 的反正切 (在 $-\pi/2$ 到 $+\pi/2$ 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。

(27) atan2f (仅正常模式)

功能

atan2f 计算 y/x 的反正切。

头文件

math.h

函数原型

float atan2f (float y, float x);

函数	参数	返回值
atan2f	<p>x ... 进行操作的数值</p> <p>y ... 进行操作的数值</p>	<p>通常</p> <p>...y/x 的反正切</p> <p>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 $\pm\infty$ 时</p> <p>... NaN</p> <p>当出现下溢时</p> <p>... 非标准数</p>

说明

- 计算 y/x 的反正切（在 $-\pi$ 到 $+\pi$ 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。

(28) cosf (仅正常模式)

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 = ±∞ 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。

(29) `sinf` (仅正常模式)

功能

`sinf` 计算正弦。

头文件

`math.h`

函数原型

`float sinf (float x);`

函数	参数	返回值
<code>sinf</code>	<code>x</code> ... 进行操作的数值	通常 ... <code>x</code> 的正弦 当 <code>x = NaN</code> 、 <code>x = ±∞</code> 时 ... <code>NaN</code> 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。

(30) `tanf` (仅正常模式)

功能

`tanf` 计算正切。

头文件

`math.h`

函数原型

`float tanf (float x) ;`

函数	参数	返回值
<code>tanf</code>	<code>x ...</code> 进行操作的数值	通常 ... <code>x</code> 的正切 当 <code>x = NaN</code> 、 <code>x = ±∞</code> 时... <code>NaN</code> 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。

(31) `coshf` (仅正常模式)

功能

`coshf` 计算双曲余弦。

头文件

`math.h`

函数原型

`float coshf (float x);`

函数	参数	返回值
<code>coshf</code>	<code>x</code> ... 进行操作的数值	通常 ... <code>x</code> 的双曲余弦 当出现上溢、 <code>x = ±∞</code> 时 ... HUGE_VAL (带正号) <code>x = NaN</code> ... NaN

说明

- 计算 `x` 的双曲余弦。
- 若 `x` 为非数值，返回 **NaN**。
- 若 `x` 为无穷大，返回正无穷大值。
- 如果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**。

(32) `sinhf` (仅正常模式)

功能

`sinhf` 计算双曲正弦。

头文件

`math.h`

函数原型

`float sinhf (float x);`

函数	参数	返回值
<code>sinhf</code>	<code>x</code> ... 进行操作的数值	通常 ... <code>x</code> 的双曲正弦 当出现上溢时... <code>HUGE_VAL</code> (带上溢值的符号) <code>x = NaN ... NaN</code> 当 <code>x = ±∞</code> 时... <code>±∞</code> 当出现下溢时 ... <code>±0</code>

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`。

(33) **tanhf** (仅正常模式)

功能

tanhf 计算双曲正切。

头文件

math.h

函数原型

float tanhf (float x) ;

函数	参数	返回值
tanhf	x ... 进行操作的数值	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**。

(34) `expf` (仅正常模式)

功能

`expf` 计算指数函数。

头文件

`math.h`

函数原型

`float expf (float x);`

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

说明

- 计算 `x` 的指数函数。
- 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`。

(35) frexpf (仅正常模式)

功能

frexpf 计算尾数和指数部分。

头文件

math.h

函数原型

float frexpf (float x, int *exp) ;

函数	参数	返回值
frexpf	x ... 进行操作的数值 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 $x = m \cdot 2^n$ ，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 $\pm\infty$ ，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。

(36) **ldexpf** (仅正常模式)

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 = $\pm\infty$ 时... $\pm\infty$ 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 $\pm\infty$ ，返回 $\pm\infty$ 。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 。

(37) `logf` (仅正常模式)

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
<code>logf</code>	<code>x ...</code> 进行操作的数值	通常... <code>x</code> 的自然对数 当 <code>x</code> 为非数值时... <code>NaN</code> 当 <code>x</code> 为无穷大时... $+\infty$ 当 <code>x</code> ≤ 0 时... <code>HUGE_VAL</code> (带负号)

说明

- 计算 `x` 的自然对数。
- 若 `x` 为非数值, 返回 `NaN`。
- 若 `x` 为 $+\infty$, 返回 $+\infty$ 。
- 在 `x < 0` 的域错误情况下, 返回带负号的 `HUGE_VAL`, 且将 `EDOM` 设置给 `errno`。
- 若 `x = 0`, 返回带负号的 `HUGE_VAL` 且将 `ERANGE` 设置给 `errno`。

(38) log10f (仅正常模式)

功能

log10f 计算 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 为非数值时... NaN 当 x = $+\infty$ 时... $+\infty$ 当 x ≤ 0 时... HUGE_VAL (带负号)

说明

- 计算 **x** 的 10 底对数。
- 若 **x** 为非数值，返回 **NaN**。
- 若 **x** 为 $+\infty$ ，返回 $+\infty$ 。
- 在 **x** < 0 的域错误情况下，返回带负号的 **HUGE_VAL**，且将 **EDOM** 设置给 **errno**。
- 若 **x** = 0，返回带负号的 **HUGE_VAL** 且将 **ERANGE** 设置给 **errno**。

(39) **modff** (仅正常模式)

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** 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。

(40) `powf` (仅正常模式)

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
<code>powf</code>	<p>x ... 进行操作的数值</p> <p>y ... 乘数</p>	<p>通常... x^y</p> <p>当</p> <p>$x = \text{NaN}$ 或 $y = \text{NaN}$ 时</p> <p>$x = +\infty$ 且 $y = 0$ 时</p> <p>$x < 0$ 且 y 不为整数时</p> <p>$x < 0$ 且 $y = \pm\infty$ 时</p> <p>$x = 0$ 且 y 不等于 0 时... NaN</p> <p>当出现下溢时 ... 非标准数</p> <p>当出现上溢时...HUGE_VAL</p> <p>(带上溢值的符号)</p> <p>由于下溢导致有效位丢失时 0</p>

说明

- 计算 x^y 。
- 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**。
- 当 $x = \text{NaN}$ 或 $y = \text{NaN}$ 时，返回 **NaN**。
- 当 $x = +\infty$ 且 $y = 0$ 、或 $x < 0$ 且 y 不为整数时、或 $x < 0$ 且 $y = \pm\infty$ 时，或 $x = 0$ 且 $y \leq 0$ 时，返回 **NaN** 且将 **EDOM** 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。

(41) `sqrtf` (仅正常模式)

功能

`sqrtf` 计算平方根。

头文件

`math.h`

函数原型

`float sqrtf (float x) ;`

函数	参数	返回值
<code>sqrtf</code>	<code>x</code> ... 进行操作的数值	当 <code>x ≥ 0</code> 时... <code>x</code> 的平方根 当 <code>x = ±0</code> 时... <code>±0</code> 当 <code>x < 0</code> 时... NaN

说明

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

(42) `ceilf` (仅正常模式)

功能

`ceilf` 计算不小于 x 的最小整数。

头文件

`math.h`

函数原型

`float ceilf (float x);`

函数	参数	返回值
<code>ceilf</code>	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 $x = \pm\infty$ 时... NaN 当 $x = -0$ 时... <code>+0</code> 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 。

(43) **fabsf** (仅正常模式)

功能

fabsf 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**。

(44) **floorf** (仅正常模式)

功能

floorf 计算不大于 **x** 的最大整数。

头文件

math.h

函数原型

float floorf (float x) ;

函数	参数	返回值
floorf	x ... 进行操作的数值	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-0，返回+0。
- 若 **x** 为无穷大，则返回 **NaN** 且将 **EDOM** 设置给 **errno**。
- 若不大于 **x** 的最大整数无法表达，返回 **x**。

(45) **fmodf** (仅正常模式)

功能

fmodf 计算 x/y 的余数。

头文件

math.h

函数原型

float fmodf (float x, float y) ;

函数	参数	返回值
fmodf	x ... 进行操作的数值 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 $\pm\infty$ 时... NaN 当 x $\pm\infty$ 且 y $=\pm\infty$ 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** $=\pm\infty$, 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)。

10.4.8 诊断函数

(1) `__assertfail` (仅正常模式)

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
<code>__assertfail</code>	<p><code>__msg</code> ... 指针指向的字符串指示将被传递给 <code>printf</code> 函数的输出转换说明</p> <p><code>__cond</code> ... <code>assert</code> 宏的实际参数</p> <p><code>__file</code> ... 源文件名</p> <p><code>__line</code> ... 源行号</p>	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。

10.5 更新启动例程及库函数的批处理文件

本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 10-18 所示。

注意事项 BAT 目录下的 **d002.78k** 和 **d014.78k** 文件用来在批处理文件激活期间更新库，而不是用于开发。在开发系统时，必须相关的设备文件（另外单独提供）。

表 10-18 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[n].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 SM78K0S。 当需要用 SM78K0S 检查 putchar 函数的输出时，用本批处理文件对库进行更新。
repselo.bat	保存/恢复编译器的保留区（_@KREGxx），且将其作为 setjmp/longjmp 函数保存/恢复处理（默认为非保存/恢复）的必须部分。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。

10.5.1 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编译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 RA78K0S 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 RA78K0S 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)
054 ... PD78054 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 054

(2)固件 ROM 例程的更新

reprom 芯片类型 乘/除法指令存在

例 reprom 054

(3)getchar 函数的更新

repgetc 芯片类型乘/除法指令存在

例 repgetc 054

(4)putchar 函数的更新

repputc 芯片类型乘/除法指令存在

例 repputc 054

(5) putchar 函数（支持 SM78K0S）的更新

reputcs 芯片类型乘/除法指令存在

例 reputcs 054

(6) setjmp/longjmp 函数的更新（带恢复/保存处理）

repselo 芯片类型乘/除法指令存在

例 repselo 054

(7) setjmp/longjmp 函数的更新（不带恢复/保存处理）

repselon 芯片类型乘/除法指令存在

例 repselon 054

(8) 中断向量表的更新

repvect 芯片类型乘/除法指令存在

例 repvect 054

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 78K/0 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 K0S 微控制器的特有功能。有关 C 源程序面向其他微控制器的可移植性，他们在 C 语言上是兼容的。因此，使用这些扩展函数开发的 C 源程序可以方便的移植到其他微控制器上，并且易于修改，

备注 在本章介绍过程中，“RTOS”表示 78K/0 系列实时操作系统。

11.1 宏名称

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 `__K0S__` 和 `__054__`。

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
  >CC78K0 -C054 prime.c ...

指定设备类型:
  #pragma pc (054)
```

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 语言规范选项（-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
<code>__callt</code>	<code>callt</code>	<code>callt/_callt</code> 函数
<code>__callf</code>	<code>callf</code>	<code>callf/_callf</code> 函数
<code>__sreg</code>	<code>sreg</code>	<code>sreg/_sreg</code> 变量
	<code>noauto</code>	<code>noauto</code> 函数
<code>__leaf</code>	<code>norec</code>	<code>norec/_leaf</code> 函数
<code>__boolean</code>	<code>boolean</code>	<code>boolean</code> 型/ <code>__boolean</code> 类型变量
	<code>bit</code>	<code>bit</code> 型变量
<code>__interrupt</code>		硬件中断
<code>__interrupt_brk</code>		软件中断
<code>__asm</code>		汇编语句
<code>__rtos_interrupt</code>		为 RTOS 分配的句柄
<code>__pascal</code>		Pascal 函数
<code>__flash</code>		固件 ROM 函数
<code>__flashf</code>		<code>__flashf</code> 函数
<code>__directmap</code>		绝对地址分配
<code>__temp</code>		临时变量

(1) 函数

关键字 `callt`，`__callt`，`callf`，`__callf`，`noauto`，`norec`，`__leaf`，`__interrupt`，`__interrupt_brk`，`__rtos_interrupt`，`__flash`，`__flashf` 以及 `__pascal` 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]

<code>__callt int func (int) ;</code>

属性修饰词规范仅限于以下所列各项。（`noauto` 和 `norec/__leaf` 修饰词不能同时指定。）将 `callt` 和 `__callt`、`callf` 和 `__callf`，`norec` 和 `__leaf` 看作相同规范。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`
- `callf`
- `noauto`
- `norec`
- `callt noauto`
- `callt norec`
- `noauto callt`
- `norec callt`
- `callf noauto`
- `callf norec`
- `noauto callf`
- `norec callf`
- `__interrupt`
- `__interrupt_brk`
- `__rtos_interrupt`
- `__pascal`
- `__pascal noauto`
- `__pascal callt`
- `__pascal callf`
- `noauto __pascal`
- `callt __pascal`
- `callt __pascal`
- `__flash`
- `__flashf`

(2) 变量

- 适用于 `sreg` 或 `__sreg` 规范的规则对 C 语言中 **寄存器** 同样有效（如需详细信息，请参阅 [11.5 \(3\) 如何使用 `saddr` 区域](#)）。
- 适用于 `bit`、`boolean` 或 `__boolean` 规范的规则对 C 语言中的 `char` 或 `int` 型说明符同样有效。然而，这些类型只能用来指定函数外部定义的变量（外部变量）。

- 适用于 `__directmap` 规范的规则对 C 语言中有关类型修饰词同样有效（如需详细信息，请参阅 [11.5 \(44\) 绝对地址分配规范](#)）。
- 适用于 `__temp` 规范的规则对 C 语言中有关类型修饰词同样有效（如需详细信息，请参阅 [11.5 \(46\) 临时变量](#)）。

11.3 存储器

存储模式由目标设备的存储空间决定。

(1) 存储模式

因为存储空间最大为 64 KB，该模式为由代码部和数据部组合而成的 64 KB。通过使用 bank 函数代码划分可能得到大于 64K 的存储器空间。

(2) 寄存器组

- 启动时寄存器组被设置为“RB0”（在该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。
- 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

该 C 编译器使用存储空间的描述如下。

(a) 正常模式（默认）

图 11-1 存储空间的利用（正常模式）

地址	用途	大小（字节）
00	40 至 7FH CALLT 表	64
0800-0FFFH	CALLT 入口区	2048
FE	20 至 B7H sreg 变量, boolean 型变量	152
FE	B8 至 BFH 运行时刻库的参数	8
FE	C0 至 C7H norec 函数的参数	8
FE	C8 至 CFH norec 函数的自动变量	8
FE	D0 至 DFH 寄存器变量	16
FE	E0 至 F7H RB3-RB1 工作寄存器 ^注	24
	F8 至 FFH RB0 工作寄存器	8
FF	00 至 FFH sfr 变量	256

注 仅当指定了该组寄存器时才会使用。

(b) 静态模式（以-SM16 规范）

图 11-2 存储空间的利用（静态模式）

地址	用途	大小（字节）
00 : 40 至 7FH	CALLT 表	64
0800-0FFFH	CALLT 入口区	2048
FE : 20 至 CFH	sreg 变量, boolean 型变量	176
FE : D0 至 DFH	共享区域 ²	16
FE : 在 20 与 DFH 之间的连续区域	用于参数, 自动变量和工作区 ³	8
FE : E0 至 F7H	RB3-RB1 工作寄存器 ¹	24
FE : F8 至 FFH	RB0 工作寄存器	8
FF : 00 至 FFH	sfr 变量	256

- 注
1. 仅当指定了该组寄存器时才会使用。
 2. 编译器使用的区域根据-SM 选项指定的参数会有所变化。存放共享数据之后剩余的区域用来存放 **sreg** 变量和 **boolean** 型变量。
 3. 仅当指定静态模式扩展说明选项 (-ZM) 时有效。

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。

该 C 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 (4) 如何使用 sfr 区域
#pragma asm	在 C 源程序中嵌入 ASM 语句 → 11.5 (8) ASM 语句
#pragma vect #pragma interrupt	描述 C 源程序中处理的中断 → 11.5 (9) 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 (11) 中断函数 (#pragma DI, #pragma EI)
#pragma halt #pragma stop #pragma nop	描述 C 源程序中的 CPU 控制指令 → 11.5 (12) CPU 控制指令
#pragma access	使用绝对地址访问函数 → 11.5 (14) 绝对地址访问函数
#pragma section	改变编译器输出区段的名称，并指定区段的位置 → 11.5 (16) 改变编译器输出部分名称
#pragma name	改变模块名称 → 11.5 (18) 模块名称改变函数
#pragma rot	使用移位函数 → 11.5 (19) 移位函数
#pragma mul	使用乘法函数 → 11.5 (20) 乘法函数
#pragma div	使用除法函数 → 11.5 (21) 除法函数
#pragma bcd	使用 BCD 运算函数 → 11.5 (22) BCD 运算函数
#pragma opc	使用数据插入函数 → 11.5 (25) 数据插入函数

#pragma 指令	应用
#pragma rtos_interrupt	使用实时操作系统（RTOS）的中断处理程序→ 11.5（26）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（RTOS）的任务函数→ 11.5（28）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（34）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（35）从 boot 区域到 flash 区域的函数调用（#pragma ext_func）
#pragma realregister	使用寄存器直接引用函数 → 11.5（39）寄存器直接引用函数
#pragma hromcall	使用片上固件自编程子程序直接调用函数→ 11.5（41）片上固件自编程子程序直接调用函数（#pragma hromcall）
#pragma inline	扩展标准库 memcpy 和 memset 内联函数 → 11.5（43）存储器处理功能

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) **noauto**函数(**noauto**)
- (6) **norec**函数(**norec**)
- (7) 位变量与布尔型变量(**bit / boolean / __boolean**)
- (8) **ASM** 语句 (**#asm #endasm / __asm**)
- (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) **callf**函数 (**callf / __callf**)
- (14) 绝对地址访问函数(**#pragma access**)
- (15) 位域声明
- (16) 更改编译器输出区名称的函数(**#pragma section ...**)
- (17) 二进制常量描述函数(**Binary constant 0bxxx**)
- (18) 模块名更改函数(**#pragma name**)
- (19) 循环移位函数(**#pragma rot**)
- (20) 乘法函数(**#pragma mul**)
- (21) 除法函数(**#pragma div**)
- (22) **BCD**操作函数(**#pragma bcd**)
- (23) **Bank**函数
- (24) 常量地址上的**Bank**函数
- (25) 数据插入函数(**#pragma opc**)
- (26) 实时操作系统 (RTOS) 的中断处理程序(**RTOS**) (**#pragma rtos_interrupt ...**)
- (27) 实时操作系统 (RTOS) 的中断处理程序修饰词(**RTOS**) (**__rtos_interrupt**)
- (28) 实时操作系统 (RTOS) 的任务函数(**#pragma rtos_task**)
- (29) 静态模式
- (30) 类型更改(**-ZI**)

- (31) **Pascal函数(__pascal)**
- (32) **函数调用接口的自动pascal函数化(-ZR)**
- (33) **Flash区域分配方法(-ZF)**
- (34) **Flash区域跳转表(#pragma ext_table)**
- (35) **从boot区域到flash区域的函数调用(#pragma ext_func)**
- (36) **固件ROM函数(__flash)**
- (37) **参数/返回值的 int 扩展限制方法(-ZB)**
- (38) **数组偏移量计算简化方法(-QW2 / -QW3)**
- (39) **寄存器直接引用函数(#pragma realregister)**
- (40) **[HL+B]基址变址寻址使用方法(-QE)**
- (41) **片上固件自编程子程序直接调用函数(#pragma hromcall)**
- (42) **__flashf函数(__flashf)**
- (43) **内存操控函数(#pragma inline)**
- (44) **绝对地址分配规范(__directmap)**
- (45) **静态模式展开规范(-ZM)**
- (46) **临时变量(__temp)**
- (47) **支持序言/尾声的库(-ZD)**

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。

(1) callt 函数 (callt/ __callt)**功能**

- **callt** 指令会将调用函数的地址存储在称作 **callt** 表的区域[40H 至 7FH]中，于是使用更短的代码来调用该函数，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

- 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 40H 至 7FH。
- 当 **callt** 表中的 **callt** 属性函数数量超出允许范围时，将出现编译错误。

- 指定 **-QL** 选项才能够使用 **callt** 表。因此，每个加载模块所允许的 **callt** 属性数和链接模块中允许的总数量如表 11-4 所示。
- 当支持序言/结尾的的库选项 (**-ZD** 选项) 被指定时，不能使用 **-QL4** 选项。同样，因为正常模式下序言/结尾库需要使用两个 **callt** 入口，在静态模式下序言/结尾库需要使用的 **callt** 入口高达十个。所以正常模式下可用的 **callt** 入口的数量会减少两个，和静态模式下可用的 **callt** 入口的数量会减少，减少数量不超过十个。

表 11-3 指定 **-QL** 选项时可以使用的 **callt** 属性函数的数量

选项	-QL1	-QL2	-QL3	-QL4
正常模式	29	27	7	0
静态模式	32	28	19	10

- 其中不使用 **-QL** 选项的默认选项如下所示。

表 11-4 **callt** 函数用法的限制

callt 函数	限定值	
	正常模式	静态模式
每个加载模块中允许的数量	最大值 29	最大值 32
链接的模块中允许的总数量	最大值 29	最大值 32

备注 指定正常模式时，**bank** 函数调用库会占用 3 个 **callt** 表。关于 **bank** 函数的具体细节，敬请参阅“[\(23\) bank 函数](#)”

示例

(C 源程序)

```

===== ca1.c =====
__callt extern int tsub ( ) ;

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

===== ca2.c =====
__callt int tsub ( )
{
    int val;
    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
    @@CODE   CSEG
    _tsub:
        .
        .
        .
        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>

- 如果不使用关键字 **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](#)。

(2) 寄存器变量 (register)

功能

- 将声明的变量（包括函数参数）分配到寄存器（HL）和 **saddr 区域**（_**@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。
- 根据引用的次数来进行分配。因此，不能确定定义的寄存器变量被分配到哪个寄存器或 **saddr 区域**中的具体哪个位置。
- 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](#)。
- 在如下所示的编译条件时，寄存器变量会被分配到不同区域（如需有关每个选项的信息，请参阅 [CC78K0 C 编译器 操作篇](#)）。
 - (i). 在正常模式情况下，寄存器变量根据引用次数被分配到寄存器 **HL** 或 **saddr 区域**[**FED0H** 至 **FEDFH**]。如果不存在堆栈帧，则寄存器变量分配到寄存器 **HL**。仅当指定了**-QR** 选项时，寄存器变量才会被分配到 **saddr 区域**。
 - (ii). 在静态模式情况下，寄存器变量根据引用次数分配到**-SM** 选项参数保留的寄存器 **DE** 或_**@KREGxx**。仅当指定**-ZM2** 选项时，寄存器变量分配到_**@KREGxx**。如需**-ZM2** 选项的详细信息，请参阅 [11.5 \(45\) 静态模式扩展规范 \(-ZM\)](#)。

效果

- 分配到寄存器或 **saddr 区域**的变量的操作指令的代码长度通常比较短，短于分配到存储器的变量的操作指令代码长度。这有助于缩短目标代码，且还提高程序运行速度。

用法

如下用 **register** 存储类型符声明变量。

```
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。

(正常模式)

- **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 **pointers** 情况下，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起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 **pointers** 来说，每个函数最多可用 8 个寄存器变量。从第 9 个寄存器变量起就被分配到普通存储空间。

(静态模式)

- **char** 型占用的空间只有其他类型（整型）的一半。
- 在 **int/short** 和 **pointers** 情况下，每个函数最多可用 1 个寄存器变量。
- 从第 2 个变量起，寄存器变量被分配到普通存储空间。
- 对于 **long/float/double/long double** 型，不能声明为寄存器变量。

表 11-5 使用寄存器变量的限制

数据类型	可用数量（每个函数）	
	正常模式	静态模式
int/short	最多 8 个寄存器变量	最多 1 个寄存器变量
Pointer	最多 8 个寄存器变量 (如果函数不具有栈帧，则最多可用 9 个寄存器变量。当使用 bank 函数 (-MF) 时，最多有 4 个变量可用作函数指针)	最多 1 个寄存器变量

示例

(C 源代码)

```

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

```

- 当未指定 **-SM** 选项时（寄存器变量分配到寄存器 **HL** 和 **saddr** 区域的示例）

下列标签由启动例程来声明（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
_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	;
movw	hl, #00H	; 以下代码在函数中间输出
movw	_@KREG00, #01H	;
movw	ax, _@KREG00	;
xch	a, x	;
add	l, a	;
xch	a, x	;
addc	h, a	;
call	!_func	;
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		

- 当指定 **-SM** 选项时（寄存器变量分配到寄存器 **DE** 的示例）

_main:		
push	de	; 在函数开始处保存寄存器的内容
movw	de, #00H	;
movw	ax, #01H	;
movw	!L0003, ax	;
xch	a, x	;
add	e, a	;
xch	a, x	;
addc	d, a	;
call	!_func	;
pop	de	; 在函数结束处恢复寄存器的内容
ret		

说明

- 要使用寄存器变量，仅需要用 **register** 存储类型标志符进行声明。
- 在库中用 **PUBLIC** 声明的模块中的标签诸如 **_@KREG00**，会附加到该 C 编译器的输出文件中。

兼容性

<从另一种 C 编译器至本 C 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，在到程序中添加变量的 **register** 声明。

<从本 C 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。

(3) 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 [FE20H 至 FEB7H]（正常模式）和 [FE20H 至 FECFH]（静态模式），并且可以重定位。当这些变量超出以上区域范围时，出现编译错误。
- `sreg` 变量的处理方式和 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 [FE20H 至 FEFFH]。该范围 [FEB8H 至 FEFFH]（正常模式）和 [FED0H 至 FEFFH]（静态模式）由编译器使用，因此必须谨慎处理（请参阅表 11-2 存储空间的利用）。

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 **pointers** 情况下，每个加载模块最多可使用 92 个变量（当使用 **saddr** 区域[FE20H 至 FED7H] 时）。注意可使用的变量数量会减少，在使用 **bit** 型和 **boolean** 型变量、寄存器变量或 **norec** 和 **noauto** 函数时可使用的变量数量会减少（正常模式）。
- 在 **int/short** 和 **pointers** 情况下，每个加载模块最多可使用 104 个变量（当使用 **saddr** 区域[FE20H 至 FEEFH] 时）。注意可使用的变量数量在使用 **bit** 型和 **boolean** 型变量和共享区域时会减少（静态模式）。

下表说明每个加载模块最多可用的 **sreg** 变量数量。

表 11-6 使用 **sreg** 变量的限制

数据类型	sreg 变量的可用数量（每个加载模块）	
	使用 saddr 区域[FE20H 至 FEB7H]时	使用 saddr 区域[FE20H 至 FECFH]时
int/short 、 pointer	最多 76 个变量 [*] (当使用 bank 函数 (-MF) 时，最多只有 38 个变量可以使用)	最多 88 个变量 [*] (当使用 bank 函数 (-MF) 时，最多只有 44 个变量可以使用)

注 使用 **bit** 和 **boolean** 型变量时，可用的变量数量减少。

示例

(C 源代码)

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

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

以下示例展示了由用户创建的 **sreg** 变量的定义代码。如果未在 C 源代码中作出 **extern** 声明，C 编译器会输出以下代码。在此情况下，将 **ORG** 伪指令无法输出。

<汇编源程序>

```

PUBLIC  _hsmm0          ;声明
PUBLIC  _hsmm1          ;
PUBLIC  _hsptr          ;

@@DATS  DSEG  SADDRP    ; 分配到段
        ORG   0FE20H    ;
_hsmm0: DS   (2)        ;
_hsmm1: DS   (2)        ;
_hsptr:  DS   (2)        ;
```

在函数中输出以下代码。

<编译器的输出目标>

```
movw    ax,_hsmm0
xch     a,x
sub     a,_hsmm1
xch     a,x
subc    a,_hsmm1+1
movw    _hsmm0,ax
```

兼容性

<从另一种 C 编译器至本 C 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。

<从本 C 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** 变量的分配可以如表所示。

表 11-7 通过-RD 选项分配到 **saddr** 区域的变量

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 pointer 型变量（当使用 bank 函数 (-MF) 时， pointer 类型变量不能用于函数）
4	当 n = 2 时的变量，另外加上 long , unsigned long , float , double 和 long double 型变量以及用于函数的 pointer 类型变量（当使用 bank 函数 (-MF) 时）
M	结构体、共用体以及数组
当忽略时	所有变量

- 不管以上规范如何，以关键字 **sreg** 声明的变量必定会分配到 **saddr** 区域。
- 以上规则还适用于 **extern** 声明引用的变量，则处理过程内容相同，效果和这些变量被分配到 **saddr** 区域一样。
- 通过此选项分配到 **saddr** 区域的变量，和 **sreg** 变量的处理方法相同。这些变量的函数和限制在 (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。

表 11-8 通过-RS 选项分配到 `saddr` 区域的变量

n 的值	分配到 <code>saddr</code> 区域的变量
1	<code>char</code> 和 <code>unsigned char</code> 型变量
2	当 <code>n = 1</code> 时的变量，另外加上 <code>short</code> , <code>unsigned short</code> , <code>int</code> , <code>unsigned int</code> , <code>enum</code> 和 <code>pointer</code> 型变量（当使用 <code>bank</code> 函数（-MF）时， <code>pointer</code> 类型变量不能用于函数）
4	当 <code>n = 2</code> 时的变量，另外加上 <code>long</code> , <code>unsigned long</code> , <code>float</code> , <code>double</code> 和 <code>long double</code> 型变量以及用于函数的 <code>pointer</code> 类型变量（当使用 <code>bank</code> 函数（-MF）时）
M	结构体、共用体以及数组
忽略时	所有变量（只有这种情况下才会包括结构体、共用体以及数组）

- 不管以上规范如何，以关键字 `sreg` 声明的变量总会被分配到 `saddr` 区域。
- 通过此选项分配到 `saddr` 区域的变量其处理方法和 `sreg` 变量相同。这些变量的函数和限制如（1）所述。

规范方法

指定-RS `[n][M]` (`n`: 1, 2 或 4) 选项。

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

(d) 参数/自动变量的 saddr 自动分配选项的使用方法 (-RK)**功能**

- 参数和自动变量（除 **const** 型）自动分配到 **saddr 区域**，而不管是否有 **sreg** 声明。
- 取决于 n 的值以及是否指定 M，可以分配到 **saddr 区域** 的参数和自动变量类型的具体分配情况如下所示。

表 11-9 通过-RK 选项分配到 saddr 区域的变量

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 pointer 型变量（当使用 bank 函数 (-MF) 时， pointer 类型变量不能用于函数）
4	当 n = 2 时的变量，另外加上 long , unsigned long , float , double 和 long double 型变量以及用于函数的 pointer 类型变量（当使用 bank 函数 (-MF) 时）
M	结构体、共用体以及数组
忽略时	所有变量

- 不管以上规范如何，以关键字 **sreg** 声明的变量总会被分配到 **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（1）所述。

使用方法

- 指定 **-RK [n] [M]** 选项（其中 n 为 1、2 或 4）。
- 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。

限制

- 仅支持静态模式。当未指定 **-SM** 选项时，输出警告消息，自动分配无法进行。
- 已声明为寄存器变量的参数/变量不分配到 **saddr 区域**。
- 当同时还指定 **-QV** 选项时，优先分配到寄存器 **DE**。

示例**<C 源代码>**

```
sub(int hsmarg)
{
    int hsmauto;
    hsmauto = hsmarg;
}
```

<编译器的输出目标>

```
@@DATS      DSEG  SADDRP
?L0003: DS   (2)
?L0004: DS   (2)
@@CODE      CSEG
_sub:
    movw    ?L0003, ax
    movw    ?L0004, ax    ; hsmauto
    ret
```

(4) 如何使用 sfr 区域 (sfr)

功能

- **sfr** 指的是一组特殊函数寄存器，比如 78K/0 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 FF00H 至 FFFFH 组成的范围内。

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，也没有引用变量或函数
- 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 __K0S__
    #pragma sfr
#endif

void main()
{
    P0 -= RXB00;
    /* RXB00 = 10;          ==> error */
}

```

不输出有关声明的代码，且会有以下代码在函数中输出。

<编译器的输出目标>

```

mov    a, P0
sub    a, RXB00
mov    P0, a

```

兼容性

<从另一种 C 编译器至本 C 编译器>

- C 源程序中，和设备或编译器无关的部分无需修改。

<从本 C 编译器至另一种 C 编译器>

- 删除 `#pragma sfr` 声明或通过 `#ifndef` 排序以及添加原来为 `sfr` 变量的变量声明。示例如下所示。

```

#ifdef __K0S__
    #pragma sfr
#else
    /* Declaration of variables */
    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。

(5) noauto 函数 (noauto)**功能**

- **noauto** 函数对自动变量加以限制，不输出预处理/后处理代码（生成堆栈帧）。
- 所有参数分配到寄存器或 **saddr** 区域（FEDCH 至 FEDFH）用作寄存器变量。如果有参数不能分配到寄存器，则会出现编译错误。
- 只有在参数分配完成之后，所有的自动变量都能够分配到寄存器或 **saddr** 区域内时，才可以使用自动变量，用作寄存器变量。
- **noauto** 函数将参数分配到 **sadd** 区以当作寄存器变量使用，使用 **noauto** 必须在编译期间指定 **-QR** 选项时。
- 除分配到寄存器的参数之外的参数会被 **noauto** 函数存储在 **saddr** 区域用作寄存器变量，且将参数按升序存储（请参阅附录 A **saddr** 区域标签列表）。
- 调用 **noauto** 函数时输出的代码和调用普通函数的输出代码相同。
- 当指定 **-SM** 选项时，警告消息仅输出首次描述 **noauto** 的行号，然后所有的 **noauto** 函数按普通函数进行处理。

效果

- 可以缩短目标代码，并改善运行速度。

使用方法

在函数声明时，在函数前加上 **noauto** 关键字就为函数加上了 **noauto** 属性，如下所示。

```
noauto 类型-名称 函数-名称
```

限制

- 当指定 **-ZA** 选项时，**noauto** 函数被禁止。
- **noauto** 函数的参数和自动变量的类型和数量有一定限制。以下展示 **noauto** 函数内可以使用的参数类型。除 **long/signed long/unsigned long**，**float/double/long double** 之外的参数被分配到寄存器 **HL**。

- **Pointer**
- **char/signed char/unsigned char**
- **int/signed int/unsigned int**
- **short/signed short/unsigned short**
- **long/signed long/unsigned long**
- **float/double/long double**

- 可以使用的参数和自动变量的总大小最多 6 个字节。

- 编译时会对这些限制进行检查。
- 如果用 **register** 声明参数，则忽略 **register** 声明。

示例

当指定**-QR**选项时

< C 源代码 >

```
noauto short nfunc(short a, short b, short c) ;
short l, m;
void main()
{
    static short ii, jj, kk;
    l = nfunc(ii, jj, kk) ;
}
noauto short nfunc(short a, short b, short c)
{
    m = a + b + c;
    return(m) ;
}
```

< 编译器的输出目标 >

```
@@CODE CSEG
_main:
;line      5: static short ii, jj, kk;
;line      6: l = nfunc(ii, jj, kk) ;
    mov     a, !?L0005                ; kk
    push   ax
    mov     a, !?L0004                ; jj
    push   ax
    mov     a, !?L0003                ; ii
    call   !_nfunc                    ; 调用 nfunc (a, b, c) 函数
    pop    ax
    pop    ax
    movw   ax, bc
    mov    !_l, ax                    ; 将返回值赋给外部变量 l
;line      7: }
    ret
;line      8: noauto short nfunc (short a, short b, short c)
;line      9: {
    _nfunc:
    push   hl                          ; 保存 HL
    xch    a, x
    xch    a, @_KREG12                 ; 将参数 a 设置为 @KREG12
    xch    a, x
    xch    a, @_KREG13                 ;
    push   ax                          ; 保存 @KREG12
    movw   ax, @_KREG14                ;
```



```

push    ax                ; 保存_@KREG14
movw   ax, sp            ;
movw   hl, ax            ;
mov    a,[hl+10]         ;
xch    a, x              ;
mov    a,[hl+11]         ;
movw   _@KREG14, ax      ; 将参数 c 分配到_@KREG14
mov    a,[hl+8]          ;
xch    a, x              ;
mov    a,[hl+9]          ;
movw   hl, ax            ; 将参数 b 分配到 HL
;line   10: m = a + b + c;
movw   ax, hl            ;
xch    a, x              ;
add    a, _@KREG12       ;
xch    a, x              ;
addc   a, _@KREG13       ;
xch    a, x              ;
add    a, _@KREG14       ;
xch    a, x              ;

addc   a, _@KREG15       ; 将 b(HL) 和 c(_@KREG14) 添加到 a(_@KREG12)
mov    !_m, ax           ; 将计算结果赋给外部变量 m
;line   11: return(m) ;
movw   bc, ax            ; 返回外部变量 m 的内容
pop    ax                ;
movw   _@KREG14, ax      ; 恢复_@KREG14
pop    ax                ;
movw   _@KREG12, ax      ; 恢复_@KREG12
pop    hl                ; 恢复 HL
ret

```

说明

- 在以上示例中，**noauto** 属性标志符加在 C 源代码的函数之前位置。声明 **noauto** 且不执行堆栈帧格式。

兼容性

<从另一种 C 编译器至本 C 编译器>

- 如果不使用关键字 **noauto**，则无需修改 C 源程序。
- 要将变量改为 **noauto** 变量，请根据以上**使用方法**中描述的程序修改程序。

<从本 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 源代码的修改](#)。

(6) norec 函数 (norec)

功能

- 如果函数自身不调用另一函数，可以改为 **norec** 函数。
- 通过 **norec** 函数，不输出预处理和后处理（栈帧格式）代码。
- **norec** 函数的参数分配到寄存器和 **saddr 区域**（FEC0H 至 FEC7H）。
- 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**（FEC0H 至 FEC7H）中，并调用 **norec** 函数。
- 自动变量分配到 **saddr 区域**（FEC8H 至 FECFH），寄存器变量同样处理。
- 当编译期间指定 **-QR** 选项时，不能分配到 **saddr 区域**。
- 如果使用 **long/float/double/long double** 型之外的参数，则第一个参数存储在寄存器 **AX** 中，第二个参数存储在寄存器 **DE**，第三个参数及随后的参数存储在 **saddr 区域**。请注意，不管参数的类型如何，寄存器 **AX** 和 **DE** 中只能存储一个参数。
- 如果在 **norec** 函数开始时，寄存器 **DE** 中没有存储 **norec** 传递的参数，则将存储在寄存器 **AX** 中的参数复制到寄存器 **DE**。如果寄存器 **DE** 中已经存储了参数，则存储在 **AX** 中的参数被复制到 **_@RTARG6** 和 **_@RTARG7**。
- 如果使用了 **long/float/double/long double** 型之外的自动变量，则分配之后剩余的参数按声明的顺序存储到 **DE**，**_@RTARG6** 和 **_@RTARG7**，**_@NRARG0**，**_@NRARG1...**
如果使用 **long/float/double/long double** 型的自动变量，则分配之后剩余的参数按声明的顺序存储到 **_@NRARG0**，**1...**
剩余参数按声明的顺序存储在 **saddr 区域**（请参阅附录 A **saddr 区域标签列表**）。

效果

- 可以缩短目标代码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**。
- 当指定-**SM** 选项时，警告消息仅输出首次描述 **norec** 的行号，然后所有的 **norec** 函数按普通函数进行处理。
- 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** 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** 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** 之外的类型，则自动变量可以使用最多 4 个字节。参数不能用 **long/float/double/long double** 型。

(当指定-**QR** 选项时)

- 如果使用 **long/float/double/long double** 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** 型，则为 2 个变量。
- 由使用的各种数据类型决定的总字节数大小以及 **saddr** 区域保留字节数量，自动变量可以使用其中参数未使用的剩余空间，也可以使用 **saddr** 区域保留但未使用的剩余空间。如果使用 **long/float/double/long double** 之外的类型，则自动变量可以使用最多 20 个字节，如果使用 **long/float/double/long double** 型，则自动变量可以使用最多 16 个字节。
- 编译时会检验这些限制，且如果不满足这些限制条件，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    _@NRARG1          ;
EXTRN    _@NRARG6          ;
:
_@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

mov     c, #02H
xch    a, x
add    a, a
xch    a, x
rolc   a, 1
dbnz   c, $$-5
xch    a, x
add    a, _@NRARG1        ; 使用 saddr 区域的自动变量
xch    a, x
addc   a, _@NRARG1+1      ; 使用 saddr 区域的自动变量
movw   bc, ax
ret

```

说明

在以上示例中，还为在 **rout** 函数的定义添加了 **norec** 属性，以声明该函数为 **norec** 函数。

兼容性

<从另一种 C 编译器至本 C 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。

<从本 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 源代码的修改](#)。

(7) 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** (溢位) 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（除非是静态模式）。
- 仅使用 **bit** 型变量，每个加载模块最多可以使用 1472 个变量（当使用 **saddr** 区域[FE20H 至 FEB7H]时）（正常模式）。
- 仅使用 **bit** 型变量，每个加载模块最多可以使用 1664 个变量（当使用 **saddr** 区域[FE20H 至 FECFH]时）（静态模式）。
- 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。
- 如表 11-11 所示，仅可以用 0 和 1 进行操作符和常数的操作。
- *, &（指针引用、地址引用）以及 **sizeof** 都不能执行。
- 当指定 **-ZA** 选项时，仅启用 **__boolean**。

表 11-10 仅使用常数 0 或 1 的操作符（通过 Bit 型变量）

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

备注 如果使用 **sreg** 变量，或指定了 **-RD**、**-RS** 和 **-RK**（**saddr** 自动分配选项）选项，则可用的 **bit** 型变量数量减少。

示例

< C 源代码 >

```

#define ON 1
#define OFF 0

extern bit data1;
extern bit data2;

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

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

```

此示例中的内容是当用户为 **bit** 型变量编写了定义代码的情况。如果未添加 **extern** 声明，则在编译器输出以下代码。此时不输出 **ORG** 伪指令。

< 汇编源程序 >

```

PUBLIC    _data1                ; 声明
PUBLIC    _data2

@@BITS   BSEG                  ; 分配到字段
          ORG    0FE20H

_data1    DBIT
_data2    DBIT

```

函数中输出以下代码。

< 编译器的输出目标 >

```

set1      _data1                ; 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 编译器至本 C 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。

<从本 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。）。

(8) ASM 语句 (#asm, #endasm/ __asm)**功能****(a) #asm - #endasm**

- 通过使用预处理指令 **#asm** 和 **#endasm** 可以将用户编写的汇编源代码程序嵌入由本 C 编译器输出的汇编源代码文件中。
- **#asm** 和 **#endasm** 行不作输出。

(b) __asm

- 汇编指令将汇编代码描述输出为字符串文字，并嵌入汇编源文件。

效果

- C 源代码的全局变量可以在汇编源代码中操作。
- 可以补充某些无法以 C 源代码来完成的函数。
- 由 C 编译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**(a) #asm - #endasm**

- 以 **#asm** 指令指示汇编源代码开始且以 **#endasm** 指令指示汇编源代码结束。在 **#asm** 与 **#endasm** 之间进行汇编源代码的描述。

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

(b) __asm

- 如需使用 **__asm**，要在 **ASM** 语句出现的模块开始处用 **#pragma asm** 规范进行声明（**#pragma** 以后的关键字区分大写字母和小写字母）。
- 以下各项可以在 **#pragma asm** 之前进行说明。
 - 注释
 - 其他 **#pragma** 指令
 - 既未定义又未引用变量或函数的预处理指令
- **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 规范，且使用转义字符（\n: 换行，\t: 制表键）或 **¥** 可以使一行连续，也可以连接字符串。

限制

- **#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** 的 **CSEG**。

示例**(a) #asm - #endasm****<C 源代码>**

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

由用户编写的汇编源代码输出到汇编源代码文件。

<编译器的输出对象>

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

说明

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

(b) `__asm`

<C 源代码>

```
#pragma asm

int a, b;

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

<汇编源程序>

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

兼容性

- 如果 C 编译器支持 `#asm`，可以根据 C 编译器指定的格式对程序进行修改。
- 如果目标设备不同，则修改程序的汇编源代码部分。

(9) 中断函数 (#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) 寄存器变量的 **saddr** 区域
- (3) **norec** 函数参数/自动变量的 **saddr** 区域（不管使用参数还是变量）
- (4) 运行时刻库的 **saddr** 区域（仅正常模式）

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

- 如果指定“不改变”，则不管是否使用这些代码，改变寄存器组的代码或者对 **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。

(正常模式)

- 如果编译时未指定 **-QR** 选项，则 **norec** 函数中寄存器变量的 **saddr** 区域和参数/自动变量的 **saddr** 区域不使用；因此，保存/恢复的代码不作输出。
如果保存代码的大小小于恢复代码的大小，则输出恢复代码。
- 表 11-12 总结了以上内容并展示了和保存/恢复区。

表 11-11 当使用中断函数时保存/恢复区域

保存/恢复区	无 BANK	调用函数				不调用函数			
		未指定 -QR		已指定 -QR		未指定 -QR		已指定 -QR	
		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
使用的运行时刻库的 saddr 区域	NG	NG	NG	NG	NG	OK	OK	OK	OK
所有运行时刻库的 saddr 区域	NG	OK	OK	OK	OK	NG	NG	NG	NG
使用的寄存器变量的 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

(静态模式)

- 当编译期间指定 **-SM** 选项时，寄存器变量的 **saddr** 区域、自动变量或 **norec** 函数参数的 **saddr** 区域和运行时刻库的 **saddr** 区域都不使用，所以保存和恢复代码区域的情况如下所示。

表 11-12 当使用中断函数时保存/恢复区域（静态模式）

保存/恢复区域	无 BANK	调用函数		不调用函数	
		堆栈	RBn	堆栈	RBn
使用的寄存器	NG	NG	NG	OK	NG
所有寄存器	NG	OK	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

但是，当指定的 **leafwork 1 至 16** 时，在堆栈中保存和恢复 **leafwork** 指定字节数量的代码，在中断函数的开始和结束处从共享区的高端地址输出（请参阅 **11.5 (29)** 当未指定 **-ZM** 选项时的静态模式，以及 **11.5 (45)** 当指定 **-ZM** 选项时的**静态模式扩展规范**）。

注意事项 如果中断函数中存在 **ASM** 语句，且在此 **ASM** 语句中使用了为编译器的寄存器所保留的区域，则该区域的内容必须由用户自行保存。

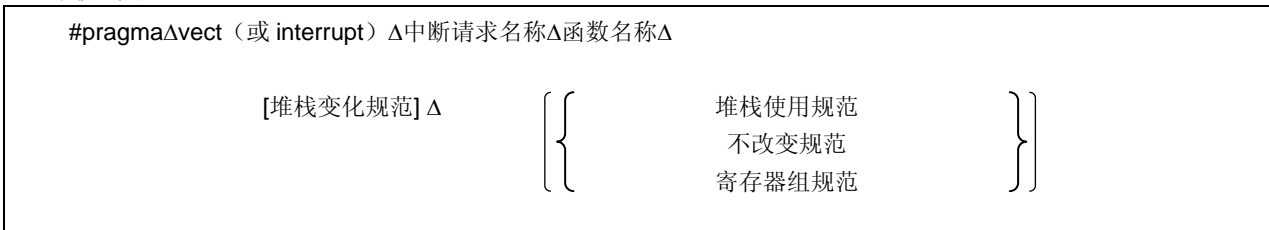
效果

- 可以用 **C** 源代码进行中断函数的描述。
- 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。

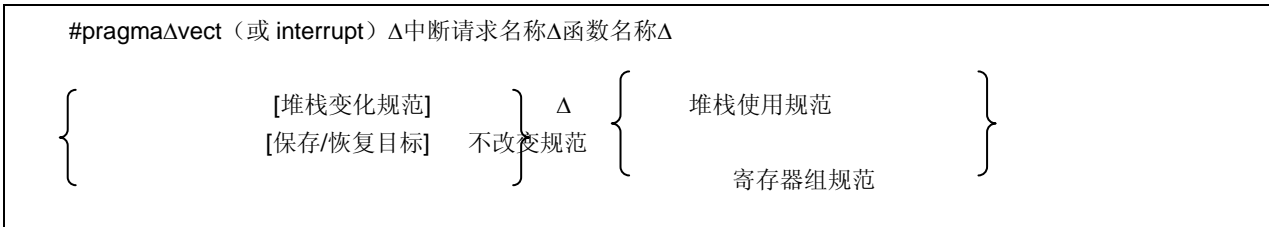
使用方法

- 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。
 - 注释语句
 - 既未定义又未引用变量或函数的预处理指令

<正常模式时>



<静态模式时>



<p>中断请求名称:</p> <p>函数名称:</p> <p>堆栈变化规范:</p>	<p>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）。</p> <p>描述中断过程的函数名称。</p> <p>SP = 数组名称 [+ 偏移位置]（示例：SP = buff + 10）。</p> <p>通过 unsigned char 定义数组（示例：无符号 char buff [10];）。</p>
--	--

堆栈使用规范:	STACK (默认)
不改变规范:	NOBANK
寄存器组规范:	RB0/ RB1/ RB2/ RB3
共享区 保存/恢复规范:	leafwork 1 至 16 (当指定 -SM 选项时)
保存/恢复目标:	SAVE_R 保存/恢复限制于寄存器的目标 SAVE_RN 保存/恢复限制于寄存器和_ @NRATxx 的目标 (当指定 -SM, -ZM 选项时)
Δ:	空格

限制

- 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回相互重叠。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”规范) 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/ callf/noauto/norec/ _callt/ _callf/ _leaf/ _rtos_interrupt/ _pascal/ _flash/ _flashf**/函数不能被指定为中断函数。
- 因为中断函数不能带参数或返回值，所以指定具有 **void** 型的中断函数 (示例: **void func (void) ;**)。
- 即使在中断函数中有 **ASM** 语句，保存所有寄存器和变量区域的代码也不作输出。因此，如果为编译器所保留的区在中断函数的 **ASM** 语句中被使用，或在 **ASM** 语句中进行了函数调用，则用户必须自行保存寄存器和变量区。
- 如果在未指定**-SM** 选项时指定 **leafwork** 为 **1 至 16**，则输出警告，且忽略共享区的保存/恢复规范。
- 当指定堆栈变化时，堆栈指针就被改为数组名称符号加偏移量的位置。数组名称的范围不通过**#pragma** 指令保留。数组需要作为全局 **unsigned char** 型数组单独定义。

- 改变堆栈指针的代码放于函数开始处，且设置堆栈指针返回的代码放在函数结束处。
- 当关键字 **sreg/ __sreg** 添加到数组用于堆栈变化时，假定定义了两个或两个以上具有不同属性和相同名称的变量，且出现编译错误。可能通过 **-RD** 选项将其中一个数组分配于 **saddr** 区域中，但无法改善代码大小和执行效率，因为数组被用作堆栈。不建议将 **saddr** 区域作为堆栈使用。
- 在“不改变”的情况下不能同时指定堆栈变化。如果这样指定，则会出现错误。
- 堆栈改变必须在堆栈使用规范之前说明。如果在堆栈使用规范之后说明堆栈改变，则出现错误。
- 如果某函数被指定为“不改变”、不在同一模块中定义的寄存器组或堆栈就被作为 **#pragma vect/ #pragma interrupt** 规范的保存目的地，则输出警告消息且忽略堆栈变化。在此情况下，使用默认堆栈。

示例

- 指定了寄存器组时

< 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 >

```
@@CODE CSEG
_inter :
    ; 切换寄存器组的代码
    ; 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
    reti
@@VECT02 CSEG AT 02H          ; NMI
_@vect02 :
    DW _inter
```

- 指定堆栈和寄存器组的改变

< C 源代码 2>

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

<编译器输出对象>

```
@CODE CSEG
_inter :
    sel RB2                ; 切换寄存器组
    push ax                ; 改变堆栈指针
    movw ax , sp           ; "
    movw sp , #_buff + 10 ; "
    push ax                ; "
    movw ax , @_RTARG0     ; 保存编译器所用的 saddr 区域
    push ax                ; "
    movw ax , @_RTARG2     ; "
    push ax                ; "
    movw ax , @_RTARG4     ; "
    push ax                ; "
    movw ax , @_RTARG6     ; "
    push ax                ; "
    call !_func
    pop ax                 ; 恢复编译器所用的 saddr 区域
    movw @_RTARG6         ; "
    pop ax                ; "
    movw @_RTARG4         ; "
    pop ax                ; "
    movw @_RTARG2         ; "
    pop ax                ; "
    movw @_RTARG0         ; "
    pop ax                 ; 将堆栈指针返回原来的位置
    movw sp , ax          ; "
    pop ax                ; "
    reti
@@VECT06 CSEG AT 0006H
_@vect06 :
DW _inter
```

- 指定共享区域保存/恢复（仅静态模式）

< C 源代码 3 >

```
#pragma interrupt INTP0 inter leafwork4
void func();
void inter()
{
    func();
}
```

<编译器输出对象>

```
EXTRN  _@KREG12
EXTRN  _@KREG14

@@CODE CSEG
_inter:
    push    ax                ;保存寄存器
    push    bc                ;"
    push    hl                ;"
    movw    ax, _@KREG12      ;保存共享区
    push    ax                ;"
    movw    ax, _@KREG14      ;"
    push    ax                ;"
    call    !_func
    pop     ax                ;恢复共享区
    movw    _@KREG14, ax      ;"
    pop     ax                ;"
    movw    _@KREG12, ax      ;"
    pop     hl                ;恢复寄存器
    pop     bc                ;"
    pop     ax                ;"
    reti

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

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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范来改变程序。

(10) 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）中断函数，并将寄存器和变量区（1）以及（4）以下保存/恢复到堆栈，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。

- | |
|--|
| <p>(1) 寄存器</p> <p>(2) 寄存器变量所用的 <code>saddr</code> 区域</p> <p>(3) <code>norec</code> 函数的参数/自动变量的 <code>saddr</code> 区域（不管是否使用）</p> <p>(4) 运行时刻库的 <code>saddr</code> 区域</p> |
|--|

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。如果编译时指定 `-SM` 选项，则不输出保存/恢复代码，因为不使用（2），（3）和（4）区。

效果

- 通过此修饰词来声明函数，向量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

- 将 `__ interrupt` 描述为中断函数的修饰词。

(对于不可屏蔽/可屏蔽中断函数)

<code>__ interrupt void func() {processing}</code>
--

(对于软件中断函数)

<code>__ interrupt_brk void func() {processing}</code>
--

限制

- `callt/ callf/ noauto/ norec/ __ callt/ __ callf/ __ leaf/ __ rtos_interrupt/ __ pascal/ __ 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**）规范，即使未加此修饰词，由**#pragma vect**（或 **interrupt**）指定的函数名称也会被判定为中断函数（如需**#pragma vect/interrupt** 的详细信息，请参阅 [11.5 \(9\) 中断函数](#)）。

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 编译器至本 C 编译器>

- 除非支持中断函数，否则无需修改 C 源程序。
- 若需要，请根据以上**使用方法**中描述的过程对中断函数进行修改。

<从本 C 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范修改程序。

(11) 中断函数 (#pragma DI, #pragma EI)

功能

- **DI** 和 **EI** 代码输出到对象，并创建一个目标文件。
- 如果忘记了 **#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 __KOS__
#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 2 >

```
#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 编译器至本 C 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。

<从本 C 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 __K0S__ ... #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范修改程序。

(12) 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

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

使用方法

- 在 C 源代码开始处说明 **#pragma HALT**，**#pragma STOP**、**#pragma NOP** 和 **#pragma BRK** 指令。
- 以下各项可以在 **#pragma** 指令之前说明。
 - (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 编译器至本 C 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。

<从本 C 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范修改程序（诸如 **#asm**，**#endasm** 和 **asm();**）。

(13) callf 函数 (callf/__callf)

功能

- **callf** 指令会将函数体存储在称作 **callf** 表区域，这样可以得到比普通 **call** 指令更短的代码。
- 如果要调用 **callf** 区域中的函数却没有函数的原型声明，该函数只能通过普通的 **call** 指令来调用。
- 被调用的函数（被别的函数调用）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 **callf/__callf** 属性，方法如下所示。

<pre>callf extern 类型名称 函数名称 __callf extern 类型名称 函数名称</pre>
--

限制

- 用 **callf/__callf** 关键字声明的函数分配在 **callf** 入口区域内，每个函数在该区域内的具体地址将会在链接目标模块时决定。因此，在汇编源代码模块中使用 **callf** 函数时，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 **callf** 函数的数量。
- **callf** 入口区域的范围为 800H 至 FFFH。
- 可以声明为 **callf** 属性的函数数量没有限制。
- 指定 **-ZA** 选项时，**__callf** 函数可用。
- 指定 **-ZA** 选项时，**callf** 函数不可用，如果有此定义，则发生错误。

示例

<pre>< C source 1 > __callf extern int fsub () ; void main () { int ret_val ; ret_val = fsub () ; }</pre>	<pre>< C source 2 > __callf int fsub () { int val ; return val ; }</pre>
---	---

<编译器的输出目标>

<pre>< C source 1 > EXTRN _fsub ; Declaration callf !_fsub ; Call < C source 2 > (将会被分配到 callf 入口区域) PUBLIC _fsub ; Declaration @@CALF CSEG FIXED _fsub : ; Function definition : ; function body :</pre>
--

兼容性

<从另一种 C 编译器至该 C 编译器>

- 如果不使用关键字 **callf**/**__callf**，则不需要修改 C 源程序。
- 要将函数改变成 **callf** 属性函数，请注意以上**使用方法**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** 来允许将 **callf** 函数被当做普通函数来处理。

(14) 绝对地址访问函数 (#pragma access)

功能

- 访问普通 RAM 空间的代码输出到目标，通过直接内联扩展而不是通过函数调用方式，且可以创建目标文件。
- 如果未说明 **#pragma** 指令，则访问绝对地址的函数被视为普通函数。

效果

- 普通存储空间中的特定地址可以很方便的地通过 C 语言描述来访问。

使用方法

- 在 C 源代码开始处说明 **#pragma access** 指令。
- 在源文件中说明该指令的方式和函数调用的格式相同。
- 以下各项可以在 **#pragma access** 之前说明。
 - (i) 注释语句
 - (ii) 其他 **#pragma** 指令
 - (iii) 既未定义又未引用变量或函数的预处理指令
- **#pragma** 以后的关键字可以用大写或小写字母表示。

以下四个函数名称在绝对地址访问过程中可用。

peekb, peekw, pokeb, pokew

[绝对地址访问函数表]

(a) **unsigned char peekb (addr);**
unsigned int addr;

返回 1 个字节地址 **addr** 的内容。

(b) **unsigned int peekw (addr);**
unsigned int addr;

返回 2 个字节地址 **addr** 的内容。

(c) **void pokeb (addr, data);**
unsigned int addr;
unsigned char data;

将 1 个字节的数据内容写入由地址 **addr** 指示的位置。

(d) **void pokew (addr, data);**
unsigned int addr;
unsigned int data;

将 2 个字节的数据内容写入由地址 **addr** 指示的位置。

限制

- 不要随便使用绝对地址访问的函数名称。
- 以小写字母表示绝对地址访问的函数。以大写字母表示的函数按普通函数进行处理。

示例**< C 源代码 >**

```
#pragma access

char a;
int b;

void main ( )
{
    a = peekb (0x1234) ;
    a = peekb (0xfe23) ;
    b = peekw (0x1256) ;
    b = peekw (0xfe68) ;

    pokeb (0x1234, 5) ;
    pokeb (0xfe23, 5) ;
    pokew (0x1256, 7) ;
    pokew (0xfe68, 7) ;
}
```

<输出汇编源程序 >

```

    . . .
    . . .
    mov     a, !01234H
    mov     !_a, a
    mov     a, 0FE23H
    mov     !_a, a
    movw   ax, !01256H
    movw   !_b, ax
    movw   ax, 0FE68H
    movw   !_b, ax

    mov     a, #05H
    mov     !01234H, a
    mov     0FE23H, #05H
    movw   ax, #07H
    movw   !01256H, ax
    movw   0FE68H, #07H
```

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用绝对地址访问的函数，则无需修改源程序。
- 如果使用绝对地址访问的函数，则根据以上**使用方法**描述的过程对程序进行修改。

<从此编译器至另外某个 C 编译器>

- 删除“**#pragma access**”声明或用**#ifdef** 进行屏蔽，绝对地址访问的函数名称可以用作函数名称。
- 要使用绝对地址访问的函数，请根据每个编译器的具体规范修改程序（**#asm, #endasm, asm** 等）。

(15) 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** 型位域不可跨字节界限进行分配，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

- 可以保存内存的内容，可以缩短目标代码且可以提高运行速度。

使用方法

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

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

示例

```
struct 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 编译器至本 C 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。

<从本 C 编译器至另外某个 C 编译器>

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

(b) 位域的分配方向

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

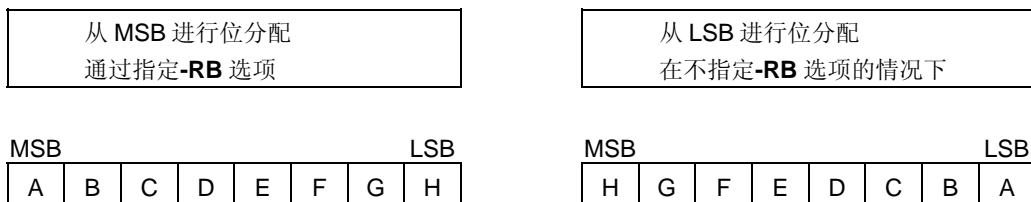
<位域声明>

```
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;
};
```

说明

因为 A 到 H 为 8 位或更少，所以可以其分配在同一个字节单元中。

图 11-3 通过位域声明的位分配（示例 1）



示例 2

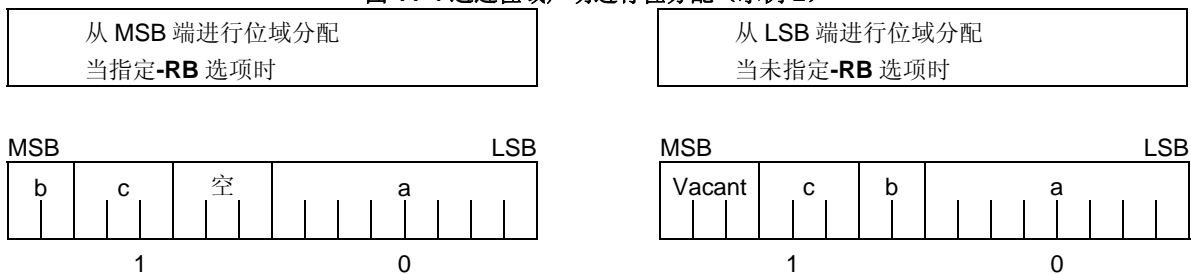
<位域声明>

```

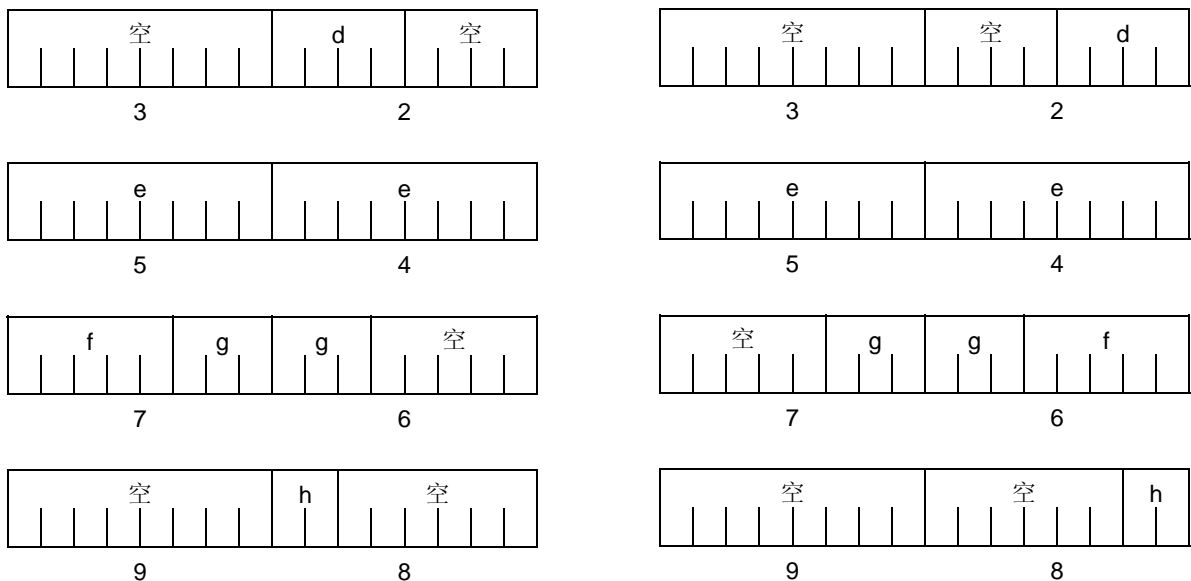
struct t {
    char          a;
    unsigned char b:2;
    unsigned char c:3;
    unsigned char d:4;
    int           e;
    unsigned char f:5;
    unsigned char g:6;
    unsigned char h:2;
    unsigned int  i:2;
};
    
```

说明

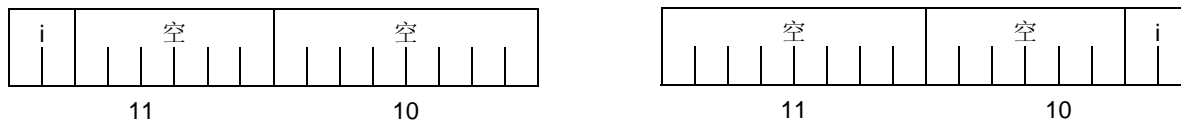
图 11-4 通过位域声明进行位分配（示例 2）



char 型的成员 a 分配到第一字节单元，unsigned char 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。



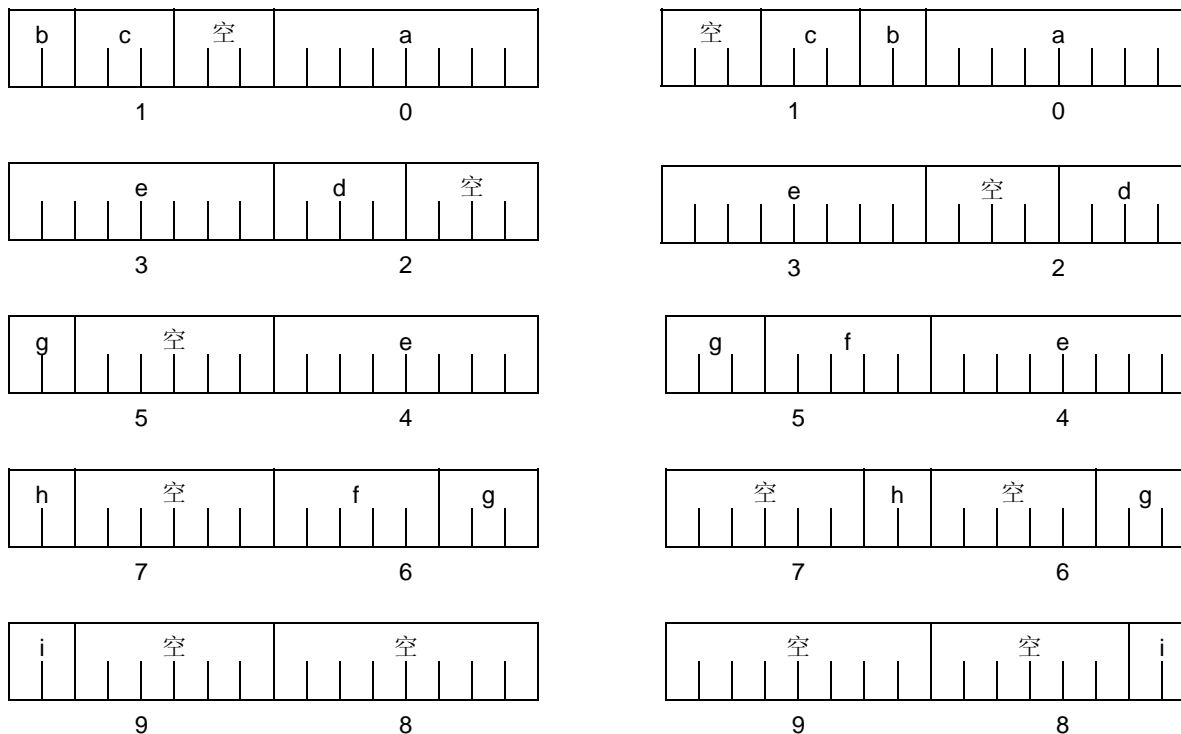
因为成员 **g** 为 **unsigned int** 型的位域，所以可以跨字节界限进行分配。因为 **h** 为 **unsigned char** 型的位域，所以其不能和 **unsigned int** 型的 **g** 位域分配在相同的字节单元中，而是分配在下一字节单元。



因为 **i** 为 **unsigned int** 型的位域，所以其分配在下一字节单元。

当指定 **-RC** 选项时（以封装结构成员），以上位域成为以下形式。

图 11-5 位域声明决定的位分配（指定了 **-RC** 选项）



备注 分配图下方的数字表示从该结构开始的字节偏移值。

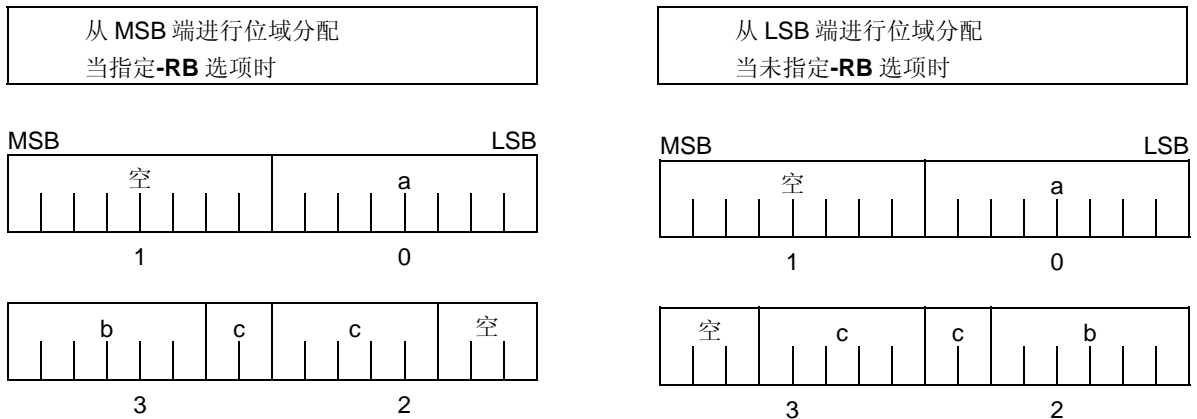
示例 3

<位域声明>

```

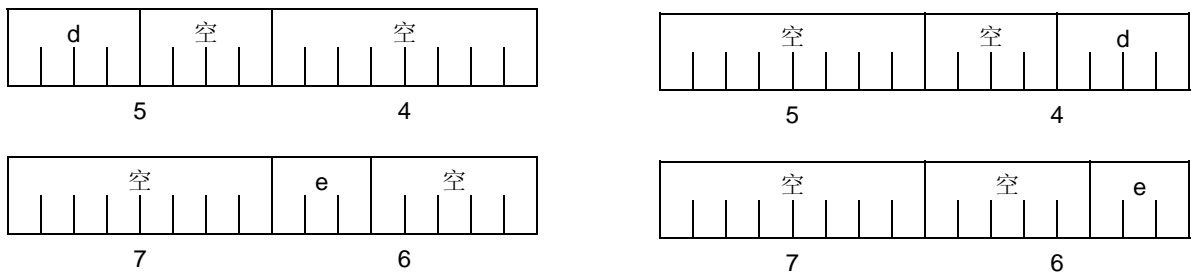
struct t {
    char          a;
    unsigned int  b:6;
    unsigned int  c:7;
    unsigned int  d:4;
    unsigned char e:3;
    unsigned char f:10;
    unsigned char g:2;
    unsigned char h:5;
    unsigned int  i:6;
};
    
```

图 11-6 根据位域声明分配位 (示例 3)

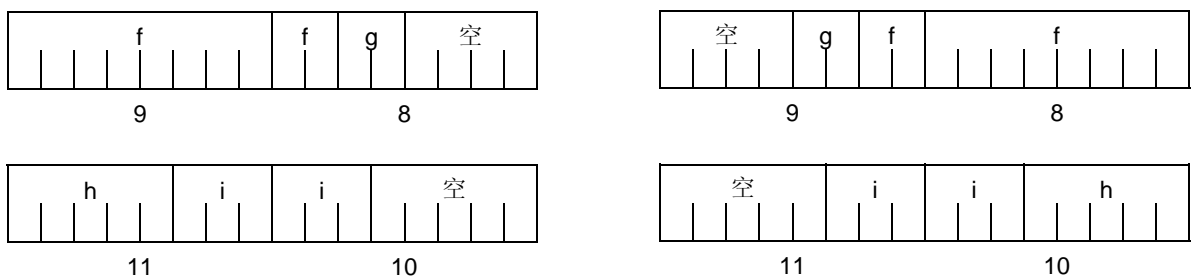


因为 b 和 c 为 **unsigned int** 型的位域，所以其从下一字单元分配。

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



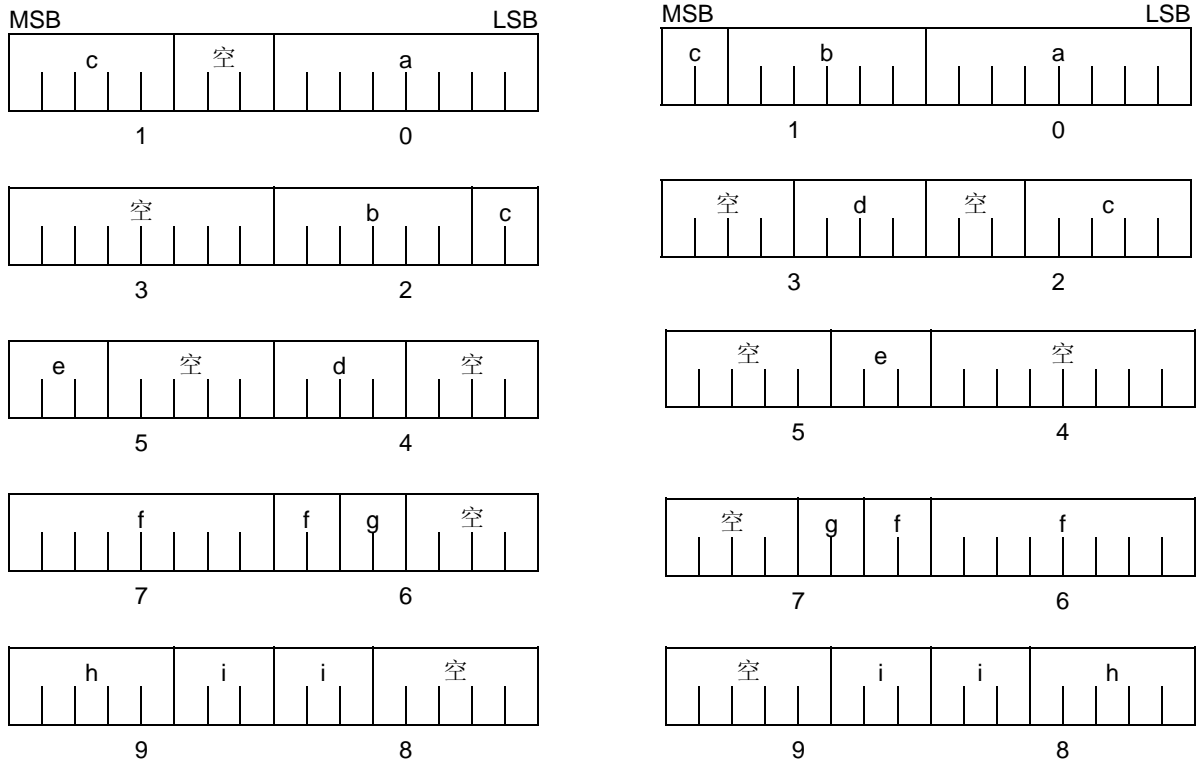
因为 e 为 **unsigned char** 型的位域，所以其分配到下一字节单元。



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。

图 11-7 根据位域声明分配位 (示例 3) (指定了-RB 选项)



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 编译器至本 C 编译器>

- 无需修改源程序。

<从本 C 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。

(16) 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 RA78K0S 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新定义，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范（字段名称最多可以使用八个字母）。
- 仅可以使用 C 语言的十六进制数和汇编程序的十六进制数来说明起始地址。

[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（请参阅示例 1 中的错误说明）。
- #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。（请参阅示例 2 中的错误说明）。
- 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（请参阅示例 3 中的错误说明）。

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#pragma 指令之后的主程序 C 代码

a3.h
#pragma section @@DATA ??DATA3 // 文件仅包含#pragma 区段。

a4.h
#pragma section @@DATA ??DATA3
extern int func2 (void) ; //包括主程序 C 代码的文件。

a.c
#include "a1.h"
#include "a2.h"
#include "a3.h" // ← 导致错误。
// 因为 a2.h 文件包含主程序 C 代码，其之后为此#pragma 指令，
// 不能包括文件 a3.h，其中只有此#pragma 指令。

#include "a4.h"

```

编码错误示例 2

```

b1.h
const int i;

b2.h
const int j;
#include "b1.h" // 这不会导致错误，因为这不是文件 (b.c)，b.c 中的
// 主程序 C 代码之后是此#pragma 指令。

b.c
const int k;
#pragma section @@DATA ??DATA1
#include "b2.h" // ← 导致错误
// 因为#include 语句随后不能在文件 (b.c) 之后出现，因为
// b.c 中主程序 C 代码之后是此#pragma 指令。

```

编码错误 示例 3

```

c1.h
extern int j;
#pragma section @@DATA ??DATA1 // 这不导致错误出现，因为在 c3.h 处理之前，包含并处理
// #pragma 指令。

c2.h
extern int k;
#pragma section @@DATA ??DATA2 // ← 导致错误。
// 此#include 语句在 c3.h 的主程序 C 代码之后指定
// 且此后不能指定#pragma 指令。

c3.h
#include "c1.h"
extern int i;
#include "c2.h"
#pragma section @@DATA ??DATA3 // ← 导致错误。
// 此#include 语句在主程序 C 代码之后指定，
// 且此后不能指定#pragma 指令。

c.c
#include "c3.h"
#pragma section @@DATA ??DATA4 // ← 导致错误。
// 此#include 语句在 c3.h 的主程序 C 代码之后指定
// 且此后不能指定#pragma 指令。

int i;

```

兼容性

<从另外某个 C 编译器至本 C 编译器>

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

<从本 C 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范修改程序。

限制

- 无法改变给出了向量表相关的字段（例如，**@@VECT02** 等）区段名称。
- 如果在于另一文件中出现具有相同名称的两个或两个以上区段，其中某个区段用 **AT** 指定了起始地址，则出现链接错误。
- 用来指示该区段用于 **bank** 函数的区段名称（**@@BANK1** 等）不能改变。
- 当改变编译器输出区段名称 **@@DATS**，**@@BITS** 和 **@@INIS** 时，指定地址的范围限制在 0FE20H 至 0FEB7H 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被和另一符号同名。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（*），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。

(*) **ROMization**-相关区段名称

```
@@R_INIT, @@R_INIS, @@INIT, @@INIS
```

要改变 **ROMization** 相关的区段，需要使用启动例程，下文将描述一个改变结束模块的示例。

[通过启动例程改变 ROMization 相关区段名称的示例]

下文为改变 **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**）和结束模块（**rom.asm**）的示例。

(C 源代码)

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

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

因此，如下所示，将以下两项添加到启动例程：新区段第一个标志声明和拷贝初值的部分，并将结束标签的声明添加到结束模块。

RTT1_S 和 **RTT1_E** 为区段 **RTT1** 的开始和结束标签名称，且 **TT1_S** 和 **TT1_E** 为区段 **TT1** 的开始和结束标签名称。

(a) 改变启动例程 `cstartx.asm` 的示例

<i> 添加标签的声明，指示改变名称的区段的结束

```

.
.
EXTRN  _main,_exit,@STBEG
EXTRN  _?R_INIT,_?R_INIS,_?DATA,_?DATS

EXTRN  RTT1_E,TT1_E    ← 添加 RTT1_E 和 TT1_E 的 EXTRN 声明
.
.

```

<ii> 添加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,#?DATS
    BZ      $LDATS2
    MOV     A,#0
    MOV     [HL],A
    INCW    HL
    BR      $LDATS1

LDATS2:
    MOVW    DE,#TT1_S
    MOVW    HL,#RTT1_S

LTT1:
    MOVW    AX,HL
    CMPW    AX,#RTT1_E    ; 添加的代码，用来将初值从已改变名称
    BZ      $LTT2          ; 的 RTT1 区段复制到 TT1 区段
    MOV     A,[HL]
    MOV     [DE],A
    INCW    HL
    INCW    DE
    BR      $LTT1

LTT2:
;
    CALL    !_main        ; main ();
    MOVW    AX,#0
    CALL    !_exit        ; exit (0);
    BR      $$
;

```

<iii> 设置已改变名称的区段开始标签

```

      .
      .
@@R_INIT CSEG UNITP
_@R_INIT :
@@R_INISCSEG UNITP
_@R_INIS :
@@INIT DSEG UNITP
_@INIT :
@@DATA DSEG UNITP
_@DATA :
@@INIS DSEGSADDRP
_@INIS :
@@DATS DSEGSADDRP
_@DATS :

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

@@CALTCSEGCALLTO
@@CALFCSEGFIXED
@@CNST CSEG UNITP
@@BITSBSEG

      END

```

(b) 改变结束模块 rom.asm 的示例

(i) 添加标签声明，表示已改变名称的区段的结束

```

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

PUBLIC    RTT1_E,TT1_E ← 添加 RTT1_E 和 TT1_E

;
@@R_INIT  CSEG      UNITP
_?R_INIT:
@@R_INIS  CSEG      UNITP
_?R_INIS:
@@INIT    DSEG      UNITP
_?INIT:
@@DATA    DSEG      UNITP
_?DATA:
@@INIS    DSEG      SADDRP
_?INIS:
@@DATS    DSEG      SADDRP
_?DATS
.
.
.

```

(ii) 设置指示结束的标签

```

.
.
.
RTT1      CSEG      UNITP ; 添加标签的设置指示 RTT1 区段结束。
RTT1_E:   ; 添加标签设置

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

;
END

```


(17) 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。以下展示二进制常量的使用方法。

0b 二进制数
0B 二进制数

备注 二进制数：'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。

(i)	下标二进制数:	int, unsigned int, long int
(ii)	下标 u 或 U:	unsigned long int unsigned int, unsigned long int
(iii)	下标 l 或 L:	long int unsigned long int
(iv)	下标 u 或 U 和下标 l 或 L:	unsigned long int

示例**<C 源代码>**

<pre>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;</pre>
--

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 无需修改。

<从本 C 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范。
- 如果编译器不支持二进制常量，则需要修改为其他整数格式，诸如八进制、十进制或十六进制。

(18) 模块名称改变函数 (#pragma name)**功能**

- 将指定模块名称的前八个字母输出到目标模块文件的符号信息表。
- 当指定了 **-G2** 选项，将指定的模块名称的前八个字母输出到汇编表文件作为符号信息 (**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九个或九个以上字母，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都会被启用，且输出警告消息。

效果

- 对象的模块名称可以改为任何名称。

使用方法

- 以下展示说明方法。

```
#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‘（’‘）’之外。区分大写和小写字母。

示例

```
#pragma name module1
.
.
```

兼容性

<从另外某个 C 编译器至本 C 编译器>

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

<从本 C 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范修改程序。

(19) 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。

备注 上述函数声明不受-ZI 选项效果。

- 通过模块的**#pragma rot** 指令声明循环移位函数的使用方法。
然而，以下各项可以在**#pragma rot** 之前说明。
 - (i) 注释
 - (ii) 其他**#pragma** 指令
 - (iii) 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    a,!_b
mov    c,a
mov    a,!_a
ror    a, 1
dbnz   c,$$-1
mov    !_c,a
```

限制

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

兼容性

<从另外某个 C 编译器至本 C 编译器>

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

<从本 C 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范（**#asm**，**#endasm** 或 **asm()**；等）修改程序。

(20) 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

- 乘法函数的调用通过由库的调用实现，如果目标设备不支持乘法指令。
- 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    a,!_b
mov    x,a
mov    a,!_a
mulu   x
movw   !_i, ax
```

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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范修改程序（**#asm**，**#endasm** 或 **asm()**；等）。

(21) 除法函数 (#pragma div)**功能**

- 输出从对象除以表达式值的代码，通过直接内联展开而不是函数调用，并生成对象文件。
- 如果没有对应的**#pragma** 指令，则乘法函数被视为普通函数。

效果

- 生成的代码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。以下两个函数名称可用于除法。

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。

备注 上述函数声明不受**-ZI** 选项影响。

- 通过模块化的**#pragma div** 指令声明除法函数的使用方法。然而，以下各项可以在**#pragma div** 之前说明。
 - (i) 注释
 - (ii) 其他**#pragma** 指令
 - (iii) 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

- 除法函数的调用通过由库的调用实现，如果目标设备不支持除法指令。
- 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    a,!_b
mov    c,a
movw   ax, !_a
divuw  c
movw   !_i, ax
mov    a, !_b
mov    c, a
movw   ax, !_a
divuw  c
mov    a, c
mov    !_c, a
```

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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范修改程序（**#asm**，**#endasm**或**asm()**；等）。

(22) BCD 运算函数 (#pragma bcd)

功能

- 输出对象中的表达式的值执行 **BCD** 操作的代码，通过直接内联展开而不是函数调用，并生成对象文件。但是，**bcdtob**, **btobcd**, **bcdtow**, **wtobcd** 和 **bbcd** 函数没有对应的内联形式。
- 如果没有对应的 **#pragma** 指令，则 **BCD** 操作函数被视为普通函数。

效果

- 即使未对 **BCD** 运算的过程进行描述，**BCD** 运算函数也可以通过 **C** 源代码或 **ASM** 语句来实现。

使用方法

- 在源文件中描述的方法和函数调用的格式相同。存在 **BCD** 运算的 13 类函数名称，如下列出。如需更多信息，请参阅本章稍后介绍的 [BCD 运算函数表](#)。

adbcd, sbbcd, adbcdb, sbbcdbe, adbcdbw, sbbcdw, adbcdbwe,
sbbcdwe, bcdtob, btobcde, bcdtow, wtobcd, btobcd

- 除法函数的用法通过模块的 **#pragma bcd** 指令声明。然而，以下各项可以在 **#pragma bcd** 之前编码。
 - 注释
 - 其他 **#pragma** 指令
 - 既未定义又未引用变量或函数的预处理指令
- 可以在 **#pragma** 之后用大写或小写字母作为关键字来说明。

限制

- **BCD** 运算函数名称不能用作函数名称。
- **BCD** 运算函数用小写字母编码。如果使用大写字母，则这些函数被视为普通函数。
- 静态模式不支持 **adbcdbwe** 和 **sbbcdwe**。

示例

<C 源代码>

```
#pragma bcd

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

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

<输出汇编源程序>

```
mov     a, !_a
add     a, !_b
adjba
mov     !_c, a
mov     a, !_b
sub     a, !_a
adjbs
mov     !_c, a
```

[BCD 运算函数表]

(1) **unsigned char adbcdb (x, y) ;**

unsigned char x ;

unsigned char y ;

通过 **BCD** 调整指令实现十进制加法运算。

(2) **unsigned char sbbcdb (x, y) ;**

unsigned char x ;

unsigned char y ;

通过 **BCD** 调整指令实现十进制减法运算。

(3) **unsigned int adbcdb (x, y) ;**

unsigned char x ;

unsigned char y ;

通过 **BCD** 调整指令实现十进制加法（带结果展开）运算。

(4) unsigned int sbbcdbe (x, y) ;**unsigned char x ;****unsigned char y ;**

通过 BCD 调整指令实现十进制减法（带结果展开）运算。如果出现借位，则更高的位设定为 0x99。

(5) unsigned int adbcdw (x, y) ;**unsigned int x ;****unsigned int y ;**

通过 BCD 调整指令实现十进制加法运算。

(6) unsigned int sbbcdw (x, y) ;**unsigned int x ;****unsigned int y ;**

通过 BCD 调整指令实现十进制减法运算。

(7) unsigned long adbcdwe (x, y) ;**unsigned int x ;****unsigned int y ;**

通过 BCD 调整指令实现十进制加法（带结果展开）运算。

(8) unsigned long sbbcdwe (x, y) ;**unsigned int x ;****unsigned int y ;**

通过 BCD 调整指令实现十进制减法（带结果展开）运算。如果出现借位，则更高的位设定为 0x9999。

(9) unsigned char bcdtob (x) .**unsigned char x ;**

十进制数转换成二进制数。

(10) unsigned int btobcde (x) ;**unsigned char x ;**

二进制数转换成十进制数。

(11) unsigned int bcdtow (x) ;**unsigned int x ;**

十进制数转换成二进制数。

(12) unsigned int wtobcd (x) ;**unsigned int x ;**

十进制数转换成二进制数。然而，如果 x 的值超过 10000，则返回 0xffff。

(13) unsigned char btobcd (x) ;**unsigned char x ;**

十进制数转换成二进制数。然而，舍弃溢位。

备注 上述函数声明不受-ZI 和-ZL 选项影响。**兼容性**

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用 **BCD** 运算函数，则无需修改。
- 要将另一函数改为 **BCD** 运算函数函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

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

(23) bank 函数

功能

- 使用函数信息文件指定选项-MF 来说明函数分配在 bank 区域还是 common 区域。
- 分配在 bank 区域的函数在调用时都是通过 bank 函数调用库实现的。
- 分配在 common 区域的函数其调用方式和普通函数调用方式相同。
- 当函数信息文件不含有源文件中的函数信息时（源文件中的函数被分配在 common 区域），使用函数信息文件指定选项-MF 向函数信息文件中添加相关信息。
- 静态（static）函数的调用方式和普通函数相同。
- 为了调用常量地址上的 bank 函数，使用__BANK0，__BANK1，...，__BANK15，这些函数都适合引用常量地址上的 bank 函数。

敬请参阅“[\(24\) 常量地址上的 Bank 函数](#)”。

- 所有待连接的源文件都应使用同一个函数信息文件。如果有某个输出文件使用的函数信息文件和别的文件连接时使用的函数信息文件不同，则在连接时会发生错误。
- 函数信息文件中指定的输出目标和未在函数信息文件中指定的目标相连接时，会发生错误。

效果

- 可以分配到代码块中的函数可以大于 64KB。

使用方法

- 使用函数信息文件指定选项-MF 为所有待连接的源文件指定函数信息文件。
- 如果指定的函数信息文件不存在，则会新建一个函数信息文件。
- 当指定的函数信息文件不含有某个源文件中的函数信息时，将相关函数信息添加到指定的函数信息文件中。该源文件中的函数被分配在 common 区域。
- 如果连接时发生分配错误，编辑函数信息文件将某些文件重新定位到 bank 区域。
- 只有分配的变化会反映到源文件。
- 关于如何编辑函数信息文件，请参阅“[CC78K0 C 编译器操作篇用户手册](#)”。

<函数信息文件>

```

/#0xxxx
// 78K/0 Series C Compiler Vx.xx 函数信息文件

file name := allocation (C...common area, 0-15...bank number) (code size)
{
function name1 ;
function name2 ;
}

// *** Code Size Information ***
// COMMON      : 指定分配到 common 区域的文件代码量字节总和
// BANK00      : 指定分配到 bank 0 区域的文件代码量字节总和
// BANK01      : 指定分配到 bank 1 区域的文件代码量字节总和

```

示例

- 使用函数信息文件指定选项-MF 指定函数信息文件来编译源文件。

<a.c>

```

extern    int func1 ( ) ;
extern    int func2 ( ) ;
int func3 ( ) ;
int a = 0 , b ;
void func ( )
{
    b = func1 ( a ) ;
    :
    b = func2 ( a ) ;
    :
    b = func3 ( a ) ;
}
int func3 ( int a )
{
    :
}

```

<b.c>

```

int func1 ( int a )
{
    :
}

```

<C.C>

```
int func2 ( int a )
{
:
}
```

将会创建一个函数信息文件，如下所示

<函数信息文件>

```
#0xxxx
// 78K/0 Series C Compiler Vx.xx Function Information File
a.c := C (3000)          <- a.c 文件被分配到 common 区域
{
func ;
func3 ;
}
b.c := C (1000)         <- b.c 文件被分配到 common 区域
{
func1 ;
}
c.c := C (2500)         <- c.c 文件被分配到 common 区域
{
func2 ;
}
// *** Code Size Information ***
// COMMON : 6500 byte
// BANK00 : 0 bytes
// BANK01 : 0 bytes
// BANK02 : 0 bytes
```


在初次创建的函数信息文件中，所有的文件都分配到 **common** 区域。

如果所有的文件无法全都分配到 **common** 区域，连接时会发生错误。通过参考函数信息文件中的输出代码量信息，可以编辑函数信息文件使得某些源文件重新分配到 **bank** 区域。

<函数信息文件>

```
 / #0xxxx
 // 78K/0 Series C Compiler Vx.xx Function Information File
 a.c := 0 (3000)                <- a.c 文件被分配到 bank 0 区域
 {
 func ;
 func3 ;
 }
 b.c := 1 (1000)               <- b.c 文件被分配到 bank 0 区域
 {
 func1 ;
 }
 c.c := C (2500)               <- c.c 文件被分配到 common 区域
 {
 func2 ;
 }
 // *** Code Size Information ***
 // COMMON : 6500 byte
 // BANK00 : 0 bytes
 // BANK01 : 0 bytes
 // BANK02 : 0 bytes
```

重定位的结果就是函数的代码量有所改变。

使用函数信息文件指定选项-MF 来指定被编辑的函数信息文件，并重新编译。

编译器输出目标，于是根据函数信息文件的内容每个文件被分配到 common 区域或 bank 区域中。

输出的代码如下所示。

<编译器的输出目标>

```
@@BANK0 CSEG BANK0
```

```
_func :
```

```
:
```

```
push hl
```

```
movw hl, #_func1
```

```
mov e, #BANKNUM _func1
```

```
callt [ @@bcall ]
```

```
pop hl
```

```
:
```

```
call !_func2
```

```
:
```

```
push hl
```

```
movw hl, #_func3
```

```
callt [ @@bcals ]
```

```
pop hl
```

```
:
```

```
_func3 :
```

```
:
```

```
@@BANK1 CSEG BANK1
```

```
_func1 :
```

```
:
```

```
@@CODE CSEG
```

```
_func2 :
```

```
:
```

下面是编译器调用的 `bank` 函数调用例程

```

@@CALT CSEG CALLT0
@@bcall : DW ?@bcall
@@bcals : DW ?@bcals
@@bcsub : DW ?@bcsub

@@LCODE CSEG
?@bcall :
    xch a , e
    xch a , BANK
    push ax
    mov a , e
    callt [ @@bcsub ]
    pop ax
    mov BANK , a
    ret

?@bcals :
    push de
    callt [ @@bcsub ]
    pop ax
    ret

?@bcsub :
    push hl
    ret

```

限制

- 只有函数体被分配到 `bank` 区域，数据无法分配到该区域。
- 同一个文件中的所有函数被分配到同一个 `bank` 中。
- `callt`, `callf`, `noauto`, `norec`, `__callt`, `__callf`, `__leaf`, `__interrupt`, `__interrupt_brk`, `__rtos_interrupt`, `__pascal`, `__flash`, 和 `__flashf` 不能被指定为 `bank` 函数。
- 如果指定了 `-MF` 选项，则 `-SM` 选项无效。
静态模式无法使用。
- 如果指定了 `-MF` 选项，则 `-ZR` 选项无效。
`Pascal` 函数接口无法使用。
- 在函数信息文件中只能编辑源文件的分配情况。
- 函数信息文件中的输出代码量不包括 `ASM` 语句的代码大小。

- 函数信息文件中不能添加注释。
- 当某个函数被分配到 **bank** 区域，调用该函数的 **block** 的输出代码以及被调用函数所在的 **block** 的输出代码都会改变。当某个源文件被重定位，在上述两个 **block** 中的文件都要重新编译。
- 当某个文件中的函数大小超过了 **bank** 区域的容量，该函数不能被成功分配到 **bank** 区域中。

如下函数不能被分配到 **bank** 区域。

- 启动例程
- 库
- 中断函数，RTOS 的中断函数，RTOS 的任务函数
- **callt** 和 **callf** 函数
- **noauto**, **norec** 和 **pascal** 函数

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 该 C 源程序无需修改。

<从本 C 编译器至另外某个 C 编译器>

- 该 C 源程序无需修改。

注意事项

- 当使用 **bank** 函数 (**-MF**) 时，函数指针大小为 4 字节。
- 分配在 **common** 区域的函数调用速度要快于分配在 **bank** 区域的函数调用。
- 分配在 **bank** 区域的函数调用速度较慢，是因为需要通过 **bank** 函数调用例程来实现调用过程。
- 同一个 **bank** 中的函数调用速度要快于跨 **bank** 进行的函数调用。
- 将不被调用的函数改为静态函数，静态函数的调用速度和 **common** 区域的函数调用速度同样快。

(24) 常量地址上的 bank 函数**功能**

- 创建的代码用来引用常量地址上的 **bank** 函数。
- 只有当设备支持 **bank** 函数是可用。

效果

- 可以调用常量地址上的 **bank** 函数

使用方法

- 格式和函数调用类型，**bank** 函数在源文件中使用大写字母描述。
- 因为常量地址上的 **bank** 函数的函数名称是 `__BANK0`, `__BANK1`, ..., 和 `__BANK15`。

[引用常量地址上的 bank 函数的函数]

```

unsigned long __BANK0 ( unsigned int addr );
unsigned long __BANK1 ( unsigned int addr );
unsigned long __BANK2 ( unsigned int addr );
unsigned long __BANK3 ( unsigned int addr );
unsigned long __BANK4 ( unsigned int addr );
unsigned long __BANK5 ( unsigned int addr );
unsigned long __BANK6 ( unsigned int addr );
unsigned long __BANK7 ( unsigned int addr );
unsigned long __BANK8 ( unsigned int addr );
unsigned long __BANK9 ( unsigned int addr );
unsigned long __BANK10 ( unsigned int addr );
unsigned long __BANK11 ( unsigned int addr );
unsigned long __BANK12 ( unsigned int addr );
unsigned long __BANK13 ( unsigned int addr );
unsigned long __BANK14 ( unsigned int addr );
unsigned long __BANK15 ( unsigned int addr );

```

`addr` 用来指定所获得的最低 2 个字节用来存放 2 字节常量的值；倒数第三个低字节获取 **bank** 编号；最高的字节获得常量 1，代表 **bank** 是函数，按照这个格式存入 4 字节数据。

只有常量可以用作参数 (`addr`) 描述，`addr` 是 CPU 地址。

限制

- 支持 **bank** 函数的设备不能使用函数名称来引用常量地址上的 **bank** 函数。
- 不支持 **bank** 函数的设备中，即使描述了某个函数来引用常量地址上的 **bank** 函数，也只会做普通函数处理。
- `__BANK0`, `__BANK1`, ..., and `__BANK15` 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 源程序>

```

#define FUNC_CALL ( addr ) ( ( void ( * ) ( ) ) ( addr ) ) ( )
#define FUNC_ADDR ( addr ) ( void ( * ) ( ) ) ( addr )
void ( *fp ) ( ) ;
void func ( )
{
    fp = FUNC_ADDR ( __BANK1 ( 0x8000 ) ) ;
    FUNC_CALL ( 0x2000 ) ;                /* 普通函数调用 */
    FUNC_CALL ( __BANK2 ( 0x9000 ) ) ;    /* Bank 函数调用 */
}

```

<编译器输出目标>

```

_func :
; line 6 :      fp = FUNC_ADDR ( __BANK1 ( 0x8000 ) ) ;
               movw      ax , #08000H
               movw      !_fp , ax
               movw      ax , #0101H
               movw      !_fp + 2 , ax
; line 7 :      FUNC_CALL ( 0x2000 ) ;                /*普通函数调用*/
               call      !02000H
; line 8 :      FUNC_CALL ( __BANK2 ( 0x9000 ) ) ;    /* Bank 函数调用*/
               push     hl
               movw      hl , #09000H
               mov       e , #02H
               callt     [ @@bcall ]
               pop      hl
; line 9 : }
               ret

```

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用函数引用常量地址上的 bank 函数，则无需修改。
- 要使用函数来引用常量地址上的 bank 函数，请根据以上描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 引用常量地址上的 bank 函数的函数名称可以被用作普通函数名称。
- 要使用函数来引用常量地址上的 bank 函数，请根据每个 C 编译器的具体规范修改程序。

(25) 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。
 - 注释
 - 其他**#pragma** 指令
 - 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 (0xBF);
    __OPC (0xA1, 0x12);
    __OPC (0x10, 0x34, 0x12);
}
```

<编译器的输出对象>

```
_main :  
; line 4 : __OPC (0xBF) ;  
    DB    0BFH  
; line 5 : __OPC (0xA1, 0x12) ;  
    DB    0A1H  
    DB    012H  
; line 6 : __OPC (0x10, 0x34, 0x12) ;  
    DB    010H  
    DB    034H  
    DB    012H  
; line 7 : }  
    ret
```

兼容性

<从另外某个 C 编译器至本 C 编译器>

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

<从本 C 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 编译器的具体规范修改程序（**#asm**，**#endasm** 或 **asm()**；等）。

(26) 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 78K0 系列 RTOS (实时操作系统) RX78K0 的中断函数。将数据常量插入到当前地址。
- 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。
- 当指定有堆栈改变, 堆栈指针改变的位置为指定数组名称符号加上偏移量得到。数组名称的区域不能由**#pragma** 指令来保留。它需要被单独定义为全局无符号字符型数组。
- 可以在中断函数中为 RTOS 调用两个系统调用函数 **ret_int/ret_wup** (关于系统调用函数的具体细节, 请参阅下文中出现的[\[系统调用函数列表\]](#))
- 如果原形声明或 **ret_int/ret_wup** 的实体定义, 在 RTOS 的中断函数之外, 且 **ret_int/ret_wup** 在中断函数之外的地方被调用, 都会有错误发生。
- 两个系统调用函数 **ret_int/ret_wup** 用无条件跳转指令来调用。
- 如果在 RTOS 的中断函数中既没有 **ret_int** 也没有 **ret_wup** 系统调用函数, 会有错误发生。
- 如果中断请求名称被忽略, 则只有两个函数 **ret_int/ret_wup** 可用。
- 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) 改变堆栈指针(只有在指定了改变堆栈时)
- (4) 保留本地变量区域(只有当本地变量存在时)
- (5) 函数体
- (6) 释放本地变量区域(只有当本地变量存在时)
- (7) 恢复堆栈指针(只有在指定了改变堆栈时)
- (8) 恢复编译器使用的 **saddr** 区域
- (9) 恢复所有的寄存器
- (10) 返回(RETI)

如果在该函数中出现 **ret_int/ret_wup**, 则每次在无条件跳转指令之前马上产生 (6) 项和 (7) 项的代码。
如果 **ret_int/ret_wup** 位于函数的最后, 不产生 (8) 项至 (10) 项的代码

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。
 - 注释
 - 既未定义又未引用变量或函数的预处理指令

#pragma Δ rtos_interrupt [Δ 中断请求名称 Δ 函数名称 Δ 堆栈改变规范]
--

备注 堆栈改变规范: SP = 数组名称[+ 偏移位置]

- **#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。

[RTOS 系统调用函数列表]

(1) void ret_int ();

调用RTOS系统调用函数**ret_int**

(2) void ret_wup (x);

unsigned char *x;

调用 RTOS 系统调用函数 **ret_wup**, x 作为参数

限制

- 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。
- 在向量中断处理期间再次发生中断 (同名或者另一个中断) 会根据中断优先级标志寄存器、中断屏蔽寄存器等。。控制寄存器的值来决定。如果指定了堆栈改变, 堆栈的内容被更新, 这可能会导致问题。但是编译器无法检测到这种错误, 所以请务必小心。
- **callt/callf/noauto/norec/_callt/_callf/_leaf/_interrupt/_interrupt_brk/_pascal/_flash/_flashf** 不能被指定为 RTOS 的中断函数。

RTOS 系统调用函数名称 **ret_int/ret_wup** 不能被用作函数名称。

如果函数使用**#pragma rtos_interrupt** 指定了堆栈变化, 但是未在同一模块中定义, 则会输出警告信息并且堆栈不做改变。

- 静态模式中不支持 RTOS 的中断函数。

示例

(a) 没有指定堆栈改变

<C 源代码>

```
#pragma rtos_interrupt INTPO intp  
  
int i ;  
  
void intp ( ) {  
    int a ;  
    a = 1 ;  
    if ( i == 1 ) {  
        ret_int ( ) ;  
    }  
}
```

<编译器的输出对象>

```

@@CODE      CSEG
_intp :
    push    ax                ; Saves registers
    push    bc                ;
    push    de                ;
    push    hl                ;
    movw    ax , _@RTARG0     ; Saves saddr area used the compiler
    push    ax                ;
    movw    ax , _@RTARG2     ;
    push    ax                ;
    movw    ax , _@RTARG4     ;
    push    ax                ;
    movw    ax , _@RTARG6     ;
    push    ax                ;
    movw    hl , #01H        ; 1
    movw    ax , !_i
    cmpw    ax , #01H        ; 1
    bnz     $?L0003
    br      !_ret_int

?L0003 :
    pop     ax                ; Restores saddr area used by the compiler
    movw    _@RTARG6 , ax    ;
    pop     ax                ;
    movw    _@RTARG4 , ax    ;
    pop     ax                ;
    movw    _@RTARG2 , ax    ;
    pop     ax                ;
    movw    _@RTARG0 , ax ;
    pop     hl                ; Restores registers
    pop     de                ;
    pop     bc                ;
    pop     ax                ;
    reti

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

```

(b) 指定了堆栈改变

<C 源代码>

```
#pragma rtos_interrupt INTPO intp sp = buff + 10

int i ;

unsigned char buff [10] ;

extern unsigned char TaskID1 ;

void intp ( ) {

    int a ;

    a = 1 ;

    if ( i == 1 ) {

        ret_wup ( &TaskID1 ) ;

    }

}
```

<编译器的输出对象>

```

_intp :
    push    ax                ; Saves registers
    push    bc                ;
    push    de                ;
    push    hl                ;
    movw   ax , _@RTARG0      ; Saves saddr area used by the compiler
    push    ax                ;
    movw   ax , _@RTARG2      ;
    push    ax                ;
    movw   ax , _@RTARG4      ;
    push    ax                ;
    movw   ax , _@RTARG6      ;
    push    ax                ;
    movw   ax , sp           ; Switch stack pointer
    movw   sp , #_buff + 10   ;
    push    ax                ;
    movw   hl , #01H          ; 1
    movw   ax , !_i
    cmpw   ax , #01H          ; 1
    bnz    $?L0003
    movw   hl , #_TaskID1
    pop    ax                ; Returns stack pointer
    movw   sp , ax           ;
    br     !_ret_wup
?L0003 :
    pop    ax                ; Returns stack pointer
    movw   sp , ax           ;
    pop    ax                ; Restores saddr area used by the compiler
    mov    w_@RTARG6 , ax    ;
    pop    ax                ;
    mov    w_@RTARG4 , ax    ;
    pop    ax                ;
    movw   _@RTARG2 , ax     ;
    pop    ax                ;
    movw   _@RTARG0 , ax     ;
    pop    hl                ; Restores registers

```

```
pop     de                ;
pop     bc                ;
pop     ax                ;

reti

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

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 编译器至另外某个 C 编译器>

- 通过删除**#pragma opc** 声明，函数名称可以用作函数名称。
- 要用作 RTOS 中断函数使用，请根据每个 C 编译器的具体规范修改程序。

(27) 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。
- 在使用 `__rtos_interrupt` 修饰词声明的函数中可以调用两个系统调用函数 `ret_int/ret_wup` (关于系统调用函数 `ret_int/ret_wup` 的具体细节, 请参阅下文中出现的[\[系统调用函数列表\]](#))。
- 如果原形声明或 `ret_int/ret_wup` 的实体定义, 在 RTOS 的中断函数之外, 且 `ret_int/ret_wup` 在中断函数之外的地方被调用, 都会有错误发生。
- 两个系统调用函数 `ret_int/ret_wup` 用无条件跳转指令来调用。
- 如果在 RTOS 的中断函数中既没有 `ret_int` 也没有 `ret_wup` 系统调用函数, 会有错误发生。

效果

- 中断向量的设置和 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 }
```

[RTOS 系统调用函数列表]

(a) `void ret_int () ;`

调用 RTOS 系统调用函数 `ret_int`

(b) `void ret_wup (x) ;`

`unsigned char *x ;`

调用 RTOS 系统调用函数 `ret_wup`, `x` 作为参数

限制

- `callt/callf/noauto/norec/_callt/_callf/_leaf/_interrupt/_interrupt_brk/_pascal/_flash/_flashf` 不能被指定为 RTOS 的中断函数。
- RTOS 系统调用函数名称 `ret_int/ret_wup` 不能被用作函数名称。
- 静态模式中不支持 `__rtos_interrupt` 修饰词。此时会出现警告信息并指向首次出现 `__rtos_interrupt` 修饰词的位置, `__rtos_interrupt` 修饰词被忽略, 该函数被当作普通函数处理。

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用另外的**#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 编译器至本 C 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。

<从本 C 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 编译器的具体规范修改程序。

(28) 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。
- 有下列的系统调用函数可用

[RTOS 系统调用函数]

(a) void ext_tsk(void);

调用 RTOS 系统调用函数 **ext_tsk**。

但是, 如果 **ext_tsk** 在 **ext_tsk** 原形声明和实体定义、中断函数、RTOS 中断函数中调用, 则会有错误发生。

- 系统调用函数 **ext_tsk** 用无条件跳转指令来调用, 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 Δrtos_task [Δ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。
- 注释
 - 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/callf/noauto/norec/__callt/__callf/__leaf/__interrupt/__interrupt_brk/__rtos_interrupt__pascal/__flash/__flashf** 不能被指定为 RTOS 的任务函数。
- 不能使用普通函数的调用方式来调用 RTOS 的任务函数。
RTOS 系统调用函数名称 **ext_tsk** 不能被用作函数名称。
- 静态模式中不支持 RTOS 的任务函数。

示例

<C 源代码>

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

<编译器的输出对象>

```
@@CODE CSEG
_main :
    push    hl
    movw   hl, #01H    ; 1
    br     !_ext_tsk   ; 结尾程序不做输出
_func :
    push   ax
    push   ax    ; 帧指针没有保存
    movw  ax, sp
    movw  hl, ax
    movw  ax, #01H    ; 1
    mov   [hl + 1], a ; x
    xch   a, x
    mov   [hl], a ; x
    movw  ax, #02H    ; 2
    mov   [hl + 3], a ; r
    xch   a, x
    mov   [hl + 2], a ; r
    br    !_ext_tsk   ; 结尾程序不做输出
```

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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 编译器的具体规范修改程序。

(29) 静态模式**功能**

- 所有参数通过寄存器传输（请参阅 [11.7.5 静态模式函数调用接口](#)）。
- 通过寄存器传输的函数参数，并将其分配在函数-特定的静态区域。
- 自动变量分配到函数-特定的静态区域。
- 在使用 **leaf** 函数^注情况下，参数和自动变量分配到 **saddr** 区域内 0FEDFH 以下的位置，按照描述顺序从高位地址开始存放。因为 **saddr** 区域通常被所有模块中的 **leaf** 函数使用，所以此区称作共享区。共享区的最大尺寸通过 **-SM** 选项的参数来指定。

```
-SM [nn]: nn = 0 to 16
```

Nn 个字节分配给共享区，剩余部分分配到函数-专用静态区。如果指定 **nn = 00** 或忽略此指定，则不使用共享区。

注 对于函数体内部未调用函数的情况，不需要说明 **norec/__leaf**，因为编译器会自动执行判定。

- 可以将 **sreg/__sreg** 关键字添加到函数参数和自动变量之前。被声明为 **sreg/__sreg** 的函数参数和自动变量会被分配到 **saddr** 区域。结果，可以很方便的进行位处理。
- 通过指定 **-RK** 选项，函数参数和自动变量（除函数的静态变量）分配到 **saddr** 且可移进行位处理（请参阅 [11.5 \(3\) 如何使用 saddr 区域](#)）。
- 编译器自动执行以下宏定义。

```
#define __STATIC_MODEL__ 1
```

效果

- 通常情况下，访问静态区域的指令比访问静态帧的指令更短、更快。因此可已在缩短对象代码量的同时并改善运行速度。
- 不执行自动变量和变量（中断函数的寄存器变量、**norec** 函数参数/自动变量、运行时库参数）的保存/恢复操作，而使用 **saddr** 区域来保存/恢复，结果，可以提高中断处理的速度。
- 因为数据区通常由几个 **leaf** 函数使用，所以可以节约存储空间。

使用方法

- 编译期间指定 **-SM** 选项。
静态模式调用在此情况下的对象，而正常模式调用不具有 **-SM** 选项规范的对象。

示例

-SM4 规范的示例如下所示。

<C 源代码>

```

void sub ( char, char, char ) ;
void main ( )
{
    char i = 1 ;
    char j, k ;
    j = 2 ;
    k = i + j ;
    sub ( i, j, k ) ;
}
void sub ( char p1, char p2, char p3 )
{
    char a1, a2 ;
    a1 = 1<<p1 ;
    a2 = p2 + p3 ;
}

```

<编译器的输出对象>

```

@@DATA DSEG UNITP
L0003: DS ( 1 ) ;main函数的自动变量k
        DS ( 1 )
; line 1 : void sub ( char , char , char ) ;
; line 2 : void main ( )
; line 3 : {
@@CODE CSEG
_main :
        push    de
; line 4 : char i = 1 ;
        mov     e , #01H; 1 ; 自动变量 i
; line 5 : char j, k ;
; line 6 : j = 2 ;
        mov     d , #02H; 2 ; 自动变量 j
; line 7 : k = i + j ;
        mov     a , e
        add     a , d ; 将i 和 j 相加
        mov     !?L0003, a ; k ; 代替 k
; line 8 : sub ( i , j , k ) ;
        mov     h , a ; 通过寄存器H传输 k
        push   de
        pop     bc ; 通过寄存器B传输 j
        mov     a , e ; 通过寄存器A传输 i
        call   !_sub
; line 9 : }
        pop     de

```

<编译器的输出对象 ...接上页>

```

ret
; line 10 : void sub ( char p1 , char p2 , char p3 )
; line 11 : {
_sub :
    mov    l , a                ; 将第一个参数分配到 l
    mov    a , h
    mov    @_KREG15 , a        ; 将第三个参数分配到共享区
; line1 2 : char a1, a2 ;
; line1 3 : a1 = 1<<p1 ;
    mov    a , l                ; 第一个参数 p1
    mov    c , a
    mov    a , #01H
    dec    c
    inc    c
    bz     $?L0006
    add    a , a
    dbnz   c , $$ - 2
?L0006 :
    mov    @_KREG14 , a; a1     ; 自动变量 a1 处于共享区
; line14 : a2 = p2 + p3 ;
    mov    a , b                ; 第二个参数 p2
    add    a , @_KREG15 ; p3    ; 添加第三个参数 p3
    mov    @_KREG13 , a ; a2    ; 自动变量 a2 处于共享区
; line 15 : }
ret

```

限制

- 静态模式模块不能与正常模式的模块连接。但是即使共享区的最大尺寸不同，静态模式模块也可以彼此连接。
- 不支持浮点数。如果出现 **float** 和 **double** 关键字，则出现致命错误。
- 参数限制为最多 3 个参数，总共 6 个字节。
- 因为不通过堆栈传输参数，所以不能使用可变长度参数。使用可变长度参数导致错误。
- 不能使用结构体/共用体的参数和返回值。这种类型的参数和返回值会导致错误。
- 不能使用 **noauto/norec/__leaf** 函数。输出警告消息且忽略说明（请参阅 [11.5 \(5\) noauto 函数](#), [11.5 \(6\) norec 函数](#)）。
- 不能使用递归函数。因为函数参数和自动变量区域是静态保留的，所以不能使用递归函数。编译器可以检测递归函数，并产生错误。
- 不能省略原型声明。如果既没有函数的真实定义也没有原型声明，则不管是否存在函数调用，都会产生错误。
- 由于参数的限制，以及不能使用递归函数，因此某些标准库无法使用。
- 如果未指定 **-ZL** 选项，则输出警告并且处理过程和指定了 **-ZL** 选项一样。因此 **long** 型始终视为 **int** 型（请参阅 [11.5 \(30\) 类型调整 \(-ZI\)](#)）。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 当创建正常模式对象时，除非指定 **-SM** 选项，否则无需修改源文件。
- 要创建静态模式对象，根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 如果由另一编译器进行重新编译，则无需修改源文件。

注意事项

- 因为参数/自动变量保留为静态，所以可能破坏递归函数中参数/自动变量的内容。当函数自身直接调用时，出现错误。但是，因为编译器无法检测此过程，所以在调用其他函数之后又进行函数自身调用时，不出现错误报警。
- 中断期间，如果通过中断服务（中断函数和在中断服务函数中调用的函数）调用处理的函数，则可能破坏参数/自动变量的内容。
- 中断期间，即使处理中的函数正在使用共享区时，也不进行共享区保存/恢复。

(30) 类型调整 (-ZI)**(a) 从 int/short 型改为 char 型****功能**

- **int** 和 **short** 型视为 **char** 型。换句话说，**int** 和 **short** 描述等于 **char** 描述。
- 下文给出了类型调整的详细内容（某些受**-QU**选项效果）。

表 11-13. 类型调整的详细信息（从 int 和 short 型改为 char 型）

在 C 源代码中说明的类型	选项	修改之后的类型
short, short int, int	具有 -QU	unsigned char
short, short int, int	不具有 -QU	signed char
unsigned short, unsigned short int, unsigned, unsigned int	–	unsigned char
signed short, signed short int, signed, signed int	–	signed char

- 警告消息会指向 **int** 或 **short** 关键字在 C 源代码中第一次出现的行号。
- 不管是否指定，**-QC** 选项都会有效。当未进行**-QC**选项规范时，输出警告消息且**-QC**选项变得有效。
- 如果同时指定**-ZA**选项（诸如**-ZAI**选项），输出警告消息（仅当指定**-W2**警告等级时）。
- 以下类型的语句可以通过类型声明符说明，如果没有特别声明，即被视为 **char** 型。
 - 函数的自动变量和返回值
 - 类型声明符忽略变量/函数声明
- 编译器自动执行以下宏定义。

```
#define __FROM_INT_TO_CHAR__ 1
```

- 某些标准库不能使用。

使用方法

- 指定**-ZI**选项。

限制

- 指定的**-ZI**和未指定的**-ZI**模块无法连接。

(b) 从 long 型改为 int 型

功能

- long 型视为 int 型。换句话说，long 描述等于 int 描述。
- 下文给出了类型调整的详细内容。

表 11-14. 类型调整的详细内容（从 long 型改为 int 型）

C 源代码中的类型说明	修改之后的类型
unsigned long, unsigned long int	unsigned int
long, long int, signed long, signed long int	signed int

- 警告消息会指向 long 关键字在 C 源代码中第一次出现的行号。
- 如果同时指定 -ZA 选项（-ZAL），则输出警告消息（仅当指定 -W2 警告等级时）。
- 编译器自动执行以下宏定义。

```
#define __FROM_LONG_TO_INT__ 1
```

- 某些标准库不能使用。

使用方法

- 指定 -ZL 选项。

限制

- 指定的 -ZL 和未指定的 -ZL 模块无法连接。

(31) Pascal 函数 (__pascal)**功能**

- 函数调用期间生成将参数放入堆栈的代码由被调用方产生，而不是由发起函数调用的一方负责。

效果

- 如果在程序中多次出现函数调用，则可以缩短目标代码。

使用方法

- 当声明函数时，将__pascal 属性修饰词添加到开始处。

限制

- **pascal** 函数不支持可变长度参数。如果定义可变长度参数，则输出警告并忽略__pascal 关键字。
- 在 **pascal** 函数中不能指定关键字 **norec**、**__interrupt**、**__interrupt_brk**、**__rtos_interrupt**、**__flash**、**__flashf**。如果出现了这些关键字，在 **norec** 关键字情况下，忽略__pascal 关键字；在**__interrupt**、**__interrupt_brk**、**__rtos_interrupt**、**__flash**、**__flashf** 关键字情况下，会输出错误。
- 如果原型声明不完整，则不能正常操作，因此当 **pascal** 函数缺少定义或原型声明缺失时，输出警告消息。
- 当指定 (**-SM**) 静态模式选项时，不支持 **Pascal** 函数。如果当使用 **pascal** 函数时又同时指定了**-SM** 选项，则输出的警告消息指向__pascal 关键字第一次出现的位置，且忽略输入文件中的__pascal 关键字。

说明

- 指定**-ZR** 选项，使所有函数都变成 **pascal** 函数。但是，如果 **pascal** 函数被调用的次数很少，则可能增加目标代码。

示例**<C 源代码>**

```
__pascal int func (int a, int b, int c) ;
void main ()
{
    int ret_val;

    ret_val = func (5, 10, 15) ;
}
__pascal int func (int a, int b, int c) ;
{
    return (a + b + c) ;
}
```

<编译器的输出对象>

```

_main:
    push    hl
    movw   ax, #0FH                ; 参数需要消耗4字节的堆栈
    push   ax                      ;
    mov    x, #0AH                ;
    push   ax                      ;
    mov    x, #05H                ;
    call   !_func
                                   ; 此处不调整堆栈

    movw   ax, bc
    movw   hl, ax
    pop    hl
    ret

_func :
    push   hl
    push   ax
    movw   ax, sp
    movw   hl, ax
    mov    a, [ hl + 6 ]
    add    a, [ hl ]
    xch    a, x
    mov    a, [ hl + 7 ]
    addc   a, [ hl + 1 ]
    xch    a, x
    add    a, [ hl + 8 ]
    xch    a, x
    addc   a, [ hl + 9 ]
    movw   bc, ax
    pop    ax
    pop    hl
    pop    de                      ; 获得返回地址
    pop    ax                      ;
    pop    ax                      ; 调用方所消耗的4字节堆栈在此调整
    push   de                      ; 重新加载返回地址
    ret

```

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用保留字 **__pascal**，则无需修改。
- 要改为 **Pascal** 函数，请根据以上 **使用方法** 描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 使用 **#define** 保持兼容性。
- 通过这样转换，**pascal** 函数被视为普通函数。

(32) 函数调用接口的自动 pascal 功能化 (-ZR)**功能**

- 除了 `norec/__interrupt/__interrupt_brk/__rtos_interrupt/__flash/__flashf` 可变长度参数函数，所有函数都被添加 `__pascal` 属性。

使用方法

- 编译期间指定 `-ZR` 选项。

限制

- 指定了 `-ZR` 选项的模块和其中未指定 `-ZR` 选项的模块无法连接。如果进行链接，则导致链接错误。
- 静态模式规范选项 (`-SM`) 和 `-ZR` 选项不能同时指定。如果同时指定，则输出警告消息并且忽略 `-ZR` 选项。
- 因为数学函数标准库不支持 `pascal` 函数，所以当使用数学函数标准库时，`-ZR` 选项无法使用。

备注 如需 `pascal` 函数调用接口的详细信息，请参阅 [11.7.6 Pascal 函数调用接口](#)。

(33) Flash 区域分配方法 (-ZF)

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。
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。

(34) Flash 区域跳转表 (#pragma ext_table)

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。
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，或者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 32 个地址专用于中断函数（包括启动和例程函数），每个占用 3 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 + 3*32”之后。在设备支持 bank 函数时，跳转表中每项占用 8 字节空间；设备不支持 bank 函数则跳转表中每项占用 3 字节空间。
- 在设备支持 bank 函数时，跳转表中各项占用的位置是 $3*32 + 8 * (\text{ext_func 最大 ID 值} + 1)$ 字节。当设备不支持 bank 函数时，跳转表中各项占用的位置是 $3 * (32 + \text{ext_func 最大 ID 值} + 1)$ 字节。
关于 ext_func 最大 ID 值，请参阅“[\(35\) 从 boot 区域到 flash 区域的函数调用 \(#pragma ext_func\)](#)”。

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，即使设备支持 bank 函数，跳转表还是无法被分配到 bank 区域中。但是首地址值必须和 -ZB 连接选项指定的 flash 起始地址相匹配，如果地址不匹配，则会导致连接错误。

- 为了配合指定的跳转表首地址，必须为中断向量（`_@vect00` 到 `_@vect3e`）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 `\NECTools32\src\cc78k0\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 `/NECTools32\src\cc78k0\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 `\NECTools32\src\cc78k0\lib to \NECTools32\lib78k0` 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 编译器至本 C 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。

<从本 C 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
@@CODE        CSEG
_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
_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 $2000H + 3 * (0006H/2)$ ，由于跳转表的首地址是 $2000H$ ，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
```

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

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。
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。

- 注释
- 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。
 - #pragma ext_func f1 3**
#pragma ext_func f1 4
 - #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** 和 **callf** 函数只能存放在 boot 区域。如果在 flash 区域出现 **callt** 和 **callf** 函数定义（指定了**-ZF** 选项），则导致错误。

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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：设备不支持 bank 函数>

(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 $2000H + 3 \times 32 + 3 \times 3$ 。

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 $2000H + 3 \times 32 + 3 \times 4$ 。

备注 3. 跳转表开始的 3×32 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

```
#pragma  ext_table  0x2000
#pragma  ext_func   f1 3
#pragma  ext_func   f2 4
__banked  extern void f1 ( void );
__non_banked      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
__banked  void f1 ( )
{
}
__non_banked void f2 ( )
{
}
```

备注 4. **#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 $2000H + 3 \times 32 + 8 \times 3$ 。

备注 5. **#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 $2000H + 3 \times 32 + 8 \times 4$ 。

备注 6. 跳转表开始的 3×32 个字节专用于中断函数（包括启动例程）。

<编译器输出目标: 设备不支持 bank 函数>

(1) Boot 区域方面(未指定-ZF 选项)

```

@@CODE    CSEG
_func :
        call    !02069H
        call    !0206CH
        ret

```

(2) Flash 区域方面(指定了-ZF 选项)

```

@ECODE CSEG
_f1 :
        ret
_f2 :
        ret
@EXT03 CSEG AT 02069H
        br     !_f1
        br     !_f2

```

<编译器输出目标: 设备支持 bank 函数>

(1) Boot 区域方面(未指定-ZF 选项)

```

@@CODE    CSEG
_func :
        push   hl
        call   !02078H
        pop    hl
        call   !02080H
        ret

```

(2) Flash 区域方面(指定了-ZF 选项)

```

@@BANK0    CSEG BANK0
_f1 :
        ret
_f2 :
        ret
@EXT03     CSEG AT 02078H
        movw  hl, #_f1
        mov  e, #BANKNUM _f1
        br   !?@bcall
        br   !_f2
        DB   ( 5 )

```

(36) 固件 ROM 函数 (__flash)

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。
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

指针 BC

4 字节 数据 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。
- 指定了静态模式时，不支持 4 字节的数据。

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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。

(37) 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     a , !_e
    mov     x , a                ; 不执行int 扩展
    push   ax
    mov     a , !_d
    mov     x , a                ; 不执行int 扩展
    call   !_func1
    pop     ax
    mov     a , c
    mov     !_c , a
; line 6 : c = func2 ( d , e );
    mov     a , !_e
    mov     x , #00H             ; 0
    xch     a , x                ; 因为没有原形声明, 执行int扩展
    push   ax
    mov     a , !_d
    mov     x , #00H             ; 0
    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
    ret
; line 8 : unsigned char func1 ( unsigned char x , unsigned char y )
_func1 :
    push   hl
    push   ax
    movw   ax , sp
    movw   hl , ax
    mov    a , [ hl ]
    xch    a , x
    mov    a , [ hl + 6 ]
    movw   hl , ax
; line 10 : return x + y ;
    mov    a , l
    add    a , h
    mov    c , a                ; 不要执行int扩展
    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 编译器至本 C 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。

<从本 C 编译器至另外某个 C 编译器>

- 无需修改。

(38) 数组偏移量计算简化方法 (-QW2, -QW3, -QW4, -QW5)**功能**

- 当计算 **char/unsigned char/unsigned int/short/unsigned short** 型数组的偏移量，且索引为 **unsigned char**-型变量时，生成的代码仅对低位字节进行计算，默认为不存在进位。
- 当指定 **-QW2** 选项时，仅当使用 **unsigned char** 变量引用 **saddr** 区域配置序列时，生成的代码仅对低位字节进行计算来产生偏移量。
- 当指定 **-QW3** 选项时，不管配置区域如何，都使用 **unsigned char** 变量引用序列时，生成的代码仅对低位字节进行计算来产生偏移量。

效果

- 因为简化了偏移量计算的代码，所以实现目标代码的缩减和运行速度的提高。

使用方法

- 编译期间指定 **-QW2** 和 **-QW3** 选项。

示例**<C 源代码>**

```
unsigned char c ;
unsigned char ary [10] ;
sreg unsigned char sary [10] ;
void main ( )
{
    unsigned char a ;
    a = ary [c] ;
    a = sary [c] ;
}
```

当指定-QW3 时

<编译器对象的输出>

```

_main :
    push    hl
    push    ax
    movw   ax, sp
    movw   hl, ax
; line 6 :    unsigned char a ;
; line 7 :
; line 8 :    a = ary [c] ;
    mov    a, !_c
    add    a, #low (_ary)
    mov    e, a                ;仅计算低字节
    mov    d, #high (_ary)
    mov    a, [de]
    mov    [hl + 1], a        ; a
; line 9 :    a = sary [c] ;
    mov    a, !_c
    add    a, #low (_sary)
    mov    e, a                ;仅计算低字节
    mov    d, #0FEH          ; 254
    mov    a, [de]
    mov    [hl + 1], a        ; a
; line 10 :   }
    pop    ax
    pop    hl
    ret

```

限制

- 如果为偏移量计算简化目标所生成的序列配置地址超出 256 个字节的限制，程序可能操作错误。
- 不支持-QW4 和-QW5 选项

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 分配规划以使得其不超过 256 个字节。或者，不指定-QW2 和-QW3 选项。

<从本 C 编译器至另外某个 C 编译器>

- 无需修改。

(39) 寄存器直接引用函数 (#pragma realregister)**功能**

- 输出访问对象寄存器的代码，通过直接内联展开而不是函数调用，并生成对象文件。
- 如果没有对应的**#pragma** 指令，寄存器直接引用函数被视为普通函数。

效果

- 可以通过 C 源文件中的描述方便的进行寄存器的访问。

使用方法

- 在源文件中描述的方法和函数调用的格式相同（请参阅本章稍后介绍的[\[寄存器直接引用函数表\]](#)）。存在 21 种类型的寄存器直接引用函数名称。

__geta,	__seta,	__getax,	__setax,	__getcy,	__setcy,	__set1cy,
__clr1cy,	__not1cy,	__inca,	__deca,	__rora,	__rorca,	__rola,
__rolca,	__shla,	__shra,	__ashra,	__nega,	__coma,	__absa

- 通过在模块中使用 **#pragma realregister** 指令，即可声明寄存器直接引用函数。然而，以下各项可以在**#pragma realregister** 指令之前说明。
 - 注释
 - 其他**#pragma** 指令
 - 既未定义又未引用变量或函数的预处理指令

示例**<C 源代码>**

```
#pragma realregister
unsigned char c = 0x88, d, e;
void main ()
{
    __seta (c);           /*设置 A 寄存器的 C 变量          */
    __shla ();           /*逻辑左移 1 位                */
    d = __geta ();       /*设置变量 d 的 A 寄存器的值      */
    if (__getcy ()) {    /*引用 CY (检验溢位)            */
        e = 1;          /*当 CY == 1 时将 e 设定为 1     */
    }
}
```

<编译器的输出对象>

```

_main :
; line 5 : __seta (c) ;          /*设置 A 寄存器的 C 变量          */
        mov    a, !_c
; line 6 : __shla () ;          /* 逻辑左移 1 位          */
        add    a, a
; line 7 : d = __geta () ;      /* 设置变量 d 的 A 寄存器的值      */
        mov    !_d, a
; line 8 : if (__getcy ()) {    /* 引用 CY (检验溢位)          */
        bnc    $?L0003
; line 9     e = 1 ;            /*当 CY == 1 时将 e 设定为 1      */
        mov    a, #01H          ; 1
        mov    !_e, a
?L0003 :
; line 10 :    }
; line 11 :    }
        ret

```

[寄存器直接引用函数表]

- (1) **unsigned char __geta (void) ;**
获得 **A** 寄存器的值。
- (2) **void __seta (unsigned char x) ;**
设置 **A** 寄存器的 **x**。
- (3) **unsigned int __getax (void) ;**
获得 **AX** 寄存器的值。
- (4) **void __setax (unsigned int x) ;**
设置 **AX** 寄存器的 **x**。
- (5) **bit __getcy (void) ;**
获得 **CY** 标志值。
- (6) **void __setcy (unsigned char x) ;**
设置 **CY** 标志的 **x** 的低 1 位。
- (7) **void __set1cy (void) ;**
生成 **set1 CY** 指令。

(8) **void __clr1cy (void) ;**

生成 Clr1 CY 指令。

(9) **void __not1cy (void) ;**

生成 not1 CY 指令。

(10) **void __inca (void) ;**

生成 inc a 指令。

(11) **void __deca (void) ;**

生成 dec a 指令。

(12) **void __ror a (void) ;**

生成 1 ror a,指令。

(13) **void __rorca (void) ;**

生成 1 rorc a,指令。

(14) **void __rol a (void) ;**

生成 1 rol a,指令。

(15) **void __rolca (void) ;**

生成 1 rolc a,指令。

(16) **void __shla (void) ;**

生成执行 A 寄存器逻辑左移 1 位的代码。

(17) **void __shra (void) ;**

生成执行 A 寄存器逻辑右移 1 位的代码。

(18) **void __ashra (void) ;**

生成执行 A 寄存器算术右移 1 位的代码。

(19) **void __nega (void) ;**

生成获得 A 寄存器二补数的代码。

(20) **void __coma (void) ;**

生成获得 A 寄存器一补数的代码。

(21) **void __absa (void) ;**

生成获得 A 寄存器绝对值的代码。

限制

- 寄存器直接引用的函数名称只能用作函数名称。寄存器直接引用函数以小写字母表示。以大写字母表示的函数被视为普通函数。
- 通过 `__seta`, `__setax` 和 `__setcy` 函数设定的 **A** 和 **AX** 寄存器和 **CY** 标志的值, 在下一次代码的生成过程中不保留。
- 由 **A** 和 **AX** 寄存器和 **CY** 标志引用的时序, 对应于表达式的评估序列, 其中 **A**、**AX** 和 **CY** 寄存器通过 `__geta`, `__getax` 和 `__getcy` 函数获得。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用寄存器直接引用函数, 则无需修改。
- 要改为寄存器直接引用函数, 请根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 删除 `#pragma realregister` 指令, 或使用 `#ifdef` 进行屏蔽, 寄存器直接引用函数名称可以当作函数名称使用。
- 要将 `pragma realregister` 用作寄存器直接引用函数, 请根据每个 C 编译器的具体规范修改程序 (`#asm`, `#endasm` 或 `asm()`;等)。

注意事项

- 在执行寄存器直接引用函数之前不保证 **CY**, **A**, **AX** 的内容将按照设计意图进行保存。因此, 建议在其值改变之前使用此函数, 方法是在表达式的开头项中使用

(40) [HL+B]基址变址寻址使用方法 (-QE)**功能**

- 使用 **unsigned char** 变量作为索引来引用 **char/unsigned char** 型数组和 **char/unsigned char** 型指针，生成的代码中会包括**[HL+B]**进行基址变址寻址。

效果

- 可以缩短目标代码，并改善程序执行速度。

使用方法

- 编译时指定**-QE** 选项。

示例**<C 源代码>**

```
unsigned    char c , d ;  
unsigned    char ary [ 10 ] ;  
char    *p ;  
void    main ( )  
{  
    ary [ c ] *= d + 1 ;  
    * ( p + c ) * = 4 ;  
}
```


当指定了-SM 和-QCE 选项时

<编译器输出目标>

```

_main :
; line 6 :      ary [ c ] *= d + 1 ;
    mov     a , !_d
    inc     a
    mov     x , a
    mov     a , !_c
    mov     b , a
    movw   hl , #_ary
    mov     a , [ hl + b ]           ;使用[HL + B]基址变址寻址
    mulux
    mov     a , x
    mov     [ hl + b ] , a         ;使用[HL + B]基址变址寻址
; line 7 :
; line 8 :      * ( p + c ) * = 4 ;
    mov     a , !_c
    mov     b , a
    movw   ax , !_p
    movw   hl , ax
    mov     a , [ hl + b ]         ;使用[HL + B]基址变址寻址
    add     a , a
    add     a , a
    mov     [ hl + b ] , a         ;使用[HL + B]基址变址寻址
; line 9 : }
    ret

```

限制

- 对于某些源程序描述语句，目标代码可能会增加。在正常模式下，本函数被禁用。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 无需修改。

<从本 C 编译器至另外某个 C 编译器>

- 无需修改。

(41) 片上固件自编程子程序直接调用函数 (#pragma hromcall)

通过 C 规范说明可以很方便的进行寄存器调用，在源程序中以函数调用的格式指定这个函数，或者在模块中使用 **#pragma hromcall** 指令声明片上固件自编程子程序直接调用函数。

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。
本函数用来使能设备写入 flash 存储器的功能。

功能

- 输出片上固件自编程子程序直接调用函数代码，通过直接内联展开而不是函数调用，并生成目标文件。
- 如果没有对应的**#pragma** 指令，则片上固件自编程子程序直接调用函数被视为普通函数。
- **__setup** 函数将 SP（堆栈指针）设置为指定地址。
- **__hromcall** 函数要调用指定地址，需要将寄存器组临时切换到寄存器组 3，并将函数编号置入 C 寄存器，入口 RAM 区域起始地址置入 HL。B 寄存器中存放的内容是返回值。
- **__hromcalla** 函数要调用指定地址，需要将寄存器组临时切换到寄存器组 3，并将函数编号置入 C 寄存器，入口 RAM 区域起始地址置入 HL。B 寄存器中存放的内容是返回值。

效果

- 可以在 C 源程序中轻松调用片上固件自编程子程序。

使用方法

- 在源文件中描述的方法和函数调用的格式相同。下列 3 个函数是片上固件自编程子程序直接调用函数名称（请参阅下文中的[\[片上固件自编程子程序直接调用函数列表\]](#)）。

__hromcall, __hromcalla, __setsp

- 模块中的**#pragma hromcall** 指令声明片上固件自编程子程序直接调用函数。但是，以下各项可以在**#pragma hromcall** 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 di
#pragma sfr
#pragma hromcall
unsigned char entryram [ 32 ];
unsigned char ret ;
void func ( )
{
    /* 禁止中断 */
    DI ( ) ;
    /* 进入自编程模式 */
    FLSPM0 = 1 ;

    /* 调用__hromcall 子程序调用函数 */
    __hromcall ( 0x8100 , 0 , entryram ) ;
    /* 设置写入时间数据 */
    entryram [ 7 ] = 0x20 ;
    /* 设置删除时间数据 */
    entryram [ 8 ] = 0x4c ;
    entryram [ 9 ] = 0x4c ;
    entryram [ 10 ] = 0x00 ;
    /* 设置收敛时间数据 */
    entryram [ 11 ] = 0x01 ;
    entryram [ 12 ] = 0x3d ;
    /* 调用__hromcall 子程序 */
    ret = __hromcall ( 0x8100 , 1 , entryram ) ;
    :
}

```

<编译器的输出对象>

```

_func ;
    di
; line 8 : /* 禁止中断 */
; line 9 : DI ( ) ;
; line 10 : /* 进入自编程模式 */
; line 11 : FLSPM0 = 1 ;
        set1        FLSPM0
; line 12 :
; line 13 : /*调用__hromcall 子程序调用函数 */
; line 14 : __hromcall ( 0x8100 , 0 , entryram ) ;
        push        psw                ; 保存当前程序状态字
        sel         rb3                ; 切换到寄存器组 3
        movw       hl , #_entryram

```

```

    mov     c, #00H                                ; 0
    call   !08100H
    pop    psw                                     ; 返回当前程序状态字
    mov    a, 0FEE3H
; line 15 : /* 设置写入时间数据 */
; line 16 : entryram [ 7 ] = 0x20 ;
    mov    a, #020H                                ; 32
    mov    !_entryram + 7, a
; line 17 : /* 设置删除时间数据 */
; line 18 : entryram [ 8 ] = 0x4c ;
    mov    a, #04CH                                ; 76
    mov    !_entryram + 8, a
; line 19 : entryram [ 9 ] = 0x4c ;
    mov    !_entryram + 9, a
; line 20 : entryram [ 10 ] = 0x00 ;
    mov    a, #00H                                ; 0
    mov    !_entryram + 10, a
; line 21 : /* 设置收敛时间数据 */
; line 22 : entryram [ 11 ] = 0x01 ;
    inc    a
    mov    !_entryram + 11, a
; line 23 : entryram [ 12 ] = 0x3d ;
    mov    a, #03DH                                ; 61
    mov    !_entryram + 12, a
; line 24 : /* 调用 __hromcall 子程序 */
; line 25 : ret = __hromcall ( 0x8100, 1, entryram );
    push   psw                                     ; 保存当前程序状态字
    sel    rb3                                     ; 切换到寄存器组 3
    movw   hl, #_entryram
    mov    c, #01H                                ; 1
    call   !08100H
    pop    psw                                     ; 返回当前程序状态字
    mov    a, 0FEE3H
    mov    !_ret, a
    ret

```

[片上固件自编程序子程序直接调用函数列表]

- (1) unsigned char __hromcall (unsigned int entryaddr, unsigned char funcno, void *entrydata) ;
在临时切换到寄存器组 3 之后调用 entryaddr 地址，将 entryaddr 地址设置到 HL 寄存器中，funcno 存放到 C 寄存器。B 寄存器中存放的内容是返回值。
- (2) unsigned char __hromcalla (unsigned int entryaddr, unsigned char funcno, void *entrydata) ;
在临时切换到寄存器组 3 之后调用 entryaddr 地址，将 entryaddr 地址设置到 HL 寄存器中，funcno 存放到 C 寄存器。A 寄存器中存放的内容是返回值。
- (3) void __setsp (unsigned int spaddr) ;
在 SP (堆栈指针) 中设置 spaddr 的值。

限制

- 片上固件自编程子程序直接调用函数的名称不能用作普通函数名称。
- 该函数在不包含固件的设备中不能使用，固件中可以直接调用自编程子程序。
- 如果片上固件自编程子程序的说明和如下语句不符，该函数无法使用
 - (i) 使用寄存器组 3
 - (ii) 在 C 寄存器中设置函数编号
 - (iii) 在 HL 寄存器中设置入口 RAM 区域的起始地址
- `__hromcall` 和 `__hromcalla` 函数中的第一个和第二个参数只能使用常量，如果指定的不是常量，则会导致错误发生。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果未使用片上固件自编程子程序直接调用函数，则无需修改。
- 如果要改用片上固件自编程子程序直接调用函数，请根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 删除 `#pragma hromcall` 语句或用 `#ifdef` 将其屏蔽。片上固件自编程子程序直接调用函数的名称可以被用作普通函数名称。
- 要使用 `#pragma hromcall` 调用片上固件自编程子程序直接调用函数，请根据每个编译器的具体规范（`#asm`，`#endasm` 或 `asm()`；等）修改程序。

注意事项

- 在调用该函数之前，参数要存放在 RAM 区域中（请参阅相关设备用户手册查看在入口 RAM 区域设置的值）。
- 该函数不进行中断禁止处理，也不进行转换到自编程模式的相关处理。所以应该在使用该函数之前完成这些处理过程。
- 关于 `__hromcall` 和 `__hromcalla` 函数中设置的固件入口地址，以及函数编号中设置的值，请参阅相关设备的用户手册。

(42) __flashf 函数 (__flashf)

当设备不具有 flash 区域自写入功能时，请勿使用本 flash 函数，如果使用，操作无法保证。

本函数用来使能设备写入 flash 存储器的功能。

功能

- 在函数的开始处将程序状态字保存到堆栈之后，该函数禁止中断，并切换到寄存器组 3。
- 保存在堆栈中的程序状态字在函数结尾处进行恢复。
- 不管有没有 #pragma hromcall 指令，“(41) 片上固件自编程子程序直接调用函数 (#pragma hromcall)”中所使用的函数都有效。
- 函数调用方将参数保存到 A（1 个字节数据）或 AX（2 个字节数据）；正常模式下函数定义方会将传入 A 或 AX 的参数拷贝到 saddr 区域（FEBAH 到 FEBFH）
- 自动变量被分配到 saddr 区域（正常模式下为 FEBAH 到 FEBFH），包括寄存器变量。

效果

- 如果函数定义时有 __flash 属性，在该函数中出现“(41) 片上固件自编程子程序直接调用函数 (#pragma hromcall)”，则在每次调用时不生成用于 bank 保存/恢复和切换到寄存器组 3 的代码。

使用方法

- 如果函数定义时，添加 __flash 属性。

示例

<C 源代码>

```
#pragma di
#pragma sfr
#pragma hromcall
unsigned char entryram [ 32 ];
unsigned char ret ;
__flashf void func ( )
{
    /* 换为自编程模式*/
    FLSPM0 = 1 ;
    /* 调用__hromcall 子程序*/
    __hromcall ( 0x8100 , 0 , entryram ) ;
    /*设置写入时间数据*/
    entryram [ 7 ] = 0x20 ;
    /*设置删除时间数据*/
    entryram [ 8 ] = 0x4c ;
    entryram [ 9 ] = 0x4c ;
    entryram [ 10 ] = 0x00 ;
    /*设置收敛时间数据*/
    entryram [ 11 ] = 0x01 ;
    entryram [ 12 ] = 0x3d ;
    /* 调用__hromcall 子程序*/
    ret = __hromcall ( 0x8100 , 1 , entryram ) ;
    :
}
```

<编译器的输出对象>

```

_func ;
    push    psw          ; 保存当前程序状态字          ; 这三行程序为编译器自动生成
    di                      ; 禁止中断
    sel     rb3          ; 切换到寄存器组 3
; line 7 : /*换为自编程模式*/
; line 8 : FLSPM0 = 1 ;
        set1    FLSPM0
; line 9 :
; line 10 : /*调用__hromcall 子程序*/
; line 11 : __hromcall ( 0x8100 , 0 , entryram ) ;
        movw   hl , #_entryram
        mov    c , #00H ; 0
        call   !08100H
; line 12 : /*设置写入时间数据*/
; line 13 : entryram [ 7 ] = 0x20 ;
        mov    a , #020H ; 32
        mov    [ hl + 7 ] , a
; line 14 : /*设置删除时间数据*/
; line 15 : entryram [ 8 ] = 0x4c ;
        mov    a , #04CH ; 76
        mov    [ hl + 8 ] , a
; line 16 : entryram [ 9 ] = 0x4c ;
        mov    [ hl + 9 ] , a
; line 17 : entryram [ 10 ] = 0x00 ;
        mov    a , #00H ; 0
        mov    [ hl + 10 ] , a
; line 18 : /*设置收敛时间数据*/
; line 19 : entryram [ 11 ] = 0x01 ;
        inc    a
        mov    [ hl + 11 ] , a
; line 20 : entryram [ 12 ] = 0x3d ;
        mov    a , #03DH ; 61
        mov    [ hl + 12 ] , a

```



```

; line 21 : /*调用__hromcall 子程序*/
; line 22 : ret = __hromcall ( 0x8100 , 1 , entryram ) ;
        mov     c , #01H ; 1
        call   !08100H
        mov    a , b
        mov    !_ret , a
        :
        pop    psw          ; 返回当前程序状态字          ; 本行亦由编译器自动生成
        ret

```

限制

- 从__flash 函数中只能调用“(41) 片上固件自编程子程序直接调用函数 (#pragma hromcall) ”。
- 只有 char/unsigned char/int/unsigned int/short/unsigned short/pointer 型可以定义为自动变量。
- 参数和自动变量最多一共可以定义 6 个字节。
- 无法进行 long 型操作。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果未使用__flash 关键字，则无需修改。
- 要使用__flash 函数，请根据以上使用方法描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 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](#)。

(43) 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。
 - 注释
 - 其他 `#pragma` 指令
 - 既未定义又未引用变量或函数的预处理指令

示例

<C 源代码>

```
#pragma inline
char  ary1[100], ary2[100];
void  main()
{
    memset (ary1, 'A', 50) ;
    memcpy (ary1, ary2, 50) ;
}
```

当未指定-**SM** 时

<编译器的输出对象>

```

_main:
    push    hl
;line 5: memset (ary1, 'A', 50) ;
    movw   de,#_ary1
    mov    a, #041H ; 65
    mov    c, #032H ; 50
    mov    [de], a
    incw   de
    dbnz   c, $$-2
;line 6: memcpy (ary1, ary2, 50) ;
    movw   de, #_ary1
    movw   hl, #_ary2
    mov    c, #032H ; 50
    mov    a, [hl]
    mov    [de], a
    incw   de
    incw   hl
    dbnz   c, $$-4
;line 7: }
    pop    hl
    ret

```

当指定-**SM** 时

```

_main:
    push    de
;line 5: memset (ary1, 'A', 50) ;
    movw   hl,#_ary1
    mov    a, #041H ; 65
    mov    c, #032H ; 50
    mov    [hl], a
    incw   hl
    dbnz   c, $$-2
;line 6: memcpy (ary1, ary2, 50) ;
    movw   hl, #_ary1
    movw   de, #_ary2
    mov    c, #032H ; 50
    mov    a, [de]
    mov    [hl], a
    incw   de
    incw   hl
    dbnz   c, $$-4
;line 7: }
    pop    de
    ret

```

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用内存运算函数，则无需修改。
- 当改变内存运算函数时，请根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 删除**#pragma inline** 指令，或用**#ifndef** 对其限制。

(44) 绝对地址分配规范 (__directmap)**功能**

- 由函数中__directmap 和 static 变量声明的外部变量，其初值被视为分配的地址，且该变量分配到指定地址。
- C 源代码中的__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范范围、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。

地址规范范围	保留区域范围	双重检验范围
0x80 至 0xffff	0xfd00 至 0xfeff	0xf000 至 0xfeff

- 如果指定的地址在地址规范范围之外，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 **W762** 警告消息并显示有重复的变量名称。
- 如果地址规范范围在 saddr 区域之内，则自动作出__sreg 声明，并生成 saddr 指令。
- 如果按二进制引用由__directmap 声明的 char/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long 型变量，则 sreg/__sreg 必须和 __directmap 连同使用。如果不指定，则引用时会出现错误。

效果

- 一个或一个以上的变量可以分配到相同属性的地址。

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

- 如果为结构体/共用体/数组声明__directmap，则将地址放入括号{}中。
- 在__directmap 外部变量被引用的模块中，__directmap 不需要声明，仅声明 extern 即可。

extern		类型名称	变量名称;
extern	__sreg	类型名称	变量名称;
- 如果模块中有某个位于 saddr 区域内的__directmap 外部变量被引用，为了生成 saddr 指令，则必须使用 __sreg 来进行“extern __sreg 类型名称变量名称;”。

示例

<C 源代码>

```

__directmap char c = 0xfe00;
__directmap __sreg char d = 0xfe20;
__directmap __sreg char e = 0xfe21;
__directmap struct x {
    char a;
    char b;
} xx = {0xfe30};
void main()
{
    c = 1;
    d = 0x12;
    e.5 = 1;
    xx.a = 5;
    xx.b = 10;
}

```

<输出对象>

```

        PUBLIC  _c
        PUBLIC  _d
        PUBLIC  _e
        PUBLIC  _xx
        PUBLIC  _main
_c      EQU     0FE00H      ;由__directmap 声明的变量的地址
_d      EQU     0FE20H      ;由 EQU 定义
_e      EQU     0FE21H      ;
_xx     EQU     0FE30H      ;
        EXTRN  __mmfe00    ;链接保留区域模块的 EXTRN 输出
        EXTRN  __mmfe20    ;
        EXTRN  __mmfe21    ;
        EXTRN  __mmfe30    ;
        EXTRN  __mmfe31    ;
@@CODE CSEG
_main:
;line  10:      c = 1;
        mov     a,#01H ;1
        mov     !_c,a
;line  11:      d = 0x12;
        mov     _d,#012H    ;因为地址指定在 saddr 区域内, 输出 saddr 指令
;line  12:      e.5 = 1;
        set1    _e.5        ;因为使用了 __sreg, 可以进行位处理

```

```

;line 13:    xx.a = 5;
           mov    _xx,#05H      ; 因为地址指定在 saddr 区域内, 输出 saddr 指令
;line 14:    xx.b = 10;
           mov    _xx+1,#0AH    ; 因为地址指定在 saddr 区域内, 输出 saddr 指令
;line 15:    }
           ret

```

限制

- 不能将函数参数、返回值或自动变量指定为 `__directmap`。如果有类型指定，则出现错误。
- 如果 `short/unsigned short/int/unsigned int/long/unsigned long` 型变量分配在奇地址，则由 `__directmap` 声明的文件中会生成校正代码，而如果由 `extern` 声明从外部文件引用这些变量，则将生成非法代码。
- 如果指定的地址位于保留区域范围之外，则变量区域不作保留，有必要指定指令文件或为保留该区域创建单独的模块。

兼容性

<从另外某个 C 编译器至本 C 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。

<从本 C 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范，请根据每个编译器的具体规范来修改程序。

(45) 静态模式展开规范 (-ZM)

功能

- `_@NRAT00` 至 `_@NRAT07` 的 8 个字节 **saddr** 区域被编译器保留，为参数和工作使用。
- 将参数和自动变量声明 `__temp` 就可以使用临时变量（如需详细信息，请参阅 11.5 (46) 临时变量）。
- 可以声明的参数数量范围为 3 到 6 个整型变量，对于字符型变量为 3 至 9 个。第四个及以后的参数被发起调用方设置在 `_@NRAT00` 至 `_@NRAT05` 的区域，并由被调用方拷贝到单独区域。但是，如果已为 **leaf** 函数或参数声明了 `__temp`，被调用方将不会进行参数拷贝，且其中存放参数的 `_@NRATxx` 区将按其正常用途使用。
- 参数可以使用 2 个字节以内的结构体/共用体。
- 函数返回值可以使用结构体/共用体。如果结构体/共用体为 2 个字节或更少字节，则返回该值。如果结构体/共用体为 3 个字节或更多字节，则将返回值存储在专门保存返回值的保留静态区，并返回该区的最高地址。
- `_@NRAT00` 至 `_@NRAT07` 的 8 个字节区还用作 **leaf** 函数共享区。在共享区的分配过程中，首先分配到 `_@NRAT00` 至 `_@NRAT07` 的 8 个字节区，然后再使用指定 **-SM** 选项指定的 `_@KREGxx` 区。
- 数组、结构体/共用体也可以分配到 `_@NRATxx` 和 `_@KREGxx`，只要其大小适合可以放入 `_@KREGxx` 区，`_@KREGxx` 区通过 `_@NRATxx` 和 **-SM** 选项来指定。
- 中断函数需要保存的目标在下表 11-15 中给出。

表 11-15 中断函数存储目标

恢复/保存区	无 BANK	调用函数				不调用函数			
		-ZM1		-ZM2		-ZM1		-ZM2	
		RBn	堆栈	RBn	堆栈	RBn	堆栈	RBn	堆栈
使用的寄存器	NG	NG	NG	NG	NG	NG	OK	NG	OK
所有寄存器	NG	NG	OK	NG	OK	NG	NG	NG	NG
整个 <code>_@NRATxx</code> 区	NG	OK	OK	OK	OK	NG	NG	NG	NG
整个 <code>_@KREGxx</code> 区	NG	OK	OK	NG	NG	NG	NG	NG	NG
使用的 <code>_@KREGxx</code> 区	NG	NG	NG	OK	OK	NG	NG	OK	OK

堆栈：指定使用堆栈

RBn：指定寄存器组

OK： 保存

NG： 不保存

注 然而，当指定 `#pragma interrupt` 时，中断函数的存储目标可以通过如下指定来限制。

`SAVE_R`（保存/恢复目标限制于寄存器中）

`SAVE_RN`（保存/恢复目标限制于寄存器和 `_@NRATxx`）。

- `-ZM1` 与 `-ZM2` 选项的差别，只是在处理 `-SM` 选项保留的 `_@KREGxx` 区有所不同。
当指定 `-ZM1` 选项时，`_@KREGxx` 区仅用于 `leaf` 函数共享区。
当指定 `-ZM2` 选项时，对 `_@KREGxx` 区进行保存/恢复，且参数和自动变量分配也在此（正常模式下与 `-QR` 选项兼容）。
- 如果指定 `-ZM` 选项的同时未指定 `-SM` 选项，则输出 `W055` 警告消息且忽略 `-ZM` 选项规范。

效果

可以减少对现有静态模式的限制，提高描述的方便性。

使用方法

编译期间指定 `-SM` 选项及 `-ZM` 选项。

示例 1

<C 源代码>

```
char func1 (char a, char b, char c, char d, char e) ;
char func2 (char a, char b, char c, char d) ;
void main ()
{
    char a = 1, b = 2, c = 3, d = 4, e = 5, r;
    r = func1 (a, b, c, d, e) ;
}
char func1 (char a, char b, char c, char d, char e)
{
    char r;

    r = func2 (a, b, c, d) ;
    return e + r;
}
char func2 (char a, char b, char c, char d)
{
    return a + b + c + d;
}
```

当指定 **-SM8**, **-ZM1** 和 **-QC** 时

<输出对象>

```

_main:
; line 5 :      char a = 1, b = 2, c = 3, d = 4, e = 5, r;
      mov     a,#01H          ;1
      mov     !L0003,a        ;a
      inc     a
      mov     !L0004,a        ;b
      inc     a
      mov     !L0005,a        ;c
      inc     a
      mov     !L0006,a        ;d
      inc     a
      mov     !L0007,a        ;e

; line 6 :
; line 7 :      r = func1 (a, b, c, d, e);
      mov     @_NRAT01,a      ;将第五个参数设定到 saddr 区域以便接收和传输参数
      mov     a,!L0006        ;d
      mov     @_NRAT00,a      ;将第四个参数设定到 saddr 区域以便接收和传输参数
      mov     a,!L0005        ;c
      mov     h,a
      mov     a,!L0004        ;b
      mov     b,a
      mov     a,!L0003        ;a
      call    !_func1
      mov     !L0008,a        ;r

; line 8 : }
      ret

; line 9 :      char func1 (char a, char b, char c, char d, char e)
; line 10 : {
_func1:
      mov     !L0011,a
      mov     a,b
      mov     !L0012,a
      mov     a,h
      mov     !L0013,a
      mov     a,_@NRAT00      ;复制到静态区
mov     !L0014,a ;
      mov     a,_@NRAT01      ;复制到静态区
      mov     !L0015,a        ;

; line 11 :      char r;
; line 12 :
; line 13 :      r = func2 (a, b, c, d);
      mov     a,!L0014        ;d
      mov     @_NRAT00,a      ;将第四个参数设定到 saddr 区域以便接收和传输参数
      mov     a,!L0013        ;c
      mov     h,a

```

```

        mov     a,!L0012          ;b
        mov     b,a
        mov     a,!L0011          ;a
        call    !_func2
        mov     !L0016,a          ;r
; line 14 : 返回 e + r;
        add     a,!L0015          ;e
; line 15 : }
        ret
; line 16 :     char func2 (char a, char b, char c, char d)
; line 17 :     {
_func2:
        mov     @_NRAT01,a
        mov     a,b
        mov     @_NRAT02,a
        mov     a,h
        mov     @_NRAT03,a
; line 18 : 返回 a + b + c + d;
        mov     a,@_NRAT01        ;a
        add     a,@_NRAT02        ;b
        add     a,@_NRAT03        ;c
        add     a,@_NRAT00        ;d 使用 leaf 函数的 @_NRAT00
; line 19 : }
        ret

```

当指定 **-SM8**, **-ZM2**, **-QC** 时

```

@@CODE CSEG
_main:
        movw   ax,@KREG10         ;
        push   ax                 ;保存 @KREG10 至 @KREG15 区
        movw   ax,@KREG12         ;
        push   ax                 ;
        movw   ax,@KREG14         ;
        push   ax                 ;
; line 5 : char a = 1, b = 2, c = 3, d = 4, e = 5, r;
        mov     @_KREG15,#01H      ;a,1 将变量分配到 @_KREG11 至 @_KREG15
        mov     @_KREG14,#02H      ;b,2
        mov     @_KREG13,#03H      ;c,3
        mov     @_KREG12,#04H      ;d,4
        mov     @_KREG11,#05H      ;e,5
; line 6 :
; line 7 : r = func1 (a, b, c, d, e);
        mov     a,@_KREG11         ;e
        mov     @_NRAT01,a         ;将第五个参数设定到 saddr 区域以便接收和传输参数
        mov     a,@_KREG12         ;d
        mov     @_NRAT00,a         ;将第四个参数设定到 saddr 区域以便接收和传输参数

```

```

        mov     a,_@KREG13           ;c
        mov     h,a
        mov     a,_@KREG14         ;b
        mov     b,a
        mov     a,_@KREG15         ;a
        call    !_func1
        mov     _@KREG10,a         ;r
; line 8 : }
        pop     ax                  ;
        movw    _@KREG14,ax        ;恢复_@KREG10 至_@KREG15 区
        pop     ax                  ;
        movw    _@KREG12,ax        ;
        pop     ax                  ;
        movw    _@KREG10,ax        ;
        ret
; line 9 : char func1 (char a, char b, char c, char d, char e)
; line 10 : {
_func1:
        mov     _@NRAT06,a         ;保存寄存器 a
        movw    ax,_@KREG10        ;
        push    ax                 ;保存_@KREG10 至_@KREG15 区
        movw    ax,_@KREG12        ;
        push    ax                 ;
        movw    ax,_@KREG14        ;
        push    ax                 ;
        mov     a,_@NRAT06         ;恢复寄存器 a
        mov     _@KREG15,a
        movw    ax,bc
        mov     _@KREG14,a
        movw    ax,hl
        mov     _@KREG13,a
        mov     a,_@NRAT00         ;复制到_@KREG12
        mov     _@KREG12,a         ;
        mov     a,_@NRAT01         ;复制到_@KREG11
        mov     _@KREG11,a         ;
; line 11 :     char r;
; line 12 :
; line 13 :     r = func2 (a, b, c, d) ;
        mov     a,_@KREG12         ;d
        mov     _@NRAT00,a         ;将第四参数设定到 saddr 区域以便传输参数
        mov     a,_@KREG13         ;c
        mov     h,a
        mov     a,_@KREG14         ;b
        mov     b,a
        mov     a,_@KREG15         ;a
        call    !_func2
        mov     _@KREG10,a         ;r
; line 14 : 返回 e + r;

```

```

        add    a,_@KREG11      ;e
L0004:
; line 15 : }
        movw  hl,ax           ;保存寄存器 a
        pop   ax              ;
        movw  @_KREG14,ax     ;恢复_@KREG10 至_@KREG15 区
        pop   ax              ;
        movw  @_KREG12,ax     ;
        pop   ax              ;
        movw  @_KREG10,ax     ;
        movw  ax,hl           ;恢复寄存器 a
        ret
; line 16 : char func2 (char a, char b, char c, char d)
; line 17 : {
_func2:
        mov   @_NRAT01,a
        mov   a,b
        mov   @_NRAT02,a
        mov   a,h
        mov   @_NRAT03,a
; line 18 : 返回 a + b + c + d;
        mov   a,_@NRAT01      ;a
        add   a,_@NRAT02      ;b
        add   a,_@NRAT03      ;c
        add   a,_@NRAT00      ;d 使用 leaf 函数的_@NRAT00
L0006:
; line 19 : }
        ret

```

示例 2

<C 源代码>

```

__sreg struct x {
    unsigned char    a;
    unsigned char    b:1;
    unsigned char    c:1;
} xx,yy;
__sreg struct y {
    int a;
    int b;
} ss, tt;
struct x func1 (struct x) ;
struct y func2();
void main()
{
    yy = func1 (xx) ;
    tt = func2();
}
struct x func1 (struct x aa)
{
    aa.a = 0x12;
    aa.b = 0;
    aa.c = 1;
    return aa;
}
struct y func2 ( )
{
    return tt;
}

```

当指定**-SM** 和**-ZM** 时

<输出对象>

```

@@CODE CSEG
_main:
;line   14:   yy = func1(xx);
        movw  ax,_xx
        call  !_func1
        movw  _yy,ax
;line   15:   tt = func2();
        call  !_func2
        movw  hl,ax
        push  de
        movw  de,#_tt
        mov   c,#04H           ;4
        mov   a,[hl]
        mov   [de],a

```

```

incw    hl
incw    de
dbnz   c,$$-4
pop     de
;line 16: }
ret
;line 17: struct x func1(struct x aa)
;line 18: {
_func1:
movw   _@NRAT00,ax
;line 19: aa.a = 0x12;
mov    _@NRAT00,#012H      ; aa,18
;line 20: aa.b = 0;
clr1   _@NRAT01.0
;line 21: aa.c = 1;
set1   _@NRAT01.1

;line 22: return aa;
movw   ax,_@NRAT00      ; aa 因为 2 个字节或更少的字节返回的值
;line 23: }
ret
;line 24: struct y func2()
;line 25: {
;line 26: return tt;
movw   hl,#_tt          ; 因为3个字节或更多字节
push   de               ; 复制到保护静态区的返回值
movw   de,#L0007
mov    c,#04H          ;4
mov    a,[hl]
mov    [de],a
incw   hl
incw   de
dbnz   c,$$-4
pop    de
movw   ax,#L0007      ; 返回静态区的前列地址
;line 27: }
ret

```

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 无需修改源程序。

<从本 C 编译器至另外某个 C 编译器>

- 无需修改源程序。

(46) 临时变量 (`__temp`)

功能

- 参数和自动变量分配到 `_@NRAT00` 至 `_@NRAT07` 区域，而不管其是否需要对应 `leaf` 函数。如果参数和自动变量未分配到 `_@NRAT00` 至 `_@NRAT07` 区，则处理方法与未声明 `__temp` 时的处理方式相同。
- 函数调用时舍弃由 `__temp` 声明的参数和自动变量的值。
- 不能将外部和静态变量声明为 `__temp`。
- 如果同时还声明了 `__sreg`，则 `char/unsigned char/short/unsigned short/int/unsigned int` 变量可以进行位操作。
- 如果在未指定 `-SM` 和 `-ZM` 选项时声明 `__temp`，则输出 `W0339` 警告消息，且忽视文件中的 `__temp` 声明。

效果

- 因为由 `__temp` 声明的参数和自动变量共享 `_@NRAT00` 至 `_@NRAT07` 区域，可以保留参数和自动变量区域。
- 如果可以清晰标识包含参数的区段和包含自动变量的区段，并且 `__temp` 声明应用于函数调用前后不需要保证值匹配的变量，则可以保留内存。

使用方法

编译期间指定 `-SM` 和 `-ZM` 选项，并将参数和自动变量声明为 `__temp`。

示例

<C 源代码>

```

void func1 ( __temp char a, char b, char c, __sreg __temp char d ) ;
void func2 ( char a ) ;
void main ( )
{
    func1 ( 1, 2, 3, 4 ) ;
}
void func1 ( __temp char a, char b, char c, __sreg __temp char d )
{
    __temp char r;

    d.1 = 0;
    r = a + b + c + d;
    func2 ( r ) ;
}
void func2 ( char r )
{
    int a = 1, b = 2;
    r++;
}

```

当指定 **-SM**, **-ZM** 和 **-QC** 时

<输出对象>

```

@@CODE CSEG
_main:
; line 5:      func1 ( 1, 2, 3, 4 ) ;
            mov     a,#04H           ; 4
            mov     @_NRAT00,a
            mov     h,#03H           ; 3
            mov     b,#02H           ; 2
            sub     a,#01H           ; 1
            call   !_func1
; line 6:      }
            ret
; line 7:      void func1 ( __temp char a, char b, char c, __sreg __temp char d )
; line 8:      {
_func1:
            mov     @_NRAT01,a      ; 分配到 @_NRAT01
            movw   ax,bc
            mov     !L0005,a
            movw   ax,hl
            mov     !L0006,a
                                   ; 分配到 @_NRAT00 的参数没有改变
; line 9:      __temp char r;
; line 10:
; line 11:     d.1 = 0;

```

```

        clr1    @_NRAT00.1      ;可能会进行位操作
; line      12 : r = a + b + c + d;
        mov     a,_@NRAT01      ; a
        add    a,!L0005         ; b
        add    a,!L0006         ; c
        add    a,_@NRAT00      ; d
        mov    @_NRAT02,a       ; r 使用_@NRAT02
; line      13 :      func2 ( r );
        call   !_func2

; line      14 :      }          ;_@NRAT00 至_@NRAT02 的值在返回后被改变
        ret

; line      15 :      void func2 ( char r)
; line      16 :      {
_func2:
        mov     @_NRAT00,a
; line      17 :      int a = 1, b = 2;
        movw   ax,#01H ; 1
        movw   @_NRAT02,ax      ; a,1
        incw   ax
        movw   @_NRAT04,ax      ; b, 2
; line      18 :      r++;
        inc    @_NRAT00
; line      19 : }
        ret

```

限制

如果调用的函数参数小于等于 3 个，则针对函数调用的自动变量可以将参数和自变量声明为 `__temp`。如果存在 4 个或更多的参数，则在参数估计期间可能会舍弃参数的值，所以不能保证描述的值。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 如果不使用保留字 `__temp`，则无需修改。
- 要改为临时变量，请根据以上 [使用方法](#) 描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 使用 `#define` 可以实现兼容性（如需详细信息，请参阅 [11.6 C 源代码的修改](#)）。
这个修改意味着 `__temp` 变量按照常规变量进行处理。

(47) 支持序言/结尾的库 (-ZD)**功能**

- 指定模式的序言/结尾代码可以通过库调用来替换。
- 用户可以使用的 **callt** 入口数量在正常模式下减少两个，在静态模式下最多可能占用十个。
- 在正常模式下库替换模式如下所示。
HL, @_KREGxx 保存/复制, 堆栈帧保护 → **callt [@@cprep2]**
HL, @_KREGxx 恢复, 堆栈帧释放 → **callt [@@cdisp2]**
- 在静态模式下, 参数分配到 **_@NRATxx** 和 **_@KREGxx**, 使得前 3 个参数符合以下说明的模式。当 **char** 和 **int** 混合使用时, 调整分配间隔按照多个 **int** 型参数模式处理。
- 在静态模式下库替换模式如下所示。

<对于 char 2 参数>

mov	_@NRAT00,a	→	callt [@@nrcp2]
movw	ax,bc		
mov	_@NRAT01,a		
mov	_@KREG15,a	→	callt [@@krpc2]
movw	ax,bc		
mov	_@KREG14,a		

<对于 char 3 参数>

mov	_@NRAT05,a	→	callt [@@nrcp3]
movw	ax,bc		
mov	_@NRAT06,a		
movw	ax,hl		
mov	_@NRAT07,a		
mov	_@KREG15,a	→	callt [@@krpc3]
movw	ax,bc		
mov	_@KREG14,a		
movw	ax,hl		
mov	_@KREG13,a		
mov	_@NRAT06,a	→	call !@@nkrc3
movw	ax,bc		
mov	_@NRAT07,a		
movw	ax,hl		
mov	_@KREG15,a		

<对于 int 2 参数>

movw	_ _@NRAT00,ax	→	callt [@ _@nrip2]
movw	ax,bc		
movw	_ _@NRAT02,ax		
movw	_ _@KREG14,ax	→	callt [@ _@krip2]
movw	ax,bc		
movw	_ _@KREG12,ax		

<对于 int 3 参数>

movw	_ _@NRAT02,ax	→	callt [@ _@nrip3]
movw	ax,bc		
movw	_ _@NRAT04,ax		
movw	ax,hl		
movw	_ _@NRAT06,ax		
movw	_ _@KREG14,ax	→	callt [@ _@krip3]
movw	ax,bc		
movw	_ _@KREG12,ax		
movw	ax,hl		
movw	_ _@KREG10,ax		
movw	_ _@NRAT04,ax	→	call! @ _@nkri31
movw	ax,bc		
movw	_ _@NRAT06,ax		
movw	ax,hl		
movw	_ _@KREG14,ax		
movw	_ _@NRAT06,ax	→	call! @ _@nkri32
movw	ax,bc		
movw	_ _@KREG14,ax		
movw	ax,hl		
movw	_ _@KREG12,ax		

<对于保存/恢复>

_ _@NRAT00 to _ _@NRAT07 save	→	callt [@ _@nrsave]
_ _@NRAT00 to _ _@NRAT07 restore	→	callt [@ _@nrload]
_ _@KREG14 to 15 save	→	call ! @ _@krs02
_ _@KREG12 to 15 save	→	call ! @ _@krs04
	→	call ! @ _@krs04i
_ _@KREG10 to 15 save	→	call ! @ _@krs06
	→	call ! @ _@krs06i
_ _@KREG08 to 15 save	→	call ! @ _@krs08
	→	call ! @ _@krs08i

_ _	@KREG06 to 15 save	→ →	call !@@krs10 call !@@krs10i
_ _	@KREG04 to 15 save	→ →	call !@@krs12 call !@@krs12i
_ _	@KREG02 to 15 save	→ →	call !@@krs14 call !@@krs14i
_ _	@KREG00 to 15 save	→ →	call !@@krs16 call !@@krs16i
_ _	@KREG14 to 15 restore	→	call !@@krl02
_ _	@KREG12 to 15 restore	→ →	call !@@krl04 call !@@krl04i
_ _	@KREG10 to 15 restore	→ →	call !@@krl06 call !@@krl06i
_ _	@KREG08 to 15 restore	→ →	call !@@krl08 call !@@krl08i
_ _	@KREG06 to 15 restore	→ →	call !@@krl10 call !@@krl10i
_ _	@KREG04 to 15 restore	→ →	call !@@krl12 call !@@krl12i
_ _	@KREG02 to 15 restore	→ →	call !@@krl14 call !@@krl14i
_ _	@KREG00 to 15 restore	→ →	call !@@krl16 call !@@krl16i

效果

通过以库替换开端和结尾代码，可以缩短目标代码。

使用方法

编译期间指定 **-ZD** 选项。

示例 1

<C 源代码>

```

int func1 (int a, int b, int c) ;
int func2 (int a, int b, int c) ;
void main ()
{
    int r;

    r = func1 (1, 2, 3) ;
}
int func1 (int a, int b, int c)
{
    return func2 (a+1, b+1, c+1) ;
}
int func2 (int a, int b, int c)
{
    return a+b+c;
}

```

当指定 **-SM**, **-ZM2D** 和 **-QC** 时

```

@@CODE CSEG
_main:
    movw    ax, @_KREG14
    pushw  ax
;line    5:    int r;
;line    6:
;line    7:    r = func1(1, 2, 3) ;
    movw    hl, #03H ; 3
    movw    bc, #02H ; 2
    movw    ax, #01H ; 1
    call   !_func1
    movw    @_KREG14, ax; r
;line    8:    }
    popw   ax
    movw    @_KREG14, ax
    ret
;line    9:    int func1(int a, int b, int c)
;line    10:   {
_func1:
    call   !@@krs06
    callt  [@@krip3]
;line    11:   return func2 (a+1, b+1, c+1) ;
    movw    ax, @_KREG10; c
    incw   ax
    movw    hl, ax

```

```

        movw    ax, _@KREG12; b
        incw   ax
        movw   bc, ax
        movw   ax, _@KREG14; a
        incw   ax
        call   !_func2

L0004:
;line    12:  }
        call  !@@kr106
        ret

;line    13:  int func2 (int a, int b, int c)
;line    14:  {
_func2:
        callt [@@nrip3]

;line    15:  return a+b+c;
        movw  ax, _@NRAT02    ; a
        xch  a, x
        add  a, _@NRAT04    ; b
        xch  a, x
        addc a, _@NRAT05    ; b
        xch  a, x
        add  a, _@NRAT06    ; c
        xch  a, x
        addc a, _@NRAT07    ; c

L0006:
;line    16:  }
        ret

```

示例 2

<C 源代码>

```

int func(register int a, register int b) ;
void main()
{
    register int a = 1, b = 2, c = 3, r;

    r = func(a, b) ;
}
int func (register int a, register int b)
{
    register int r;

    r = a + b;
    return r;
}

```

当指定-QR 和-ZD 时

<输出对象>

```

@@CODE CSEG
_main:
    movw    de,#0300H
    callt   [@@cprep2]
; line    4 :    register int a = 1, b = 2, c = 3, r;
    movw    hl,#01H                ; 1
    movw    @_KREG00, #02H          ; b,2
    movw    @_KREG02, #03H          ; c,3
; line    5 :
; line    6 :    r = func(a, b);
    movw    ax,@_KREG00            ; b
    push    ax
    movw    ax,hl
    call    !_func
    pop     ax
    movw    ax,bc
    movw    @_KREG04,ax            ; r
; line    7 : }
    movw    ax,#0300H
    callt   [@@cdisp2]
    ret
; line    8 :    int func (register int a,register int b)
; line    9 :    {
_func:
    movw    de,#0C940H
    callt   [@@cprep2]
; line   10 :    register int r;
; line   11 :
; line   12 :    r = a + b;
    movw    ax,hl
    xch     a,x
    add     a,@_KREG12            ; a
    xch     a,x
    addc    a,@_KREG13            ; a
    movw    @_KREG00,ax            ; r
L0004:
; line   14 :    }
    movw    ax,#0C940H
    callt   [@@cdisp2]
    ret

```


限制

- 优化规范选项-QL4 和-ZD 选项不能同时指定。如果同时指定，则输出 W0052 警告消息，并用-QL3 选项替换-QL4 选项进行处理。
- flash 区域分配选项-ZF 不能和-ZD 选项同时指定，如果同时指定，则输出 W0054 警告消息，并且-ZD 选项被放弃。

注意事项

仅当前 3 个参数没有任何一个被指定 **register**，或将前 3 个参数都指定为 **__temp** 时，在静态模式下的参数拷贝模式是模式-匹配的。因此，如果指定-**QV** 选项，或为前 3 个参数有某个被指定为 **register/__temp**，则不进行模式-匹配，所以不可能替换-**ZD** 选项。

兼容性

<从另外某个 C 编译器至本 C 编译器>

- 无需修改源程序。
- 要以库替换序言/结尾代码，请根据以上**使用方法**描述的过程对源程序进行修改。

<从本 C 编译器至另外某个 C 编译器>

- 无需修改源程序。

11.6 C 源代码的修改

通过使用本 C 编译器的扩展函数，可以产生效率更高的目标。但是，希望这些扩展函数仅适用于 78K/0 系列。因此，要将其用于其他设备，可能需要对 C 源代码进行修改。此处，介绍了如何使 C 源代码可以从另一 C 编译器移植到本 C 编译器以及相反操作。

<从另一 C 编译器至本 C 编译器>

- **#pragma**[±]

如果另一 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一 C 编译器的规范。

- 扩展的规范

如果另一 C 编译器已扩展了规范，诸如添加了新的关键字，则必须修改 C 源代码。修改 C 源代码的方法和程度取决于另一 C 编译器的规范。

注 **#pragma** 为 ANSI 支持的预处理指令。

#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。

<从本 C 编译器至另一 C 编译器>

- 因此此 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一 C 编译器。

示例

<1> 要使关键字无效（同样应用于 **callf**, **sreg**, **noauto** 和 **norec** 等）

```
#ifndef __K0S__
#define callt /* 将 callt 作为普通函数 */
#endif
```

<2> 要从一种类型变为另一种

```
#ifndef __K0S__
#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) 存储参数的位置和顺序
 - (c) 存储自动变量的位置和顺序
3. **noauto** 函数调用接口
 - (a) 传输参数
 - (b) 存储参数的位置和顺序
 - (c) 存储自动变量的位置和顺序
4. **norec** 函数调用接口
 - (a) 传输参数
 - (b) 存储参数的位置和顺序
 - (c) 存储自动变量的位置和顺序
5. 静态模式函数调用接口
 - (a) 传输参数
 - (b) 存储参数的位置和顺序
 - (c) 存储自动变量的位置和顺序
6. **Pascal** 函数调用接口

11.7.1 返回值

调用的函数将返回值存储在寄存器和进位标志中，如表 11-16 所示。

表 11-16 存储返回值的位置

模 式 类型	正常模式下的存放位置	静态模式下的存放位置
1-字节整数	BC	A
2-字节整数		AX
4-字节整数	BC（低） DE（高）	不支持
指针（未使用 bank 函数（-MF））	BC	AX
指针（使用 bank 函数（-MF））	BC（数据指针） BC（低），DE（高）（函数指针）	不支持
结构体，共用体	BC（如果拷贝到函数专用区，则为结构体/共用体的起始地址）	不支持
1 位	CY（进位标志）	CY（进位标志）
浮点数（float 型）	BC（低），DE（高）	不支持
浮点数（double 型）	BC（低），DE（高）	不支持

11.7.2 普通函数调用接口

当所有参数分配到寄存器且未使用自动变量时，普通函数调用接口与 **noauto** 函数调用接口相同。

(1) 传输参数

- 存在两类参数：分配到寄存器的参数，普通参数。
- 只要寄存器和 **_@KREGxx** 还有剩余空间，分配到寄存器的参数预先有 **register** 声明并且被成功分配到寄存器或 **_@KREGxx**。但是，仅当指定 **-QR** 时，参数才会被分配到 **_@KREGxx**。下文称分配到寄存器或 **_@KREGxx** 的参数为寄存器参数。
- 有关 **_@KREGxx** 的详细信息，请参阅附录 **saddr 区域域标签列表**。
- 剩余参数分配到堆栈。
- 在函数发起调用方，以 **register** 声明的参数和普通参数传输方式相同。第二个及以后的参数通过堆栈传输且第一个参数通过寄存器或堆栈传输。
- 在函数定义方，通过寄存器或堆栈传输的参数被保存在参数分配的位置。
- 寄存器参数拷贝到寄存器或 **_@KREGxx**。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 普通参数通过堆栈传递。当参数通过堆栈传输时，其中参数被传入的区域就成为参数分配到的区域。
- 对存放参数的寄存器进行保存和恢复，这个工作在函数定义方进行。
- 第一个参数传入的位置如表 11-17 所示。

表 11-17. 第一个参数传入的位置（函数调用方）

类型	选项	正常模式
1-字节数据 ^注		AX
2-字节数据 ^注		AX, BC
3-字节数据 ^注		AX, BC
4-字节数据 ^注		AX, BC
浮点数（float 型）		AX, BC
浮点数（double 型）		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

- 存在两类参数：分配到寄存器的参数和普通参数。用 **register** 声明的参数，当指定 **-QV** 时的参数都会被分配到寄存器。
- 未分配到寄存器的参数分配到堆栈。分配到堆栈的参数从最后一个参数开始按顺序放于堆栈上。
- 对存放参数的寄存器进行保存和恢复，这个工作在函数定义方进行。
- 在函数定义方，通过寄存器或堆栈传输的参数存储在参数分配的区域。
- 寄存器参数被拷贝到寄存器或 **_@KREGxx**。仅在指定了 **-QR** 选项时才会拷贝到 **_@KREGxx** 中。即使参数通过寄存器传输时，也必须拷贝寄存器，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 在函数调用方，寄存器参数和普通参数的传输方法相同。第二个或随后的参数通过堆栈传输。第一个参数通过寄存器或堆栈传输。如需了解其中第一个参数的传入位置，请参阅表 11-17。

（要使用的寄存器）

HL

当存在堆栈帧时，参数不会分配到 **HL**。

（要使用的 **saddr** 区域）

_@KREG12 至 15

(分配顺序)

- 寄存器
 - char** 型: 顺序为 L-H。
 - int, short** 和 **enum** 型: **HL**
- **saddr** 区域
 - char** 型: 顺序为 **_@KREG12, _@KREG13, _@KREG14** 和 **_@KREG15**。
 - int, short** 和 **enum** 型: 顺序为 **_@KREG12 至 13** 和 **_@KREG14 至 15**。
 - long, float, double** 型: 顺序为 **_@KREG12 至 13 (低)** 到 **_@KREG14 至 15 (高)**。

(3) 自动变量存储的位置和顺序

- 存在两类自动变量：分配到寄存器的自动变量和普通自动变量。用 **register** 声明的自动变量和指定 **-QV** 选项时的自动变量被分配到寄存器。只要寄存器和 **_@KREGxx** 存在空间，就尽量分配到寄存器和 **_@KREGxx**。但是，仅当指定 **-QR** 选项时，自动变量才会分配到 **_@KREGxx**。
下文称分配到寄存器和 **_@KREGxx** 的自动变量为寄存器变量。
- 如需 **_@KREGxx** 的详细信息，请参阅附录 [saddr 区域标签列表](#)。
- 寄存器参数分配完成之后，才会分配寄存器变量。因此，当在寄存器参数分配完成之后如果存在过剩的寄存器时，寄存器变量分配到寄存器。
- 未分配到寄存器的自动变量分配到堆栈。
- 将寄存器和 **_@KREGxx** 保存和恢复到自动变量分配区域的工作在函数定义方进行。

(a) 自动变量分配顺序

将自动变量分配到 **_@KREGxx** 的顺序如下所示。

(将要使用的寄存器)

HL

当存在堆栈帧时，自动变量不分配到 **HL**。

(要使用的 **saddr** 区域)

_@KREG00 至 11

(分配顺序)

- 寄存器
 - char** 型: 顺序为 L 和 H。
 - int, short** 和 **enum** 型: **HL**
- **saddr** 区域
 - char** 型: 顺序为 **_@KREG00, _@KREG01 ...**和 **_@KREG11**。
 - int, short** 和 **enum** 型: 顺序为 **_@KREG00 至 01, _@KREG02 至 03 ...**和 **_@KREG10 至 11**。
 - long, float, double** 型: 顺序为 **_@KREG00 至 03, _@KREG04 至 07** 和 **_@KREG12 至 11**。
- 分配到堆栈的自动变量按声明的顺序分配到堆栈。

[示例]

在正常模式下

<C 源代码 1>

```

void func0 (register int, int) ;

void main ()

{
    func (0x1234, 0x5678) ;
}

void func (register int p1, int p2)

{
    register int r ;

    int a ;

    r = p2 ;

    a = p1 ;

}

```

<输出代码>

```

_main:
; line 4:      func0 (0x1234, 0x5678) ;
    movw      ax, #05678H          ; 22136
    push      ax                    ;通过堆栈接收/传递参数
    movw      ax, #01234H          ; 4660    ;将第一个参数传输至寄存器
    call      !_func0              ;函数调用
    pop       ax                    ;通过堆栈接收/传递参数
; line 5: }
    ret
; line 6:      void func0(register int p1, int p2)
; line 7:      {
_func0:
    push      hl
    xch       a, x
    xch       a, @_KREG12
    xch       a, x
    xch       a, @_KREG13          ;将寄存器参数 p1 分配到 @_KREG12
    push      ax                    ;保存寄存器参数的 saddr 区域
    movw      ax, @_KREG00
    push      ax                    ;保存寄存器变量的 saddr 区域
    push      ax                    ;为自动变量 a 保留的区域

```

```

movw    ax, sp
movw    hl, ax

```

```

; line 8:    register int r ;
; line 9:    int a ;
; line 10:   r = p2 ;
            mov    a, [hl + 10]    ; p2    ;分配参数 p2, 通过堆栈接收/传输
            xch    a, x
            mov    a, [hl + 11]    ; p2
            movw   @_KREG00, ax    ; r    ;至寄存器变量 _KREG14
; line 11:   a = p1 ;
            movw   ax, @_KREG12    ; p1    ;将寄存器参数 _KREG12 分配到
            mov    [hl + 1], a     ; a
            xch    a, x
            mov    [hl], a        ; a    ;自动变量 a
; line 12:   }
            pop    ax              ;释放为自动变量 a 保留的区域
            pop    ax
            movw   @_KREG00, ax    ;恢复寄存器变量的 saddr 区域
            pop    ax
            movw   @_KREG12, ax    ;恢复寄存器参数的 saddr 区域
            pop    hl
            ret

```


<C 源代码 2>

```

void func1 (int, register int) ;
void main ()
{
    func1 (0x1234, 0x5678) ;
}
void func1 (int p1, register int p2)
{
    register int r ;
    int a ;
    r = p2 ;
    a = p1 ;
}

```

<输出代码>

```

_main:
; line 4:          func1 (0x1234, 0x5678) ;
    movw    ax, #05678H          ; 22136
    push   ax                    ; 通过堆栈接收/传递参数
    movw    ax, #01234H          ; 4660    ; 将第一个参数传递到寄存器
    call   !_func1              ; 函数调用
    pop    ax                    ; 通过堆栈接收/传递参数
; line 5:          }
    ret

; line 6:          void func1 (int p1,register int p2)
; line 7:          {
_func1:
    push   hl
    push   ax                    ; 将第一个参数 p1 加载于堆栈上
    movw   ax, @_KREG00
    push   ax                    ; 保存寄存器参数的 saddr 区域
    movw   ax, @_KREG12
    push   ax                    ; 保存寄存器参数的 saddr 区域
    push   ax                    ; 为自动变量 a 保留的区域
    movw   ax, sp
    movw   hl, ax
    mov    a, [hl + 12]          ; 将参数 p2 从堆栈传输到 saddr 区域

```

```

    xch    a, x
    mov    a, [hl + 13]
    movw  @_KREG12, ax           ; 将寄存器参数分配到 _@KREG12

; line    8:                    register int r ;
; line    9:                    int a ;
; line    10:                   r = p2 ;
    movw  ax, @_KREG12         ; p2
    movw  @_KREG00, ax        ; r           ; 寄存器变量 _@KREG00
; line    11:                   a = p1 ;
    mov   a, [hl + 6]         ; p1         ; 将参数 p1 (低) 从寄存器传输至堆栈
    mov   [hl], a             ; a         ; 自动变量 a (低)
    xch   a, x
    mov   a, [hl + 7]         ; p1         ; 将参数 p1 (高) 从寄存器传输至堆栈
    mov   [hl + 1], a        ; a         ; 自动变量 a (高)
; line    12:                   }
    pop   ax                  ; 释放自动变量 a 的区
    pop   ax
    movw  @_KREG12, ax        ; 为寄存器变量恢复的 saddr 区域
    pop   ax
    movw  @_KREG00, ax       ; 为寄存器变量恢复的 saddr 区域
    pop   ax
    pop   hl
    ret

```

11.7.3 noauto 函数调用接口（仅在正常模式可用）

（1）传递参数

- 在函数调用方，参数的方式传输和普通函数相同。请参阅 [11.7.2 普通函数调用接口](#)。
- 在函数定义方，通过寄存器或堆栈传输的参数被拷贝到寄存器及 `_@KREG12 至 15` 中。仅当指定 `-QR` 选项时，才会有参数被拷贝 `_@KREG12 至 15`。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 对存放参数的寄存器进行保存和恢复，这个工作在函数定义方进行。

（2）参数存储的位置和顺序

- 在函数定义方，所有参数被分配到寄存器和 `_@KREG12 至 15`。但是，仅当指定 `-QR` 选项时，参数才会分配到 `_@KREG12 至 15`。
- 如果有参数既没有分配到寄存器也没有分配到 `_@KREG12 至 15`，则出现错误。
- 在函数调用方，参数的传输方式和普通函数相同（请参阅 [11.7.2 普通函数调用接口](#)）。
- 在函数定义方，通过寄存器或堆栈传输的参数被拷贝到寄存器及 `_@KREG12 至 15` 中。仅当指定 `-QR` 选项时，才会有参数被拷贝 `_@KREG12 至 15`。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 对存放参数的寄存器进行保存和恢复，这个工作在函数定义方进行。

（分配顺序）

- 分配顺序与普通函数相同（请参阅 [11.7.2 普通函数调用接口](#)）。

（3）存储自动变量的位置和顺序

自动变量分配到寄存器和 `_@KREG12 至 15` 中。但是，仅当指定 `-QR` 选项时，自动变量才会分配到 `_@KREG12 至 15`。如需 `_@KREG12 至 15` 的详细信息，请参阅 [附录 saddr 区域标签列表](#)。

当参数分配到寄存器完成之后，如果还有空闲的寄存器，则自动变量也分配到寄存器。当指定 `-QR` 选项时，自动变量也被分配到 `_@KREG12 至 15`。

如果有自动变量既没有分配到寄存器也没有分配 `_@KREG12 至 15`，则出现错误。

对存放自动变量的寄存器 `_@KREG12 至 15` 进行保存和恢复，这个工作在函数定义方进行。

(分配顺序)

- 将自动变量分配到寄存器的顺序和参数分配的顺序相同。
- 分配到 `_@KREG12` 至 `15` 的自动变量按声明的顺序进行。

[示例]

<C 源代码>

```
noauto void func2 (int, int) ;  
void main ()  
{  
    func2 (0x1234, 0x5678) ;  
}  
noauto void func2 (int p1, int p2)  
{  
    .  
    .  
    .  
}
```

<输出代码>

```

_main:
; line 4:      func2 (0x1234, 0x5678) ;
      movw    ax, #05678H           ; 22136
      push   ax                    ;通过堆栈传输的参数
      movw    ax, #01234H           ; 4660   ;通过寄存器传输的第一个参数
      call   !_func2                ;函数调用
      pop    ax                    ;通过堆栈传输的参数
; line 5:      }
      ret
; line 6:      noauto void func2 (int p1, int p2)
; line 7:      {
_func2:
      push   hl                    ;为参数保留的寄存器
      xch    a, x
      xch    a, @_KREG12            ;将参数 p1 分配到_@KREG12 (低)
      xch    a, x
      xch    a, @_KREG13            ;将参数 p1 分配到_@KREG13 (高)
      push   ax                    ;为参数保留的 saddr 区域
      movw   ax, sp
      movw   hl, ax
      mov    a, [hl + 6]            ;寄存器 p2 (低) 通过堆栈传输, 通过寄存器接收
      xch    a, x
      mov    a, [hl + 7]            ;寄存器 p2 (高) 通过堆栈传输, 通过寄存器接收
      movw   hl, ax                ;将参数分配到 HL
      pop    ax
      movw   @_KREG12, ax           ;恢复自动变量的 saddr 区域
      pop    hl                    ;恢复参数的寄存器
      ret

```

11.7.4 norec 函数调用接口 (正常模式)

(1) 传输参数

所有参数分配到_@NRARGx 和_@RTARG6 至 7。在函数调用方, 参数通过寄存器_@NRARGx 传递。

在函数定义方, 通过寄存器传输的参数拷贝到寄存器, 或拷贝到_@RTARG6 至 7 (请参阅附录 saddr 区域标签列表)。

(2) 存储参数的位置和顺序

- 在函数定义方, 所有参数分配到寄存器, _@NRARGx, _@RTARG6 至 7。仅当指定了-QR 选项时, 参数才会分配到_@NRARGx。
- 仅当 DE 中存在参数时, 自动变量分配到_@RTARG6 至 7 (请参阅附录 saddr 区域标签列表)。

- 如果有参数既没有分配到寄存器也没有分配 `_@NRARGx`, `_@RTARG6` 至 `7`, 则发生错误。
- 在函数调用方, 参数通过寄存器和 `_@NRARGx` 传递。
- 在函数定义方, 通过寄存器传输的参数被拷贝到寄存器或 `_@RTARG6` 至 `7`。即使参数通过寄存器传递, 也必须进行寄存器拷贝, 因为函数调用方 (发送方) 的寄存器和函数定义方 (接收方) 的寄存器不同。如果参数通过寄存器传输, 则参数传入的区域就成为为其分配的区域。
- 如果参数不再通过寄存器传输, 则可以分配到 `_@NRARGx` 且通过该处传输。在此情况下, 通过寄存器和 `_@NRARGx` 进行联合传输。

(参数分配顺序)

- 分配到 `_@NRARGx` 的参数按照声明的顺序分配。
- 分配到寄存器的参数根据以下规则分配到寄存器 `_@RTARG6` 至 `7`。

(要使用的寄存器)

- 当一个参数为 `char`, `int`, `short`, `enum` 或指针型时: **AX** 传输, **DE** 接收
- 当两个或两个以上参数为 `char`, `int`, `short`, `enum` 或指针型时: **AX** 和 **DE** 传输, `_@RTARG6`, `7` 和 **DE** 接收

(分配顺序)

- `char`, `int`, `short`, `enum` 和指针型: 顺序为 **DE**, `_@RTARG6` 至 `7`

(3) 存储自动变量的位置和顺序

只要寄存器和 `_@NRARGx` 还有空间, 自动变量就尽可能分配到寄存器和 `_@NRARGx`。如果不存在空闲的寄存器, 则分配到 `_@NRATxx`。但是, 仅当指定 `-QR` 选项时, 自动变量才会分配到 `_@NRARGx` 和 `_@NRATxx`。

如需 `_@NRATxx` 的详细信息, 请参阅[附录 saddr 区域标签列表](#)。

如果有自动变量既没有分配到寄存器也没有分配 `_@NRARGx` 和 `_@NRATxx`, 则出现错误。

对存放自动变量的寄存器进行保存和恢复, 这个工作在函数定义方进行。

(分配顺序)

- 将自动变量分配到寄存器, `_@RTARG6`至 `7` 的顺序与分配参数的顺序相同。
- 分配到 `_@NRARGx`, `_@NRATxx`的自动变量按照声明的顺序分配。

[例]

在正常模式下

<C 源代码>

```

norec void func3 (char, int, char, int) ;
void main ()
{
    func3 (0x12, 0x34, 0x56, 0x78) ;
}
norec void func3 (char p1, int p2, char p3, int p4)
{
    int a ;
    a = p2 ;
}

```

当指定-QR 时

<输出代码>

```

_main :
; line 4 :      func3 (0x12, 0x34, 0x56, 0x78) ;
    movw      @_NRARG1, #078H ;120      ;通过_@NRARG1 传输的参数
    mov       @_NRARG0, #056H ;86       ;通过_@NRARG0 传输的参数
    movw      de, #034H      ;52       ;通过寄存器 DE 传输的参数
    mov       a, #012H      ;18       ;通过寄存器 A 传输的参数
    call      !_func3      ;函数调用
    ret
; line 6 :      norec void func3 (char p1, int p2, char p3, int p4)
; line 7 :      {
_func3 :
    mov       @_RTARG6, a      ;将参数 p1 分到_@RTARG6
; line 8 :      int a ;
; line 9 :      a = p2 ;
    movw      ax, de          ;参数 p2
    movw      @_NRARG2, ax    ;a      ;自动变量 a
    ret

```

11.7.5 静态模式函数调用接口

(1) 传输参数

- 在函数调用方，寄存器参数的传输方式和普通参数相同。
所有参数要通过寄存器传输，最多可以有三个参数共 6 个字节。
- 在函数定义方，通过寄存器传输的参数被存储为其分配的区域。寄存器参数拷贝到寄存器。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。
- 普通函数分配到函数专用区。

(2) 存储参数的位置和顺序

(a) 参数存储位置

- 存在两类自动变量：分配到寄存器的参数和普通参数。
- 分配到寄存器的参数就是使用 **register** 声明过的参数。
- 在函数定义方，通过寄存器或堆栈传输的参数存储参数所分配的区域。
寄存器参数拷贝到寄存器。即使参数通过寄存器传递，也必须要进行寄存器拷贝，因为函数调用方（发送方）的寄存器和函数定义方（接收方）的寄存器不同。普通参数分配到函数专用区。
- 对存放参数/自动变量的寄存器进行保存和恢复，这个工作在函数定义方进行。
- 剩余的自动变量分配到函数专用区。
- 在函数调用方，寄存器参数的传输方式和普通参数相同。所有参数要通过寄存器传输，最多可以有三个参数共 6 个字节。表 11-18 展示了参数传入的位置。

表 11-18 静态模式下传输参数的区

数据大小	第一个参数	第二个参数	第三个参数
1-字节数据 [#]	A	B	H
2-字节数据 [#]	AX	BC	HL
4-字节数据 [#]	分配到 AX 和 BC 和剩余参数分配到 H 或 HL 。		

注 在 1 至 4-字节数据中不能包括有结构体或共用体。

(b) 参数分配顺序

- 分配到函数专用区的参数按照顺序从最后一个参数分配。
- 在符合以下规则时，寄存器参数分配到寄存器 **DE**。

(要使用的寄存器)

DE

(分配顺序)

char 型: **D, E** 序列

int, short, enum 型: **DE**

(3) 存储自动变量的位置和顺序**(a) 自动变量的存储位置**

- 存在两类自动变量：分配到寄存器的自动变量和普通自动变量。
- 用 **register** 声明的自动变量和当指定 **-QV** 时的自动变量分配到寄存器。
- 在寄存器参数分配完成之后，再分配寄存器变量。因为这种原因，仅当寄存器参数分配完成之后，寄存器还有空闲，才会将寄存器变量分配至寄存器。
- 剩余的自动变量分配到函数专用区。
- 将寄存器保存和恢复到分配参数的区域在函数定义方进行。

(b) 自动变量分配顺序

- 在符合以下规则时，自动变量分配到寄存器 DE。

(要使用的寄存器)

DE

(分配顺序)

char 型: 顺序为 **E, D**

int, short, enum 型: **DE**

- 分配到函数专用区的自动变量按声明的顺序进行分配。

[示例 1]**<C 源代码>**

```
void func4 (register int, char);
void main ()
{
    func4 (0x1234, 0x56);
}
void func4 (register int p1, char p2)
{
    register char r;
    int a;
    r = p2;
    a = p1; func();
}
```

<输出代码>

```

@@DATA      DSEG
L0005:      DS          (1)          ; 参数 p2
L0006:      DS          (1)          ; 自动变量 r
L0007:      DS          (2)          ; 自动变量 a

; line 1:   void func4 (register int, char) ; void func(void);
; line 2:   void main ()
; line 3:   {

@@CODE      CSEG
_main:
; line 4:   func4 (0x1234, 0x56) ;
    mov     b, #056H             ; 86          ;通过寄存器 B 传输第二个参数
    movw   ax, #01234H          ; 4660       ;通过寄存器 AX 传输第一个参数
    call   !_func4              ;函数调用
; line 5:   }
    ret
; line 6:   void func4 (register int p1, char p2)
; line 7:   {
_func4:
    push   de                    ;为寄存器参数保存寄存器
    movw   de, ax                ;将寄存器参数 p1 分配到 DE
    movw   ax, bc
    mov    !L0005, a              ;将参数 p2 拷贝到 L0005
; line 8:   register char r ;
; line 9:   int a ;
; line 10:  r = p2 ;
    mov    !L0006, a              ; r    ; 自动变量 r
; line 11:  a = p1 ; func();
    movw   ax, de                ;寄存器参数 p1
    movw   #L0007, ax            ; a    ; 自动变量 a
    callt  !_func
; line 12:  }
    pop    de                    ;为寄存器参数恢复寄存器
    ret

```

[示例 2]

<C 源代码>

```
void func5 (int,register char) ; void func();
void main ()
{
    func5 (0x1234, 0x56) ;
}
void func5 (int p1,register char p2)
{
    register char r ;
    int a ;
    r = p2 ;
    a = p1 ; func ();
}
```

当指定了 **-NQ** 选项时

<输出代码>

```

@@DATA  DSEG  UNITP
L0005 :   DS      (2)
L0006 :   DS      (2)

; line 1 :      void func5 (int,register char) ; void func(void);
; line 2 :      void main ( )
; line 3 :      {

@@CODE    CSEG
_main :
; line 4 :      func5 (0x1234, 0x56) ;
                mov     b, #056H      ; 86    ;通过寄存器 B 传输第二个参数
                movw   ax, #01234H    ; 4660  ;通过寄存器 AX 传输第一个参数
                call   !_func5        ;函数调用

; line 5 :      }
                ret

; line 6 :      void func5 (int p1,register char p2)
; line 7 :      {
_func5 :
                push   de              ;为寄存器参数保存寄存器
                                           ;寄存器参数。
                movw  !L0005, ax      ;将参数 p1 拷贝到 L0005
                mov   a, b
                mov   d, a            ;将寄存器参数 p2 分配到 d.

; line 8 :      register char r ;
; line 9 :      int a ;
; line 10 :     r = p2 ;
                mov   a, d            ;寄存器参数 p2
                mov   e, a            ;寄存器变量 r

; line 11 :     a = p1 ; func();
                movw  ax, !L0005      ; p1    ; 参数 p1
                movw  !L0006, ax     ; a     ; 自动变量 a
                call  !_func

; line 12 :     }
                pop   de              ;为寄存器参数恢复寄存器
                ret

```

11.7.6 Pascal 函数调用接口

此函数接口与其他函数接口之间的不同在于：当调用函数时，为了加载参数对堆栈进行的校正操作在被调用方进行，而不是在发起函数调用方进行。其他所有操作和函数的指定属性相同。

[为参数分配的区域]

[参数分配的顺序]

[为自动变量分配的区域]

[自动变量分配的顺序]

- 如果同时指定有 **noauto** 属性，则与调用 **noauto** 函数时的特征相同（请参阅 [11.7.3 noauto 函数调用接口](#)）。
- 如果同时未指定 **noauto** 属性，则与调用普通函数时的特征相同（请参阅 [11.7.2 普通函数调用接口](#)）。

示例 1

<C 源代码>

```
__pascal void func0 (register int, int) ;
void main ()
{
    func0 (0x1234, 0x5678) ;
}
__pascal void func0 (register int p1, int p2)
{
    register int r ;
    int a ;
    r = p2 ;
    a = p1 ;
}
```

当指定-**QR** 选项时

<输出代码>

```
_main:
; line 4 :      func0 (0x1234, 0x5678) ;
    movw    ax, #05678H          ; 22136
    push   ax                    ;堆栈通过参数传输
    movw    ax, #01234H          ; 4660      ;通过寄存器传输的第一个参数
    call   !_func0              ;函数调用
                                ;此处不校正堆栈

; line 5:      }
    ret

; line 6 :      __pascal void func0 (register int p1, int p2)
; line 7 :      {
_func0:
    push   hl
    xch    a, x
    xch    a, @_KREG12
    xch    a, x
    xch    a, @_KREG13          ;将寄存器参数 p1 分配到 @_KREG12
```

```

    push    ax                ;保存寄存器参数使用的 saddr 区域
    movw   ax, @_KREG00
    push    ax                ;保存寄存器变量使用的 saddr 区域
    push    ax                ;保留自动变量 a 区域
    movw   ax, sp
    movw   hl, ax
; line 8 :   register int r ;
; line 9 :   int a;
; line 10 :  r = p2;
    mov    a, [hl + 10]      ; p2      ;堆栈传输参数 p2
    xch    a, x
    mov    a, [hl + 11]      ; p2
    movw   @_KREG00, ax     ; r      ;指定到寄存器变量 _KREG14
; line 11 :  a = p1 ;
    movw   ax, @_KREG12     ; p1      ;寄存器参数 _KREG12
    mov    [hl + 1], a      ; a
    xch    a, x
    mov    [hl], a         ; a      ;赋值到自动变量 a
; line 12 :  }
    pop    ax                ;释放自动变量 a 区域
    pop    ax
    movw   @_KREG00, ax     ;恢复寄存器变量使用的 saddr 区域
    pop    ax
    movw   @_KREG12, ax     ;恢复寄存器参数使用的 saddr 区域
    pop    hl
    pop    de                ;获得返回地址
    pop    ax                ;因为参数传输使用堆栈, 对堆栈进行校正
    push   de                ;重新加载返回地址
    ret

```

示例 2

<C 源代码>

```

__pascal noauto void func2 (int, int) ;
void main ()
{
    func2 (0x1234, 0x5678) ;
}
__pascal noauto void func2 (int p1, int p2)
{
    .
    .
    .
}

```

当指定-QR 选项时

<输出代码>

```

_main:
; line 4:      func2 (0x1234, 0x5678) ;
      movw    ax, #05678H      ; 22136
      push   ax                ;通过堆栈传输的参数
      movw    ax, #01234H      ; 4660    ;通过寄存器传输的第一个参数
      call   !_func2          ;函数调用
                                   ;此处不校正堆栈

; line 5:      }
      ret

; line 6:      __pascal noauto void func2 (int p1, int p2)
; line 7:      {
_func2:
      push   hl                ;保存自动变量的寄存器
      xch    a, x
      xch    a, @_KREG12      ;将参数 p1 分配到_@KREG12 (低)
      xch    a, x
      xch    a, @_KREG13      ;将参数 p1 分配到_@KREG13 (高)
      push   ax                ;保存参数使用的 saddr 区域
      movw   ax, sp
      movw   hl, ax
      mov    a, [hl + 6]      ;参数 p2 (低) 通过堆栈传输且通过寄存器接收
      ;
      xch    a, x
      mov    a, [hl + 7]      ;参数 p2 (高) 通过堆栈传输且通过寄存器接收
      ;
      movw   hl, ax          ;将参数分配到 HL
      ;
      pop    ax
      movw   @_KREG12, ax    ;恢复自动变量使用的 saddr 区域
      pop    hl              ;恢复参数使用的寄存器
      pop    de              ;获得返回地址
      pop    ax              ;因为参数传输使用堆栈, 对堆栈进行校正
      push   de              ;重新加载返回地址
      ret

```

第 12 章 汇编程序的引用

本章描述如何链接用汇编语言编写的程序。

如果在 C 源程序中调用由其它编程语言编写的函数，那么这两种目标模块要通过连接器进行连接。本章就是描述在 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描述如何使用 RA78K0 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

该 C 编译器访问参数和自动变量的过程描述如下。

12.1.1 普通模式

- 在函数调用方，寄存器参数的传递方式和普通参数传递方式相同。
函数第一个参数使用如下寄存器和堆栈进行传递，后续的有关参数使用堆栈进行传递。

表 12-1. 参数传递（函数调用方）

类型	传递位置（第一个参数）	传递位置（第二个及更靠后的参数）
1 字节, 2 字节数据	AX	堆栈传递
3 字节, 4 字节数据	AX, BC	堆栈传递
浮点数	AX, BC	堆栈传递
其它数据	堆栈传递	堆栈传递

备注 1 至 4 字节的数据可以包括结构体和共用体。

- 在函数定义方，通过寄存器或堆栈传递的参数都存储在参数分配单元内。
寄存器参数拷贝到寄存器中或 **saddr** 区域（**_@KREGxx**）中。即使参数已经由寄存器完成传递，由于函数调用方（参数传递方）与函数定义方（参数接受方）的寄存器不同，所以必须要执行寄存器拷贝。
通过寄存器传递的普通参数在函数定义方被压入堆栈中，如果由堆栈进行普通参数的传递，那么传递的位置就是参数分配的单元位置。
将寄存器值保存和恢复至参数分配位置的工作是由函数定义方执行的。
- 函数的参数和在函数内部声明的自动变量的值被存储在如下寄存器中，或者使用选项可以将其存储到 **saddr** 区域或堆栈帧中。当存储在栈帧中时，使用 **HL** 寄存器作为基址指针。
如果函数参数被声明为寄存器类型，或由 **-QV** 选项和 **-QR** 选项指定函数参数时，该参数将分配至 **saddr** 区域。

表 12-2. 参数/自动变量的存储（被调用函数内）

选项	参数/自动变量	存储位置	优先级
-QV (寄存器分配选项)	声明的参数或自动变量	HL 寄存器 (仅当不需要基址指针时)	字符型: L, H, 依此顺序; int, short, enum 型: HL
-QR (saddr 分配选项)	声明为寄存器的参数或自动变量	HL 寄存器 (仅当不需要基址指针时) 参数: _@KREG12 至 15 [0FEDCH 至 0FEDFH] 自动变量: _@KREG00 至 11 [0FED0H 至 0FEDBH]	根据被引用次数, 分配的字节数量仅等于变量或参数大小。 按照字符型分配至寄存器: L, H 这样的顺序。 Int, short, enum 型: HL
-QRV	声明的参数或自动变量	HL 寄存器 (仅当不需要基址指针时) 参数: _@KREG12 至 15 [0FEDCH 至 0FEDFH] 自动变量: _@KREG00 至 11 [0FED0H 至 0FEDBH]	根据被引用次数, 分配的字节数量仅等于变量或参数大小。 按照字符型分配至寄存器: L, H 这样的顺序。 Int, short, enum 型: HL
缺省	声明的参数或自动变量	堆栈帧	依照出现的先后顺序

下面示例显示了函数的调用情况。

(C 源程序: 普通模式 并指定了**-QRV** 参数)

```
void func0 (register int, int) ;
void main () {
    func0 (0x1234, 0x5678) ;
}
void func0 (register int p1, int p2) {
    register int r;
    int a;
    r=p2;
    a=p1;
}
```

<输出的汇编源程序>

```

EXTRN    _@KREG12
EXTRN    _@KREG13
EXTRN    _@KREG00
EXTRN    _@KREG02
PUBLIC   _func0
PUBLIC   _main

@@CODE   CSEG
_main:
    movw  ax,#05678H           ;22136
    push  ax                   ;通过堆栈传递的参数
    movw  ax,#01234H           ;4660   ;通过寄存器传递第一个参数
    call  !_func0              ;函数调用
    pop   ax                   ;通过堆栈传递参数
    ret

_func0:
    push  hl                   ;保留寄存器用来传递参数
    xch   a,x
    xch   a,_@KREG12
    xch   a,x
    xch   a,_@KREG13           ;将寄存器参数 p1 分配至 _@KREG12.
    push  ax                   ;寄存器参数用 saddr 区域保存.
    movw  ax,_@KREG00
    push  ax                   ;寄存器变量用 saddr 区域保存.
    movw  ax,_@KREG02
    push  ax                   ;自动变量用 saddr 区域保存.
    movw  ax,sp
    movw  hl,ax
    mov   a,[hl+10]           ;通过堆栈传递参数 p2
    xch   a,x
    mov   a,[hl+11]
    movw  hl,ax               ;分配至 HL
    movw  ax,hl               ;参数 p2
    movw  _@KREG00,ax         ;r   ;分配至寄存器变量 r.
    movw  ax,_@KREG12         ;p1  ;寄存器参数 p1
    movw  _@KREG02,ax         ;a   ;分配至自动变量 a.
    pop   ax
    movw  _@KREG02,ax         ;用 saddr 区域恢复寄存器变量
    pop   ax
    movw  _@KREG00,ax         ;用 saddr 区域恢复自动变量
    pop   ax
    movw  _@KREG12,ax         ;用 saddr 区域恢复寄存器参数
    pop   hl                  ;恢复参数占用的寄存器
    ret
END

```

12.1.2 静态模式

- 在函数调用方，寄存器参数的传递方式和普通参数德传递方式相同。
- 通过一个寄存器就可以传递多达 3 个参数，或总数为 6 个字节的参数。

表 12-3. 参数传递（函数调用方）

类型	传递位置 (第一个参数)	传递位置 (第二个参数)	传递位置 (第三个参数)
1 字节数据	A	B	H
2 字节数据	AX	BC	HL
4 字节数据	分配到 AX 和 BC，其余分配至 H 或 HL		

备注 1 至 4 字节数据不包括结构体和共用体。

- 在函数定义方，通过寄存器传递的参数存储在参数分配单元内。声明为寄存器的参数（寄存器参数）总是尽可能分配至寄存器中，普通参数则分配至为特殊函数保留的区域中。
- 所有的寄存器参数都是通过寄存器来进行传递的，但是由于函数调用方（参数传递方）与函数定义方（参数接受方）的寄存器不同，所以必须要执行寄存器拷贝。
- 对分配给参数/自动变量的寄存器的保存和恢复工作在函数定义方进行。
- 通过使用选项，将函数参数及函数内部声明的自动变量值存储在如下所列的特殊函数区域。特殊函数区域是 RAM 中为每个函数预留的静态区域。

表 12-4. 参数/自动变量存储（被调用函数内）

选项	参数/自动变量	存储位置	优先级
-QV (寄存器分配选项)	声明的参数或自动变量	DE 寄存器	参数： 字符型：D, E 依此顺序 int, short, enum 型： DE 自动变量： 字符型：E, D 依此顺序 int, short, enum 型： DE
缺省	声明的参数，自动变量	特殊函数区域	从第一个参数起进行分配，自动变量按照其出现的先后次序进行分配
缺省	参数，声明为寄存器变量的寄存器变量	DE 寄存器	根据被引用次数，分配的字节数量仅等于变量或参数大小。 除了变量或参数必需的字节数量，其它都被分配该函数专用区域。

下面示例显示了函数的调用情况。

< C 源程序: 静态模式 - SM 和-QV 选项指定时 >

```

void sub();
void func (register int, char);
void main(){
    func (0x1234, 0x56);
}
void func (register int p1, char p2){
    register char r;
    int a;
    r=p2;
    a=p1;
    sub();
}

```

<输出的汇编源程序>

```

PUBLIC    _func
PUBLIC    _main

@@DATA    DSEG
?L0005:   DS (1)           ; 参数 p2
?L0006:   DS (1)           ; 寄存器变量 r
?L0007:   DS (2)           ; 自动变量 a

@@CODE    CSEG
_main:
    mov     b,#056H         ;86      ; 通过寄存器 B 传递第二个参数
    movw   ax,#01234H      ;4660  ; 通过寄存器 AX 传递第一个参数
    call   !_func          ;      ; 函数调用
    ret

func:
    push   de               ;      ; 为寄存器参数保留寄存器
    movw  de,ax             ;      ; 将寄存器参数 p1 分配至 DE
    mov   a,b               ;      ;
    mov   !?L0005,a         ;r      ; 将器参数 p2 拷贝至 ?L0005.
    mov   !?L0006,a         ;      ; 分配至寄存器变量 r
    movw  ax,de             ;a      ; 寄存器参数 p1
    movw  !?L0007,ax        ;      ;
    call  !_sub             ;      ;
    pop   de                ;      ; 恢复寄存器参数使用的寄存器.
    ret
END

```

12.2 返回值的存储

函数调用期间的返回值存储在寄存器中及进位标志位中。
返回值存储位置如下表所示。

表 12-5. 返回值存储位置

类型	普通模式	静态模式
1 字节整型数	BC	A
2 字节整型数		AX
4 字节整型数	BC (低字节), DE (高字节)	不支持
指针型 (不使用 bank 函数(-MF)时)	BC	AX
指针型 (使用 bank 函数(-MF)时)	BC (数据指针) BC (低端), DE (高端) (函数指针)	不支持
指针型	BC	AX
结构体, 共用体	BC (结构体或共用体的起始地址拷贝至特殊函数区域)	不支持
1 位	CY (进位标志)	CY (进位标志)
浮点数	BC (低端), DE (高端)	不支持

12.3 在 C 语言程序中调用汇编语言程序

本节显示的是普通模式（缺省）下的示例。如果指定了 **-QV** 选项，**-QR** 选项及 **-QRV** 选项，那么参数就按表 12-2 所示进行存储。然而，只有不需要基址指针（不使用基址指针时），才会分配 **HL** 寄存器。

由 C 语言程序调用汇编语言程序过程描述如下。

- [修改函数信息文件](#)
- [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 修改函数信息文件

当某个汇编源文件被分配到 **bank** 区域，必须将函数相关信息加入函数信息文件中。

本例为函数信息文件中的代码示例。

<Sample.asm 以及其中的函数 FUNC 被分配到 bank2>

```
sample.asm := 2 (0)
{
    FUNC ;
}
```

12.3.2 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 */
}
```

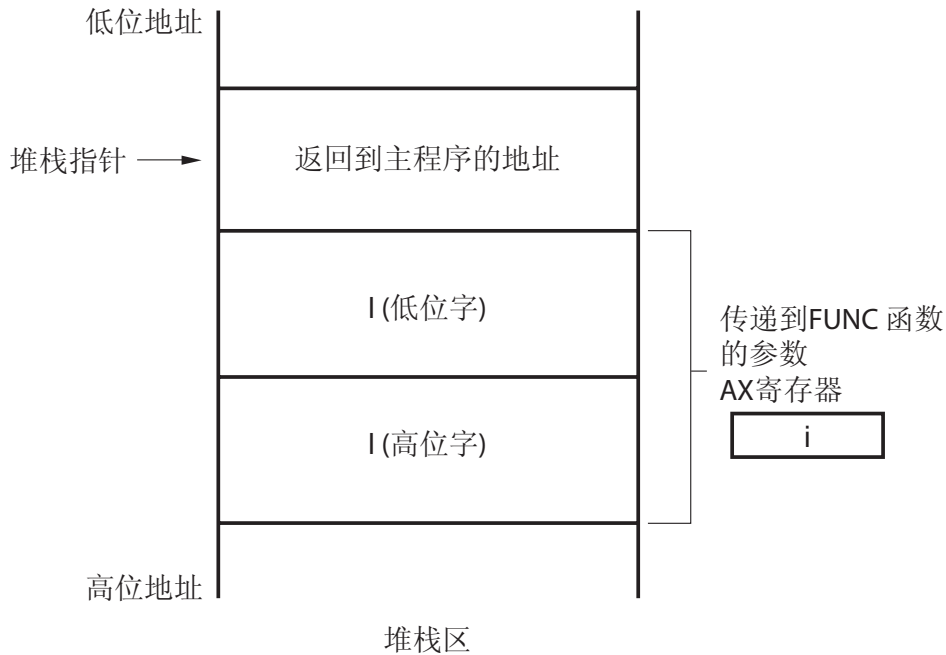
在该示例程序中，程序接口及程序执行的流程如下所述。

[调用常规汇编语言例程]

- (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。

图 12-1 调用后的堆栈区

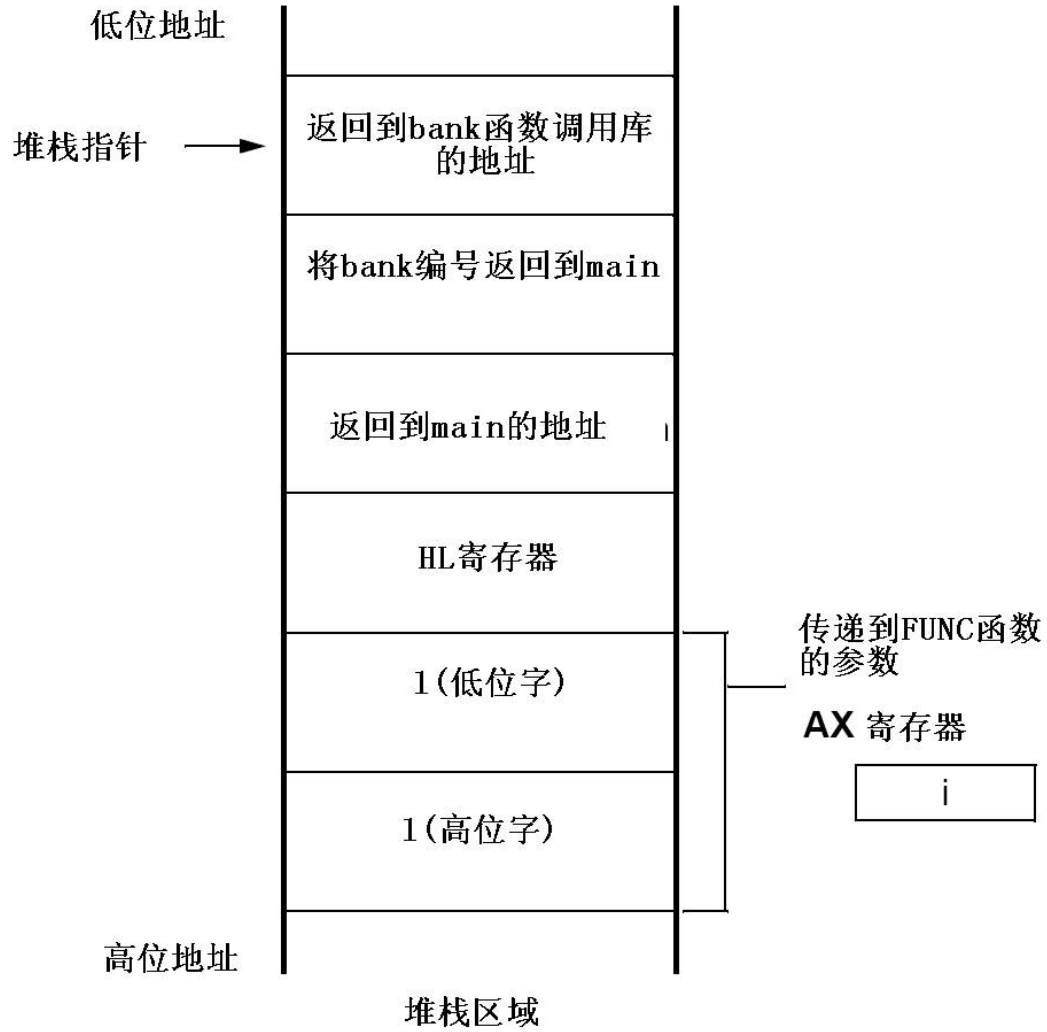


[调用位于 bank 区域的汇编语言例程]

- (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) 读入寄存器中的 **FUNC** 函数入口地址和 **bank** 编号，使用 **bank** 函数调用库将控制权移交给 **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。

图 12-2 调用后的堆栈区



12.3.3 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(054)

    PUBLIC _FUNC
    PUBLIC _DT1
    PUBLIC _DT2

@@DATA    DSEG
?DT1: DS  (2)
?DT2: DS  (4)

@@CODE    CSEG
_FUNC:
    PUSH    HL                ; 保存基址指针----- (1)
    PUSH    AX
    MOVW    AX,SP             ; 拷贝堆栈指针----- (2)
    MOVW    HL,AX
    MOV     A,[HL]            ; arg1
    XCH     A, X
    MOV     A,[HL+1]          ; arg1
    MOV     !_DT1,A           ; 传递第一个参数 (i)
    MOV     A, [HL + 8]       ; arg2 (参数在 bank 区域时偏移量加 6)
    XCH     A, X
    MOV     A, [HL + 9]       ; arg2 (参数在 bank 区域时偏移量加 6)
    MOVW    !_DT2 + 2, AX
    MOV     A, [HL + 6]       ; arg2 (参数在 bank 区域时偏移量加 6)
    XCH     A, X
    MOV     A, [HL + 7]       ; arg2 (参数在 bank 区域时偏移量加 6)
    MOVW    !_DT2, AX         ; 传递第一个参数 (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）

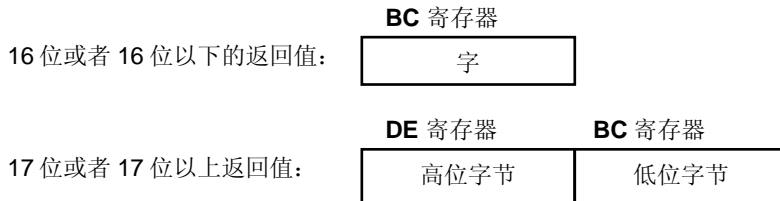
由于函数内部的‘PUSH, POP’指令改变了堆栈指针（SP）的值，因此要将堆栈指针拷贝至‘HL’寄存器，用做参数的基址指针。

(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。

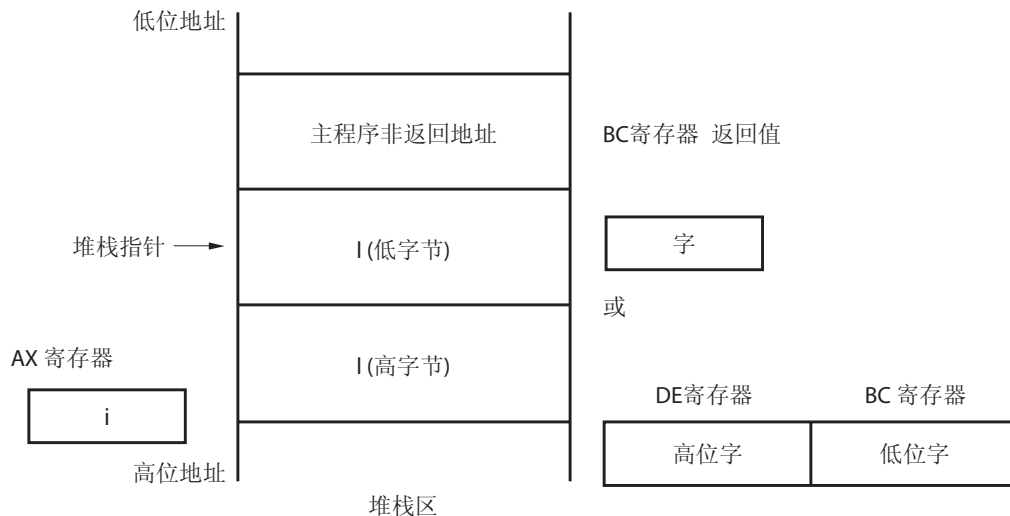


(5) 恢复寄存器值

恢复保存的基址指针和工作寄存器。

(6) 返回至主函数

图 12-3. 返回后的堆栈区



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：

[调用常规 C 语言例程]

- (1) 将参数保存到堆栈。
- (2) 保存 C 工作寄存器 (**AX**, **BC**, 及 **DE**) 。
- (3) 调用 C 语言函数。
- (4) 根据参数所占的字节数增加堆栈指针 (**SP**) 值。
- (5) 引用 C 语言函数的返回值 (在 **BC** 或 **DE** 及 **BC** 中) 。

下例为汇编语言程序示例。

```

$PROCESSOR (054)

        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**)

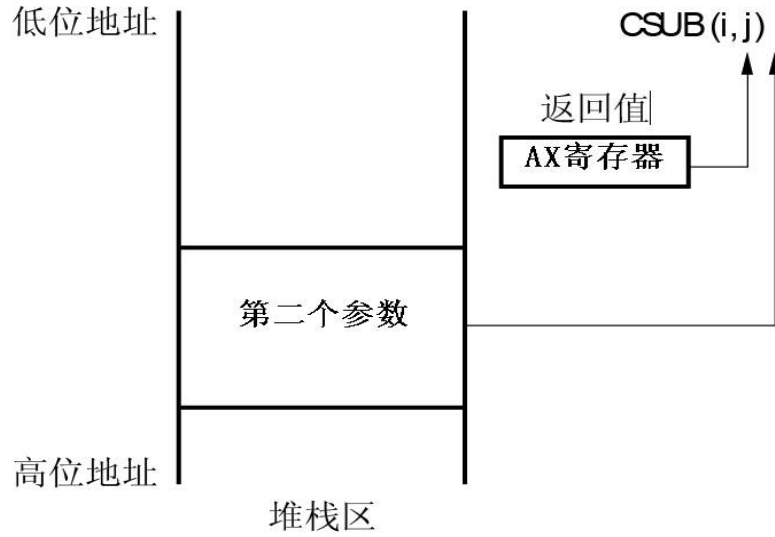
在 C 语言中使用三对寄存器 **AX**, **BC** 及 **DE**, 在调用返回时并不恢复这些寄存器的值, 因此, 如果需要使用这些寄存器中的值, 就要在函数调用方将这些值保存起来。

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。

图 12-4 参数入栈



(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。



[调用 bank 区域的 C 语言例程]

- (1) 保存 C 工作寄存器 (**AX, BC, 及 DE**) 。
- (2) 将参数保存到堆栈。
- (3) 保存 **HL** 寄存器, 并将 C 语言函数的首地址放入 **HL** 寄存器中。
- (4) 将 C 语言函数所在的 **bank** 区域编号保存到 **E** 寄存器。
- (5) 用 **callt** 指令调用 **bank** 函数调用库。
- (6) 恢复 **HL** 寄存器。
- (7) 根据参数所占的字节数增加堆栈指针 (**SP**) 值。
- (8) 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 (054)
```

```

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)
    push    hl
    movw    hl, #_CSUB         ;设置第一个参数 (i)
    movw    e, #BNAKNUM_CSUB   ;设置第一个参数 (i)
    callt   [@@bcall]         ;调用 "CSUB (i, j)"函数
    pop     hl
    pop     ax                ;
    ret
END

```

- (1) 保存工作寄存器 (**AX, BC** 和 **DE**)

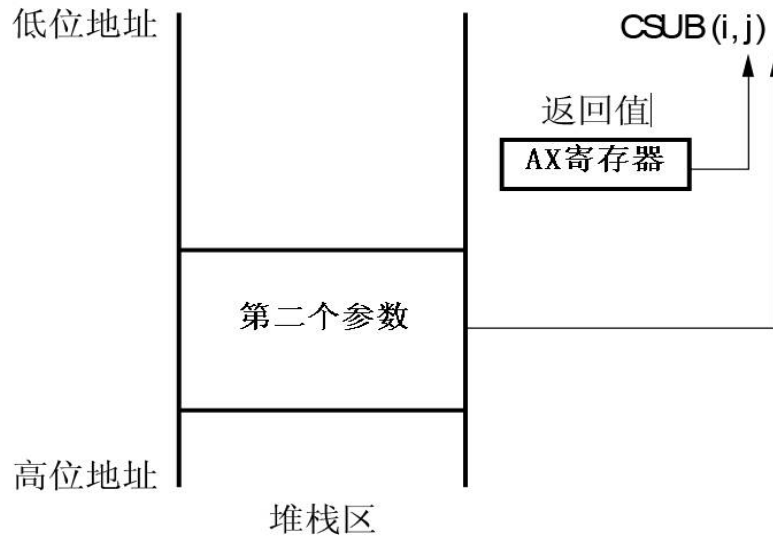
在 C 语言中使用三对寄存器 **AX, BC** 及 **DE**, 在调用返回时并不恢复这些寄存器的值。

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。在参数传递代码前后, 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。在参数之后保存的 **HL** 寄存器被放入堆栈。

- (2) 参数入栈

任何参数均放置在堆栈中。图 12-5 显示了参数传递情况。

图 12-5 参数入栈



(4) 。

(5) 用 `callt` 指令调用 `bank` 函数调用库。

(6) 恢复 HL 寄存器。

(3) 保存 HL 寄存器，并将 C 语言函数的首地址放入 HL 寄存器中

保存 HL 寄存器，并将 C 语言函数的首地址放入 HL 寄存器中，该 C 语言函数用 `bank` 函数调用库执行调用。

(4) 保存 C 语言函数的 `bank` 编号

如果有 C 语言函数通过 `bank` 函数调用库调用，将 C 语言函数所在的 `bank` 区域编号保存到 E 寄存器。

(5) 调用 `bank` 函数调用库

使用 `callt` 指令调用 `bank` 函数调用库 `@@bcall`。

(6) 恢复 HL 寄存器

恢复在步骤 (3) 中保存的 HL 寄存器。

(7) 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。

(8) 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 C 语言定义的变量

如果在汇编语言程序中引用 C 语言定义的外部变量，就必须使用 **EXTRN** 进行外部声明。要引用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 (054)
```

```
    PUBLIC _subf
    EXTRN _c
    EXTRN _i
```

```
@@CODE CSEG
```

```
_subf:
    MOV     a, #04H
    MOV     !_c, a
    MOVW   ax, #07H      ;7
    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;

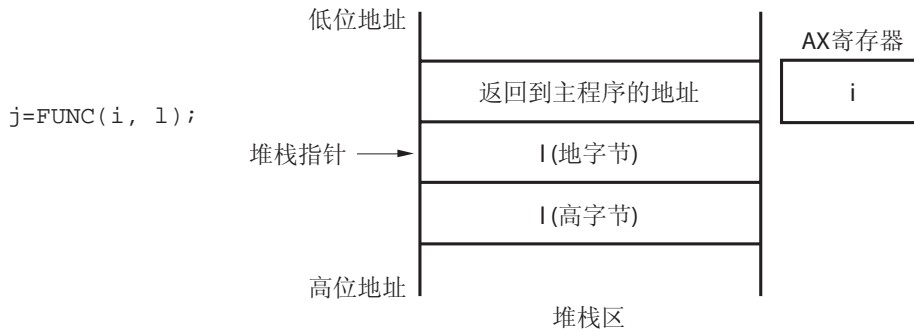
    i = 1;
    l = 0x54321;
    j = FUNC (i, l); /* 函数调用 */
}
```

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。

图 12-6 参数在堆栈中的位置



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 78K/0 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** 表或 **callf** 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 (**callf** 区可用)
 - └─ 用作 **__callf/callf** 函数 (可以有效改善执行速度)
 - └─ 如果 (没有递归调用)
 - └─ 用作 **__leaf/norec** 函数
 - └─ 如果 (没有使用自动变量)
 - └─ 用作 **noauto** 函数
 - └─ 如果 (使用自动变量 并且 **saddr** 区域可用)
 - └─ 声明为 **register** 型

(4) 优化选项

特别强调目标代码大小的最优选项如下。

<特别强调目标代码大小>

-QX3

将变量定义为 `__sreg`，可以进一步缩减代码大小并提高程序执行的速度。但是，这种方法限于 `saddr` 区域可用情况下，如果区域空间不足以及 `saddr` 区域不能用，就会出现编译错误。

如果需要更进一步提高程序执行的速度，请指定 `-QX2` 为缺省选项。

另外，还可以在 C 源程序中使用该编译器所支持的扩展函数，这样可以提高目标代码的效率。

(5) 使用扩展描述

• 函数定义



• 函数没有被递归调用

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 \(6\) norec 函数](#)和 [11.7.4 norec 函数调用接口 \(正常模式\)](#)。

• 不使用自动变量的函数

对那些不使用自动变量的函数应该将其定义为 `noauto` 函数。该函数不输出堆栈帧信息代码，而且其参数尽可能通过寄存器进行传递，这样便有助于缩短目标代码长度，提高程序执行速度。

备注 有关 `noauto` 函数定义的（`noauto int sub1 (int i) ...`），请参阅 [11.5 \(5\) noauto 函数](#)，[11.7.3 noauto 函数调用接口 \(正常模式\)](#)。

• 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，我们知道，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 \(2\) 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 \(3\) 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 INTPO 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 CC78K0 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.1 正常模式

(a) 寄存器变量

表 A-1 寄存器变量（正常模式）

标签名	地址
<code>__KREG00</code>	0FED0H
<code>__KREG01</code>	0FED1H
<code>__KREG02</code>	0FED2H
<code>__KREG03</code>	0FED3H
<code>__KREG04</code>	0FED4H
<code>__KREG05</code>	0FED5H
<code>__KREG06</code>	0FED6H
<code>__KREG07</code>	0FED7H
<code>__KREG08</code>	0FED8H
<code>__KREG09</code>	0FED9H
<code>__KREG10</code>	0FEDA H
<code>__KREG11</code>	0FEDB H ^注
<code>__KREG12</code>	0FEDC H ^注
<code>__KREG13</code>	0FEDD H ^注
<code>__KREG14</code>	0FEDE H ^注
<code>__KREG15</code>	0FEDF H ^注

注 当函数参数由 `register` 声明时，或者指定了 `-QV` 选项和 `-QR` 选项时，那么参数便被分配至 `saddr` 区域。

(b) norec 函数的参数

表 A-2 norec 函数的参数（正常模式）

标签名	地址
_ @NRARG0	0FEC0H
_ @NRARG1	0FEC2H
_ @NRARG2	0FEC4H
_ @NRARG3	0FEC6H

(c) norec 函数的自动变量

表 A-3 norec 函数的自动变量

标签名	地址
_ @NRAT00	0FEC8H
_ @NRAT01	0FEC9H
_ @NRAT02	0FECAH
_ @NRAT03	0FECBH
_ @NRAT04	0FECCH
_ @NRAT05	0FECDH
_ @NRAT06	0FECEH
_ @NRAT07	0FECFH

(d) 运行时刻的参数

表 A-4 运行时刻的参数

标签名	地址
_ @RTARG0	0FEB8H
_ @RTARG1	0FEB9H
_ @RTARG2	0FEBAH
_ @RTARG3	0FEBBH
_ @RTARG4	0FEBCH
_ @RTARG5	0FEBDH
_ @RTARG6	0FEBEH
_ @RTARG7	0FEBFH

A.2 静态模式

(a) 共享区域

表 A-5 共享区域（静态模式）

标签名	地址
_ @KREG00	0FED0H
_ @KREG01	0FED1H
_ @KREG02	0FED2H
_ @KREG03	0FED3H
_ @KREG04	0FED4H
_ @KREG05	0FED5H
_ @KREG06	0FED6H
_ @KREG07	0FED7H
_ @KREG08	0FED8H
_ @KREG09	0FED9H
_ @KREG10	0FEDA H
_ @KREG11	0FEDB H
_ @KREG12	0FEDC H
_ @KREG13	0FEDD H
_ @KREG14	0FEDE H
_ @KREG15	0FEDF H

(b) 用于参数，自动变量及工作区的标签名

表 A-6 用于参数，自动变量及工作区

标签名	地址
_ @NRAT00	0FE xx H [#]
_ @NRAT01	_ @NRAT00 + 1
_ @NRAT02	_ @NRAT00 + 2
_ @NRAT03	_ @NRAT00 + 3
_ @NRAT04	_ @NRAT00 + 4
_ @NRAT05	_ @NRAT00 + 5
_ @NRAT06	_ @NRAT00 + 6
_ @NRAT07	_ @NRAT00 + 7

注 **saddr** 区域中的任意地址

附录 B 区段名称列表

这一章说明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 40H 至 7FH 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: 0000H 至 FEFFH)。
FIXED:	将指定段的起始地址定位在 800H 至 0FFFH 范围内。
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: 80H 至 0FA7EH)。

(2) DSEG 重定位属性

SADDRP:	将指定段起始地址定位在 <code>saddr</code> 区域的 FE20H 至 FEFFH 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

表 B-1 区段名称列表 (程序区域和数据区域)

区段名称	区段类型	重定位属性	说明
@@CODE	CSEG		存放代码的区段
@@LCODE	CSEG		存放库代码的区段
@@CNST	CSEG	UNITP	const 常量段
@@R_INIT	CSEG	UNITP	初始化所需的数据区段 (具有初始值)
@@R_INIS	CSEG	UNITP	初始化所需的数据区段 (具有初始值的 sreg 变量)
@@CALF	CSEG	FIXED	callf 函数区段
@@CALT	CSEG	CALLT0	callt 函数表区段
@@VECTnn	CSEG	AT 00nnH	中断矢量表区段 ^注
@@INIT	DSEG	UNITP	数据区段 (具有初始值)
@@DATA	DSEG	UNITP	数据区段 (没有初始值)
@@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		boolean 型 (布尔型) 及 bit 型 (位型) 变量段
@@BANK0 至 @@BANK15	CSEG	BANK0 至 BANK15	bank 函数区段

注 nn 的具体数值由中断类型决定。

B.1.2 flash 存储器区域

表 B-2 区段名称列表 (flash 存储器区域)

区段名称	区段类型	重定位属性	说明
@@ECODE	CSEG		存放代码的区段
@@LECODE	CSEG		存放库代码的区段
@@ECNST	CSEG	UNITP	const 常量段
@@ER_INIT	CSEG	UNITP	初始化所需的数据区段 (具有初始值)
@@ER_INIS	CSEG	UNITP	初始化所需的数据区段 (具有初始值的 sreg 变量)
@@ECALF	CSEG	FIXED	callf 函数区段
@@ECALT	CSEG	CALLT0	callt 函数表区段
@@EVECTnn	CSEG	AT 20mmH	中断矢量表区段 ^{注1}
@@EXTxx	CSEG	AT 2yyyH	Flash 区域跳转表区段 (当指定了 -ZF 选项时) ^{注1}
@@EINIT	DSEG	UNITP	数据区段 (具有初始值)
@@EDATA	DSEG	UNITP	数据区段 (没有初始值)
@@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		boolean 型 (布尔型) 及 bit 型 (位型) 变量段

注 1 nn 和 mm 的具体数值由中断类型决定。

注 2 xx 和 yyy 的具体数值由 flash 区域函数的 ID 值决定。

B.2 区段位置

表 B-3 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.3C 源程序示例

```
#pragma INTERRUPT INTPO 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*/
{
    unsigned char uc = 0;
    uc++;
    if (b_bit)
        b_bit = 0;
}

callt void f_clt () /* callt 函数定义 */
{
}

callf void f_clf () /* callf 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 RA78K0S 在线帮助。

```

; 78K/0 系列 C 编译器 V1.30 汇编源程序
;   日期:xx xxx xxxx 时间:xx:xx:xx

; 命令 : -c014 sampk0s.c -sa -ng
; 输入文件 : sample.c
; 产生的汇编文件 : sample.asm
; 参数文件 :

$PROCESSOR(014)
$NODEBUG
$NODEBUGA
$KANJI CODE SJIS
$TOL_INF03FH, 0130H, 00H, 020H, 00H

        PUBLIC _inter
        PUBLIC _i_cnst
        PUBLIC ?f_clt
        PUBLIC _f_clf
        PUBLIC _b_bit
        PUBLIC _l_init
        PUBLIC _i_data
        PUBLIC _sr_inis
        PUBLIC _sr_dats
        PUBLIC _main
        PUBLIC _f_clt
        PUBLIC _@vect06

@@BITS BSEG ; boolean 型变量段
_b_bit DBIT

@@CNST CSEGUNITP ; const 常量区段
_i_cnst: DW 01H ; 1

@@R_INITCSEGUNITP ; 初始化所需的数据段 (具有初始值的外部变量)
DW00002H,00000H ; 2

@@INITDSEGUNITP ; 数据区段 (具有初始值的外部变量)
_l_init: DS (4)
@@DATADSEGUNITP ; 数据区段 (不具有初始值的外部变量)
_i_data: DS (2)

@@R_INISCSEGUNITP ; 初始化数据段 (具有初始值的 sreg 变量)
DW 03H ; 3
@@INISDSEGSADDRP ; 数据区段 (具有初始值的 sreg 变量)
_sr_inis: DS (2)

```

@@DATSDSEGSADDRP ;数据区段 (不具有初始值的 sreg 变量)

_sr_dats: DS (2)

@@CALTCSEGCALLT0 ;callt 函数区段

?f_clt:DW _f_clt

```
; line 1 : #pragma INTERRUPT INTP0 inter rb1 /*中断矢量*/
; line 2 :
; line 3 : void inter (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 : callf void f_clf (void); /*callf 函数原型声明*/
; line 7 : boolean b_bit; /*boolean 型变量*/
; line 8 : long l_init=2; /*具有初始值的外部变量*/
; line 9 : int i_data; /*不具有初始值的外部变量*/
; line 10 : sreg int sr_inis=3; /*具有初始值的 sreg 变量*/
; line 11 : sreg int sr_dats; /*不具有初始值的 sreg 变量*/
; line 12 :
; line 13 : void main() /*函数定义*/
; line 14 : {
```

@@CODECSEG ;代码部分的区段

_main:

```
pushhl ;[INF] 1, 4
; line 15 : int i;
; line 16 : i=100;
movw hl,#064H; 100 ;[INF] 3, 6
; line 17 : }
pop hl ;[INF] 1, 4
ret ;[INF] 1, 6
; line 18 :
; line 19 : void inter() /*中断函数定义*/
; line 19 : {
_inter:
sel RB1 ;[INF] 2, 4 选择寄存器组 1
pushhl ;[INF] 1, 4
; line 21 : unsigned char uc=0;
movl,#00H ; 0 ;[INF] 2, 4
; line 22 : uc++;
inc l ;[INF] 1, 2
; line 23 : if(b_bit)
bf _b_bit,$L0005 ;[INF] 4,10
; line 24 : b_bit=0;
clr1 _b_bit ;[INF] 2, 4
```



```

L0005:
; line 25 : }
  pop hl  ;[INF] 1, 4
  reti   ;[INF] 1, 6
; line 26 :
; line 27 : callt void f_clt() /*callt 函数定义*/
; line 28 : {
_f_clt:
; line 29 : }
  ret   ;[INF] 1, 6
; line 30 :
; line 31 : callf void f_clf ( ) /* callf 函数定义*/
; line 32 : {

@@CALF CSEG FIXED ; callf 函数区段
_f_clf :
; line 33 : }
ret   ;[INF] 1, 6

@@VECT06 CSEGAT0006H ; 中断矢量
_@vect06:
  DW _inter
  END

; *** 代码信息(Code Information) ***
;
; $FILE C:\NECTools32\work\sampk0s.c
;
; $FUNC main ( 14 )
; void = ( void )
; CODE SIZE = 6 字节 , CLOCK_SIZE = 20 clocks , STACK_SIZE = 2 字节
;
; $FUNC inter ( 20 )
; void = ( void )
; CODE SIZE = 14 字节, CLOCK_SIZE = 38 clocks , STACK_SIZE = 2 字节
;
; $FUNC f_clt ( 28 )
; void = ( void )
; CODE SIZE = 1 字节, CLOCK_SIZE = 6 clocks , STACK_SIZE = 0 字节
;
; $FUNC f_clf ( 32 )
; void = ( void )
; CODE SIZE = 1 字节, CLOCK_SIZE = 6 clocks , STACK_SIZE = 0 字节
; Target chip : uPD78014
; Device file : Vx . xx

```

附录 C 运行时刻库列表

表 C-1 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	所支持的模式		函数
		普通模式	静态模式	
加 1	lsinc	OK	-	有符号长整型数据加 1
	luinc	OK	-	无符号长整型数据加 1
	finc	OK	-	浮点型数据加 1
减 1	lsdec	OK	-	有符号长整型数据减 1
	ludec	OK	-	无符号长整型数据减 1
	fdec	OK	-	浮点型数据减 1
符号取反	lsrev	OK	-	有符号长整型数据的符号取反
	lurev	OK	-	无符号长整型数据的符号取反
	frev	OK	-	浮点型数据的符号取反
1 的补码	lscom	OK	-	得到有符号长整型数据的 1 的补码
	lucom	OK	-	得到无符号长整型数据的 1 的补码
逻辑非	lsnot	OK	-	有符号长整型数据求反
	lunot	OK	-	无符号长整型数据求反
	fnot	OK	-	浮点型数据符号求反

表 C-1. 运行时刻库列表

分类	函数名称	所支持的模式		函数
		普通模式	静态模式	
乘法	csmul	OK	OK	在有符号字符型数据之间执行乘法运算
	cumul	OK	OK	在无符号字符型数据之间执行乘法运算
	ismul	OK	OK	在有符号整型数据之间执行乘法运算
	iumul	OK	OK	在无符号整型数据之间执行乘法运算
	ismul	OK	-	在有符号长整型数据之间执行乘法运算
	lumul	OK	-	在无符号长整型数据之间执行乘法运算
	fmul	OK	-	在浮点型数据之间执行乘法运算
除法	csdiv	OK	OK	在有符号字符型数据之间执行除法运算
	cudiv	OK	OK	在无符号字符型数据之间执行除法运算
	isdiv	OK	OK	在有符号整型数据之间执行除法运算
	iudiv	OK	OK	在无符号整型数据之间执行除法运算
	lsdiv	OK	-	在有符号长整型数据之间执行除法运算
	ludiv	OK	-	在无符号长整型数据之间执行除法运算
	fdiv	OK	-	在浮点型数据之间执行除法运算
取余	csrem	OK	OK	在有符号字符型数据之间执行除法运算后, 获取余数
	curem	OK	OK	在无符号字符型数据之间执行除法运算后, 获取余数
	isrem	OK	OK	在有符号整型数据之间执行除法运算后, 获取余数
	iurem	OK	OK	在无符号整型数据之间执行除法运算后, 获取余数
	lsrem	OK	-	在有符号长整型数据之间执行除法运算后, 获取余数
	lurem	OK	-	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	OK	-	在有符号长整型数据之间执行加法运算
	luadd	OK	-	在无符号长整型数据之间执行加法运算
	fadd	OK	-	在浮点数据之间执行加法运算

表 C-1 运行时刻库列表

分类	函数名	所支持的模式		函数
		普通模式	静态模式	
减法	lssub	OK	-	在有符号长整型数据之间执行减法运算
	lusub	OK	-	在无符号长整型数据之间执行减法运算
	fsub	OK	-	在浮点型数据之间执行加碱运算
左移	lslsh	OK	-	将有符号长整型数据左移
	lulsh	OK	-	将无符号长整型数据左移
右移	lsrsh	OK	-	将有符号长整型数据右移
	lursh	OK	-	将无符号长整型数据右移
比较	cscmp	OK	OK	有符号字符型数据比较
	iscmp	OK	OK	有符号整型数据比较
	lscmp	OK	-	有符号长整型数据比较
	lucmp	OK	-	无符号长整型数据比较
	fcmp	OK	-	浮点型数据比较
按位与	lsband	OK	-	在有符号长整型数据之间执行与（AND）运算
	luband	OK	-	在无符号长整型数据之间执行与（AND）运算
按位或	lsbor	OK	-	在有符号长整型数据之间执行或（OR）运算
	lubor	OK	-	在无符号长整型数据之间执行或（OR）运算
按位异或	lsbxor	OK	-	在有符号长整型数据之间执行异或（XOR）运算
	lubxor	OK	-	在无符号长整型数据之间执行异或（XOR）运算
逻辑 与	fand	OK	-	在两个浮点数据之间执行一次逻辑与（AND）运算
逻辑 或	for	OK	-	在两个浮点数据之间执行一次逻辑或（OR）运算
浮点数的转换	ftols	OK	-	将浮点数转换成有符号长整型数
	ftolu	OK	-	将浮点数转换成无符号长整型数
转换为浮点数	lstof	OK	-	将有符号长整型数转换成浮点数
	lutof	OK	-	将无符号长整型数转换成浮点数
位数据的转换	btol	OK	-	将位型数据转换成长整型数据

表 C-1 运行时库列表

分类	函数名	所支持的模式		函数
		普通模式	静态模式	
启动例程	cstart	OK	OK	启动模块 <ul style="list-style-type: none"> 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。 <pre> @@VECT00 CSEG AT 0000H DW _@cstart </pre> 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	OK	–	函数的预处理
	cdisp	OK	–	函数的后处理
	cprep2	OK	–	函数预处理 (包括 saddr 区域 用于寄存器变量)
	cdisp2	OK	–	函数后处理(包括 saddr 区域 用于寄存器变量)

表 C-1 运行时刻库列表

分类	函数名	所支持的模式		函数
		普通模式	静态模式	
函数的前处理和 后处理	nrcp2	-	OK	用于拷贝参数
	nrcp3	-	OK	
	krcp2	-	OK	
	krcp3	-	OK	
	nkrc3	-	OK	
	nrip2	-	OK	
	nrip3	-	OK	
	krip2	-	OK	
	krip3	-	OK	
	nkri31	-	OK	
	nkri32	-	OK	
	nrsave	-	OK	
	nrload	-	OK	用于恢复 _@NRATxx
	krs02	-	OK	用于保存 _@KREGxx
	krs04	-	OK	
	krs04i	-	OK	
	krs06	-	OK	
	krs06i	-	OK	
	krs08	-	OK	
	krs08i	-	OK	
	krs10	-	OK	
	krs10i	-	OK	
	krs12	-	OK	
	krs12i	-	OK	
	krs14	-	OK	
	krs14i	-	OK	
	krs16	-	OK	
	krs16i	-	OK	

表 C-1 运行时库列表

分类	函数名	所支持的模式		函数
		普通模式	静态模式	
函数的前处理和后处理	kr102	-	OK	用于恢复_ @KREGxx
	kr104	-	OK	
	kr104i	-	OK	
	kr106	-	OK	
	kr106i	-	OK	
	kr108	-	OK	
	kr108i	-	OK	
	kr110	-	OK	
	kr110i	-	OK	
	kr112	-	OK	
	kr112i	-	OK	
	kr114	-	OK	
	kr114i	-	OK	
	kr116	-	OK	
	kr116i	-	OK	
	hdwinit	OK	OK	在 CPU 复位后，立即执行对外围设备(sfr)的初始化处理
Bank 函数	bcall	OK	-	调用 bank 函数
	bcalls	OK	-	
BCD 类型转换	bcdtob	OK	OK	将 1 个字节的 bcd 码转换成单字节 二进制 数
	btobcd	OK	OK	将 1 个字节的 二进制 数转换成 2 个字节的 bcd 码
	bcdtow	OK	OK	将 2 个字节的 bcd 码转换成 2 个字节的 二进制 数
	wtobcd	OK	OK	将 2 个字节的 二进制 数转换成 2 个字节的 bcd 码
	bbcd	OK	OK	将 1 个字节的 二进制 数转换成 1 个字节的 bcd 码

表 C-1 运行时刻库列表

分类	函数名	所支持的模式		函数
		普通模式	静态模式	
辅助函数	mulu	OK	OK	兼容 mulu 指令
	mulue	OK	OK	兼容 mulu 指令
	divuw	OK	OK	兼容 divuw 指令
	divuwe	OK	OK	兼容 divuw 指令
	addwbc	OK	OK	用来替代固定型指令模式
	clra0	OK	OK	
	clra1	OK	OK	
	clrx0	OK	OK	
	clrax0	OK	OK	
	clrax1	-	OK	
	clrbc0	OK	-	
	clrbc1	OK	-	
	cmpa0	OK	OK	
	cmpa1	OK	OK	
	cmpc0	OK	-	
	cmpax1	OK	OK	
	ctoi	OK	OK	
	maxde	OK	OK	
	mdeax	OK	OK	
	incde	OK	OK	
	decde	OK	OK	
	maxhl	OK	OK	
	mhlax	OK	OK	
	incl	OK	OK	
	dechl	OK	OK	
	dellab	OK	-	
	dell03	OK	-	
	della4	OK	-	
	delsab	OK	-	
	dels03	OK	-	

表 C-1. 运行时库列表

分类	函数名称	所支持的模式		函数
		普通模式	静态模式	
辅助函数	hlllab	OK	-	用来替代固定型指令模式
	hlll03	OK	-	
	hllla4	OK	-	
	hllsab	OK	-	
	hlls03	OK	-	
	apinch	OK	OK	
	apdech	OK	OK	
	incwhl	OK	OK	
	decwhl	OK	OK	
	shl4	OK	OK	
	shr4	OK	OK	
	swap4	OK	OK	
	tableh	OK	OK	
	utctoi	OK	OK	

附录 D 库占用的堆栈列表

表 D-1 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	普通模式	静态模式
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longjmp	2	2
stdarg.h	va_arg	0	—
	va_start	0	—
	va_starttop	0	—
	va_start_banked	0	—
	va_starttop_banked	0	—
	va_end	0	—

表 D-1 标准库堆栈占用空间列表

分类	函数名称	普通模式	静态模式
stdio.h	sprintf	52 (72) ^{#1}	—
	sscanf	290 (304) ^{#1}	—
	printf	54 (72) ^{#1}	—
	scanf	294 (304) ^{#1}	—
	vprintf	52 (72) ^{#1}	—
	vsprintf	52 (72) ^{#1}	—
	getchar	0	0
	gets	6	6
	putchar	0	0
	puts	4	4

表 D-1 标准库堆栈占用空间列表

分类	函数名称	普通模式	静态模式
stdlib.h	atoi	4	4
	atol	10	—
	strtol	20	—
	strtoul	20	—
	calloc	14	14
	free	8	8
	malloc	6	6
	realloc	10	12
	abort	0	0
	atexit	0	0
	exit	$2 + n^{\#2}$	$2 + n^{\#2}$
	abs	0	0
	div	$6 (3)^{\#3}$	—
	rand	$14 (15)^{\#3}$	—
	labs	2	—
	ldiv	14	—
	brk	0	0
	sbrk	4	4
	atof	33	—
	strtod	33	—
	itoa	10	10
	ltoa	16	—
	ultoa	16	—
	rand	14	—
	srand	0	—
	bsearch	$32 + n^{\#3}$	—
	qsort	$16 + n^{\#4}$	—
	strbrk	0	0
	strsbrk	4	4
	strtoa	10	10
	strltoa	16	—
	strultoa	16	—

表 D-1 标准库堆栈占用空间列表

分类	函数名称	普通模式	静态模式
string.h	memcpy	4	6
	memmove	4	8
	strcpy	2	4
	strncpy	4	6
	strcat	2	4
	strncat	4	6
	memcmp	2	4
	strcmp	2	2
	strncmp	2	4
	memchr	2	2
	strchr	2	0
	strcspn	6	6
	strpbrk	4	4
	strrchr	4	4
	strspn	6	6
	strstr	4	4
	strtok	4	4
	memset	4	4
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	4
math.h	acos	22	—
	asin	22	—
	atan	22	—
	atan2	23	—
	cos	24 (34) ^{‡ 5}	—
	sin	24 (34) ^{‡ 5}	—
	tan	28 (34) ^{‡ 5}	—
	cosh	24	—
	sinh	27	—
	tanh	32	—
	exp	24	—
	frexp	2 (10) ^{‡ 6}	—
	ldexp	2 (10) ^{‡ 6}	—
	log	24 (34) ^{‡ 6}	—
	log10	22 (32) ^{‡ 6}	—
	modf	2 (10) ^{‡ 6}	—

表 D-1 标准库堆栈占用空间列表

math.h	pow	26 (36) ^{#5}	—
	sqrt	16	—
	ceil	2(10) ^{#6}	—
	fabs	0	—
	floor	2(10) ^{#6}	—
	fmod	2 (10) ^{#6}	—
	matherr	0	—
	acosf	22	—
	asinf	22	—
	atanf	22	—
	atan2f	23	—
	cosf	24 (34) ^{#6}	—
	sinf	24 (34) ^{#6}	—
	tanf	28 (34) ^{#6}	—
	coshf	24	—
	sinhf	27	—

表 D-1 标准库堆栈占用空间列表

math.h	tanhf	32	—
	expf	24	—
	frexpf	$2(10)^{\#6}$	—
	ldexpf	$2(10)^{\#6}$	—
	logf	$24(34)^{\#6}$	—
	log10f	$22(32)^{\#6}$	—
	modff	$2(10)^{\#6}$	—
	powf	$26(36)^{\#6}$	—
	sqrtf	16	—
	ceilf	2	—
	fabsf	0	—
	floorf	$2(10)^{\#6}$	—
	fmodf	$2(10)^{\#6}$	—
assert.h	__assertfail	$64(82)^{\#7}$	—

- 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 $(20 + \text{qsort 函数调用的的外部函数的堆栈占用空间}) \times (1 + \text{递归调用次数})$ 。
 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	普通模式	静态模式
加 1	lsinc	0	—
	luinc	0	—
	finc	$16(26)^{\#1}$	—
减 1	lsdec	0	—
	ludec	0	—
	fdec	$16(26)^{\#1}$	—
符号取反	lsrev	0	—
	lurev	0	—
	frev	0	—
二进制反码	lscom	0	—
	lucom	0	—
逻辑非	lsnot	0	—
	lunot	0	—
	fnot	0	—

表 D-2 运行时刻库堆栈占用空间列表

乘法	csmul	2	4
	cumul	2	4
	ismul	6 (1) ^{註 2}	6
	iumul	6 (1) ^{註 2}	6
	lsmul	6 (7) ^{註 2}	—
	lumul	6 (7) ^{註 2}	—
	fmul	10 (20) ^{註 1}	—
除法	csdiv	8	8
	cudiv	2	4
	isdiv	10 (3) ^{註 2}	12
	iudiv	6 (1) ^{註 2}	6
	lsdiv	10	—
	ludiv	6	—
	fdiv	10 (20) ^{註 1}	—
取余	csrem	8	8
	curem	2	4
	isrem	10 (3) ^{註 2}	12
	iurem	6 (1) ^{註 2}	6
	lsrem	10	—
	lurem	6	—
加法	lsadd	0	—
	luadd	0	—
	fadd	10 (20) ^{註 1}	—
减法	lssub	0	—
	lusub	0	—
	fsub	10 (20) ^{註 1}	—
左移	lslsh	2	—
	lulsh	2	—
右移	lsrsh	2	—
	lursh	2	—

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	静态模式
比较	cscmp	0	2
	iscmp	0	2
	lscmp	2	—
	lucmp	2	—
	fcmp	4 (16) ^{#1}	—
按位与	lsband	0	—
	luband	0	—
按位或	lsbor	0	—
	lubor	0	—
按位异或	lsbxor	0	—
	lubxor	0	—
逻辑与	fand	0	—
逻辑或	for	0	—
浮点数的转换	ftols	8	—
	ftolu	8	—
转换为浮点数	lstof	12 (22) ^{#1}	—
	lutof	12 (22) ^{#1}	—
位数据的转换	btol	0	—
启动例程	cstart	2	2
函数的前处理和 后处理	cprep	2 + n ^{#2}	—
	cdisp	0	—
	cprep2	自动变量的空间大小 + 寄存器变量	—
	cdisp2	0	—
	nrcp2	—	0
	nrcp3	—	0
	krcp2	—	0
	krcp3	—	0
	nkrc3	—	0
	nrip2	—	0
	nrip3	—	0
	krip2	—	0
	krip3	—	0
	nkri31	—	0
	nkri32	—	0
	nrsave	—	8
	nrload	—	0

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	静态模式
函数的前处理和 后处理	krs02	—	2
	krs04	—	4
	krs04i	—	4
	krs06	—	6
	krs06i	—	6
	krs08	—	8
	krs08i	—	8
	krs10	—	10
	krs10i	—	10
	krs12	—	12
	krs12i	—	12
	krs14	—	14
	krs14i	—	14
	krs16	—	16
	krs16i	—	16
	kr102	—	0
	kr104	—	0
	kr104i	—	0
	kr106	—	0
	kr106i	—	0
	kr108	—	0
	kr108i	—	0
	kr110	—	0
	kr110i	—	0
	kr112	—	0
	kr112i	—	0
	kr114	—	0
	kr114i	—	0
	kr116	—	0
	kr116i	—	0
	hdwinit	0	0
	Bank 函数	bcall	6
bcalls		6	—
BCD 型转换	bcdtob	4	4
	btobcd	4	8
	bcdtow	4	4
	wtobcd	6	10
	bbcd	4	8

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	静态模式
Bank 函数	bcall	6	—
	bcalls	6	—
BCD 型转换	bcdtob	4	4
	btobcd	4	4
	bcdtow	4	4
	wtobcd	6	6
	bbcd	4	4
辅助函数	mulu	4	4
	mulue	4	4
	divuw	6	6
	divuwe	6	6
	addwbc	0	0
	clra0	0	0
	clra1	0	0
	clrx0	0	0
	clrax0	0	0
	clrax1	—	0
	clrbc0	0	—
	clrbc1	0	—
	cmpa0	0	0
	cmpa1	0	0
	cmpc0	0	—
	cmpax1	0	0
	maxde	0	0
	mdeax	0	0
	incde	0	0
	decde	0	0
	maxhl	0	0
	mhlax	0	0
	inchl	0	0
	dechl	0	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	静态模式
辅助函数	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	—
	shl4	0	—
	shr4	0	—
	swap4	0	—
	tableh	0	—
	utctoi	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)。
 2. 括号中的数值是为乘法/除法时使用的。
 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。

表 E-1 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	模式支持		备注
		普通模式	静态模式	
乘法	@ @ismul	75	73	执行两个有符号型整数数据的乘法
	@ @iumul	75	73	执行无符号型整数数据的乘法
	@ @ismul	85	—	执行两个有符号型长整数数据的乘法
	@ @lumul	85	—	执行两个无符号型长整数数据的乘法
除法	@ @isdiv	107	105	执行两个有符号型整数数据的除法
	@ @iudiv	85	83	执行两个无符号型整数数据的除法
取余	@ @isrem	107	105	执行两个有符号型整数数据的除法之后，获得余数。
	@ @iurem	85	83	执行两个无符号型整数数据的除法之后，获得余数。
stdlib.h	div	183	—	
	rand	85	—	使用 @ @lumul
	qsort	75	73	使用 @ @iumul

附录 F 索引

符号

??	32
# operator	159
## operator	159
#asm - #endasm	344
#define 指令	161
#include	52
#include 指令	156
#pragma access	364
#pragma asm	344
#pragma bcd	392
#pragma di	313
#pragma 指令	306
#pragma div	390
#pragma ei	357
#pragma halt	360
#pragma hromcall	348
#pragma inline	456
#pragma interrupt	347
#pragma mul	388
#pragma name	385
#pragma nop	360
#pragma realregister	442
#pragma rot	386
#pragma section	373
#pragma sfr	331
#pragma stop	360
#pragma vect	347
\a	31
\b	31
.....	31
\f	31
.....	31
\n	31
.....	31
\r	31
.....	31
\t	31
.....	31
\v	31
.....	31

A

abort	228
abs	230
绝对地址访问函数	27, 364
绝对地址定位规定	26, 459
acos	258
acosf	281
聚合类型	44
ANSI	307
算术运算	93
数组	138
数组声明	62

数组偏移量计算的简单方法	28, 440
数组类型	44
asin	259
asinf	282
__asm	344
ASM 语句	26, 344
汇编语言	16
assert	191
__assertfail	303
赋值运算符	110
atan	260
atan2	261
atan2f	284
atanf	283
atexit	192, 229
atof	192, 233
atoi	221
atol	221
auto	55
函数调用接口的自动 pascal 函数化	28, 427

B

Bank 函数	27, 396
常量地址上的 bank 函数	27
基址变址寻址使用方法	29, 446
BCD 码运算函数	27, 392
二进制常数	27, 383
位域	367
位域声明	27, 367
位变量	26, 340
按位与运算符	102
按位或运算符	104
按位异或运算符	103
块范围	35
__boolean	340
布尔型变量	26, 340
分支语句	117
break 语句	135
BRK	360
brk	192, 232
bsearch	237

C

C 语言	16
__callf	362
callf 函数	27, 362
calloc	224
callt/__callt	317
callt/__callt 函数	25
callt 函数	317
Cast 运算符	91
ceil	276

- ceilf 299
 改变编译器输出的段名 27
 如何改变编译器输出的段名 373
 char 型 40
 character 常量 48
 character 型 43
 逗号运算符 113
 注释 53
 兼容类型 45
 合成类型 45
 复合赋值 112
 符合语句 117
 const 61
 常量 46
 常量表达式 115
 continue 语句 134
 cos 262
 cosf 285
 cosh 265
 coshf 288
 CPU 控制指令 26, 360
 ctype 178
- ### D
- 数据插入函数 27, 405
 __DATE 168
 十进制常量 47
 分隔符 51
 DI 357
 __directmap 459
 div 192, 231
 除法函数 27, 390
 do 语句 130
- ### E
- EI 357
 357
 枚举型常量 47
 枚举型 59
 枚举型标识符 40
 等值运算符 100
 errno 184
 error 184
 ESCAPE 序列 31
 exit 192, 229
 exp 268
 expf 291
 表达式语句 117
 extern 55
 外部定义 143
 外部链接 36
 外部目标定义 146
 ext_tsk 416
- ### F
- fabsf 300
 __FILE 168
 文件范围 35
 固件 ROM 函数 28, 436
 __flash 436
 flash 区域分配方法 28, 429
 __flashf 函数 29, 452
 浮点 188
 浮点常量 46
 浮点型 41
 floor 函数 278
 fmod 279
 fmodf 302
 for 语句 131
 free 函数 225
 frexp 269
 frexpf 292
 函数 20
 函数声明 63
 函数定义 144
 从 boot 区域到 flash 区域的函数调用 28, 432
 函数原型范围 35
 函数范围 35
 函数类型 44
- ### G
- 一般整型提升 73
 getchar 217
 gets 函数 218
 goto 语句 133
- ### H
- HALT 360
 头名称 52
 十六进制常量 47, 446, 29
 如何使用 saddr 区域 324
 如何使用 sfr 区域 331
- ### I
- 标识符 36
 if ... else 语句 126
 if 语句 126
 非完整类型 44
 整型常量 46
 整数类型 40
 内部链接 36
 __interrupt 355
 中断函数 26, 347, 357
 中断函数修饰词 26, 355
 RTOS 的中断函数 27, 407
 RTOS 的中断函数修饰词 28, 414
 __interrupt_brk 355
 isalnum 198
 isalpha 198
 isascii 198
 iscntrl 198
 isdigit 198
 isgraph 198
 islower 198
 isprint 198
 ispunct 198

isspace..... 198
 isupper..... 198
 isxdigit..... 198
 迭代语句..... 117
 itoa..... 235

K

关键字..... 33

L

标签语句..... 117
 labs..... 230
 ldexp..... 270
 ldexpf..... 293
 ldiv..... 192, 231
 支持序言/尾声的库..... 29, 473
 limits..... 185
 __LINE__..... 168
 log..... 271
 log10..... 272
 log10f..... 295
 logf..... 294
 逻辑与运算符..... 106
 逻辑或运算符..... 107
 longjmp..... 192, 202
 ltoa..... 235

M

机器语言..... 16
 Macro 名..... 168
 Macro 替换..... 159
 malloc..... 226
 math..... 186
 matherr..... 280
 memchr..... 247
 memcmp..... 245
 memcpy 函数..... 242
 memmove 函数..... 242
 存储器操作函数..... 29, 456
 存储空间..... 311
 memset..... 253
 参数/返回值的 int 扩展限制方法..... 28, 437
 modf..... 273
 modff..... 296
 模块名更改函数..... 385
 乘法函数..... 27, 388

N

空链接..... 36
 noauto 函数..... 26, 333
 NOP..... 360
 norec/ __leaf 函数..... 26
 norec 函数..... 336

O

对象类型..... 38
 八进制常量..... 47
 片上固件自编程子程序直接调用函数..... 29, 448
 __OPC..... 405

P

__Pascal..... 425
 Pascal 函数..... 28, 425
 Pascal 函数调用接口..... 498
 peekb..... 364
 peekw..... 364
 指针..... 138
 指针声明符..... 62
 pokeb..... 364
 pokew..... 364
 后缀操作符函数..... 80
 pow..... 274
 powf..... 297
 预处理指令..... 147
 printf..... 192, 213
 putchar..... 219
 puts..... 220

Q

__QE..... 446
 __QL 选项..... 317
 qsort..... 238
 -QW2..... 440
 -QW3..... 440

R

rand..... 192, 236
 realloc..... 227
 重入..... 192
 寄存器..... 55, 320
 寄存器组..... 311
 指定寄存器组..... 347
 寄存器直接访问函数..... 29, 442
 寄存器变量..... 25, 320
 关系运算符..... 98
 return 语句..... 136
 rolb..... 386
 rolw..... 386
 rorb..... 386
 rorw..... 386
 移位函数..... 27, 386
 RTOS..... 307
 __rtos_interrupt 修饰词..... 414

S

sbrk..... 192, 232
 标量类型..... 45
 scanf..... 192, 214
 ROMization 相关的区段名称..... 379
 Selection 语句..... 117
 setjmp..... 179, 192, 202
 sfr 区域..... 26
 sfr 变量..... 331
 移位运算符..... 96
 有符号整型..... 40
 简化赋值..... 111
 sin..... 263
 sinf..... 286

sinh 266
sinhf 289
sprintf 192, 205
sqrt 275
sqrtf 298
srand 192, 236
sreg 声明 324
sscanf 192, 209
堆栈更改设置 349
启动例程 379
启动例程 304
静态标识符 55
静态模式 28
静态模式扩展设置 29, 462
stdarg 180
__STDC__ 168
stddef 186
stdlib 181
STOP 360
存储类型标识符 55
strbrk 239
strcat 244
strchr 248
strcmp 246
strcoll 256
strcpy 243
strcspn 249
string 183
字符串文字 49
strtoa 241
strlen 255
strltoa 241
strncat 244
strncmp 246
strncpy 243
strpbrk 250
strrchr 248
strsbrk 240
strspn 249
strstr 251
strtod 192, 233
strtok 192, 252
strtol 222
strtol 222
struct 137
结构 137
结构指针 138
结构标识符 57
结构类型 44
结构变量 137
strutoa 241
strxfrm 257
switch 语句 127

T

标记 60
tan 264
tanf 287
tanh 267
tanhf 290
任务 416
RTOS 的任务函数 28, 416
__temp 470
临时变量 29, 470
__TIME__ 168
toascii 200
tolower 201
_tolower 199
tolower 199
toup 201
_toupper 201
toupper 199
三字符序列 32
类型修改 28, 423
Type 名称 64
Type 标识符 56
typedef 55

U

ultoa 235
单目运算符 86
union 140
联合类型 44
无符号整型 40
saddr 区域使用方法 25

V

va_arg 203
va_end 203
va_start 203
va_start_banked 203
va_starttop_banked 203
void 75
空指针 75
volatile 61
vprintf 192, 215
vsprintf 192, 216

W

while 语句 129

Z

-ZB 437
-ZD 473
-ZF 428
-ZI 423
-ZM 462
-ZR 427

