致尊敬的顾客

关于产品目录等资料中的旧公司名称

NEC电子公司与株式会社瑞萨科技于2010年4月1日进行业务整合（合并），整合后的新公司暨“瑞萨电子公司”继承两家公司的所有业务。因此，本资料中虽还保留有旧公司名称等标识，但是并不妨碍本资料的有效性，敬请谅解。

瑞萨电子公司网址：http://www.renesas.com

2010年4月1日
瑞萨电子公司

【发行】瑞萨电子公司（http://www.renesas.com）

【业务咨询】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.

User's Manual

H8S、 H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序

编译程序包 Ver.6.01

瑞萨单片机开发环境系统

瑞萨电子
www.renesas.com

Rev.1.00　2005.10

Keep safety first in your circuit designs!

1. Renesas Technology Corp. puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corp. product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corp. or a third party.
2. Renesas Technology Corp. assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corp. without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corp. assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corp. by various means, including the Renesas Technology Corp. Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corp. assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corp. semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corp. is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corp. for further details on these materials or the products contained therein.

注意

本文只是参考译文，前页所载英文具有正式效力。

请遵循安全第一进行电路设计

1. 虽然瑞萨科技尽力提高半导体产品的质量和可靠性，但是半导体产品也可能发生故障。半导体的故障可能导致人身伤害、火灾事故以及财产损害。在电路设计时，请充分考虑安全性，采用合适的如冗余设计、利用非易燃材料以及故障或者事故防止等的安全设计方法。

关于利用本资料时的注意事项

1. 本资料是为了让用户根据用途选择合适的瑞萨科技产品的参考资料，不转让属于瑞萨科技或者第三者所有的知识产权和其它权利的许可。
2. 对于因使用本资料所记载的产品数据、图、表、程序、算法以及其它应用电路的例子而引起的损害或者对第三者的权力的侵犯，瑞萨科技不承担责任。
3. 本资料所记载的产品数据、图、表、程序、算法以及其它所有信息均为本资料发行时的信息，由于改进产品或者其它原因，本资料记载的信息可能变动，恕不另行通知。在购买本资料所记载的产品时，请预先向瑞萨科技或者经授权的瑞萨科技产品经销商确认最新信息。
   本资料所记载的信息可能存在技术不准确或者印刷错误。因这些错误而引起的损害、责任问题或者其它损失，瑞萨科技不承担责任。
   同时也请通过各种方式注意瑞萨科技公布的信息，包括瑞萨科技半导体网站。
   （http：//www.renesas.com）
4. 在使用本资料所记载部分或者全部数据、图、表、程序以及算法等信息时，在最终做出有关信息和产品是否适用的判断前，务必对作为整个系统的所有信息进行评价。由于本资料所记载的信息而引起的损害、责任问题或者其它损失，瑞萨科技不承担责任。
5. 瑞萨科技的半导体产品不是为在可能和人命相关的环境下使用的设备或者系统而设计和制造的产品。在研讨将本资料所记载的产品用于运输、交通车辆、医疗、航空宇宙用、原子能控制、海底中继器的设备或者系统等特殊用途时，请与瑞萨科技或者经授权的瑞萨产品经销商联系。
6. 未经瑞萨科技的书面许可，不得翻印或者复制全部或者部分资料的内容。
7. 如果本资料所记载的某产品或者技术内容受日本出口管理限制，必须在得到日本政府的有关部门许可后才能出口，并且不准进口到批准目的地国家以外的国家。
   禁止违反日本和（或者）目的地国家的出口管理法和法规的任何转卖、挪用或者再出口。
8. 如果需要了解本资料所记载的信息或者产品的详细，请与瑞萨科技联系。

前　言

本手册叙述了“H8S,H8/300 系列 C/C++编译程序、汇编程序、优化连接编辑程序”的使用方法。

本产品是将 C、C++语言以及汇编语言记述的源程序转换成 H8SX 系列、H8S/2600 系列、H8S/2000 系列、H8/300 系列、H8/300 系列或者 H8/300L 系列的目标程序和装入模块的软件系统。

在使用前，请务必仔细阅读并理解用户手册。

■记载上的注意事项

在本手册的说明中使用的符号表示以下的含义：

〈　〉	表示指定此符号括起来的内容。
［　］	表示能省略的项目。
…	表示前述的项目被重复指定。
△	表示1 个以上的空格。
(RET)	表示回车（也称为回车键）。
｜	表示能选择用“|”分隔的项目。
(CNTL)	表示在按下 CNTL 键的同时输入后面的字符。

本手册对应在 UNIX*^{1}、PC-9801*^{2} 系列、IBM PC*^{3} 及其兼容机上运行的 Microsoft® Windows® 98 operating system、Microsoft® Windows® Millennium Edition operating system、 Microsoft® Windows NT® operating system、Microsoft® Windows® 2000 operating system、Microsoft® Windows® XP operating system*^{4}。UNIX 上运行的软件系统称为 UNIX 版，在 PC-9801 系列、IBM PC 及其兼容机上运行的软件系统称为 PC 版。

【注】*1　UNIX 是 X/Open 公司在美国及其它国家的注册商标。
*2　PC-9801 是日本电气株式会社的商标。
*3　IBM PC 是美国 International Business Machines Corporation 的注册商标。
*4　Microsoft®, Windows®, Windows NT®是美米国 Microsoft Corporation 在美国及及其它国家的注册商标或者商标。

目　录

第 1 章　概要 1
- 1.1　程序的开发步骤 1
- 1.2　编译程序的概要 2
- 1.3　汇编程序的概要 2
- 1.4　优化连接编辑程序的概要 2
- 1.5　预连接程序的概要 3
- 1.6　标准库生成工具的概要 3
- 1.7　栈解析工具的概要 3
- 1.8　格式转换程序的概要 3

第 2 章　C/C++编译程序的操作方法 5
- 2.1　选项指定规则 5
- 2.2　选项解说 5
  - 2.2.1　源选项 5
  - 2.2.2　目标选项 9
  - 2.2.3　列表选项 20
  - 2.2.4　优化选项 22
  - 2.2.5　其它选项 36
  - 2.2.6　CPU 选项 43
  - 2.2.7　剩余选项 52

第 3 章　汇编程序的操作方法 55
- 3.1　选项指定规则 55
- 3.2　选项解说 55
  - 3.2.1　源选项 55
  - 3.2.2　目标选项 60
  - 3.2.3　列表选项 66
  - 3.2.4　调整选项 71
  - 3.2.5　其它选项 73
  - 3.2.6　CPU 选项 74
  - 3.2.7　剩余选项 79

第 4 章　优化连接编辑程序的操作方法 85
- 4.1　选项指定规则 85
  - 4.1.1　命令行的格式 85
  - 4.1.2　子命令文件的格式 85

4.2 选项解说 86
4.2.1 输入选项 86
4.2.2 输出选项 90
4.2.3 列表选项 98
4.2.4 优化选项 100
4.2.5 段选项 105
4.2.6 验证选项 107
4.2.7 其它选项 108
4.2.8 子命令文件选项 115
4.2.9 CPU 选项 116
4.2.10 剩余选项 117
第 5 章 标准库生成工具的操作方法 119
5.1 选项指定规则 119
5.1.1 命令行的格式 119
5.2 选项解说 119
5.2.1 追加选项 119
5.2.2 不可指定的选项 122
5.2.3 指定选项时的注意事项 123
第 6 章 栈解析工具的操作方法 125
6.1 栈解析工具的启动 125
第 7 章 环境变量 127
7.1 环境变量一览表 127
7.2 编译程序的隐式声明 129
第 8 章 文件规格 131
8.1 文件名的命名方法 131
8.2 编译列表的参照方法 132
8.2.1 编译列表的结构 132
8.2.2 源列表信息 132
8.2.3 错误信息 134
8.2.4 符号分配信息 135
8.2.5 目标信息 137
8.2.6 统计信息 139
8.3 汇编列表的参照方法 140
8.3.1 汇编列表的结构 140
8.3.2 源列表信息 141
8.3.3 参照表 142
8.3.4 段信息列表 143
8.4 连接列表的参照方法 144

8.4.1 连接列表的结构 144
8.4.2 选项信息 144
8.4.3 错误信息 145
8.4.4 连接映像信息 145
8.4.5 符号信息 146
8.4.6 符号删除优化信息 147
8.4.7 变量存取优化对象符号信息 147
8.4.8 函数存取优化对象符号信息 148
8.4.9 参照表信息 149
8.5 库列表的参照方法 150
8.5.1 库列表的结构 150
8.5.2 选项信息 150
8.5.3 错误信息 151
8.5.4 库信息 151
8.5.5 库内模块、段、符号信息 152

第 9 章 编程 153

9.1 程序结构 153
9.1.1 段 153
9.1.2 C/C++程序的段 154
9.1.3 汇编程序的段 157
9.1.4 段的连接 158
9.2 初始设定程序的建立 161
9.2.1 存储区的分配 162
9.2.2 执行环境的设定 170
9.3 C/C++程序和汇编程序的连接 199
9.3.1 外部名的相互参照方法 199
9.3.2 函数调用的接口 200
9.3.3 参数分配的具体例子 208
9.3.4 寄存器和堆栈区的使用方法 215
9.4 程序建立时的注意事项 220
9.4.1 编程时的注意事项 220
9.4.2 用 C++编译程序编译 C 程序时的注意事项 223
9.4.3 程序开发时的注意事项 224

第 10 章 C/C++语言规格 225

10.1 语言规格 225
10.1.1 编译程序的规格 225
10.1.2 数据的内部表现 230
10.1.3 浮点数的规格 239
10.1.4 运算符的评价顺序 245

10.2 扩展功能......246
10.2.1 #pragma 扩展名和关键字......246
10.2.2 段地址运算符......279
10.2.3 内部函数......280
10.3 C/C++库......307
10.3.1 标准 C 库......307
10.3.2 嵌入式 C++类库......423
10.3.3 可重入库......493
10.3.4 未支持库......495
第 11 章 汇编程序的语言规格......497
11.1 程序的元素......497
11.1.1 源语句......497
11.1.2 保留字......499
11.1.3 符号......500
11.1.4 常数......502
11.1.5 定位计数器......503
11.1.6 表达式......504
11.1.7 字符串......508
11.1.8 局部标号......508
11.2 执行指令......510
11.2.1 执行指令的概要......510
11.2.2 有关执行指令的注意事项......512
11.3 汇编程序控制指令......522
11.4 文件包含功能......567
11.5 条件汇编功能......569
11.5.1 条件汇编功能的概要......569
11.5.2 有关条件汇编功能的控制语句......574
11.6 宏功能......584
11.6.1 宏功能的概要......584
11.6.2 有关宏功能的控制语句......586
11.6.3 宏体......589
11.6.4 宏调用......592
11.6.5 字符串操作函数......594
11.7 结构化汇编功能......597
11.7.1 有关结构化汇编功能的注意事项......598
11.7.2 有关结构化汇编功能的控制语句......599
第 12 章 编译程序的错误信息......615
12.1 错误格式和错误级......615
12.2 信息一览......615

12.3 C库函数的错误信息 686

第 13 章 汇编程序的错误信息 689

13.1 错误格式和错误级 689
13.2 信息一览 689

第 14 章 优化连接编辑程序的错误信息 705

14.1 错误格式和错误级 705
14.2 信息一览 705

第 15 章 标准库生成工具/格式转换程序的错误信息 719

15.1 错误格式和错误级 719
15.2 信息一览 719

第 16 章 极限值 723

16.1 编译程序的极限值 723
16.2 汇编程序的极限值 725

第 17 章 面向 AE5 的功能支持 727

17.1 编译程序的功能 727
  17.1.1 概要 727
  17.1.2 编译程序选项 728
  17.1.3 内部函数 729
17.2 汇编程序的功能 732

第 18 章 版本升级的注意事项 733

18.1 版本升级时的注意事项 733
  18.1.1 程序的运行保证 733
  18.1.2 与旧版本的兼容性 734
  18.1.3 命令行接口 737
  18.1.4 提供的内容 739
  18.1.5 列表文件规格 739
18.2 追加/改善的内容 740
  18.2.1 共同的追加/改善 740
  18.2.2 编译程序的追加/改善 740
  18.2.3 汇编程序的追加/改善的功能 750
  18.2.4 优化连接编辑程序追加/改善的功能 751
18.3 格式转换程序的操作方法 753
  18.3.1 目标文件的格式 753
  18.3.2 与旧版本的兼容性 753
  18.3.3 选项指定规则 754
  18.3.4 选项解说 754

第 19 章　附录 757
19.1　S型、HEX文件格式 757
19.1.1　S 型文件格式 757
19.1.2　HEX 文件格式 759
19.2　ASCII码一览表 761
19.3　短绝对地址的存取范围 762

索引 763

第 1 章　概要

1.1　程序的开发步骤

程序的开发步骤如图 1.1 所示。阴影部分表示“Renesas C/C++ Compiler Package for H8,H8S and H8SX family”提供的软件。

本手册说明 C/C++编译程序、汇编程序、优化连接编辑程序、标准库生成工具、栈解析工具和格式转换程序。

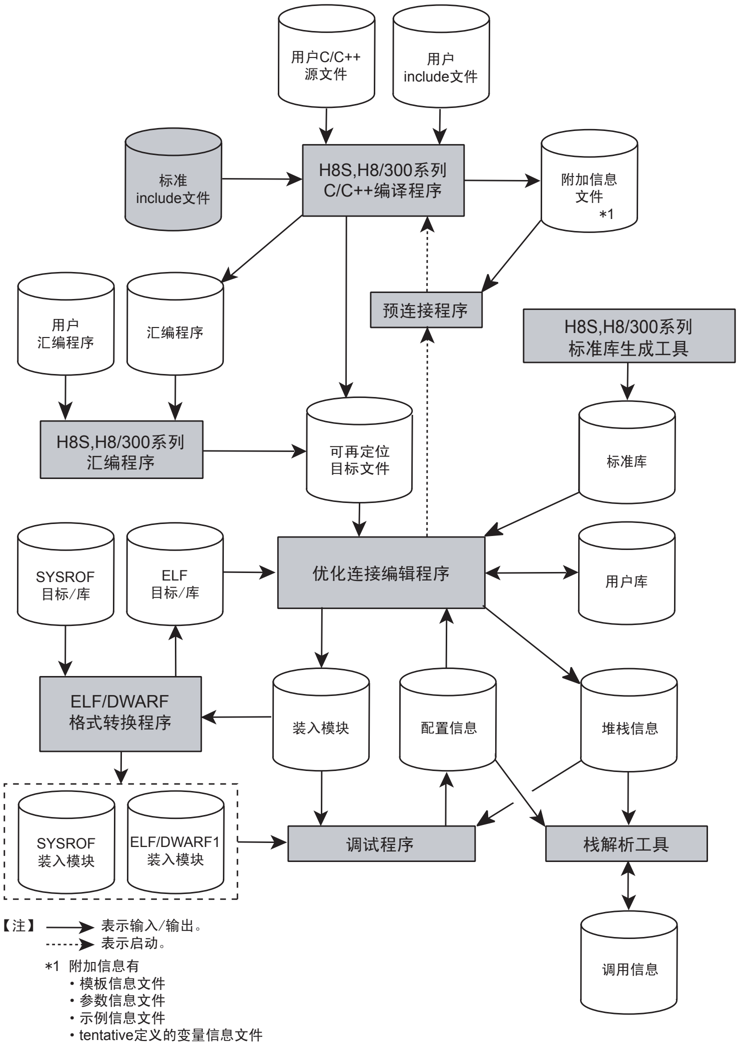


图 1.1　程序的开发步骤

以下阐述 C/C++编译程序、汇编程序、优化连接编辑程序、预连接程序、标准库生成工具、栈解析工具和格式转换程序的概要。

1.2　编译程序的概要

“H8S、H8/300 系列 C/C++编译程序”（以下简称为编译程序）对由 C 语言或者 C++语言记述的源程序进行 H8S、H8/300 系列的可再定位目标程序或者汇编源程序的生成。

本编译程序有以下特点：

(1) 生成可用于嵌入式系统且可进行 ROM 化的目标程序。

(2) 支持为了提高目标程序的执行速度和减小程序大小的优化功能。

(3) 支持为了有效使用 CPU 的短绝对寻址方式和间接寻址方式等功能的扩展功能和选项。

(4) 作为程序描述语言，支持 C 语言和 C++语言。

(5) 支持非 C/C++言語标准的中断函数和系统指令描述等编制嵌入式程序所必需的扩展功能。

(6) 能指定由调试程序进行 C/C++源级调试的调试信息输出。

(7) 能选择输出汇编源程序或者可再定位目标程序。

(8) 能指定由优化连接编辑程序进行优化的模块间优化信息输出。

1.3　汇编程序的概要

“H8S、H8/300 系列汇编程序”（以下简称为汇编程序）对由汇编语言记述的源程序进行 H8S、H8/300 系列的可再定位目标程序的生成。

本汇编程序有以下特点：

(1) 能通过如下所示的预处理功能高效率地记述源程序：

- 文件包含功能
- 带条件的汇编功能
- 宏功能
- 结构化汇编功能

(2) 执行指令和汇编程序控制指令的名称（助记符）符合 IEEE-694 规格规定的命名规则，形成统一的体系。

1.4　优化连接编辑程序的概要

“优化连接编辑程序”对由编译程序和汇编程序输出的多个目标程序进行装入模块或者库文件的生成。

本优化连接编辑程序有以下特点：

(1) 对编译程序不能优化的多个目标程序间的存储器配置和函数调用关系进行优化。

(2) 能选择输出以下 5 种装入模块：

- 可再定位 ELF 格式
- 绝对 ELF 格式
- S 型格式
- HEX 格式
- 二进制格式

(3) 能建立和编辑库文件。

(4) 能输出符号参照次数列表。

(5) 能删除库文件和装入模块文件的调试信息。

(6) 能指定栈解析工具使用的堆栈信息文件的输出。

1.5　预连接程序的概要

“预连接程序”由优化连接编辑程序调用，在使用 C++程序模板和运行时型检查功能时，启动编译程序并生成需要的目标文件。

通常不必在意预连接程序，但是，在不使用C++程序模板和运行时型信息时，能通过指定优化连接编辑程序的noprelink选项抑制预连接程序的启动，提高连接速度。

1.6　标准库生成工具的概要

“H8S、H8/300 系列标准库生成工具”（以下简称为标准库生成工具）是通过用户指定选项建立编译程序提供的标准库文件的软件系统。

本编译程序提供的标准库函数有 C 库函数群、面向嵌入式系统的 C++类库函数群以及运行时例程群（程序执行所必需的算术运算）。必须注意：即使在源程序中不使用库函数，有时也需要运行时例程。

1.7　栈解析工具的概要

“栈解析工具”对由优化连接编辑程序输出的堆栈信息文件进行程序堆栈使用量的计算。

1.8　格式转换程序的概要

“ELF/DWARF 格式转换程序”（以下简称为格式转换程序）将旧版本的编译程序、汇编程序输出的目标文件和库文件转换为 ELF 格式，或者将绝对 ELF 格式的装入模块转换为旧版本的连接编辑程序的输出格式。

第 2 章　C/C++编译程序的操作方法

2.1　选项指定规则

启动编译程序的命令行的格式如下：

```
ch38  [△<选项> … ] [△<文件名>[△<选项> …] …]
  <选项>：－<选项> [=<子选项>][, …]
```

2.2　选项解说

命令行格式的英文大写字母表示缩写指定时的字母，其下划线表示省略时的解释。
另外，用选项卡名<种类名>[项目]…表示对应综合开发环境的对话菜单。
选项的记述顺序对应综合开发环境的选项卡及其种类。

2.2.1　源选项

表 2.1　源种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	include 文件目录	Include = <路径名>[,…]	C/C++ <Source> [Show entries for :] [Include file directories]	指定 include 文件的路径名
2	默认 include 文件	PREInclude = <文件名>[,…]	C/C++ <Source> [Show entries for :] [Preinclude files]	将所指定的文件包含在编译单位的起始位置
3	宏名的定义	DEFine = <sub>[,…] <sub>: <宏名>[=<字符串>]	C/C++ <Source> [Show entries for :] [Defines]	将<字符串>定义为<宏名>
4	信息消息	Message NOMessage [= <错误号> [-<错误号>][,…]]	C/C++ <Source> [Show entries for :] [Messages] [Display information level messages]	输出 不输出 （可指定错误号和范围）
5	文件间直接插入展开目录的指定	FILE_INLINE_PATH= <路径名>[,...]	C/C++ <Source> [Show entries for :] [File inline path]	指定文件间直接插入展开文件的路径名

include 文件目录

Include

C/C++ <Source>[Show entries for :][Include file directories]

格　式　　`Include = <路径名>[,…]`

说　明　　指定 include 文件的路径名。
有多个路径名时，能用逗号（,）分开指定。
按照 include 选项的指定目录和环境变量 CH38 的指定目录顺序进行系统 include 文件的检索。
按照当前目录、include 选项指定目录和环境变量 CH38 指定目录的顺序进行用户 include 文件的检索。

例

```
ch38 –include=c:\usr\inc,c: \usr\CH38 test.c
```

将目录 c:\usr\inc 和 c:\usr\CH38 作为 include 文件路径进行检索。

默认 include 文件

PREInclude

C/C++ <Source>[Show entries for :][Preinclude files]

格　式　　`PREInclude = <文件名>[,…]`

说　明　　将所指定的文件内容取到编译单位的起始位置。有多个文件名时，能用逗号（,）分开指定。

例

```
ch38 –preinclude=a.h test.c
```

<test.c>的内容

```
int a;
main() {  … }
```

编译时的解释

```
#include "a.h"
int a;
main() {  … }
```

宏名的定义

DEFine

C/C++ <Source>[Show entries for :][Defines]

格　式　　DEFine = <sub>[,…]

　　　　　　<sub>：<宏名> [= <字符串>]

说　明　　获得与 C/C++源程序中记述的#define 相同的效果。

能通过<宏名>=<字符串>的记述，将<字符串>定义为宏名。

如果对子选项单独指定<宏名>，就认为该宏名已定义。在<字符串>中能记述名字或者整型常数。

信息消息

Message
NOMessage

C/C++ <Source>[Show entries for :][Messages][Display information level messages]

格　式　　Message

NOMessage [= <错误号>[-<错误号>][,…]]

说　明　　指定是否输出信息级的消息。

message 选项输出信息级的消息。

nomessage 选项抑制信息级的全部消息的输出。但是，如果通过子选项指定错误号，就只抑制指定消息的输出。

也能用如<错误号>-<错误号>的连字符（-）指定要抑制的错误号的范围。

在省略本选项时解释为 nomessage。

例　　ch38 –nomessage=5,300-306 test.c

抑制 C0005 和 C0300～C0306 的信息级消息的输出。

备　考　　在<错误号>中能指定警告级和信息级的序号。

如果使用 Ver.4.0 以下的版本指定多个 message 选项和 nomessage 选项，就只有最后指定的选项有效。但是，在 Ver.6.0 以上的版本时，抑制 nomessage 选项指定的错误号的并集消息输出。

文件间直接插入展开目录的指定

FILE_INLINE_PATH

C/C++ <Source>[Show entries for :][File inline path]

格　式　　FILE_INLINE_PATH = <路径名>[,…]

说　明　　指定文件间直接插入展开的对象文件的路径名。
有多个路径名时，能用逗号（,）分开指定。
按照 file_inline_path 选项的指定目录和当前目录的顺序，检索文件间直接插入展开的对象文件。

例

```
ch38 -file_inline_path=c:\usr\file -file_inline=test2.c test.c
```

将目录 c:\usr\file 作为文件间直接插入展开的指定目录，检索由 file_inline 选项指定的 test2.c。

备　考　　本选项在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

2.2.2　目标选项

表 2.2　目标种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	预处理器展开	PREProcessor [= <文件名>]	C/C++ <Object> [Output file type :] [Preprocessed source file]	输出预处理器展开后的源程序
2	目标格式	Code = { Machinecode | Asmcode }	C/C++ <Object> [Output file type :] [Machine code] [Assembly source code]	 输出机器语言程序 输出汇编程序
3	调试信息	DEBug NODEBug	C/C++ <Object> [Generate debug information]	输出调试信息 不输出调试信息

4	段名	SEction = <sub>[,…]
		<sub>:{
		
		Program=<段名> |
		Const = <段名> |
		Data = <段名> |
		Bss = <段名> |
		}
			C/C++ <Object>
			[Section :]
			[Program section (P)]
			[Const section (C)]
			[Data section (D)]
			[Uninitialized data section (B)]
				
				程序区的段名
				常数区的段名
				初始化数据区的段名
				未初始化数据区的段名

5	字符串输出区域	STring = { Const | Data }	C/C++ <Object> [Store string data in :]	输出到常数区(C) 输出到初始化数据区(D)
6	乘除规格的扩展解释	CPUExpand [= V6] NOCPUExpand	C/C++ <Object> [Mul/Div operation specification]	根据 CPU 指令规格进行乘除的代码展开 根据 ANSI C 语言规格进行乘除的代码展开
7	目标文件的输出指定	OBject [= <文件名>] NOOBject	C/C++ <Object> [Output directory :]	输出目标文件 不输出目标文件
8	模板示例生成功能	Template = { None Static | Used | ALl | AUto }	C/C++ <Object> [Template :]	不生成示例 只将被参照的内容作为内部连接进行生成 只将被参照的内容作为外部连接进行生成 生成被声明和被参照的内容 连接时自动生成
9	边界调整数、边界调整指定	ALign [=4] NOALign	C/C++ <Object> [Group by alignment :]	变量重新配置后分配 按照声明的顺序分配变量
10	输出目标兼容	LEgacy=v4	C/C++ <Object> [Ver.4.0 Optimization technology generation:]	通过与 H8S 的 Ver.4.0 相同的优化处理进行目标输出

预处理器展开

PREProcessor

C/C++ <Object>[Output file type :][Preprocessed source file]

格　式　　PREProcessor [= <文件名>]

说　明　　输出预处理器展开后的源程序。
在不指定文件名时，用与源文件相同的文件名建立扩展名为“p”（输入源文件为 C 程序时）或者“pp”（输入源程序为 C++程序时）的文件。
在指定 preprocessor 选项时，不输出目标文件。

备　考　　指定 preprocessor 选项时，以下的选项无效：
code、object、outcode、debug、pack、string、show=object,statistics,allocation、section、optimize、speed、goptimize、byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、cpuexpand、eepmov、regparam、stack、align/noalign、structreg、longreg、macsave、bit_order、ptr16、opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、library、volatile_loop、sbr、legacy=v4、scope、noscope、file_inline、file_inline_path、enable_register、strict_ansi、cpuexpand=v6

目标格式

Code

C/C++ <Object>[Output file type :] [Machine code] [Assembly source code]

格　式　　Code = { Machinecode ｜ Asmcode }

说　明　　指定目标程序的输出格式。
在指定 code=machinecode 选项时，输出可再定位目标（机器语言）程序。
在指定 code=asmcode 选项时，输出汇编程序。
另外，将表示各函数使用的堆栈信息的.stack 控制指令输出到汇编程序中。
在省略本选项时解释为 code=machinecode。

备　考　　如果指定 code=asmcode 选项，show=object,goptimize 选项就无效。

调试信息

DEBug
NODEBug

C/C++ <Object>[Generate debug information]

格　式　DEBug

NODEBug

说　明　在指定 debug 选项时，将源级调试所需的调试信息输出到目标文件。

即使指定了优化选项，本选项也有效。

在指定 nodebug 选项时，不将调试信息输出到目标文件中。

在省略本选项时解释为 nodebug。

段名

SEction

C/C++ <Object>[Section :] [Program section (P)] [Const section (C)]
[Data section (D)] [Uninitialized data section (B)]

格　式

```
        SEction   = <sub>[,…]
          <sub>: { Program     = <段名> |
                   Const       = <段名> |
                   Data        = <段名> |
                   Bss         = <段名> }
```

说　明　指定目标程序中的段名。

section=program=<段名>指定程序区的段名。

section=const=<段名>指定常数区的段名。

section=data=<段名>指定初始化数据区的段名。

section=bss=<段名>指定未初始化数据区的段名。

<段名>是不以数字开始的英文字母、数字、下划线(_)或者$的列。段名最大为 8192 个字符。

在省略本选项时解释为 section=program=P, const=C, data=D, bss=B。

备　考　有关程序和段名的对应关系的详细内容，请参照“9.1　程序结构”。

不能给不同区域的段指定相同的段名。如果通过 Section 选项将 P, C, B, D 段更改为 S 段，就发生警告错误。S 段是用于堆栈区的保留段名。

字符串输出区域

STring

`C/C++ <Object>[Store string data in :]`

格　式	`STring = {` <u>`Const`</u> `｜ Data }`
说　明	指定字符串的输出目标。 在指定 `string=const` 选项时，输出到常数区。 在指定 `string=data` 选项时，输出到初始化数据区。 在程序执行时，能更改已输出到初始化数据区的字符串。但是，需要在 ROM 和 RAM 中双重确保区域，并且在开始执行程序时从 ROM 传送到 RAM。有关初始化数据区的初始设定和存储器分配的方法，请参照“9.2.1 存储区的分配”。 在省略本选项时解释为 `string=const`。

乘除规格的扩展解释

CPUExpand
NOCPUExpand

C/C++ <Object>[Mul/Div operation specification]

格 式

`CPUExpand [=V6]`
`NOCPUExpand`

说 明

在指定 cpuexpand 选项时，根据 ANSI 规格进行变量乘除代码展开的扩展解释和生成。
如果指定 cpuexpand=v6，就在通过和 Ver.4.0 以下版本相同的优化处理进行生成代码时，输出和 Ver.6.0 的 cpuexpand 规格相同的代码。
通过此子选项，代码发生变化的 C 源程序的记述式如下：

```
(a)   signed long =    signed int  <<  常数
(b)   signed long =  unsigned int  <<  常数
(c) unsigned long =    signed int  <<  常数
(d) unsigned long =  unsigned int  <<  常数
(e)   signed int  = (  signed int  <<  常数) /   signed int
(f)   signed int  = (unsigned int  <<  常数) /   signed int
(g)   signed int  = (unsigned int  <<  常数) / unsigned int
(h) unsigned int  = (  signed int  <<  常数) /   signed int
(i) unsigned int  = (unsigned int  <<  常数) /   signed int
(j) unsigned int  = (unsigned int  <<  常数) / unsigned int
```

在指定 nocpuexpand 选项时，以 ANSI 规格的格式进行乘除代码展开的生成。
在省略本选项时解释为 nocpuexpand。

备　考　在指定 cpuexpand、cpuexpand=V6 选项时，因为与语言规格规定的值的保证范围不同，所以运算结果和指定 nocpuexpand 选项时的结果有可能不同。用本选项指定的乘除代码展开如表 2.3 所示。

表 2.3　cpuexpand 选项的运算规格

运算对象	us1*us2 的运算长度(H8S/2600 的例子)	
	指定 cpuexpand 时	指定 nocpuexpand 时
unsigned short us1, us2; unsigned long ul; ul= us1*us2;	用 unsigned long 保持 us1*us2 的中间结果。* 例：MOV.W　@_us1,Rd MOV.W　@_us2,Rs MULXU.W　Rs,ERd MOV.L　ERd,@_ul 将 us1*us2 结果的 4 个字节赋值到 ul。	用 unsigned short 运算 us1*us2。 例：MOV.W　@_us1,Rd MOV.W　@_us2,Rs MULXU.W　Rs,ERd EXTU.L　ERd MOV.L　ERd,@_ul 将 us1*us2 结果的低位 2 个字节进行 0 扩展后赋值到 ul。
unsigned short us1,us2,us3; unsigned short us; us= us1*us2/us3;	用 unsigned long 保持 us1*us2 的中间结果。* 例：MOV.W　@_us1,Rd MOV.W　@_us2,Rs MULXU.W　Rs,ERd MOV.W　@_us3,Rs DIVXU.W　Rs,ERd MOV.W　Rd,@_us 将 us1*us2 结果的 4 个字节作为除法指令的被除数。	用 unsigned short 运算 us1*us2。 例：MOV.W　@_us1,Rd MOV.W　@_us2,Rs MULXU.W　Rs,ERd EXTU.L　ERd MOV.W　@_us3, Rs DIVXU.W　Rs,ERd MOV.W　Rd,@_us 将 us1*us2 结果的低位 2 个字节进行 0 扩展后的值作为除法指令的被除数。

【注】* 在将 2 个字节之间的乘法结果赋值到 4 个字节时，或者在进行数据类型转换时，或者在以 2 个字节进行除法运算时，使用以 4 个字节保持的中间结果。

在 CPU 种类为 H8S 并且指定 legacy=v4 时，或者在 CPU 种类为 H8/300 和 H8/300H 时，cpuexpand=V6 子选项的指定有效。

目标文件输出指定

OBject
NOOBject

`C/C++ <Object>[Output directory :]`

格　式　`OBject [= <目标文件名>]`
`NOOBject`

说　明　指定是否输出目标文件。
在指定 noobject 选项时，不输出目标文件。
如果不用 object 选项指定目标文件名，就输出和源文件相同的文件名且扩展名为“obj”（输出文件为可再定位目标程序时）或者“src”（输出文件为汇编源程序时）的目标文件。
文件扩展名由 code 选项决定为“obj”或者“src”。
在省略本选项时解释为 object。

备　考　指定 noobject 选项时，以下的选项无效：
outcode、debug、pack、string、show=object、statistics、allocation、section、optimize、speed、goptimize、byteenum、volatile、regexpansion、cmncode、case、indirect、abs8、abs16、cpuexpand、eepmov、regparam、stack、align/noalign、structreg、longreg、macsave、bit_order、ptr16、opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、library、volatile_loop、sbr、legacy=v4、scope、noscope、file_inline、file_inline_path、enable_register、strict_ansi、cpuexpand=v6

模板示例生成功能

Template

C/C++ <Object>[Template :]

格　式

```
Template ={  None      |
             Static    |
             Used      |
             ALl       |
             AUto     }
```

说　明

指定模板的示例生成方法。
在指定 `template=none` 时，不生成示例。
在指定 `template=static` 时，只生成在编译单位中被参照的模板示例。但是，被生成的函数具有内部连接。
在指定 `template=used` 时，只生成在编译单位中被参照的模板示例。但是，被生成的函数具有外部连接。
在指定 `template=all` 时，生成在编译单位中被声明或被参照的全部模板示例。
在指定 `templete=auto` 时，生成连接时所需的示例。

备　考

在省略本选项时解释为 `template=auto`，但是在指定 `code=asmcode` 时总是为 `template=static`。

边界调整数、边界调整指定

ALign
NOALign

C/C++ <Object>[Group by alignment :]

格　式	ALign [=4] NOALign

说　明　在指定 noalign 选项时，按照声明的顺序分配被定义的变量。

在指定 align 选项时，为了缩小边界调整数的空区域而进行变量的重新分配。如果进行重新分配，通常减小空区域的部分和目标大小。

在指定 align=4 选项时，将数据分成边界调整数为 4 的段、2 的段和 1 的段。将长度为 4 的倍数的数据分配到边界调整数为 4 的段，段名为在原段名附加$4 后的段名。CPU 种类为 H8SX 时，能提高对已分配到 4 字节边界的 4 个字节变量的存取速度。

将长度为奇数的数据分配到边界调整数为 1 的其它段，段名为原段名附加$1 后的段名，由此减小空区域。

剩下的数据，即长度为偶数但不为 4 的倍数的数据留在原来的段中。

另外，即使通过 pragma 扩展指定或者选项指定进行段名的更改，对被更改的段名也附加$4 或者$1。

在省略本选项时解释为 align。

备　考　只在 CPU 种类为 H8SX 时， align=4 选项的指定有效。

在将指定 align=4 时的 1 个字节/4 个字节的数据段分配到特定的地址时，需要通过 optlnk 的 start 选项明确指定各段。在不改变边界的数据结构时，必须指定 noalign 选项。

例

```
char a;
short b;
char c;
long d;
#pragma section _v
short e;
long f;
#pragma section

main( )
{
  ...
```

- noalign指定

2个字节

a	空区域
b	b
c	空区域
d	d
d	d

【段 B】
大小： 10 bytes
边界调整数： 2

e	e
f	f
f	f

【段 B_v】
大小： 6 bytes
边界调整数： 2

- 按声明的顺序分配数据
- 由于边界调整数为2的数据必定被分配到偶数地址，所以如果奇数大小数据在前，就有可能产生不被使用的空区域。

- align指定（默认设定）

2个字节

b	b
d	d
d	d
a	c

【段 B】
大小： 8 bytes
边界调整数： 2

e	e
f	f
f	f

【段 B_v】
大小： 6 bytes
边界调整数： 2

- 为了使空区域为最小，在排列边界调整数为2的数据后，排列边界调整数为1的数据。

- align = 4指定

2个字节

d	d
d	d

【段 B$4】
大小： 4 bytes
边界调整数： 4

f	f
f	f

【段 B_v$4】
大小： 4 bytes
边界调整数： 4

b	b

【段 B】
大小： 2 bytes
边界调整数： 2

e	e

【段 B_v】
大小： 2 bytes
边界调整数： 2

a	c

【段 B$1】
大小： 2 bytes
边界调整数： 1

- 将数据分成3类
  X：大小为4的倍数的数据
  Y：大小为奇数的数据
  Z：其它（大小为偶数，
  　但不为4的倍数的数据）
- 根据数据的大小，分割段。
  例如，B段被分割成B$4、B$1、B。
  X：由大小为4的倍数的数据构成的段的边界调整数为4，在段名的末尾附加"$4"。（例：B$4）
  Y：由大小为奇数的数据构成的段的边界调整数为1，在段名的末尾附加"$1"。（例：B$1）
  Z：由其它数据构成的段留在边界调整数为2的原来的段中。（例：B）

Ver.4.0 优化目标输出

LEgacy=v4

None

格　式	`LEgacy=v4`
说　明	如果同时给本选项和 CPU 选项指定 2600A/2600N/2000A/2000N，就以和 Ver.4.0 以下版本相同的优化处理输出代码。 在不指定本选项时，以比 `Ver.4.0` 的目标更优化的处理输出目标码。
备　考	在 CPU 种类不为 2600A/2600N/2000A/2000N 时，本选项无效。 另外，如果指定 `legacy=v4` 选项，以下的选项无效： `opt_range, del_vacant_loop, max_unroll, infinite_loop, global_alloc, struct_alloc, const_var_propagate, volatile_loop, scope,noscope, strict_ansi, file_inline, file_inline_path, enable_register`

2.2.3　　列表选项

表 2.4　列表种类选项一览表

项目	命令行格式	对话菜单	指定内容
1　列表文件	List [= <文件名>]	C/C++ <List>	输出列表文件
	NOList	[Generate list file]	不输出列表文件
2　列表内容和格式	SHow = <sub>[,…]	C/C++ <List>	
	<sub>:{	[Contents :]	
	SOurce | NOSOurce |		源列表的有无
	Object | NOObject |		目标列表的有无
	STatistics | NOSTatistics |		统计信息的有无
	Allocation | NOAllocation |		符号分配列表的有无
	Expansion|NOExpansion |		宏展开后的列表的有无
	Width = <数值> |		1 行的最大字符数: 0, 80～132
	Length = <数值> |		页内的最大行数: 0, 20～255
	Tab = { 4 | 8 } }		使用制表符时的列数：4 | 8

列表文件

List
NOList

C/C++ <List>[Generate list file]

格　式　　`List [= <列表文件名>]`
`NOList`

说　明　　指定是否输出列表文件。

在指定 list 选项时，输出列表文件。

如果指定 nolist 选项，就不输出列表文件。

能按照“8.1　**文件名的命名方法**”指定<列表文件名>。

如果在 list 选项中不指定<列表文件名>，就建立和源文件相同的文件名的列表文件，其扩展名在 UNIX 版时为“lis”而在 PC 版时为“lst”（输入源文件为 C 程序时）或者“lpp”（输入源程序为 C++程序时）。

在省略本选项时解释为 list。

SHow

C/C++ <List> [Contents :]

格　式

```
SHow = <sub>[,…]
     <sub>:{SOurce       | NOSOurce      |
            Object       | NOObject      |
            STatistics   | NOSTatistics  |
            Allocation   | NOAllocation  |
            Expansion    | NOExpansion   |
            Width =〈数值〉                |
            Length=〈数值〉                |
            Tab   = {  4 |  8  }          }
```

说　明

指定编译程序输出的列表的内容及其格式和输出的解除。

有关本项记述的各列表的具体例子，请参照“8.2　编译列表的参照方法”。

在省略本选项时解释为

show=source、noobject、statistics、noallocation、noexpansion、width=0、length=0、tab=8。

备　考

子选项的一览表如表 2.5 所示。

表 2.5　show 选项的子选项一览表

子选项	含义
source	输出源程序列表。
nosource	不输出源程序列表。
object	输出目标程序列表。
noobject	不输出目标程序列表。
statistics	输出统计信息的列表。
nostatistics	不输出统计信息的列表。
allocation	输出符号分配信息的列表。
noallocation	不输出符号分配信息的列表。
expansion	输出 include 文件和宏展开后的源程序列表。如果同时指定 nosource 子选项，expansion 子选项就无效，不输出源程序列表。
noexpansion	输出 include 文件和宏展开前的源程序列表。如果同时指定 nosource 子选项，noexpansion 子选项就无效，不输出源程序列表。
width=〈数值〉	〈数值〉指定的字符数为列表的 1 行的最大字符数。〈数值〉用 10 进制数指定，能指定 0 或者从 80 到 132 之间的数值。〈数值〉为 0 时，不规定列表的 1 行的最大字符数。
length=〈数值〉	〈数值〉指定的行数为列表的 1 页的最大行数。〈数值〉用 10 进制数指定，能指定 0 或者从 20 到 255 之间的数值。〈数值〉为 0 时，不规定列表的 1 页的最大行数。
tab= { 4 | 8 }	指定列表显示时的制表符的字符数。

2.2.4　　优化选项

表 2.6　优化种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	优化级	OPtimize = { 0 | 1 }	C/C++ <Optimize> [Optimization]	不优化 优化
2	模块间的优化	Goptimize	C/C++ <Optimize> [Generate file for inter-module optimization]	用于模块间优化的附加信息输出
3	速度优先的优化	SPeed [= <sub>[,…]] <sub>: { Register | SHift | Loop [= { 1 | 2 }] | SWitch | Inline [=<数值>] | STruct | Expression }	C/C++ <Optimize> [Speed or size :] [Speed sub-options :]	指定速度优先的代码生成 进行寄存器保存/恢复的 push、pop 展开 移位运算的高速化 删除循环语句中的归纳变量 删除循环语句中的归纳变量，循环展开 switch 语句的高速化 自动直接插入展开 结构体赋值表达式的高速化 四则运算、比较、赋值表达式的高速化
4	Switch 语句展开方式	CAse = { Auto | Ifthen | Table }	C/C++ <Optimize> [Switch statement :]	通过 speed 选项指定的有无进行判断 以 if_then 方式进行展开 以表转移方式进行展开
5	存储器间接方式	INDirect = { Normal | Extended }	C/C++ <Optimize> [Function call :]	以存储器间接方式展开函数调用 以扩展存储器间接方式展开函数调用
6	指针长度的指定	PTr16	C/C++ <Optimize> [2byte pointer]	将指向变量的指针长度指定为 2 个字节
7	短绝对地址	ABS8 ABS16	C/C++ <Optimize> [Data access :]	以 8 位绝对地址存取 8 位数据 以 16 位绝对地址存取全部数据
8	外部变量的优化	Volatile NOVolatile	C/C++ <Optimize> [Details...] [Global variables] [Treat global variables as volatile qualified]	抑制外部变量的优化 优化外部变量
9	外部变量优化范围的指定	OPT_Range = { All | NOLoop | NOBlock }	C/C++ <Optimize> [Details...] [Global variables] [Specify optimizing range:]	将函数内的全部外部变量指定为优化对象 抑制循环控制变量和循环内的外部变量向循环外的移动 抑制对所有循环或分支间的外部变量的优化
10	空循环的删除	DEL_vacant_loop = { 0 | 1 }	C/C++ <Optimize> [Details...] [Miscellaneous] [Delete vacant loop]	抑制空循环的删除 删除空循环
11	循环最大展开数的指定	MAX_unroll = <数值> <数值>：1-32	C/C++ <Optimize> [Details...] [Miscellaneous] [Specify maximum unroll factor :]	指定展开循环时的最大展开数 default：1（指定 speed, speed=loop[=2] 时为 2）

	项目	命令行格式	对话菜单	指定内容
12	无限循环前的赋值表达式的删除	INFinite_loop = { 0 | 1 }	C/C++ <Optimize> [Details...] [Global variables] [Delete ssignment to global variables before an infinite loop]	抑制无限循环前的外部变量赋值表达式的删除 删除无限循环前的外部变量赋值表达式
13	外部变量的寄存器分配	GLOBAL_Alloc = { 0 | 1 }	C/C++ <Optimize> [Details...] [Global variables] [Allocate registers to global variables]	抑制外部变量的寄存器分配 进行外部变量的寄存器分配
14	结构体/联合体成员的寄存器分配	STRUCT_Alloc = { 0 | 1 }	C/C++ <Optimize> [Details...] [Global variables] [Allocate registers to struct/union members]	抑制结构体/联合体成员的寄存器分配 进行结构体/联合体成员的寄存器分配
15	Const 常数传播	CONST_Var_propagate = { 0 | 1 }	C/C++ <Optimize> [Details...] [Global variables] [propagate variables which are const qualified]	抑制由 const 声明的外部常数的常数传播 进行由 const 声明的外部常数的常数传播
16	特殊库函数的直接插入展开指定	LIBrary = { Function | Intrinsic }	C/C++ <Optimize> [Details...] [Miscellaneous] [Inline memcpy/strcpy]	将 memcpy 和 strcpy 作为函数调用进行代码生成 将 memcpy 和 strcpy 直接插入展开进行代码生成
17	优化范围的分割	SCOpe NOSCope	-	分割优化范围 不分割优化范围
18	文件间的直接插入展开	FILe_inline= <文件名>[,…]	C/C++ <Optimize> [Details…] [Inline] [inline file path]	进行文件间的直接插入展开

优化级

OPtimize

C/C++ <Optimize>[Optimization]

格　式	`OPtimize = { 0 ｜ 1 }`
说　明	指定目标程序的优化级。 在指定 optimize=0 选项时，不进行目标程序的优化。 在指定 optimize=1 选项时，进行优化。 在省略本选项时解释为 optimize=1。
备　考	在指定 optimize=0 选项时，speed=inline,loop 选项无效。

模块间优化

Goptimize

C/C++ <Optimize>[Generate file for inter-module optimization]

格　式	`Goptimize`
说　明	输出用于模式间优化的附加信息。 在连接时，指定本选项的文件为模块间优化的对象。

速度优先优化

SPeed

C/C++ <Optimize>[Speed or size :][Speed sub-options :]

格　式

```
SPeed = <sub>[,…]
  <sub>:{Register                                    |
         SHift                                       |
         Loop [={ 1 | 2 }]                           |
         SWitch                                      |
         Inline [ = <数值>]                          |
         STruct                                      |
         Expression                                  }
```

说　明

对于编译程序生成的目标，指定谋求执行速度高速化的优化。

在指定 speed=register 选项并 CPU/运行模式为 300ha、300hn,300 时，在函数的入口/出口不使用运行时例程而使用 PUSH、POP 指令展开寄存器的保存/恢复。

在指定 speed=shift 选项时，不使用运行时例程而用代码展开移位运算。

在指定 speed=loop=1 选项时，删除归纳变量。

在指定 spped=loop=2 选项时，删除归纳变量，进行循环展开的优化。

在指定 speed=switch 选项时，以速度优先选择 switch 语句判断式的代码展开方式。

在指定 speed=inline 选项时，进行小容量函数的直接插入展开。

speed=inline=<数值>能改变作为直接插入展开对象的函数的最大容量。

在 CPU 种类为 H8SX 或者 H8S（没有指定 legacy=v4）时，<数值>表示在程序大小增加到百分之几之前进行直接插入展开。例如，假设指定 speed=inline=50，就在程序大小增加到 50%之前（到 1.5 倍为止）进行直接插入展开。

在 CPU 种类为 H8/300、H8/300H 或者 H8S（指定 legacy=v4）时，<数值>表示能直接插入展开的函数的节点数（除声明以外的变量/运算符的语句总数）。即直接插入展开比<数值>所示的基准小的函数。此时，程序大小的增量取决于被直接插入展开的函数的大小和出现频率，而不像 H8SX、H8S（没有指定 legacy=v4）那样指定增量的上限。

省略<数值>时的解释:在 CPU 种类为 H8SX、H8S 时为 100，除此以外为 110。

有关直接插入展开的条件，请参照“10.2.1(2) **有关函数的扩展功能**”的#pragma inline/__inline 的项目。

如果指定 speed=struct 选项，就在对结构体型和 double 型的赋值语句进行代码展开时不使用运行时例程。

在指定 speed=expression 选项时，将四则运算、比较、赋值表达式展开成不使用运行时例程的代码（部分表达式可能为对象外）。

如果只指定 speed，就全部进行执行速度优先的优化。在省略本选项时，生成与执行速度相比更重视减小目标码大小的目标。

备　考

在指定无优化(optimize=0)时，speed=loop,inline 选项无效。

switch 语句展开方式

CAse

C/C++ <Optimize>[Switch statement :]

格　式　　CAse = { Auto | Ifthen | Table }

说　明　　指定 switch 语句的代码展开方式。

在指定 case=auto 选项时，编译程序自动选择优先减小目标大小的展开方式。

在指定 speed 选项或者 speed=switch 选项时，编译程序自动选择优先执行速度的展开方式。

在指定 case=ifthen 选项时，以 if_then 方式展开 switch 语句。if_then 方式是比较 switch 语句的评价式的值和 case 标号的值，如果一致就重复进行（case 标号的个数次）跳转到 case 标号语句处理的展开方式。此方式与 switch 语句所包含的 case 标号个数成正比，增大目标码大小。

在指定 case=table 选项时，以表方式展开 switch 语句。表方式是将 case 标号的跳转目标确保到转移表，然后通过 1 次转移表的参照，跳转到与 switch 语句的评价式一致的 case 标号语句的展开方式。此方式与 switch 语句所包含的 case 标号个数成正比，增大确保在常数区的转移表容量，但是执行速度总是固定。

在省略本选项时解释为 case=auto。

例

```
int a, b;
 :
switch(a){
 case 1:    b=3; break;
 case 2:    b=2; break;
 case 3:    b=1; break;
}
```

上述源程序的代码展开例子如下所示 (cpu=2600n)：

```
  MOV.W   @_a:16,R0                    MOV.W   @_a:16,R0
  MOV.B   R0H,R0H                      SUB.W   #H'1,R0
  BNE     Ld                           CMP.W   #H'2,R0
  CMP.B   #1,R0L                       BHI     Ld
  BEQ     L1                           MOV.B   @(Ll:16,ER0),R0L
  CMP.B   #2,R0L                       EXTU.W  R0
  BEQ     L2                           ADD.W   #LWORD Lp,R0
  CMP.B   #3,ROL                       JMP     @ER0
  BNE     L4                         Lp:
  BRA     L3                                   :
L1:                                  L1:（转移表）
  case=ifthen                          case=table
```

表 2.7　根据式的值的大小、周期的不同

a 的值	if_then 方式		表方式	
	目标码大小	执行周期	目标码大小	执行周期
1	22 字节	9	29(26+3)字节	17
3		17		

存储器间接方式

INDirect

C/C++ <Optimize>[Function call :]

格　式　　`INDirect = {Normal | Extended}`

说　明　　指定在源程序中调用函数时的存储器间接方式。
在指定 indirect=normal 选项时，通过存储器间接(@@aa:8)调用全部的函数。在指定 indirect=extended 选项时，通过扩展存储器间接(@@vec:7)调用全部的函数。
对于在源程序中定义的函数，在以下的段中建立存储器间接调用的地址表：

- 在指定 indirect=normal 选项时，为“$INDIRECT”段
- 在指定 indirect=extended 选项时，为“$EXINDIRECT”段

有关地址表的段名转换方法，请参照“**10.2.1(1) 有关存储器分配的扩展功能**”的段转换指定。

备　考　　能保存地址表的区域被限制在以下的地址范围：

- “$INDIRECT”段　：地址 0x0000～0x00FF
- “$EXINDIRECT”段：地址 0x0100～0x01FF　H8SX 正常模式
  　　　　　　　　：地址 0x0200～0x03FF　H8SX 的其它模式

在连接时，必须用 start 选项将这些段的分配明确指定到各地址范围内。
indirect=extended 指定只在 CPU 种类为 H8SX 时有效。
在只对特殊函数指定存储器间接方式时，使用#pragma indirect、_ _indirect、_ _indirect_ex。这些指定比本选项优先处理，详细内容请参照“**10.2.1(2) 有关函数的扩展功能**”。

在同一个函数的定义侧和调用侧必须统一为 normal 或者 extended。

指针长度的指定

PTr16

C/C++ <Optimize>[2byte pointer]

格　式　　`PTr16`

说　明　　将指向数据的指针长度设定为 2 个字节。

备　考　　如果不指定本选项，指向数据的指针长度就为 4 个字节。
如果指定本选项，就需要将参照目标的数据段明确分配到 16 位绝对地址区。段分配的地址通过优化连接编辑程序的 start 选项来指定，详细内容请参照“**4.2.5 段选项**”的 start 选项。另外，有关 16 位绝对地址区，请参照“**19.3　短绝对地址的存取范围**”。
本选项的指定在 CPU/运行模式为 H8SXA,H8SXX,H8S/2600A,H8S/2000A 时有效。
由于指向数据的指针长度从 4 个字节变为 2 个字节，所以不仅对源的分配而且对函数的参数传递方法和返回值的接受方法也有影响。必须注意：相同数据和相同函数的处理必须保持文件之间的一致性。

短绝对地址

ABS8
ABS16

C/C++ <Optimize>[Data access :]

格　式	ABS8 ABS16

说　明　以短绝对寻址方式存取分配在静态区的数据。

在指定 abs8 选项时，生成以 8 位绝对地址(@aa:8)存取 char 型数据、unsigned char 型数据以及包含 char 型的元素、unsigned char 型的元素以及成员的 1 个字节边界一致的复合型数据的代码。

在 CPU/运行模式为 H8SXA、H8SXX、2600a、2000a、300ha 时，abs16 选项生成由 16 位绝对地址(@aa:16)存取数据的代码；在 CPU/运行模式为 H8SXN、H8SXM、2600n、2000n、300hn、300 时，abs16 选项的指定无效。

用 abs8 选项以 8 位绝对地址存取的数据被输出到段名“$ABS8C”、“$ABS8D”或者“$ABS8B ”。另外，用 abs16 选项以 16 位绝对地址存取的数据被输出到段名“$ABS16C”、“$ABS16D”或者“$ABS16B”。

也能用#pragma abs8,#pragma abs16 的扩展名以及_ _abs8,_ _abs16 的关键字指定用短绝对寻址方式存取的变量。

如果指定选项和扩展名/关键字，就优先扩展名/关键字的指定。

备　考　在连接时，需要将由本选项输出的段分配到短绝对地址区。有关短绝对地址区的范围，请参照“19.3　短绝对地址的存取范围”。另外，有关短绝对地址区的段名转换方法，请参照“10.2.1(1)　存储器分配的扩展功能”的段转换指定。

外部变量的优化

Volatile
NOVolatile

`C/C++ <Optimize>[ Details...][Global variables][Treat global variables as volatile qualified]`

格　式　　`Volatile`
　　　　　`NOVolatile`

说　明　　在指定 `volatile` 选项时，对全部的外部变量不进行优化。
　　　　　在指定 `novolatile` 选项时，对无 `volatile` 限定词的外部变量进行优化。
　　　　　在省略本选项时解释为 `novolatile`。

例　　　　源程序

```
volatile int a;
int b;
void main(void){
  a;
  b;
}
・指定 volatile 时
      mov.w @_a,R0
      mov.w @_b,R0     ;将 b 作为 volatile 变量存取
      rts
・指定 novolatile 时
      mov.w @_a,R0
      rts              ;作为优化结果，删除 b 的存取
```

外部变量优化范围的指定

OPT_Range

`C/C++ <Optimize> [Details...][Global variables][Specify optimizing range :]`

格　式　　`OPT_Range = { All | NOLoop | NOBlock }`

说　明　　在指定 `opt_range=all` 选项时，以函数中的全部范围为对象，对外部变量进行优化。
在指定 `opt_range=noloop` 选项时，不将循环内的外部变量和循环判断式使用的外部变量作为优化对象。
在指定 `opt_range=noblock` 选项时，抑制全部分支间的外部变量的优化（包含循环）。
在省略本选项时解释为 `opt_range=all`。

例

(1) 分支间的优化例子（在指定 `opt_range=all/noloop` 时进行）

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = A;
}
```

＜优化后的源程序＞

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1;    /* 删除 A 的参照，传播 A=1 */
}
```

(2) 循环的优化例子（在指定 `opt_range=all` 时进行）

```
int A,B,C[100];          /* 外部变量 */
void f() {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

＜优化后的源程序＞

```
void f() {
    int i;
    int temp_A, temp_B; /* 局部变量 */
    temp_A = A;         /* 将循环判断式的 A 的参照移动到循环外 */
    temp_B = B;         /* 将循环内的 B 的参照移动到循环外   */
    for (i=0;i<temp_A;i++) {    /* 删除循环内的 A 的参照 */
        C[i] = temp_B;          /* 删除循环内的 B 的参照 */
    }
}
```

备　考　本选项只在 CPU 种类为 H8SX 或者 H8S（没有指定 `legacy=v4` 选项）时有效。
在指定 `opt_range=noloop` 选项时，`max_unroll=1` 总是为默认值。在指定 `opt_range=noblock` 选项时，`max_unroll=1`、`const_var_propagate=0`、`global_alloc=0` 总是为默认值。

空循环删除

DEL_vacant_loop

`C/C++ <Optimize>[Details...][Miscellaneous][Delete vacant loop]`

格　式　`DEL_vacant_loop = { 0 | 1 }`

说　明　在指定 `del_vacant_loop=0` 选项时，即使循环内没有处理，也不删除循环。
在指定 `del_vacant_loop=1` 选项时，删除循环内没有处理的循环。
在省略本选项时解释为 `del_vacant_loop=0`。

备　考　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 `legacy=v4` 选项）时有效。

循环最大展开数的指定

MAX_unroll

C/C++ <Optimize>[Details...][Miscellaneous][Specify maximum unroll factor :]

格　式　　MAX_unroll = <数值>

说　明　　指定循环展开时的最大展开数。能给<数值>指定 1 到 32 的整数，如果指定其它值，就发生错误。

省略本选项时解释：在指定 speed 或者 speed=loop[=2]选项时为 max_unroll=2，否则为 max_unroll=1。

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

在指定 opt_range=noloop/noblock 时，max_unroll=1 总是为默认值。

无限循环前的赋值表达式的删除

INFinite_loop

C/C++ <Optimize>[Details...][Global variables][Delete assignment to global variables before an infinite loop]

格　式　　INFinite_loop = {0 | 1}

说　明　　在指定 infinite_loop=0 选项时，不删除无限循环前一行的外部变量赋值表达式。

在指定 infinite_loop=1 选项时，删除出现在无限循环前一行而在无限循环内没有参照的外部变量赋值表达式。

在省略本选项时解释为 infinite_loop=0。

例

```
int A;
void f()
{
    A = 1;          /* 外部变量 A 的赋值表达式 */
    while(1) {}     /* 没有参照 A */
}

<指定 infinite_loop=1 时的源程序>
void f()
{
                    /* 删除外部变量 A 的赋值表达式 */
    while(1) {}
}
```

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

外部变量的寄存器分配

GLOBAL_Alloc

C/C++ <Optimize>[Details...][Global variables][Allocate registers to global variables]

格　式　　GLOBAL_Alloc = { 0 | 1 }

说　明　　在指定 global_alloc=0 选项时，抑制外部变量的寄存器分配。
在指定 global_alloc=1 选项时，进行外部变量的寄存器分配。
在省略本选项时解释为 global_alloc=1。

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。
在指定 opt_range=noblock 时，global_alloc=0 为默认值。

结构体/联合体成员的寄存器分配

STRUCT_Alloc

C/C++ <Optimize>[Details...][Global variables][Allocate registers to struct/union members]

格　式　　STRUCT_Alloc = { 0 | 1 }

说　明　　在指定 struct_alloc=0 选项时，抑制结构体/联合体成员的寄存器分配。
在指定 struct_alloc=1 选项时，进行结构体/联合体成员的寄存器分配。
在省略本选项时解释为 struct_alloc=1。

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。
在指定 opt_range=noblock 或者 global_alloc=0 并且指定 struct_alloc=1 时，只进行局部结构体/联合体成员的寄存器分配。

const 常数传播

CONST_Var_propagate

C/C++ <Optimize>[Details...][Global variables][Propagate variables which are const qualified]

格　式　　CONST_Var_propagate = { 0 | 1 }

说　明　　在指定 const_var_propagate=0 选项时，抑制由 const 声明的外部变量的常数传播。
在指定 const_var_propagate=1 选项时，进行由 const 声明的外部变量的常数传播。
在省略本选项时解释为 const_var_propagate=1。

例

```
const int X = 1;
int A;
void f() {
    A = X;
}
```

＜指定 const_var_propagate =1 时的源程序＞

```
void f() {
    A = 1;    /* 传播 X=1 */
}
```

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。
在指定 opt_range=noblock 时，const_var_propagate=0 为默认值。
本选项不能抑制在 C++程序中由 const 声明的变量（总是进行常数传播）。

特殊库函数的直接插入展开指定

LIBrary

C/C++ <Optimize>[Details...][Miscellaneous][Inline memcpy/strcpy]

格　式　　LIBrary = { Function | Intrinsic }

说　明　　对于库函数 memcpy, strcpy，在指定 library=function 选项时，调用该函数。
在指定 library=intrinsic 选项时，对该函数进行直接插入展开。

备　考　　library=intrinsic 的指定只在 CPU 种类为 H8SX 时有效。

优化范围的分割

SCOpe
NOSCope

None

格　式	`SCOpe` `NOSCope`
说　明	在指定 scope 选项时，对于大容量函数，将优化范围分割为多个进行编译。 在指定 noscope 选项时，不分割优化范围而进行编译。由于扩大优化范围，编译时间变长，但是一般能提高目标性能。如果寄存器的数量不足，就有可能降低目标性能。 本选项根据程序效果不同，所以推荐在性能调整时指定本选项。
备　考	本选项在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

文件间直接插入展开

FILe_inline

C/C++ <Optimize>[Details...][Inline][Inline file path]

格　式	`FILe_inline=<文件名>[,…]`
说　明	对于用文件名指定的文件，进行文件间函数的直接插入展开。
备　考	如果指定 file_inline 选项，就只对<文件名>指定的文件函数中用#pragma inline、inline 关键字指定的函数进行直接插入展开。如果同时指定-speed=inline 选项，就展开全部可直接插入的函数。 如果由 file_inline 选项指定的多个文件中定义了相同名字的全局函数，运行就无法保证（使用任选的 1 个函数定义进行直接插入展开）。 不能省略由<文件名>指定的文件名的扩展名。 不能通过 file_inline 选项指定编译对象的文件。 不能对<文件名>指定通配符(*,?)。 不展开有#pragma asm～endasm, #pragma inline_asm 和__asm 记述的文件。 本选项在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）有效。

2.2.5　　其它选项

表 2.8　其它种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	注释的嵌套	COMment	C/C++ <Other> [Miscellaneous options :] [Allow comment nest]	允许注释(/* */)的嵌套
2	嵌入式 C++言語	ECpp	C/C++ <Other> [Miscellaneous options :] [Check against EC++ language specification]	基于 EC++语言的语法检查和使用的存储器管理库的辨别
3	MAC 寄存器的保证	MAcsave	C/C++ <Other> [Miscellaneous options :] [Interrupt handler saves/restores MACH and MACL registers if used]	在中断函数前后总是保证 MAC 寄存器的内容
4	循环判断式的优化抑制	VOLATILE_Loop	C/C++ <Other> [Miscellaneous options :] [Treats loop condition as volatile qualified]	抑制循环判断式的优化
5	枚举型大小	Byteenum	C/C++ <Other> [Miscellaneous options :] [Treat enum as char if it is in the range of char]	声明的枚举型数据的 char/unsigned char 型转换
6	变量分配寄存器数的扩展	<u>Regexpansion</u> NORegexpansion	C/C++ <Other> [Miscellaneous options :] [Increase a register for register variable]	使用(E)R3～(E)R6 使用(E)R4～(E)R6
7	公用表达式的优化	CMncode	C/C++ <Other> [Miscellaneous options :] [Put common subexpression on a register temporarily]	强化公用表达式删除的优化
8	块传送指令	EEpmov	C/C++ <Other> [Miscellaneous options :] [Use EEPMOVE in block copy]	用 eepmov 指令展开结构体的赋值表达式
9	预处理器展开时的输出限制	NOLINe	C/C++ <Other> [Miscellaneous options :] [Suppress #line in preprocessed source file]	在展开预处理器时抑制#line 输出
10	消息级	CHAnge_message =<sub>[,…] <sub>:<level> [=<n>[-m],…] <level>:{Information | Warning | Error }	C/C++ <Other> [User defined options :]	改变信息级
11	register 存储类变量的优先分配	ENAble_register	C/C++ <Other> [Miscellaneous options :] [Enable register declaration]	将指定 register 存储类的变量优先进行寄存器分配
12	ANSI 依据	STRIct_ansi	C/C++ <Other> [Miscellaneous options :] [Obey ansi specifications more strictly]	依据 ANSI 进行以下处理： ・浮点运算的结合律

注释的嵌套

COMment

C/C++ <Other>[Miscellaneous options :] [Allow comment nest]

格　式　COMment

说　明　可进行嵌套的注释记述。
在省略本选项时，如果进行注释嵌套的记述，就发生错误。

例

```
/* This is an example of/* nested */ comment */
                                      ↑
                                     [1]
```

如果指定 comment 选项，就全部解释为注释，但是，如果省略本选项，就解释为在［1］处注释已结束。

嵌入式 C++语言

ECpp

C/C++ <Other>[Miscellaneous options :] [Check against EC++ language specification]

格　式　ECpp

说　明　根据 Embedded C++ 语言规格进行 C++程序的语法检查。
Embedded C++ 语言规格不支持 catch、const_cast、dynamic_cast、explicit、mutable、namespace、reinterpret_cast、static_cast、template、throw、try、typeid、typename、using。如果记述这些关键字，就输出错误信息。
另外，本选项辨别 EC++/C++使用的存储器管理库。
在使用 EC++库时，必须指定本选项。

备　考　Embedded C++ 语言规格不支持多重继承和虚基类。
如果记述多重继承或者虚基类，就输出错误信息"C5882 (E) Embedded C++ does not support multiple or virtual inheritance"。
不能同时指定本选项和 exception 选项。

MAC 寄存器的保证

MAcsave

C/C++ <Other>[Miscellaneous options :] [Interrupt handler saves/restores MACH and MACL registers if used]

格　式　　MAcsave

说　明　　在中断函数前后总是保证 MAC 寄存器。
在指定 macsave 选项时，如果中断函数内使用 MAC 寄存器或者有函数调用，就生成 MAC 寄存器的保存/恢复代码。
如果不指定 macsave 选项，就只在中断函数内使用 MAC 寄存器时生成 MAC 寄存器的保存/恢复代码。

循环判断式的优化抑制

VOLATILE_Loop

C/C++ <Other>[Miscellaneous options :][Treats loop condition as volatile qualified]

格　式　　VOLATILE_Loop

说　明　　在循环判断式包含外部变量时，不为优化对象。
但是，在伴有型转换、或者包含 2 个以上外部变量、或者进行复合运算时，可能不为优化抑制对象。

备　考　　本选项只在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。
如果指定本选项，即使不指定 volatile 选项，在循环内的该外部变量也为优化抑制对象。
在不指定本选项时，循环判断式如果在循环内不变就有可能被删除。

枚举型大小

Byteenum

C/C++ <Other>[Miscellaneous options :] [Treat enum as char if it is in the range of char]

格　式　　Byteenum

说　明　　将 enum 声明的枚举型数据作为 char 型、unsigned char 型处理。
如果指定本选项，就根据 enum 声明的枚举型成员的值的范围，选择枚举型数据的型。在值的范围为-128～127 时作为 char 型处理，在 0～255 时作为 unsigned char 型处理。
如果省略本选项或者即使指定本项而枚举型成员的值超出上述范围，就将枚举型数据作为 int 型处理。

例　　源程序

```
enum EM {a,b,c} E;
void main(void){E=b;}
```

・指定 byteenum 时

```
     mov.b #1,R0L   ;传送 1 个字节的数据
     mov.b R0L,@_E
     rts
  _E:
     .res.b 1       ;将 E 分配为 1 个字节的区域
```

・没有指定 byteenum 时

```
   mov.w #1,R0      ;传送 2 个字节的数据
   mov.w R0,@_E
   rts
  _E:
   .res.w 1         ;将 E 分配为 2 个字节的区域
```

备　考　　必须注意：相同枚举型数据的型解释必须保持文件之间的一致性。

变量分配寄存器数的扩展

Regexpansion
NORegexpansion

C/C++ <Other>[Miscellaneous options :] [Increase a register for register variable]

格　式　　Regexpansion
NORegexpansion

说　明　　在指定 regexpansion 选项时，扩展分配寄存器变量的寄存器个数。
在指定 noregexpansion 选项时，不扩展分配寄存器变量的寄存器个数。
如果扩展寄存器个数，通常分配到寄存器的变量个数就变多，变量的存取速度就变快。有关寄存器变量的分配规则，请参照“9.3.2(3) 有关寄存器的规则”。
在省略本选项时解释为 regexpansion。

备　考　　regexpansion/noregexpansion 的指定在 CPU 种类为 H8SX、H8S（没有指定 legacy=v4 选项）时无效。

公用表达式的优化

CMncode

C/C++ <Other>[Miscellaneous options :] [Put common subexpression on a register temporarily]

格 式 CMncode

说 明 通过将公用表达式转换为临时变量的优化，扩展对象表达式的个数。
如果通过指定 cmncode 选项扩展公用表达式优化的对象表达式，就将临时变量分配到寄存器，通常能提高目标性能。但是，如果寄存器的个数不足，就将临时变量分配到存储器，反而有可能降低目标性能。因为根据程序提高目标性能的效果不同，所以请在性能调整时试用本选项。

备 考 本选项在 CPU 种类为 H8/300 和 H8/300H 时或者在 CPU 种类为 H8S 并指定 legacy=v4 时有效。

块传送指令

EEpmov

C/C++ <Other>[Miscellaneous options :] [Use EEPMOVE in block copy]

格 式 EEpmov

说 明 在对结构体的赋值语句和局部变量声明的数组初始值的赋值表达式进行代码展开时，如果 CPU 种类为 H8SX 就使用块传送指令 MOVMD，否则使用块传送指令 EEPMOV。在传送量太大并且用块传送指令不能对应时用运行时例程展开。
在省略本选项时，就用 MOV 指令或者运行时例程展开结构体的赋值语句等。

备 考 CPU 种类为 300H/H8S（指定 legacy=v4）时，如果在 EEPMOV.W 指令执行中接受中断，就不能保证在中断处理结束后控制移动到下一条指令的运行结果。此时，对于包含有可能接受中断的函数的源文件，不能指定本选项。
在 CPU 种类为 H8SX/H8S（没有指定 legacy=v4）时，展开的代码对应中断处理。

预处理器展开时的输出限制

NOLINe

C/C++ <Other>[Miscellaneous options :] [Suppress #line in preprocessed source file]

格　式　　NOLINe

说　明　　如果指定本选项，就在展开预处理器时抑制#line 输出。

备　考　　本选项只在指定 preprocessor 选项时有效。

信息级

CHAnge_message

C/C++ <Other>[Use defined options :]

格　式　　CHAnge_message = <sub>[,…]

```
<sub>         : <错误级>[=<错误号>[- <错误号>][,…]]
<错误级>      : { Information | Warning | Error }
```

说　明　　更改信息、警告的错误级。

例

change_message=information=1001,5038-5047
将 C1001 和 C5038 到 C5047 的警告级的指定错误号更改为信息级。

change_message=warning=5007-5009
将 C5007 到 C5009 的信息级的指定错误号更改为警告级。

change_message=error=2-1024
将 C0002 到 C1024 的信息级和警告级的指定错误号更改为错误级。

change_message=information
将全部的警告级信息更改为信息级。

change_message=warning
将全部的信息级消息更改为警告级。

change_message=error
将全部的信息和警告信息更改为错误级。

备　考　　对于更改为信息级的消息，能通过 nomessage 选项的指定抑制输出。
忽视不存在的错误号。
如果指定多个本选项，就全部有效。但是，如果在多个指定中序号有重叠，最后的指定有效。

register 存储类变量优先分配

ENAble_register

C/C++ <Other> [Miscellaneous options :][Enable register declaration]

格　式　`ENAble_register`

说　明　将指定 register 存储类的变量优先分配到寄存器。

备　考　如果不能分配到寄存器，就输出信息消息：
C0101 (I) Register is not allocated to "变量名" in "函数名"。
但是，如果不能将参数分配到寄存器，就不输出该信息。
本选项在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

ANSI 依据

STRIct_ansi

C/C++ <Other> [Miscellaneous options :][Obey ansi specifications more strictly]

格　式　`STRIct_ansi`

说　明　依据 ANSI 进行以下处理：
(1) 浮点运算的结合律

备　考　在指定本选项时，运算结果有可能与 Ver.6.0 以下版本的编译程序不同。
本选项在 CPU 种类为 H8SX 和 H8S（没有指定 legacy=v4 选项）时有效。

2.2.6　CPU 选项

表 2.9　CPU 选项一览表

项目		命令行格式	对话菜单	指定内容
1	CPU 种类/ 运行模式	CPu = { AE5 | H8SXN [:<*2>] | H8SXM [:<*1>][:<*2>] | H8SXA [:<*1>][:<*2>] | H8SXX [:<*1>][:<*2>] | 2600N | 2600A [:<*1>] | 2000N | 2000A [:<*1>] | 300HN | 300HA[:<*1>] | 300 | 300L | 300Reg }	CPU [CPU:] [Multiple/Divide :]	AE5 *3 H8SX 正常模式 H8SX 中级模式 H8SX 高级模式 H8SX 最高模式 H8S/2600 正常模式 H8S/2600 高级模式 H8S/2000 正常模式 H8S/2000 高级模式 H8/300H 正常模式 H8/300H 高级模式 H8/300
2	参数保存 寄存器	REGParam = { 2 | 3 }	CPU [Change number of parameter registers from 2 (default) to 3]	使用(E)R0、(E)R1 使用(E)R0、(E)R1、(E)R2
3	结构体参数、 返回值的寄存 器分配	STRUctreg NOSTRUctreg	CPU [Pass struct parameter via register]	将 4byte 以下的结构体参数和返回值分配到寄存器
4	4byte 参数、 返回值的寄存 器分配	LONgreg NOLONgreg	CPU [Pass 4-byte parameter/ return value via register]	将 4byte 的参数和返回值分配到寄存器(cpu=300)
5	double→float 的转换	DOuble=Float	CPU [Treat double as float]	将 Double 型的变量/数值转换为 float 型
6	堆栈大小的 指定	STAck = { Small | Medium | Large }	CPU [Stack calculation]	指定堆栈计算时的大小 1byte 2byte 4byte
7	运行时型信息	RTti = { ON | OFf }	CPU [Enable/disable runtime information]	dynamic_cast、typeid 有效 无效
8	异常处理功能	EXception NOEXception	CPU [Use try, throw and catch of C++]	将异常处理功能设定为有效 将异常处理功能设定为无效
9	结构体、联合 体、类成员的 边界调整数	PAck = { 1 | 2 }	CPU [Pack struct, union and class]	将数据的边界调整数设定为 1 遵从数据的边界调整
10	8bit 绝对区域 的地址指定	SBr = <地址>	CPU [Specify SBR address :]	指定 8bit 绝对区域的起始地址
11	位域排列顺序 的指定	BIt_order = { Left | Right }	C/C++ <Object> [Bit field alloc-order :]	从高位开始保存成员 从低位开始保存成员

*1 : 地址空间的位宽
*2 : 指定乘除法器
*3 : 有关 AE5，请参照“第 17 章　面向 AE5 的功能支持”。

CPu

CPU [CPU:][Multiple/Divide :]

格　式

```
CPu = { AE5
        H8SXN                      [: <乘除法器指定>]|
        H8SXM[: <地址空间的位宽>][: <乘除法器指定>]|
        H8SXA[: <地址空间的位宽>][: <乘除法器指定>]|
        H8SXX[: <地址空间的位宽>][: <乘除法器指定>]|
        2600N                                      |
        2600A[: <地址空间的位宽>]                  |
        2000N                                      |
        2000A[: <地址空间的位宽>]                  |
        300HN                                      |
        300HA[: <地址空间的位宽>]                  |
        300 | 300L | 300Reg}

    <地址空间的位宽> : {20 | 24 | 28 | 32}
    <指定乘除法器>: {M | D | MD}          M:乘法器 D:除法器
```

说　明

指定生成的目标程序的 CPU 种类和运行模式。
在没有输入时，乘除法器指定作为“无乘除法器”处理。
子选项的一览表和可指定的位宽如表 2.10 所示。

表 2.10　cpu 选项的子选项一览表

子选项名	含　义	位宽度	乘除法器
AE5	用于 AE5 的目标	—	—
H8SXN	用于 H8SX 的正常模式	—	M, D, MD
H8SXM	用于 H8SX 的中级模式	20, 24	M, D, MD
H8SXA	用于 H8SX 的高级模式	20, 24, 28, 32	M, D, MD
H8SXX	用于 H8SX 的最高模式	28, 32	M, D, MD
2600n	用于 H8S/2600 的正常模式	—	—
2600a	用于 H8S/2600 的高级模式	20, 24, 28, 32	—
2000n	用于 H8S/2000 的正常模式	—	—
2000a	用于 H8S/2000 的高级模式	20, 24, 28, 32	—
300hn	用于 H8/300H 的正常模式	—	—
300ha	用于 H8/300H 的高级模式	20, 24	—
300	H8/300 的目标	—	—
300l	H8/300 的目标 为了与汇编程序兼容。	—	—
300reg	H8/300 的目标 为了与旧版本兼容。	—	—

（如果没有指定位宽，就设定为有下划线的默认值。）

例

```
-cpu = H8SXM:20      ; 无乘除法器，位宽为 20bit 的 H8SX 中级模式
-cpu = h8sxa:32:md   ; 有乘除法器，位宽为 32bit 的 H8SX 高级模式
-cpu = H8SXA:D       ; 有除法器，位宽为 24bit 的 H8SX 高级模式
```

备　考	在省略 cpu 选项时，参照 H38CPU 环境变量的内容。 在同时指定 cpu 选项和 H38CPU 环境变量时，优先 cpu 选项。 如果省略 cpu 选项和 H38CPU 环境变量，就发生错误。 有关 AE5 的 CPU 子选项，请参照“第 17 章　面向 AE5 功能的支持”。

参数保存寄存器

REGParam

CPU [Change number of parameter registers from 2(default) to 3]

格　式	`REGParam = { 2	3 }`
说　明	指定参数保存寄存器的个数。 在指定 regparam=2 选项时，将 ER0 和 ER1（H8/300 为 R0、R1）用作参数保存寄存器。 在指定 regparam=3 选项时，将 ER0、ER1 和 ER2（H8/300 为 R0、R1、R2）用作参数保存寄存器 。 在省略本选项时解释为 regparam=2。

结构体参数的寄存器分配

STRUctreg
NOSTRUctreg

CPU [Pass struct parameter via register]

格　式　STRUctreg

NOSTRUctreg

说　明　指定是否将结构体型的参数和返回值分配到寄存器。

在指定 nostructreg 选项时，不使用寄存器而使用存储器传递参数。

在指定 structreg 选项时，使用寄存器传递参数。

能作为参数传递的结构体大小的上限在 CPU=300 时为 2 个字节，其它 CPU 时为 4 个字节。

在省略本选项时解释为 nostructreg。

备　考　如果 CPU 为 H8/300 并指定 longreg 选项，就能分配最多 4 个字节的数据。

4byte 参数的寄存器分配

LONgreg
NOLONgreg

CPU [Pass 4-byte parameter/return value via register]

格　式　LONgreg

NOLONgreg

说　明　指定是否将 4 个字节的参数和返回值分配到寄存器。

通过指定本选项，分配到寄存器的变量为 long 型、unsigned long 型和 float 型。

在指定 nolongreg 选项时，不使用寄存器而使用存储器传递参数。

在指定 longreg 选项时，使用寄存器传递参数。

在省略本选项时解释为 nolongreg。

备　考　只有在 CPU 为 H8/300 时才能指定本选项。

CPU 不为 H8/300 时，总是分配 4 个字节的数据。

double→float 转换

DOuble=Float

CPU [Treat double as float]

格　式　　`DOuble=Float`

说　明　　将 double（双精度浮点）型的变量/数值作为 float（单精度浮点）型生成目标。。

堆栈大小的指定

STAck

CPU [Stack calculation :]

格　式　　`STAck = { Small | Medium | Large }`

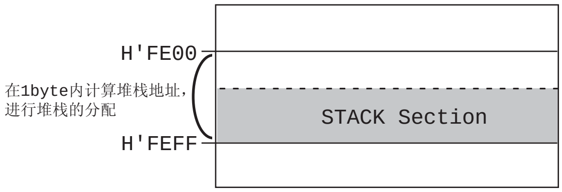
说　明　　指定堆栈大小。

在指定 stack=small 选项时，只用最低位的 1 个字节计算堆栈地址，不进位到高位字节。
在指定 stack=medium 选项时，只用最低位的 2 个字节计算堆栈地址，不进位到高位字节。
在指定 stack=large 选项时，以 4 个字节计算堆栈地址。
在省略本选项时解释为 stack=medium。

备　考

- 本选项必须在全部程序中指定相同的堆栈大小。
- 在使用的堆栈大小大于被指定的堆栈大小时，或者在超过 1 个字节、2 个字节和 4 个字节的边界值分配变量时，在编译程序中不输出错误信息或警告信息，但是，通过指定 goptimize 选项，能在连接编辑程序中输出警告信息。那时，必须增大堆栈大小。

例:
```
-stack=small
```



运行时型信息

RTti

CPU [Enable/disable runtime information]

格　式　`RTti = { ON | OFf }`

说　明　指定运行时型信息的有效/无效。
在指定 `rtti=on` 选项时，将 `dynamic_cast`、`typeid` 设定为有效。
在指定 `rtti=off` 选项时，将 `dynamic_cast`、`typeid` 设定为无效。
在省略本选项时解释为 `rtti=off`。

备　考　不能将由本选项的指定而建立的目标文件登录到库或者输出到可再定位目标文件。有可能产生符号双重定义错误或者符号未定义错误。

异常处理功能

EXception
NOEXception

CPU [Use try, throw and catch of C++]

格　式　`EXception`
`NOEXception`

说　明　在指定 `noexception` 选项时，将 C++异常处理功能设定为无效。
在指定 `exception` 选项时，将 C++异常处理功能（`try`, `catch`, `throw`）设定为有效。
如果使用异常处理功能，有可能降低代码性能。
在省略本选项时解释为 `noexception`。
不能同时指定 `exception` 选项和 `ecpp` 选项。

结构体、联合体和类成员的边界调整数

PAck

CPU [Pack struct, union and class]

格　式　　PAck = { 1 | 2 }

说　明　　指定结构体、联合体和类成员的边界调整数。

也能用#pragma　pack 扩展名指定结构体成员的边界调整数。如果指定选项和#pragma 扩展名，就优先扩展名的指定。

结构体、联合体和类成员的边界调整数与成员的最大边界调整数相同。

详细内容请参照“10.1.2(2)　复合型（C 语言）和类的型（C++语言）”。

在省略本选项时解释为 pack=2。

备　考

- 指定 pack 选项时的结构体成员的边界调整数如表 2.11 所示。

表 2.11　指定 pack 选项时的结构体、联合体和类成员的边界调整数

成员的型	pack=1	pack=2	无指定
[unsigned] char	1	1	1
[unsigned] short、[unsigned] int、[unsigned] long、浮点型、指针型	1	2	2
边界调整数为 1 的结构体、联合体和类成员	1	1	1
边界调整数为 2 的结构体、联合体和类成员	1	2	2

- 不能使用指针存取指定 pack=1 选项或者#pragma pack 1 的结构体、联合体和类成员（包含在使用指针的成员函数中的存取）。

例：（指定 cpu=2600a 和 pack=1 时）

```
struct S {
    char x;
    int y;
} s;
int *p=&s.y;    // s.y 的地址有可能为奇数
void test()
{
    s.y=1;      //能正确存取
    *p =1;      //不能正确存取
}
```

8bit 绝对区域地址值的指定

SBr

CPU [Specify SBR address :]

格　式	SBr = <地址>
说　明	本选项指定作为 8 位绝对地址区域设定的地址。 如果指定 SBR=<地址>选项，就将从<地址>开始的 1 个字节的区域用作 8 位绝对地址区域。必须用 16 进制数指定地址，忽视低 8 位。
备　考	本选项只在 CPU 种类为 H8SX 时有效。 必须在<地址>中指定能用作数据区的地址。 如果省略本选项，就给<地址>指定默认的 8 位绝对地址。有关 8 位绝对地址，请参照“19.3 短绝对地址的存取范围”。
例	`ch38 –sbr=A0000 test.c`　　假设 8 位绝对地址区域从地址 0xA0000 开始，进行编译。

位域排列顺序的指定

BIt_order

CPU [Bit field alloc-order]

格　式	BIt_order = { <u>Left</u> | Right }
说　明	指定位域成员的排列顺序。 在指定 bit_order=left 选项时，从高位开始分配成员。 在指定 bit_order=right 选项时，从低位开始分配成员。 在省略本选项时解释为 bit_order=left。
备　考	有关位域成员的分配，请参照“10.1.2　数据的内部表现”和“10.2.1　#pragma 扩展名和关键字”的#pragma bit_order。必须注意：相同数据的位域成员的排列顺序必须保持文件之间的一致性。

2.2.7　　剩余选项

表 2.12　剩余选项一览表

	项目	命令行格式	对话菜单	指定内容
1	C/C++语言的选择	LANg = { C | CPp }	— (Determined by an extension)	作为 C 程序编译 作为 C++程序编译
2	版权输出抑制	<u>LOGO</u> NOLOGO	— (nologo is always valid)	输出版权 抑制版权的输出
3	字符串中的字符码	EUc SJis LATin1	—	选择 euc 码 选择 sjis 码 选择 latin1 码
4	目标码中的汉字转换	OUtcode = { Euc | Sjis }	—	euc 码 sjis 码
5	子命令文件的选择	SUbcommand = <文件名>	—	从<文件名>指定的文件中取命令选项

C/C++语言的选择

LANg

None (Always determined by an extension)

格　式　　LANg = { C | CPp }

说　明　　指定源程序的语言。

在指定 lang=c 选项时，作为 C 程序编译。

在指定 lang=cpp 选项时，作为 C++程序编译。

如果省略本选项，就通过源程序的扩展名判断。扩展名为 c 时，作为 C 程序编译；扩展名为 cpp、cc、cp 时，作为 C++程序编译。如果不指定源程序的扩展名，就作为 C 程序编译。

例

ch38 test.c	作为 C 程序编译。
ch38 test.cpp	作为 C++程序编译。
ch38 –lang=cpp test.c	作为 C++程序编译。
ch38 test	假设为 test.c，作为 C 程序编译。

备　考　　指定 lang=c 选项时，ecpp 选项无效。

版权输出抑制

LOGO
NOLOGO

None (nologo is always available)

格　式　LOGO
NOLOGO

说　明　抑制版权的输出。
在指定 `logo` 选项时，输出版权显示。
在指定 `nologo` 选项时，抑制版权显示的输出。
在省略本选项时解释为 `logo`。

字符串中的字符码

EUc
SJis
LATin1

None

格　式　EUc
SJis
LATin1

说　明　能在字符串、字符常数和注释中记述日语或 `ISO-Latin1` 码。
主机和字符串中的代码的关系如**表 2.13** 所示。

表 2.13　主机和字符串中的代码

主机	选项指定			
	euc	sjis	latin1	无指定
PC	euc	sjis	latin1	sjis
SPARC	euc	sjis	latin1	euc
HP9000/700	euc	sjis	latin1	sjis

备　考　指定 `latin1` 选项时，`outcode` 选项无效。

目标码中的汉字码转换

OUtcode

None

格　式　　`OUtcode = { Euc | Sjis }`

说　明　　在字符串、字符常数中记述日语时，指定输出到目标程序的汉字码。

在指定 `outcode = euc` 选项时，用 `EUC` 码输出汉字码。

在指定 `outcode = sjis` 选项时，用 `SJIS` 码输出汉字码。

能通过 `euc` 或者 `sjis` 选项指定源程序中的汉字码。

子命令文件的选择

SUbcommand

None

格　式　　`SUbcommand = <子命令文件名>`

说　明　　在指定 `subcommand` 选项时，用子命令文件指定编译程序启动时的编译程序选项。子命令文件中的格式和命令行的格式相同。

例

```
opt.sub           : -show=object -debug –byteenum
命令行指定        : ch38 -cpu=2600a -subcommand=opt.sub test.c
编译程序解释      : ch38 -cpu=2600a -show=object -debug -byteenum test.c
```

第 3 章　　汇编程序的操作方法

3.1　　选项指定规则

启动汇编程序的命令行格式如下：

```
asm38 [△<选项> ...] [△<文件名> [, ...*]] [△<选项> ...]
  <选项>：-<选项> [=<子选项> [, ...]]
```

【注】 * 如果指定多个源文件名，按指定的顺序连接这些文件后的结果为汇编处理单位。因此，必须在最后的文件中记述.END 汇编程序控制指令。

3.2　　选项解说

命令行格式的英文大写字母表示缩写指定时的字母；下划线表示省略时的解释。
另外，用选项卡名<种类名>[项目]...表示对应综合开发环境的对话菜单。
选项的顺序对应综合开发环境的选项卡及其种类。

3.2.1　　源选项

表 3.1　源种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	include 文件目录	Include = <路径名>[, ...]	Assembly <Source> [Show entries for :] [Include file directories]	指定 include 文件的路径名
2	置换符号的定义	DEFine = <sub>[, ...] <sub>： <置换符号> = <字符串>	Assembly <Source> [Show entries for :] [Defines]	定义字符串的置换
3	整型预处理器 变量的定义	ASsignA = <sub>[, ...] <sub>： <变量名> = <整数常数>	Assembly <Source> [Show entries for :] [Preprocessor variables]	定义整型的预处理器变量
4	字符型预处理器 变量的定义	ASsignC = <sub>[, ...] <sub>： <变量名> = <字符串>	Assembly <Source> [Show entries for :] [Preprocessor variables]	定义字符型的预处理器变量

include 文件目录

Include

`Assembly <Source> [Show entries for :] [Include file directories]`

格　式　　`Include=<路径名>［,…］`

说　明　　`include` 选项指定包含的文件目录名。
目录名的指定遵循主机的标准指定方法。
能在 1 行的命令行中输入多个目录名的指定。
首先检索当前目录，然后按照指定的顺序检索 `include` 选项指定的目录。

例

```
asm38  aaa.mar -include=c:\usr\tmp,c: \tmp
```

如果在 `aaa.mar` 中指定 `.INCLUDE "file.h"`，就按照当前目录、`c:\usr\tmp`、`c:\tmp` 的顺序检索 `file.h`。

备　考　　与汇编程序控制语句的关系

选项	控制语句	结　果
include	（与指定无关）	由.INCLUDE 控制指令指定的目录 由 include 选项指定的目录*
（无指定）	.INCLUDE <文件名>	由.INCLUDE 控制指令指定的目录

【注】* 使用在.INCLUDE 控制指令指定的目录字符串前附加 include 选项指定的目录字符串的目录名。

置换符号的定义

DEFine

`Assembly <Source> [Show entries for :] [Defines]`

格　式　　`DEFine = <sub>[,…]`

　　　　　　`<sub>`：`<`置换符号`>=<`字符串`>`

说　明　　`define` 选项通过预处理器将置换符号置换为对应的字符串。

`define` 和 `assignc` 选项的不同功能对应 `.DEFINE` 和 `.ASSIGNC` 的不同功能。

备　考　　与汇编程序控制语句的关系

选项	控制语句	结　果
define	.DEFINE*	由 define 选项指定的字符串
	（无指定）	由 define 选项指定的字符串
（无指定）	.DEFINE	由.DEFINE 控制指令指定的字符串

【注】* 如果通过 define 选项给置换符号设定字符串，该置换符号的.DEFINE 定义就全部无效。

不能通过 `define` 选项置换下述控制指令：

`.AENDI , .AENDR , .AENDW , .AIFDEF , .END , .ENDM , .ENDF , .ENDI , .ENDS , .ENDW`

整型预处理器变量的定义

ASsignA

`Assembly <Source> [Show entries for :][Preprocessor variables]`

格　式　　ASsignA= <sub>[,…]

　　<sub>：<预处理器变量名>＝<整数常数>

说　明　　assigna 选项给预处理器变量设定整数常数。

预处理器变量名的写法和符号名的写法相同。

用基数（B'、Q'、D'、H'）和数值的组合，指定整数常数。如果省略基数而只指定数值，就作为 10 进制数处理。能指定整数常数的范围为-2,147,483,648～4,294,967,295。但是在设定负值时必须使用 10 进制数以外的基数。

例

```
asm38  aaa.mar -assigna=_$=H'FF
```

给预处理器变量_$设定 H’FF。将源程序中的预处理器变量_$的全部参照处\&_$设定为 H'FF。

备　考　　如果主机 OS 为 UNIX 并且在预处理器变量名中存在$或者表示基数的“'”符号，就必须在该符号的前面指定反斜线字符。

与汇编程序控制语句的关系

选项	控制语句	结　果
assigna	.ASSIGNA*	由 assigna 选项指定的值
	（无指定）	由 assigna 选项指定的值
（无指定）	.ASSIGNA	由.ASSIGNA 控制指令指定的值

【注】 * 如果通过 assigna 选项给预处理器变量设定值，该预处理器的 ASSIGNA 定义就无效。

字符型预处理器变量的定义

ASsignC

`Assembly <Source> [Show entries for :][Preprocessor variables]`

格　式

```
ASsignC= <sub>[,…]
     <sub>：<预处理器变量名>=<字符串>
```

说　明

assignc 选项给预处理器变量设定字符串。
预处理器变量名的写法和符号名的写法相同。
必须用双引号（"）将字符串括起来指定。
字符串最多能指定 255 个字符。

例

```
asm38  aaa.mar -assignc=_$=ON!OFF
```

给预处理器变量_$设定字符串 ON!OFF。将源程序中的预处理器变量_$的全部参照处\&_$设定为字符串 ON!OFF。

备　考

如果主机 OS 为 UNIX 并且在字符串中指定下列字符时，就必须在该字符的前面指定反斜线字符。另外，如果在下列字符的前后指定字符串，就必须用双引号（"）将字符串括起来。

- 感叹号（!）
- 双引号（"）
- 美元符号（$）
- 反引号（`）

与汇编程序控制语句的关系

选项	控制语句	结　果
assignc	.ASSIGNC*	由 assignc 选项指定的字符串
	（无指定）	由 assignc 选项指定的字符串
（无指定）	.ASSIGN	由.ASSIGNC 控制指令指定的字符串

【注】* 如果通过 assignc 选项给预处理器变量设定字符串，该预处理器变量的.ASSIGNC 定义就全部无效。

3.2.2　目标选项

表 3.2　目标种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	调试信息的输出控制	Debug NODebug	Assembly <Object> [Debug information :]	输出调试信息 不输出调试信息
2	预处理器的展开结果输出	EXPand [= <输出文件名>]	Assembly <Object> [Generate assembly source file after preprocess]	输出预处理器展开后的源程序
3	优化的指定	OPtimize NOOPtimize	Assembly <Object> [Optimize]	有优化 无优化
4	位移量大小的设定	BR_relative = < sub >	Assembly <Object> [Default of branch displacement size :]	设定分支指令的位移量的默认值大小
		< sub > ： { 8 |		设定为 8bit
		16 }		设定为 16bit
5	模块间的优化	GOptimize	Assembly <Object> [Generate file for intermodule optimization]	输出用于模式间优化的附加信息
6	目标模块的输出控制	Object [= <输出文件名>] NOObject	Assembly <Object> [Output file directory :]	输出目标文件 不输出目标文件

调试信息的输出控制

Debug
NODebug

Assembly <Object> [Debug information :]

格　式　　`Debug`
`NODebug`

说　明　　debug 选项输出调试信息。
Nodebug 选项不输出调试信息。
debug、nodebug 各选项的指定在输出目标模块时有效。

备　考　　通过调试程序进行程序调试时需要调试信息。调试信息包含源程序的行信息和符号信息（符号调试信息）等。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果（输出目标模块时）
debug	（与指定无关）	输出调试信息。
nodebug	（与指定无关）	不输出调试信息。
（无指定）	.OUTPUT DBG	输出调试信息。
	.OUTPUT NODBG	不输出调试信息。
	（无指定）	不输出调试信息。

输出预处理器的展开结果

EXPand

Assembly <Object> [Generate assembly source file after preprocess]

格　式　　`EXPand [=<输出文件名>]`

说　明　　expand 选项输出进行宏展开、条件汇编、结构化汇编和文件包含后的汇编程序源文件。
如果指定本选项，就不生成目标。
在省略输出文件的指定时，输出文件名如下：
- 在省略文件扩展名的指定时
  文件扩展名为 exp。
- 在省略主文件名和文件扩展名的指定时
  主文件名与输入的源文件（指定的第 1 个文件）相同。
  文件扩展名为 exp。

备　考　　不能给输入源文件和输出文件指定相同的文件名。

优化指定

OPtimize
NOOPtimize

`Assembly <Object> [Optimize]`

格　式　　`OPtimize`
`NOOPtimize`

说　明　　`optimize` 选项指定是否进行 PC 相对方式、带位移量的寄存器间接的位移量大小和绝对寻址方式的地址大小的优化。但是，对于 H8SX 的 MOVA 指令，优化的情况如下：

第 1 操作数	优化可否
@(disp,Reg) *1	○
@(disp,@ERn.sz) *2	○
@(disp,@±ERn.sz) *2	○
@(disp,@ERn±.sz) *2	○
@(disp,@(disp,Reg).sz) *2 *3	×
@(disp,@abs.sz) *2	×

【注】*1　Reg 为 RnL.B, RnH.B, Rn.W 或者 En.W 的情况。
*2　sz 为 B 或者 W 的情况。
*3　Reg 为 ERn, RnL.B, Rn.W 或者 ERn.L 的情况。

本选项的对象是没有指定位移量大小（：8，：16）或者没有指定绝对地址的确保长度（:8, :16, :24, :32）的执行指令。
根据 PC 相对方式的位移量的值，位移量大小设定如下：

在 H8S/2600 高级模式不指定优化时

位移量值		位移量大小
绝对值	（-32,768～32,767）	16 位*
相对值		16 位
外部参照值		16 位

【注】* 只限参照在指令后定义的绝对符号。

在 H8S/2600 高级模式指定优化时

位移量值		位移量大小
绝对值	（-128～127）	8 位
	（-32,768～-129） （128～32,767）	16 位
相对值		16 位
外部参照值		16 位

例

```
asm38 aaa.mar -optimize
```

进行目标模块的优化。

```
asm38 aaa.mar
```

不进行目标模块的优化。

备　考　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项 1	选项 2	控制指令	结　果
optimize	（与指定无关）	（与指定无关）	被优化的位数
nooptimize	br_relative	（与指定无关）	由 br_relative 选项指定的位数
	（无指定）	.DISPSIZE	.DISPSIZE 控制指令的位数
		（无指定）	8 位

【注】optimize 选项优先于目标模块输出的选项（br_relative）和控制指令（.DISPSIZE）。

位移量大小的设定

BR_relative

`Assembly <Object> [Default of branch displacement size :]`

格　式　　`BR_relative = { 8 | 16 }`

说　明　　指定分支指令的位移量为前面指令时的位移量的默认值大小。

- 8 … 指定默认值大小为 8 位。
- 16 … 指定默认值大小为 16 位。

本选项的对象是指定了位移量大小（ :8, :16 ）并且没有指定 optimize 选项时的位移量大小。

备　考　　由于在 H8/300、H8/300L 时 br_relative=8 固定，所以本选项无效。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项 1	选项 2	控制指令、CPU 种类	结　果
optimize	（与指定无关）	（与指定无关）	优化的位数
nooptimize	br_relative	（与指定无关）	由 br_relative 选项指定的位数
	（无指定）	.DISPSIZE	由.DISPSIZE 控制指令指定的位数
		（无指定） cpu= 300、300L、300HN、2000N、2600N、H8SXN	8 位
		（无指定） cpu= 300HA、2000A、2600A、H8SXM、H8SXA、H8SXX、AE5	16 位

【注】optimize 选项优先于目标模块输出的选项（br_relative）和控制指令（.DISPSIZE）。

模块间的优化

GOptimize

`Assembly <Object> [Generate file for inter-module optimization]`

格　式　　`GOptimize`

说　明　　输出用于模块间优化的附加信息。

指定本选项的文件为连接时模块间优化的对象。

目标模块的输出控制

Object
NOObject

Assembly <Object> [Output file directory :]

格　式　　<u>`Object`</u> [=<输出目标文件名>]
`NOObject`

说　明　　`object` 选项输出目标文件。
`Noobject` 选项不输出目标文件。
在省略输出目标文件的指定时，输出文件名如下：

- 在省略文件扩展名的指定时
  文件扩展名为 `obj`。
- 在省略主文件名和文件扩展名的指定时
  主文件名与输入的源文件（指定的第一个文件）相同。
  文件扩展名为 `obj`。

备　考　　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果
object	（与指定无关）	输出目标文件。
noobject	（与指定无关）	不输出目标文件。
（无指定）	.OUTPUT OBJ	输出目标文件。
	.OUTPUT NOOBJ	不输出目标文件。
	（无指定）	输出目标文件。

不能给输入源文件和输出目标文件指定相同文件名。如果指定相同文件名，输入源文件就会被盖写。

3.2.3　　列表选项

表 3.3　列表种类选项一览表

项目	命令行格式	对话菜单	指定内容
1　汇编列表的输出控制	LISt [= <输出文件名>] NOLISt [= <输出文件名>]	Assembly <List> [Generate list file]	输出列表文件 不输出列表文件
2　源程序列表的输出控制　*	SOurce NOSOurce	Assembly <List> [Source program :]	输出源程序列表 不输出源程序列表
3　源程序列表的部分输出控制　*	SHow [= <输出种类>[, …]] NOSHow [= <输出种类>[, …]] <输出种类>： {CONditionals | Definitions | CAlls | Expansions | Structured | CODe }	Assembly <List> [Source program list contents :] [Code]	控制源程序列表的部分输出
4　对照表的输出控制　*	CRoss_reference NOCRoss_reference	Assembly <List> [Cross reference :]	输出对照表 不输出对照表
5　段信息列表的输出控制　*	SEction NOSEction	Assembly <List> [Section :]	输出段信息列表 不输出段信息列表

【注】 * source/nosource、show/noshow、cross_reference/nocross_reference、section/nosection 的各选项只在指定了 list 选项时有效。

汇编列表的输出控制

LISt
NOLISt

`Assembly <List> [Generate list file]`

格　式　　`LISt [=<输出列表文件名>]`
`NOLISt [=<输出列表文件名>]`

说　明　　如果指定 `list` 选项，就输出汇编列表。
在省略输出列表文件名的指定时，输出文件名如下：
- 在省略文件扩展名的指定时
  文件扩展名为 `lis`。
- 在省略主文件名和文件扩展名的指定时
  主文件名就与输入源文件（指定的第一个文件）相同。
  文件扩展名为 `lis`。

如果指定 `nolist` 选项，就不输出汇编列表。
如果通过 `nolist` 指定文件名，就只将发生错误的行的汇编列表输出到文件中。

备　考　　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果
list	（与指定无关）	输出汇编列表。
nolist	（与指定无关）	不输出汇编列表。
（无指定）	.PRINT LIST	输出汇编列表。
	.PRINT NOLIST	不输出汇编列表。
	（无指定）	不输出汇编列表。

不能给输入源文件和输出列表文件指定相同文件名。如果指定相同文件名，输入源文件就会被盖写。

源程序列表的输出控制

SOurce
NOSOurce

Assembly <List> [Source program :]

格　式　　<u>SOurce</u>
NOSOurce

说　明　　source 选项将源程序列表附加到汇编列表。
nosource 选项不将源程序列表附加到汇编列表。
source、nosource 的指定只在输出汇编列表时有效。

备　考　　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果（汇编列表输出时）
source	（与指定无关）	输出源程序列表。
nosource	（与指定无关）	不输出源程序列表。
（无指定）	.PRINT SRC	输出源程序列表。
	.PRINT NOSRC	不输出源程序列表。
	（无指定）	输出源程序列表。

源程序列表的部分输出控制

SHow
NOSHow

Assembly <List> [Source program list contents :] [Code:]

格　式　　<u>SHow</u> [=<输出种类>[,…]]
NOSHow [=<输出种类>[,…]]
输出种类：｛CONditionals｜Definitions ｜CAlls ｜
Expansions ｜Structured ｜CODe ｝

说　明　　指定源程序列表的预处理器功能的源语句的部分输出、输出抑制以及目标码显示行的部分输出、输出抑制。

进行或者抑制由输出种类指定的项目的输出。如果省略输出种类，就进行或者抑制全部项目的输出。

・show　　・・・　输出
・noshow　・・・　抑制输出

输出种类的内容如下：

输出种类	含 义	内　容
conditionals	附带条件不成立	.AIF, .AIFDEF 的不成立部分
definitions	定义	宏定义部分 .AREPEAT, .AWHILE 定义部分 .INCLUDE 控制语句 .ASSIGNA, .ASSIGNC 控制语句
calls	调用	宏调用语句 .AIF, .AIFDEF, .AENDI 控制语句 结构化汇编控制语句
expansions	展开	宏展开部分 .AREPEAT, .AWHILE 展开部分
structured	结构化展开	结构化汇编展开部分
code	目标码显示行	控制指令的目标码显示超过源语句行数的部分

备　考　　show,noshow 的指定只在输出汇编列表时有效。
在 PC 版的情况下指定 2 个以上的输出种类时，必须用括弧（）括起来指定。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果
show=输出种类	（与指定无关）	输出
noshow=输出种类	（与指定无关）	抑制输出
（无指定）	.LIST 输出种类（输出）	输出
	.LIST 输出种类（输出抑制）	抑制输出
	（无指定）	输出

对照表的输出控制

CRoss_reference
NOCRoss_reference

`Assembly <List >[Cross reference :]`

格　式　　`CRoss_reference`
`NOCRoss_reference`

说　明　　`cross_reference` 选项将对照表附加到汇编列表。
`nocross_reference` 选项不将对照表附加到汇编列表。
`cross_reference`、`nocross_reference` 的指定只在输出汇编列表时有效。

备　考　　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果（输出汇编列表时）
cross_reference	（与指定无关）	输出对照表。
nocross_reference	（与指定无关）	不输出对照表。
（无指定）	.PRINT　CREF	输出对照表。
	.PRINT　NOCREF	不输出对照表。
	（无指定）	输出对照表。

段信息列表的输出控制

SEction
NOSEction

`Assembly <List > [Section :]`

格　式　　`SEction`
`NOSEction`

说　明　　`section` 选项将段信息列表附加到汇编列表。
`nosection` 选项不将段信息列表附加到汇编列表。
`section`、`nosection` 的指定只在输出汇编列表时有效。

备　考　　与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果（输出汇编列表时）
section	（与指定无关）	输出段信息列表。
nosection	（与指定无关）	不输出段信息列表。
（无指定）	.PRINT SCT	输出段信息列表。
	.PRINT NOSCT	不输出段信息列表。
	（无指定）	输出段信息列表。

3.2.4　　调整选项

表 3.4　调整种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	8 位或者 16 位绝对寻址方式符号的指定	ABS8 ABS16	Assembly <Tuning> [Option to set :]	指定用 8 位或者 16 位绝对寻址方式存取的符号。

8 位或者 16 位绝对地址的指定

ABS8
ABS16

Assembly <Tuning> [Option to set :]

格　式　　`ABS8[= <符号>[,…]]`
`ABS16[= <符号>[,…]]`

说　明　　abs8 选项指定用 8 位绝对寻址方式存取的符号。
abs16 选项指定用 16 绝对寻址方式存取的符号。
在省略符号时，将全部的外部参照/定义符号作为对象。
如果对于相同的符号同时指定 abs8/abs16 选项，就优先后面的指定。

- 在指定-abs8 –abs16 时
  全部的外部符号为 16 位绝对寻址方式。
- 在指定-abs8=<sym> –abs16=<sym>时
  <sym>为 16 位绝对寻址方式，而其它由 CPU 决定。

但是，如果指定符号或省略符号，处理就不同。

- 在指定-abs8=<sym> -abs16 时
  只有<sym>为 8 位绝对寻址方式，而其它为 16 位绝对寻址方式。

存取容量的优先顺序

优先顺序		存取容量的方式
高	1	绝对寻址方式的确保容量
↕	2	.IMPORT/.EXPORT/.GLOBAL 控制指令的存取容量 .ABS8/.NOABS8 控制指令
低	3	abs8/abs16 选项

例　　`asm38 aaa.mar –abs8=sym1 –abs16`

在绝对寻址方式中指定外部符号时，sym1 以 8 位绝对寻址方式存取，而其它外部符号以 16 位绝对寻址方式存取。

`asm38 aaa.mar –abs8=sym1 –abs16=sym2,sym3,sym4`

aaa.mar 的内容

```
.CPU     2600A
.IMPORT  sym1,sym2,sym3,sym5
.IMPORT  sym4:8
MOV.B    @sym1  ,R1H        ; 8 位        (指定-abs8)
MOV.B    @sym2  ,R1H        ;16 位        (指定-abs16)
MOV.B    @sym3:8,R1H        ; 8 位        (指定确保容量)
MOV.B    @sym4  ,R1H        ; 8 位        (指定.IMPORT 的存取容量)
MOV.B    @sym5  ,R1H        ;32 位        (无指定)
MOV.B    @(sym1+sym2),R1H   ; 8 位*       (混合指定-abs8,-abs16)
```

【注】如果在绝对寻址方式中记述多个外部符号，存取容量就适用最小的存取容量。

3.2.5　　其它选项

表 3.5　其它种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	未参照的外部参照符号信息的输出抑制	Exclude <u>NOExclude</u>	Assembly <Other> [Miscellaneous options :] [Remove unreferenced external symbols]	不输出未参照的外部参照符号的符号信息 输出未参照的外部参照符号的符号信息

未参照符号信息的输出抑制

Exclude
NOExclude

`Assembly <Other> [Miscellaneous options :] [Remove unreferenced external symbols]`

格　式　　Exclude

<u>NOExclude</u>

说　明　　exclude 选项不输出未参照的外部参照符号的符号信息。

noexclude 选项输出未参照的外部参照符号的符号信息。

能通过抑制输出未参照的外部参照符号的符号信息，减小目标模块的大小。

例

```
asm38 aaa.mar -exclude
```

不输出未参照的外部参照符号的符号信息。

```
asm38 aaa.mar -noexclude
```

输出未参照的外部参照符号的符号信息。

3.2.6　CPU 选项

表 3.6　CPU 选项卡选项一览表

	项目	命令行格式	对话菜单	指定内容
1	CPU 种类的指定	CPu ＝ {AE5 | H8SXN[:{M|D|MD}] | H8SXM[:<位宽>] [:{M|D|MD}] | H8SXA[:<位宽>] [:{M|D|MD}] | H8SXX[:<位宽>] [:{M|D|MD}] | 2600N | 2600A[:<位宽>] | 2000N | 2000A[:<位宽>] | 300HN | 300HA[:<位宽>] | 300 | 300L }	CPU [CPU :]	指定 CPU 种类。
2	8 位短绝对区域的基址指定	SBR	CPU [Specify SBR address :]	指定 8 位短绝对区域的基址。

CPU 种类的指定

CPu

CPU [CPU :]

格　式

```
CPu ＝ {AE5                                            |
        H8SXN [:{M|D|MD}]                              |
        H8SXM [:<地址空间的位宽>][:{M|D|MD}]           |
        H8SXA [:<地址空间的位宽>][:{M|D|MD}]           |
        H8SXX [:<地址空间的位宽>][:{M|D|MD}]           |
        2600N                                          |
        2600A [:<地址空间的位宽>]                      |
        2000N                                          |
        2000A [:<地址空间的位宽>]                      |
        300HN                                          |
        300HA [:<地址空间的位宽>]                      |
        300 | 300L                                     }
```

说　明　　指定被建立的目标程序的 CPU 种类、运行模式、地址空间的位宽和乘除法器的有无。

子选项如下：

子选项名	含　义
AE5	建立 AE5 的目标。请参照“第 17 章　面向 AE5 的功能支持”。
H8SXN[:{M|D|MD}]	建立 H8SX 正常模式的目标。 能指定乘除法器。
H8SXM [：<地址空间的位宽>][:{M|D|MD}]	建立 H8SX 中级模式的目标。 〈地址空间的位宽〉为 20 或者 24 的数值，分别表示 1M 字节、16M 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。 能指定乘除法器。
H8SXA [：<地址空间的位宽>][:{M|D|MD}]	建立 H8SX 高级模式的目标。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节和 4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。 能指定乘除法器。
H8SXX [：<地址空间的位宽>][:{M|D|MD}]	建立 H8SX 最高模式的目标。 〈地址空间的位宽〉为 28 或者 32 的数值，分别表示 256M 字节、4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 32。 能指定乘除法器。
2600N	建立 H8S/2600 正常模式的目标。
2600A [：<地址空间的位宽>]	建立 H8S/2600 高级模式的目标。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节、4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
2000N	建立 H8S/2000 正常模式的目标。
2000A [：<地址空间的位宽>]	建立 H8S/2000 高级模式的目标。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节、4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
300HN	建立 H8/300H 正常模式的目标。
300HA [：<地址空间的位宽>]	建立 H8/300H 高级模式的目标。 〈地址空间的位宽〉为 20 或者 24 的数值，分别表示 1M 字节、16M 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
300	建立 H8/300 的目标。
300L	建立 H8/300L 的目标。

如下指定乘法器/除法器的有无：

乘法器/除法器	指定方法
无/无	无指定
有/无	M
无/有	D
有/有	MD

有乘法器时的追加指令为 MAC、LDMAC、STMAC、CLRMAC、MULU/U、MULS/U。
不存在有除法器时的追加指令。

备　考

如果省略 cpu 选项，就参照 H38CPU 环境变量的内容。
如果同时指定 cpu 选项和 H38CPU 环境变量，就优先 cpu 选项。如果省略 cpu 选项和 H38CPU 环境变量，就输出错误信息 933。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	环境变量	结　果
cpu=cpu 种类	（与指定无关）	（与指定无关）	由 cpu 选项指定的 cpu 种类
（无指定）	.CPU　cpu 种类	（与指定无关）	由 cpu 控制指令指定的 cpu 种类
	（无指定）	h38cpu=cpu 种类	环境变量的 cpu 种类
		（无指定）	输出错误信息 933

8 位短绝对区域的基址指定

SBR

CPU [Specify SBR address :]

格　式　　SBR={<常数>|USER}

说　明　　SBR=<常数>以<常数>为基址，将 256 个字节的区域用作 8 位绝对地址的存取区。对于常数，指定基数 H'，最低 8 位固定为 0。

SBR=USER 根据地址空间的位宽为以下的基址：

	CPU/运行模式	8 位短绝对地址基址
H8SX 最高模式	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFF00
H8SX 高级模式	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX 中级模式	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX 正常模式	H8SXN	H'0000FF00

CPU 为 H8SXN、H8SXM、H8SXA、H8SXX 时能指定 SBR 选项。

与汇编程序控制指令的关系

选项	控制指令	8 位绝对地址存取区的基址
sbr=＜常数＞	.SBR ＜常数＞	由 SBR 控制指令指定的常数
	.SBR	由 sbr 选项指定的常数
	（无指定）	由 sbr 选项指定的常数
sbr=USER	.SBR ＜常数＞	由 SBR 控制指令指定的常数
	.SBR	由地址空间的位宽决定的值
	（无指定）	由地址空间的位宽决定的值
（无指定）	.SBR ＜常数＞	由 SBR 控制指令指定的常数
	.SBR	由地址空间的位宽决定的值
	（无指定）	由地址空间的位宽决定的值

例

```
asm38 aaa.mar -sbr=H’ff0000
   将 8 位短绝对区域设定为 H’00ff0000～H’00ff00ff。
  aaa.mar 的内容
    .CPU     H8SXX:32
    MOV.L    #H’00ff0000,ER1
    LDC.L    ER1,SBR
    MOV.B    @sym1  ,R1H        ; 8 位        （-sbr 指定 8 位短绝对区域内）
    MOV.B    @sym2  ,R1H        ;16 位        （-sbr 指定 8 位短绝对区域外）
  sym1:  .equ  H’00ff0040
  sym2:  .equ  H’ffffff40
```

备　考　　主机 OS 为 UNIX 时，对于常数，在基数 H’的“’”符号前指定反斜线符号。

3.2.7　剩余选项

表 3.7　剩余选项一览表

	项目	命令行格式	对话菜单	指定内容
1	异常结束的错误级变更	ABort={ Warning | Error }	Assembly <Other> [User defined options :]	更改汇编程序异常结束的错误级
2	ISO-Latin1 码的指定	LATIN1	Assembly <Other> [User defined options :]	选择 latin1 码
3	shift JIS 码的指定	SJIS	Assembly <Other> [User defined options :]	选项 sjis 码
4	EUC 码的指定	EUC	Assembly <Other> [User defined options :]	选项 euc 码
5	目标码内的汉字转换	OUtcode	Assembly <Other> [User defined options :]	指定输出到目标文件的汉字码
6	汇编列表的行数指定	LINes = <行数>	Assembly <Other> [User defined options :]	设定汇编列表的行数
7	汇编列表的位数指定	COlumns = <位数>	Assembly <Other> [User defined options :]	设定汇编列表的位数
8	版权输出的抑制	LOGO NOLOGO	— (nologo is always valid)	版权的输出 抑制版权的输出
9	子命令文件的指定	SUBcommand = <文件名>	—	从<文件名>指定的文件取命令选项

异常结束错误级的变更

ABort

`Assembly <Other> [User defined options :]`

格　式　　`ABort = { Warning | Error }`

说　明　　abort 选项更改错误级。
返回给 OS 的值在 1 个以上时，抑制目标模块的输出。
abort 的指定只在输出目标模块时有效。
返回给 OS 的值如下：

发生个数			指定选项时返回给 OS 的值			
警告	错误	致命错误	abort=Warning		abort=Error	
			PC	UNIX	PC	UNIX
0	0	0	0	0	0	0
1 个以上	0	0	2	1	0	0
—	1 个以上	0	2	1	2	1
—	—	1 个以上	4	1	4	1

拉丁文码字符

LATIN1

Assembly <Other> [User defined options :]

格　式	`LATIN1`
说　明	`latin1` 选项将字符串和注释中的 `ISO-Latin1` 码记述设定为可能。 `latin1` 选项不能和 `sjis`、`euc`、`outcode` 同时指定。

将汉字码指定为 shift JIS

SJIS

Assembly <Other> [User defined options :]

格　式	`SJIS`
说　明	`sjis` 选项将字符串和注释中的日语记述设定为可能。 如果指定 `sjis`，就将字符串和注释中的日语解释为 `shift JIS` 码。 如果省略 `sjis`，就将字符串和注释中的日语解释为取决于主机的日文字符码。 `sjis` 选项不能和 `latin1`、`euc` 同时指定。

将汉字码指定为 EUC

EUC

`Assembly <Other> [User defined options :]`

格　式　　EUC

说　明　　euc 选项将字符串和注释中的日语记述设定为可能。
如果指定 euc，就将字符串和注释中的日语解释为 EUC 码。
如果省略 euc，就将字符串和注释中的日语解释为取决于主机的日文字符码。
euc 选项不能和 latin1、sjis 同时指定。

设定输出汉字码

OUtcode

`Assembly <Other> [User defined options :]`

格　式　　OUtcode= <汉字码>
<汉字码> = {SJIS | EUC}

说　明　　outcode 选项将源文件中的日语记述转换为指定的汉字码并输出到文件。
根据指定的 outcode 和源文件中的汉字码（sjis、euc）输出到目标文件的汉字码如下：

outcode 的汉字码	源文件中的汉字码		
	sjis	euc	无指定
sjis	shift JIS 码	shift JIS 码	shift JIS 码
euc	EUC 码	EUC 码	EUC 码
无指定	shift JIS 码	EUC 码	默认汉字码

默认汉字码如下：

PC	shift JIS 码
SPARC 站	EUC 码
HP9000/700 系列	shift JIS 码

汇编列表的行数设定

LINes

`Assembly <Other> [User defined options :]`

格　式　　`LINes =<行数>`

说　明　　`lines` 选项设定汇编列表的每页的行数。
行数的有效值为 `20～255`。
`lines` 的指定只在输出汇编列表时有效。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果
lines=<行数>	（与指定无关）	每页为 lines 选项指定的行数。
（无指定）	.FORM LIN=<行数>	每页为由.FORM 控制指令指定的行数。
	（无指定）	每页为 60 行。

汇编列表的位数设定

COlumns

`Assembly <Other> [User defined options :]`

格　式　　`COlumns=<位数>`

说　明　　`columns` 选项设定汇编列表的每行的位数。
位数的有效值为 `79～255`。
`columns` 的指定只在输出汇编列表时有效。

与汇编程序控制指令的关系（汇编程序优先选项的指定）

选项	控制指令	结　果
columns=<位数>	（与指定无关）	每行为 columns 选项指定的位数。
（无指定）	.FORM COL=<位数>	每行为由.FORM 控制指令指定的位数。
	（无指定）	每行为 132 位。

版权输出的抑制

LOGO
NOLOGO

None (nologo is always available)

格　式　　LOGO
NOLOGO

说　明　　抑制版权的输出。
如果指定 logo，就输出版权显示。
如果指定 nologo，就抑制版权显示的输出。
省略本选项时解释为 logo。

子命令文件的指定

SUBcommand

None

格　式　　SUBcommand=<子命令文件名>

说　明　　subcommand 选项从文件输入命令行。
必须按通常指定命令行相同的顺序指定输入文件名和选项。
1 行只能指定 1 个输入文件名或者选项。
不能在子命令文件中指定 subcommand 选项。

例

```
asm38 aaa.src -subcommand=aaa.sub
```

将子命令文件的内容展开为命令行后进行汇编。

aaa.sub 的内容：

```
bbb.src
-list
-noobj
```

展开结果：

```
asm38 aaa.src,bbb.src -list -noobj
```

备　考　　整个子命令文件的大小最大为 65,535 个字节。

第 4 章　优化连接编辑程序的操作方法

4.1　选项指定规则

4.1.1　命令行的格式

命令行的格式如下：

```
optlnk [ {△<文件名> | △<选项列>}…]
  <选项列>：-<选项>[= <子选项>[,…]]
```

4.1.2　子命令文件的格式

子命令文件的格式如下：

```
  <选项> {= | △}[<子选项>[,…]] [△&] [;<注解>]
    &：继续行的指定
```

有关子命令文件格式的详细内容，请参照“4.2.8 子命令文件选项”。

4.2　选项解说

选项、子选项的英文大写字母表示缩写指定时的字母，下划线表示省略时的解释。
另外，用选项卡名<种类名>[项目]...表示对应综合开发环境的对话菜单。
选项的顺序对应综合开发环境的选项卡及其种类。

4.2.1　输入选项

表 4.1　输入种类选项一览表

项目	选项	对话菜单	指定内容
1　输入文件	Input = <sub> [{,|△}...] <sub>: <文件名> [(<模块名>[,...])]	Link/Library <Input> [Show entries for :] [Relocatable files and object files]	指定输入文件 （在命令行指定时不需要 input）
2　库文件	LIBrary = <文件名>[,…]	Link/Library <Input> [Show entries for :] [Library files]	指定输入库文件
3　二进制文件	Binary = <sub>[,...] <sub> : <文件名> (<段名> [:<边界调整数>] [,<符号名>])	Link/Library <Input> [Show entries for :] [Binary files]	指定输入二进制文件
4　符号定义	DEFine = <sub>[,...] <sub>: <符号名> = {<符号名> | <数值> }	Link/Library <Input> [Show entries for :] [Defines:]	强制定义未定义符号 定义为与符号名相同的值 数值定义
5　执行开始地址	ENTry = { <符号名> | <地址> }	Link/Library <Input> [Use entry point :]	指定入口符号 指定入口地址
6　预连接程序	NOPRElink	Link/Library <Input> [Prelinker control :]	抑制预连接程序的启动

输入文件

Input

Link/Library <Input>[Show entries for :][Relocatable files and object files]

格　式　`Input = <子选项>[{,|△}…]`
`<子选项> : <文件名>[(<模块名>[,…])]`

说　明　指定输入文件。有多个文件时，用逗号（,）或者空格分开指定。
也能指定通配符（*、?），用通配符指定的字符串按照字母顺序展开。数字先于英文字母，英文大写字母先于英文小写字母。
能指定为输入文件的有编译程序和汇编程序输出的目标文件、优化连接编辑程序输出的可再定位文件和绝对文件。另外，能通过库名（<模块名>）的格式将库中的模块指定为输入文件，指定模块名时不需扩展名。
在没有给输入文件名指定扩展名的情况下，没有模块名时假设为“obj”，有模块名时假设为“lib”。

例
```
input=a.obj lib1(e)     ; 输入 a.obj 和 lib1.lib 中的模块 e
input=c*.obj            ; 输入全部以 c 开头的扩展名为 obj 的文件
```

备　考　在指定 form=object 和 extract 时，本选项无效。
在命令行指定输入文件时不需要 input。

库文件

LIBrary

Link/Library <Input>[Show entries for :][Library files]

格　式　`LIBrary = <文件名>[,…]`

说　明　指定库文件。有多个文件时，用逗号（,）分开指定。
也能指定通配符（*、?），用通配符指定的字符串按照字母顺序展开。数字先于英文字母，英文大写字母先于英文小写字母。
没有给输入文件名指定扩展名时，假设为“lib”。
在指定 form=library 选项或者 extract 选项时，将库文件作为编辑对象库输入。
否则，在进行了被指定为输入文件的文件之间的连接处理后，从库文件检索未定义符号。
按照库选项指定的用户库文件（指定顺序）、库选项指定的系统库文件（指定顺序）和默认库（环境变量 HLNK_LIBRARY1,2,3）的顺序检索库文件中的符号。

例
```
library=a.lib,b          ; 输入 a.lib 和 b.lib。
library=c*.lib           ; 输入全部以 c 开头的扩展名为 lib 的文件。
```

二进制文件

Binary

Link/Library <Input>[Show entries for :][Binary files]

格　式

Binary = <子选项>[,…]
<子选项> : <文件名>(<段名>[:<边界调整数>][,<符号名>])
<边界调整数> : 1 | 2 | 4 | 8 | 16 | 32 (默认值为 1)

说　明

指定输入二进制文件。有多个文件时，用逗号（,）分开指定。
在没有给文件名指定扩展名时，假设为“bin”。
将输入的二进制数据作为指定段的数据进行分配。通过 start 选项指定段的地址，段不能省略。
另外，也能通过符号的指定作为定义符号进行连接。如果是在 C/C++程序中参照的变量名，就在程序中的参照名前附加“_”。
对本选项指定的段，可指定边界调整数。能给边界调整数指定的值为 2 的乘方，而不能指定其它值。
没有指定边界调整数时，默认值为 1。

例

```
input=a.obj
start=P,D*/200
binary=b.bin(D1bin),c.bin(D2bin:4,_datab)
```

将 b.bin 作为 D1bin 段，从地址 0x200 开始分配。
将 c.bin 作为 D2bin 段（边界调整数为 4），分配到 D1bin 之后。
将 c.bin 数据作为定义符号_datab 进行连接。

备　考

在指定 form={object | relocate | library}或者 strip 时，本选项无效。
另外，没有指定输入目标文件时，不能指定本选项。

符号定义

DEFine

Link/Library <Input>[Show entries for :][Defines]

格　式

DEFine = <子选项>[,…]
<子选项> : <符号名> = {<符号名> | <数值>}

说　明

用外部定义符号或者数值强制定义未定义符号。
用 16 进制数指定数值。在前头为 A～F 时，先检索符号，如果没有相应的符号，就解释为数值；在前头附加 0 时，总是解释为数值。在符号名为 C/C++变量名时，在程序中的定义名前附加“_”；为 C++函数名时（main 函数除外），将包含参数串的程序中的定义名用双引号括起来指定。但是，参数为 void 时用"函数名()"指定。

例

```
define=_sym1=data       ;将_sym1 定义为与外部定义符号 data 相同的值。
define=_sym2=4000       ;将_sym2 定义为 0x4000。
```

备　考

在指定 form={object | relocate | library}时，本选项无效。

执行开始地址

ENTry

`Link/Library <Input>[Use entry point :]`

格　式　`ENTry = {<符号名> | <地址>}`

说　明　用外部定义符号或者地址指定执行开始地址。
用 16 进制数指定地址。在前头为 A～F 时，先检索定义符号，如果没有相应的符号，就判断为地址；在前头附加 0 时，总是解释为地址。
符号名为 C 函数名时，在程序中的定义名前附加“_”；为 C++函数名时（main 函数除外），将包含参数串的程序中的定义名用双引号括起来指定。但是，参数为 void 时用"函数名()"指定。
如果在编译或者汇编时指定 entry 符号，就优先本选项的指定。

例

```
entry=_main          ;将 C/C++的 main 函数设定为执行开始地址。
entry="init()"       ;将 C++的 init 函数设定为执行开始地址。
entry=100            ;将 0x100 设定为执行开始地址。
```

备　考　在指定 form={object | relocate | library}或者 strip 时，本选项无效。
在指定未参照符号删除优化（optimize=symbol_delete）时，必须指定执行开始地址，如果没有指定，未参照符号删除优化的指定就无效。

预连接程序

NOPRElink

`Link/Library <Input>[Show entries for :][Prelinker control :]`

格　式　`NOPRElink`

说　明　抑制预连接程序的启动。
预连接程序支持 C++模板示例的自动生成功能和运行时型的检查功能。在不使用 C++模板功能和运行时型的检查功能时，请指定 noprelink 选项，提高连接速度。

备　考　在指定 extract 或者 strip 时，本选项无效。

4.2.2　输出选项

表 4.2　输出种类选项一览表

项目		选项	对话菜单	指定内容
1	输出格式	FOrm = { Absolute | Relocate | Object | Library [= {S | U}] | Hexadecimal | Stype | Binary }	Link/Library <Output> [Type of output file :]	绝对格式 可再定位格式 目标格式 库格式 HEX 格式 S 型格式 二进制格式
2	调试信息	DEBug SDebug NODEBug	Link/Library <Output> [Debug information :]	输出（输出到文件中） 输出调试信息文件 不输出
3	记录长度的统一	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }	Link/Library <Output> [Data record header:]	HEX 记录 扩展 HEX 记录 32bit HEX 记录 S1 记录 S2 记录 S3 记录
4	ROM 化支持	ROm = <sub>[,...] <sub> : <ROM 段名> =<RAM 段名>	Link/Library <Output> [Show entries for :] [ROM to RAM mapped sections:]	确保 RAM 区，用 RAM 地址再定位符号
5	输出文件	OUtput = <sub>[,…] <sub> : <文件名> [=<输出范围>] <输出范围>: { <起始地址> - <结束地址> | <段名>[:...] }	Link/Library <Output> [Show entries for :] [Output file path/ Messages] or [Divided output files:]	指定输出文件 （可指定范围，可分割输出）
6	外部符号分配信息文件	MAp [= <文件名>]	Link/Library <Output> [Generate map file]	指定外部符号分配信息文件的输出（面向 SuperH）
7	空区域的输出指定	SPace [= <数值>]	Link/Library <Output> [Specify value filled in unused area] [Output padding data]	指定空区域的输出值
8	信息消息	Message NOMessage [= <sub>[,...]] <sub> : <错误号> [- <错误号>]	Link/Library <Output> [Show entries for :] [Output file path/ Messages] [Repressed information level messages:]	输出 不输出 （可指定错误号和范围）
9	没被参照的定义符号的通知	MSg_unused	Link/Library <Output> [Show entries for :] [Notify unused symbol:]	通过信息输出通知 1 次也没被参照的定义符号
10	段内数据的紧凑分配	DAta_stuff	Link/Library <Output> [Show entries for :] [Reduce empty areas of boundary alignment:]	在紧凑编译单位之间的空区域后分配数据

输出格式

FOrm

Link/Library <Output>[Type of output file :]

格　式　　`FOrm = {Absolute | Relocate | Object | Library[={S|U}] | Hexadecimal | Stype | Binary}`

说　明　　指定输出格式。
在省略本选项时解释为 `form=absolute`。子选项一览表如表 4.3 所示。

表 4.3　form 选项的子选项一览表

子选项名		内　容
1	absolute	输出绝对文件。
2	relocate	输出可再定位文件。
3	object	输出目标文件。在通过 extract 选项从库中取出 1 个模块作为目标文件时使用。
4	library	输出库文件。 在指定 library=s 时，将输出库文件作为系统库。 在指定 library=u 时，将输出库文件作为用户库。 在省略时解释为 library=u。
5	hexadecimal	输出 HEX 文件。HEX 格式请参照“19.1.2　HEX 文件格式”。
6	stype	输出 S 型文件。S 型格式请参照“19.1.1　S 型文件格式”。
7	binary	输出二进制文件。

备　考　　输出格式与输入文件及其它选项的关系如表 4.4 所示。

表 4.4 输出格式与输入文件及其它选项的关系

	输出格式	指定选项	能输入的文件格式	能指定的选项*1
1	Absolute	有 strip	绝对文件	input, output, hide, show=symbol,reference
		其它	目标文件 可再定位文件 二进制文件 库文件	input, library, binary, debug/nodebug, sdebug, cpu, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, compress, rename, delete, define, fsymbol, stack, noprelink, memory, msg_unused, data_stuff, show=symbol, reference,xreference
2	Relocate	有 extract	库文件	library, output, show=symbol,reference
		其它	目标文件 可再定位文件 二进制文件 库文件	input, library, debug/nodebug, output, hide, rename, delete, noprelink, msg_unused, data_stuff, show=symbol,reference, xreference
3	Object	有 extract	库文件	library, output, show=symbol
4	Hexadecimal Stype Binary		目标文件 可再定位文件 二进制文件 库文件	input, library, binary, cpu, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, absolute_forbid, profile, cachesize, sbr, rename, delete, define, fsymbol, stack, noprelink, record, s9*2, memory, msg_unused, data_stuff, show=symbol,reference,xreference
			绝对文件	input, output, record, s9*2, show=symbol,reference,xreference
5	Library	有 strip	库文件	library, output, hide, show=symbol,section
		有 extract	库文件	library, output, show=symbol,section
		其它	目标文件 可再定位文件	input, library, output, hide, rename, delete, replace, noprelink, show=symbol,section

【注】 *1：总是能指定 message/nomessage、change_message、logo/nologo、form、list、subcommand。

*2：只能在输出格式为 form=stype 时指定 s9。

调试信息

DEBug
SDebug
NODEBug

Link/Library <Output>[Debug information :]

格　式	`DEBug`
	`SDebug`
	`NODEBug`

说　明	指定是否输出 debug 信息。 debug 选项将调试信息输出到输出文件中。 sdebug 选项将调试信息输出到<输出文件名>.dbg 文件中。 nodebug 选项不输出调试信息。 在指定 form=relocate 时，如果指定 sdebug 选项，就解释为 debug 选项。 在通过 output 选项指定多个文件输出时，如果指定 debug 选项，就解释为 sdebug 选项，并输出到<开始输出文件名>.dbg。 在省略本选项时解释为 debug。
备　考	在指定 form={object | library | hexadecimal | stype | binary}、strip 或者 extract 时，本选项无效。

记录长度的统一

REcord

Link/Library <Output>[Data record header :]

格　式	`REcord = {H16 | H20 | H32 | S1 | S2 | S3}`
说　明	与地址范围无关，以一定的数据记录输出。 如果存在比指定的数据记录大的地址，就按照地址选择数据记录。 在省略本选项时，输出符合各地址的数据记录。
备　考	在没有指定 form=hexadecimal 或者 stype 时，本选项无效。

ROM 化支持

ROm

Link/Library <Output>[Show entries for :][ROM to RAM mapped sections]

格　式　　ROm = <子选项>[,…]
<子选项> : <ROM 段名>=<RAM 段名>

说　明　　确保用于初始化数据区的 ROM 区和 RAM 区，将 ROM 段内的定义符号再定位为 RAM 段内的地址。
给 ROM 段指定带初始值的可再定位段。
给 RAM 段指定不存在的段或者容量为 0 的可再定位段。

例　　rom=D=R
start=D/100,R/8000
确保与 D 段相同容量的 R 段，将 D 段内的定义符号再定位为 R 段内的地址。

备　考　　在指定 form={object | relocate | library}或者 strip 时，本选项无效。

输出文件

OUtput

Link/Library <Output> [Show entries for :][Output file path/ Messages] or [Divided output files]

格　式　　OUtput ＝ <子选项>[,…]
<子选项> :　<文件名>[=<输出范围>]
<输出范围>: {<起始地址>-<结束地址> | <段名>[:…]}

说　明　　指定输出文件名。在 form={absolute | hexadecimal | stype | binary}时，能指定多个文件。用 16 进制数指定地址。前头为 A～F 时先检索段，如果没有相应的段，就判断为地址。如果在前头附加 0，总是解释为地址。
在省略本选项时解释为<第一个输入文件名>.<默认扩展名>。
默认扩展名如下：
form=absolute：「abs」、form=relocate　：「rel」、form=object：「obj」
form=library　：「lib」、form=hexadecimal：「hex」、form=stype　：「mot」
form=binary　：「bin」

例　　output=file1.abs=0-ffff,file2.abs=10000-1ffff
将 0～0xffff 输出到 file1.abs、0x10000～0x1ffff 输出到 file2.abs。

output=file1.abs=sec1:sec2,file2.abs=sec3
将 sec1 和 sec2 段输出到 file1.abs、sec3 段输出到 file2.abs。

外部符号分配信息文件的输出

MAp

Link/Library <Output>[Generate map file]

格　式　　MAp [= <文件名>]

说　明　　输出编译程序在外部变量的存取优化中使用的外部变量分配信息文件。
如果不指定<文件名>，就输出由 output 选项指定的文件名或者文件名为第一个输入文件名且扩展名为 bls 的文件。
在建立外部变量分配信息文件时的变量声明顺序和读取重新编译后的目标时的变量声明顺序发生变化时，输出错误。

备　考　　本选项只在指定 form={absolute | hexadecimal | stype | binary}时有效。

空区域的输出指定

SPace

Link/Library <Output>[Show entries for :][Specify value filled in unused area]
[Output padding data]

格　式　　SPace [=　<数值>]

说　明　　指定填入输出范围的存储器空区域的 16 进制数值。
填入空区域的方法根据 output 选项指定时的输出范围指定方法有以下不同：

- 输出范围:段指定
  如果在被指定的段之间存在空区域，就输出指定数据
- 输出范围:地址范围指定
  如果在被指定的范围内存在空区域，就输出指定数据

输出数据长度以 1、2、4 个字节单位为有效。输出数据长度由 space 选项指定的 16 进制数的个数决定。如果指定 3 个字节数据，就对高位进行 0 扩展，作为 4 个字节的数据处理。另外，如果指定奇数位数据，也对高位进行 0 扩展，作为偶数位输入处理。
空区域的容量不是输出数据长度的倍数时，尽量输出，并通过信息发出警告。

备　考　　如果没有给本选项指定数值，就不输出到空区域。
本选项只在指定 form={ binary |stype | hexadecimal}选项时有效。
如果没有通过 output 选项指定输出范围，本选项的指定无效。

信息消息

Message
NOMessage

Link/Library <Output>[Show entries for :] [Output file path/ Messages]

[Repressed information level messages :]

格　式　Message

NOMessage [= <子选项>[,…]]

<子选项> : <错误号>[-<错误号>]

说　明　指定是否输出信息级消息。

在指定 message 选项时，输出信息级消息。

在指定 nomessage 选项时，抑制信息级消息的输出。另外，如果指定错误号，就能抑制指定错误号的信息输出。也能用连字符(-)指定抑制的错误号的范围。如果将警告和错误级信息序号指定为错误号，就假设由 change_message 更改为信息级消息，抑制信息输出。

在省略本选项时解释为 nomessage。

例　nomessage=4,200-203,1300

抑制 L0004、L0200～L0203 和 L1300 的信息输出。

没被参照的定义符号的通知

MSg_unused

Link/Library <Output>[Show entries for :] [Output Messages] [Notify unused symbol:]

格　式　MSg_unused

说　明　如果指定本选项，就通过信息输出将连接处理中一次也没被参照的外部定义符号通知用户。

例　optlnk -msg_unused a.obj

备　考　在输入文件为 absolute 格式时，本选项指定无效。

为了输出信息，需要同时指定 message 选项。

在以下的情况下，因为不能正确解析参照关系，所以由信息输出通知的信息不正确。

- 汇编时不指定-goptimize 选项，并且在同一个文件中存在向同一个段转移的情况（只在 cpu 为 H8 系列时）
- 参照同一个文件中的常数符号
- 编译时指定优化，并且调用紧接的函数的情况
- 编译时 map 优化有效的情况（只在 cpu 为 SH 系列时）
- 在 C 源程序中记述#pragma tbr 时直接指定偏移量的情况（只在 cpu 为 SH2A/SH2AFPU 时）
- 通过连接时的优化，常数和字面常量被统一的情况

段内数据的紧凑分配

DAta_stuff

Link/Library <Output>[Show entries for :] [Reduce empty areas of boundary alignment:]

格　式　DAta_stuff

说　明　在连接时，紧凑分配段内的数据。作为本选项功能对象的段为常数区、初始化数据区和未初始化数据区。
如果指定本选项，就在减少由编译单位的段边界调整产生的空区域后进行连接。
但是，不改变数据的分配顺序。
在不指定本选项时，按照编译单位的段的边界调整进行连接。通过本选项的指定，能紧凑由边界调整产生的冗长的空区域，减小全数据段的容量。

例

```
<tp1.c>             <tp2.c>
----------------    -------------------
long a;             char  d;
char b,c;           long  e;
                    char  f;
```

<编译后的数据段容量（SH 编译的输出例子）>

```
  tp1.obj : 4+1+1 = 6 个字节
  tp2.obj : 1+3[*]+4+1 = 9 个字节
```

<连接后的 tp1.obj 和 tp2.obj 的数据段容量>

1) 没有 data_stuff 指定
   按照各段的边界调整连接目标文件（传统处理）。
   6 个字节[tp1] + 2 个字节[*] + 9 个字节[tp2] = 17 个字节

2) 有 data_stuff 指定
  紧凑分配段内的数据，紧凑由边界调整产生的冗长的空区域后进行连接。
  (4+1+1)字节 + 1 个字节 + 1 个字节[*] + 4 个字节 + 1 个字节 = 13 个字节

【注 1】 * : 因边界调整而产生的空区域

【注 2】 因编译后的数据段容量根据编译时的选项指定等而变，所以可能与上述例子不同。

备　考　如果在连接指定 SH 编译程序的 smap 选项的目标文件时指定本选项，就不保证运行。
对于汇编程序输出的目标文件，不适用本选项功能。
在下述的任一条件时，本选项指定无效。

- 指定 form=library,object 时
- 输入绝对下载模块时
- 指定 memory=low 时
- 没有指定 nooptimize 时

对于指定本选项而生成的可再定位文件，不适用连接时的优化。

4.2.3　　列表选项

表 4.5　列表种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	列表文件	LISt [= <文件名>]	Link/Library <List> [Generate list file]	指定列表文件输出
2	列表内容	SHow [= <sub>[,...]] <sub> : { SYmbol | Reference | SEction | Xreference }	Link/Library <List> [Contents :]	 符号信息 参照次数 段信息 参照表信息

列表文件

LISt

`Link/Library <List>[Generate list file]`

格　式　　LISt [= <文件名>]

说　明　　指定列表文件输出和列表文件名。

不指定列表文件名时，建立的列表文件的文件名为和输出（或者开始输出）文件相同的文件名，其扩展名在指定 form=library 或者 extract 时为“lbp”，否则为“map”。

列表内容

SHow

Link/Library <List>[Contents :]

格　式　`SHow[= <sub>[,…]]`

`<sub> : { SYmbol | Reference | SEction | Xreference }`

说　明　指定列表的输出内容。

子选项一览表如表 4.6 所示。

关于各列表的具体例子请参照“8.4 连接列表的参照方法”和“8.5 库列表的参照方法“。

表 4.6　show 选项的子选项一览表

	输出格式	子选项名	含义
1	指定 form=library 或者 extract 时	symbol	输出模块内符号名一览表。
		reference	不能指定。
		section	输出模块内段一览表。
		xreference	不能指定。
2	form=library 以外 或者 没有指定 extract 时	symbol	输出符号地址、长度、种类和优化内容。
		reference	输出符号的参照次数。
		section	不能指定。
		xreference	输出参照表信息。

备　考　`show=reference` 在指定 `form={ object | relocate }`时无效。

`show=xreference` 在指定 `form=library` 时无效。

关于参照表信息的输出，有以下限制:

- 输入文件为 `absolute` 格式时，不输出参照侧地址的信息。
- 在汇编时没有指定`-goptimize` 选项时，不输出有关向同一文件内转移的信息（只在 `cpu` 为 `H8` 系列时）。
- 不输出有关参照同一文件内的常数符号的信息。
- 不输出在编译时优化有效并且调用紧接的函数时的信息。
- 在 `map` 优化有效时，除了基准符号外，不输出变量的参照信息（只在 `cpu` 为 `SH` 系列时）。
- 如果在 `C` 源程序中记述`#pragma tbr` 时直接指定偏移值，就不输出该函数的信息（只在 `cpu` 为 `SH2A/SH2AFPU` 时）。
- 在连接时指定优化的情况下，如果常数和字面常量被统一，就不输出有关此常数和字面常量的参照信息。

4.2.4　　优化选项

表 4.7　优化种类选项一览表

<table>
<tr><th colspan="2">项目</th><th>命令行格式</th><th>对话菜单</th><th>指定内容</th></tr>
<tr><td>1</td><td>优化</td><td>OPtimize [= <sub>[,...]]
<sub> : { STring_unify
| SYmbol_delete
| Variable_access
| Register
| SAMe_code
| SHort_format
| Function_call
| Branch
| SPeed
| SAFe }
NOOPtimize</td><td>Link/Library
<Optimize>
[Show entries for :]
[Optimize items]
[Optimize :]</td><td>进行优化
统一常数/字符串
删除未参照符号
有效地利用短绝对寻址方式
进行寄存器保存/恢复的优化
统一共用码
缩短寻址方式
有效地利用间接寻址方式
优化分支指令
进行速度重视的优化
安全优化
不进行优化</td></tr>
<tr><td>2</td><td>共用码长度</td><td>SAMESize = <长度>
(省略时：sames=1e)</td><td>Link/Library
<Optimize>
[Eliminated size :]</td><td>指定作为共用码统一对象的最小长度</td></tr>
<tr><td>3</td><td>配置信息</td><td>PROfile = <文件名></td><td>Link/Library
<Optimize>
[Include profile :]</td><td>指定配置信息文件
（进行动态优化）</td></tr>
<tr><td>4</td><td>高速缓存容量</td><td>CAchesize = <sub>
<sub>: Size = <容量>　　|
Align = <行容量>
(省略时：ca=s=8,a=20)</td><td>Link/Library
<Optimize>
[Cache size :]</td><td>指定高速缓存的容量
指定高速缓存的行容量</td></tr>
<tr><td>5</td><td>优化的部分抑制</td><td>SYmbol_forbid =
<符号名>[,...]
SAMECode_forbid =
<函数名>[,...]
Variable_forbid =
<符号名>[,...]
FUnction_forbid =
<函数名>[,...]
Absolute_forbid =
<地址> [+ <容量>] [,...]</td><td>Link/Library
<Optimize>
[Show entries for :]
[Forbid item]</td><td>未参照符号删除的抑制符号
共用码统一的抑制符号
有效地利用短绝对寻址方式的抑制符号
有效地利用间接寻址方式的抑制符号
优化抑制的地址范围</td></tr>
</table>

OPtimize
NOOPtimize

```
Link/Library <Optimize>[Show entries for :][Optimize items][Optimize :]
```

格　式

```
OPtimize[= <子选项>[,…]]
NOOPtimize

<子选项> : {STring_unify | SYmbol_delete | Variable_access
           | Register | SAMe_code | SHort_format
           | Function_call | Branch | SPeed | SAFe}
```

说　明

指定是否进行模块之间的优化。
在指定 `optimize` 选项时，对编译或者汇编时指定 `goptimize` 选项的文件进行优化。
在指定 `nooptimize` 选项时，不进行模块的优化。
在省略本选项时解释为 `optimize`。子选项一览表如表 4.8 所示。

表 4.8　optimize 选项的子选项一览表

子选项	含　义	优化对象程序[1]			
		SHC	SHA	H8C	H8A
无参数	执行全部优化。	○	×	○	○
string_unify	对于具有 const 属性的常数，统一相同值的常数。 具有 const 属性的常数包含以下内容： • 在 C/C++程序中 const 声明的变量 • 字符串数据的初始值/字面常数	○	×	○	×
symbol_delete	删除 1 次也没有参照的变量/函数。必须指定 entry 选项。	○	×	○	×
variable_access	以 8 位/16 位绝对寻址方式将存取次数多的变量分配到可存取的区域。必须指定 cpu 选项。	×	×	○	○
register	解析函数的调用关系，重新分配寄存器并删除冗长的寄存器保存/恢复码。必须指定 entry 选项。	○	×	○	×
same_code	将多个相同的指令序列进行子例程化。	○	×	○	×
short_format	在能缩短位移量/立即数的代码长度时，置换为代码长度较短的指令。	×	×	○	○
function_call	如果在 0～0xFF 的范围内有空区域，就分配存取次数多的函数地址。另外，在 CPU 种类为 H8SX 时，也使用下述区域： H8SXN：0x100～0x1FF H8SXM、H8SXA、H8SXX：0x200～0x3FF 必须指定 cpu 选项。	×	×	○	○
branch	基于程序的分配信息，优化分支指令长度。如果执行其它优化项目，与有无指定无关，一定执行此选项。	○	×	○	○
speed	执行不会引起目标速度下降的优化项目。 与 optimize=string_unify、symbol_delete、varaible_access、register、short_format、branch 相同。	○	×	○	○
safe	执行不会由变量或函数的属性限制的优化项目。 与 optimize=string_unify、register、short_format、branch 相同。	○	×	○	○

【注】*1:　SHC：用于 SH 的 C/C++程序　SHA：用于 SH 的汇编程序
　　　　　H8C：用于 H8 的 C/C++程序　H8A：用于 H8 的汇编程序

备　考　　在指定 form=｛object | relocate | library｝或者 strip 时，本选项无效。
如果在编译时指定 map 优化，常数/字面常量的统一优化（optimize=string_unify）无效。
optimize=short_format 的指定只在 CPU 种类为 H8SX 时有效。

共用码长度

SAMESize

Link/Library <Optimize>[Eliminated size :]

格　式　　SAMESize = <长度>

说　明　　用共用码统一优化（optimize=same_code）指定优化对象的最小代码长度。必须用 8～7FFF 的 16 进制数指定。
在省略本选项时解释为 samesize=1E。

备　考　　本选项在没有指定 optimize=same_code 时无效。

配置信息

PROfile

Link/Library <Optimize>[Include profile :]

格　式　　PROfile = <文件名>

说　明　　指定配置信息文件。

能指定为配置信息文件的只有 Ver.5.0 以上的日立调试接口或者 Ver.2.0 以上的 HEW 输出的配置信息文件。

如果指定配置信息文件，就能通过模块之间的优化执行基于动态信息的优化。

受配置信息输入影响的优化如表 4.9 所示。

表 4.9　配置信息和优化的关系

子选项	含　义	优化对象程序[1]			
		SHC	SHA	H8C	H8A
variable_access	优先分配动态存取次数多的变量。	×	×	○	○
function_call	降低动态存取次数多的函数的优化优先顺序。	×	×	○	○
branch	将动态调用次数多的函数分配到调用函数的附近。 对用于 SH 的程序，根据用 cachesize 选项指定的高速缓存容量进行分配优化。	○	△[2]	○	○

【注】 *1　SHC：用于 SH 的 C/C++程序　SHA：用于 SH 的汇编程序、
H8C：用于 H8 的 C/C++程序　H8A：用于 H8 的汇编程序

*2　不进行函数单位的移动，但是进行输入文件单位的移动。

备　考　　本选项在没有指定 optimize 时无效。

高速缓存容量

CAchesize

Link/Library <Optimize>[Cache size :]

格　式　　`CAchesize = <sub>`
`<sub>:Size = <容量> | Align = <块容量>`

说　明　　指定高速缓存容量和高速缓存块容量。
在指定 profile 选项时，用于分支指令优化（optimize=branch）。
容量以千字节为单位、块容量以字节为单位，用 16 进制数指定。
在省略本选项时解释为 cachesize=size=8,align=20。

备　考　　本选项在没有指定 profile 时无效。

优化的部分抑制

SYmbol_forbid
SAMECode_forbid
Variable_forbid
FUnction_forbid
Absolute_forbid

Link/Library <Optimize>[Show entries for :][Forbid item]

格　式
```
SYmbol_forbid = <符号名>[,…]
SAMECode_forbid = <函数名>[,…]
Variable_forbid = <符号名>[,…]
FUnction_forbid = <函数名>[,…]
Absolute_forbid = <地址>[+ <容量>][,…]
```

说　明　　抑制特定的符号和地址范围的优化。必须用 16 进制数指定地址和容量。对于 C/C++变量名和 C 函数，在程序中的定义名前附加“_”。在 C++函数的情况下，将含有参数串的程序中的定义名用双引号括起来指定。但是，在参数为 void 时，用“函数名()”指定。选项一览表如**表 4.10** 所示。

表 4.10　优化的部分抑制的选项一览表

选项	参数	含　义
symbol_forbid	函数名|变量名	抑制未参照符号的删除优化。
samecode_forbid	函数名	抑制共用码的统一优化。
variable_forbid	变量名	抑制短绝对寻址方式的有效利用优化。
function_forbid	函数名	抑制间接寻址方式的有效利用优化。
absolute_forbid	地址[+容量]	抑制地址＋容量的范围优化。

例　　`symbol_forbid="f(int)"`　; 即使参照次数为 0，也不删除 C++函数 f(int)。

备　考　　本选项在没有指定 optimize 时无效。

4.2.5　段选项

表 4.11　段种类选项一览表

	项目	命令行格式	对话菜单	指定内容
1	段地址	STAR t= <sub>[,...] <sub> : <段名> [{: | ,} <段名>[,...]] [/<地址>]	Link/Library <Section> [Show entries for :] [Section]	指定段的起始地址
2	符号地址文件	FSymbol = <段名>[,...]	Link/Library <Section> [Show entries for :] [Symbol file]	输出外部定义符号地址的定义文件

段地址

STARt

Link/Library <Section>[Show entries for :][Section]

格　式

```
STARt = <子选项>[,…]
<子选项> : <段名>[{:|,} <段名>[,…]][/<地址>]
```

说　明

指定段的起始地址。必须用 16 进制数指定地址。
也能用冒号（:）的分开指定，给相同地址分配多个段。
也能用通配符（*）指定段名。按照输入顺序展开用通配符指定的段。
在对相同地址指定分配的段之间，按照指定顺序进行分配。
按照输入文件的指定顺序和输入库的指定顺序分配相同段内的目标。
在没有指定地址时，分配到地址 0。
将没有用 `start` 选项指定的段分配到最后分配地址的后面。

例

```
start=P,C,D*/100,R1:R2/8000   ; 假设以 D 开始的段有 D1,D2
ROM=D1=R1,D2=R2
```

按照 P、C、D1、D2 的顺序，从地址 0x100 开始分配。R1、R2 都被分配到地址 0x8000。

```
input=a.obj b.obj   ; a.obj 参照 d.lib 内的符号，b.obj 参照 c.lib 内的符号
library=c.lib,d.lib
start=P/100
```

P 段内的分配顺序为 a(P)、b(P)、c(P)、d(P)。

备　考

本选项在指定 `form={object | relocate | library}`或者 `strip` 时无效。

符号地址文件

FSymbol

Link/Library <Section>[Show entries for :][Symbol file]

格　式	`FSymbol = <段名>[,…]`
说　明	以汇编程序控制指令格式将指定的段内外部定义符号输出到文件。 文件名为<输出文件>.fsy。

例

```
fsymbol=sct2,sct3
output=test.abs
```

将段 sct2、sct3 的外部定义符号输出到 test.fsy。

[test.fsy 的输出例子]

```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
;fsymbol = sct2, sct3

;SECTION NAME = sct2
 .export _f
_f: .equ h'00000000
 .export _g
_g: .equ h'00000016
;SECTION NAME = sct3
 .export _main
_main: .equ h'00000020
 .end
```

备　考　本选项在指定 form={object | relocate | library}或者 strip 时无效。

4.2.6　　验证选项

表 4.12　验证种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	地址一致性的检查	CPu = { <cpu 信息文件名> | <存储器种类> = <地址范围>[,...] } <存储器种类> = { ROm | RAm | XROm | XRAm | YROm | YRAm } <地址范围>: <起始地址> - <结束地址>	Link/Library <Verify> [CPU information check :]	指定 cpu 信息文件 指定可分配的段地址范围

地址一致性的检查

CPu

Verify[CPU information check:]

格 式

```
CPu = { <cpu 信息文件名>
        | <存储器种类> = <地址范围>[,…] }
<存储器种类> = { ROm | RAm | XROm | XRAm | YROm | YRAm }
<地址范围> : <起始地址> - <结束地址>
```

说 明

检查段的分配地址的一致性。
xrom/xram 指定 DSP 的 X 存储区；yrom/yram 指定 DSP 的 Y 存储区。
必须用 16 进制数指定可分配的段地址范围。ROM/RAM 的属性用于模块间优化。
也能指定由旧产品附属的 cia（cpu information analyzer）建立的 cpu 信息文件。

例

`cpu=ROM=0-FFFF,RAM=10000-1FFFF`
检查段地址是否在 0-FFFF 或者 10000-1FFFF 之间。
在模块之间的优化中，不进行不同属性之间的目标转移。

备 考

本选项在指定 form={object | relocate | library}或者 strip 时无效。
在 CPU 种类不为 SHDSP,SH2DSP,SH3DSP,SH4ALDSP 时，存储器种类的 xrom、xram、yrom、yram 指定无效。

4.2.7　其它选项

表 4.13　其它种类选项一览表

项目		命令行格式	对话菜单	指定内容
1	结束码	S9	Link/Library <Other> [Miscellaneous options :] [Always output S9 record at the end]	总是输出 S9 记录
2	堆栈信息文件	STACk	Link/Library <Other> [Miscellaneous options :] [Stack information output]	输出堆栈使用量的信息文件
3	调试信息的压缩	COmpress NOCOmpress	Link/Library <Other> [Miscellaneous options :] [Compress debug information]	压缩调试信息 不压缩调试信息
4	存储器使用量的削减指定	MEMory = [High ｜ Low]	Link/Library <Other> [Miscellaneous options :] [Low memory use during linkage]	指定装入输入文件时的存储器使用量
5	符号名的更改	REName = <sub>[,...] <sub> : { [<文件名>] (<名称>=<名称>[,...]) | [<模块名>] (<名称>=<名称>[,...]) }	Link/Library <Other> [User defined options :]	更改符号名和段名
6	符号名的删除	DELete = <sub>[,...] <sub> : { <模块名> | [<文件名>] (<名称>[,...]) }	Link/Library <Other> [User defined options :]	删除符号名和模块名
7	模块的置换	REPlace = <sub>[,...] <sub> : <文件> [(<模块>[,...])]	Link/Library <Other> [User defined options :]	置换库文件中的同名模块
8	模块的抽出	EXTract = <模块>[,...]	Link/Library <Other> [User defined options :]	抽出库文件中指定的模块
9	调试信息的删除	STRip	Link/Library <Other> [User defined options:]	删除绝对文件和库文件的调试信息
10	信息级	CHange_message = <sub>[,...] <sub>: {Information | Warning | Error} [=<错误号> [-<错误号>] [,...]]	Link/Library <Other> [User defined options:]	更改信息级
11	局部符号名的隐含指定	Hide	Link/Library <Other> [User defined options:]	删除局部符号名的信息

结束码

S9

Link/Library <Other>[Miscellaneous options :][Always output S9 record at the end]

格　式　　S9

说　明　　即使入口地址超出 0x10000，也将 S9 记录输出到末端。

备　考　　本选项在没有指定 form=stype 时无效。

堆栈信息文件

STACk

Link/Library <Other>[Miscellaneous options :][Stack information output]

格　式　　STACk

说　明　　输出堆栈使用量的信息文件。
文件名为<输出文件名>.sni。

备　考　　本选项在指定 form={object | relocate | library}和 strip 时无效。

调试信息的压缩

COmpress
NOCOmpress

`Link/Library <Other>[Miscellaneous options :][Compress debug information]`

格　式　　`COmpress`

`NOCOmpress`

说　明　　指定是否压缩调试信息。
在指定 `compress` 选项时，压缩调试信息。
在指定 `nocompress` 选项时，不压缩调试信息。
如果压缩调试信息，就会提高调试程序的装入速度。另外，如果指定 `nocompress` 选项，就会提高连接速度。
在省略本选项时解释为 `nocompress`。

备　考　　本选项在指定 `form={object | relocate | library | hexadecimal | stype | binary}`或者 `strip` 选项时无效。

存储器使用量的削减指定

MEMory

`Link/Library <Other>[Miscellaneous options :][Low memory use during linkage]`

格　式　　`MEMory ＝ [ High | Low ]`

说　明　　指定连接时的存储器使用量。
在指定 `memory=high` 选项时，进行通常处理。
在指定 `memory=low` 选项时，以小单位装入连接时所需的信息，减少存储器的使用量。由于文件存取频率的增加，所以在存储器使用量不超过实际存储器容量的情况下，和指定 `memory=high` 选项相比处理速度慢。

在连接大规模的工程时，如果优化连接编辑程序的存储器使用量超过工作计算机的实际存储器容量并且运行变慢时，请指定 `memory=low` 选项试一试。

备　考　　如果指定下述选项，本选项指定无效。
`Optimize`、`compress`、`delete`、`rename`、`map`、`stack`、`list` 和 `show=reference`（同时指定时）
另外，根据输入文件和输出文件的格式，有些组合也可能无效。详细内容请参照“4.2.2 输出选项”的表 4.4。

符号名的变更

REName

Link/Library <Other>[User defined options :]

格　式

```
REName = <子选项>[,…]
<子选项> : { [<文件>](<名称> = <名称>[,…])
           | [<模块>](<名称> = <名称>[,…])}
```

说　明

更改外部符号名和段名。
也能更改包含在特定文件或者特定库内模块中的符号名和段名。
C/C++变量名时，在程序中的定义名前附加“_”。
不能保证更改函数名时的运行。
如果指定的名称存在于段和符号中，就优先符号名。
存在多个相同的文件名或者模块名时，优先先输入的指定。

例

```
rename=(_sym1=data)      ;将_sym1 更改为 data。
rename=lib1(P=P1)       ;将库模块 lib1 内的 P 段
                        ;更改为 P1 段。
```

备　考

本选项在指定 extract 或者 strip 时无效。

符号名的删除

DELete

Link/Library <Other>[User defined options :]

格　式

```
DELete = <子选项>[,…]
<子选项> : { [<文件>](<名称>[,…])
           | <模块> }
```

说　明

删除外部符号名或者库模块。
也能删除包含在特定文件中的符号名或者模块。
对于 C/C++变量名和 C 函数名，在程序中的定义名前附加“_”。在 C++函数时，将含有参数串的程序中的定义名用双引号括起来指定。但是，参数为 void 时，用“函数名()”指定。存在多个相同文件名时，优先先输入的指定。
如果用本选项指定符号名删除，就不删除目标而将属性更改为内部符号。

例

```
delete=(_sym1)           ;删除全部文件中的符号名_sym1。
delete=file1.obj(_sym2)  ;删除 file1.obj 中的符号名_sym2。
```

备　考

本选项在指定 extract 或者 strip 时无效。

模块的置换

REPlace

Link/Library <Other>[User defined options :]

格　式　`REPlace = <子选项>[,…]`

`<子选项> : <文件名>[(<模块名>[,…])]`

说　明　置换库模块。

将指定的文件或者库模块和由 `library` 选项指定的库内的同名模块进行置换。

例

```
replace=file1.obj        ; 将模块 file1.obj 和模块 file1 进行置换。
replace=lib1.lib(mdl1)   ; 将库文件 lib1.lib 内的模块 mdl1 和模块
                         ; mdl1 进行置换。
```

备　考　本选项在指定 `form={object | relocate | absolute | hexadecimal | stype | binary}`和 `extract`、`strip` 时无效。

模块的抽出

EXTract

Link/Library <Other>[User defined options :]

格　式　`EXTract = <模块名>[,…]`

说　明　抽出库模块。

从由 `library` 选项指定的库文件中抽出指定的库模块。

例

```
extract=file1              ;抽出模块 file1。
```

备　考　本选项在指定 `form={absolute | hexadecimal | stype | binary}`和 `strip` 时无效。

调试信息的删除

STRip

Link/Library <Other>[User defined options :]

格　式　　STRip

说　明　　删除绝对文件和库文件的调试信息。
在指定 strip 选项时，输入文件和输出文件一一对应。

例　　input=file1.abs file2.abs file3.abs
strip
删除 file1.abs、file2.abs、file3.abs 的调试信息，并且分别输出到 file1.abs、file2.abs、file3.abs。删除调试信息前的文件被备份到 file1.abk、file2.abk、file3.abk。

备　考　　本选项在指定 form={object | relocate | hexadecimal | stype | binary}时无效。

信息级

CHange_message

Link/Library <Other>[User defined options :]

格　式　　CHange_message = <子选项>[,…]
<子选项> : <错误级>[=<错误号>[-<错误号>][,…]]
<错误级> : {Information | Warning | Error}

说　明　　更改信息、警告和错误级的信息级。
能更改信息输出时的继续/中断执行。

例　　change_message=warning=2310
将 L2310 更改为警告级，并且在输出 L2310 时继续处理。

change_message=error
将全部的信息、警告消息更改为错误级。
只要输出一个信息，就中断处理。

局部符号名的隐含指定

Hide

`Link/Library <Other>[User defined options :]`

格　式　　Hide

说　明　　如果指定本选项，就删除输出文件中的局部符号名信息。

因为有关局部符号的名称信息被删除，所以即使通过二进制编辑程序等打开文件，也不能确认局部符号名。完全不影响被生成的文件的运行。

要隐含处理局部符号名时，请指定本选项。

隐含对象的符号种类如下：

- C 源程序：指定 static 限定词的变量名、函数名等
- C 源程序：goto 语句的标号名
- 汇编源程序：没有声明外部定义（参照）符号的符号名

例　　C 源程序中本选项功能有效的记述例子如下所示：

```
int g1;
int g2=1;
const int g3=3;
static int s1;          //<--- static 变量名为隐含对象
static int s2=1;        //<--- static 变量名为隐含对象
static const int s3=2;  //<--- static 变量名为隐含对象

static int sub1()       //<--- static 函数名为隐含对象
{
    static int s1;      //<--- static 变量名为隐含对象
    int l1;

    s1 = l1; l1 = s1;
    return(l1);
}

int main()
{
    sub1();
    if (g1==1)
        goto L1;
    g2=2;
L1:                     //<--- goto 语句的标号名为隐含对象
    Return(0);
}
```

备　考　　本选项只在输出文件格式为 absolute、relocate、library 时有效。

如果输入在编译/汇编时指定 goptimize 选项的文件或者输出文件格式为 relocate 或 library，就不能指定本选项。

在进行 map 优化的情况下指定本选项时，请不要在第一次连接时指定，而只能在第二次连接时指定本选项。

即使指定本选项，也不删除调试信息内的符号名。

4.2.8　　子命令文件选项

表 4.14　子命令文件种类选项一览表

	项目	子命令行格式	对话菜单	指定内容
1	子命令文件	SUbcommand = <文件名>	Link/Library <Subcommand file> [Use external subcommand file]	通过子命令文件指定选项

子命令文件

SUbcommand

Link/Library <Subcommand file> [Use external subcommand file]

格　式　　`SUbcommand = <文件名>`

说　明　　通过子命令文件指定选项。
子命令文件的格式如下：

<选项> {= | △} [<子选项> [,…]] [△&] [;<注解>]

也能用空格字符代替“=”，分开指定选项和子选项。
input 选项时，能指定空格字符分隔子选项。
在子命令文件中，1 行指定 1 个选项。
不能在 1 行中记述子选项时，能用&继续指定。
不能在子命令文件中指定 subcommand 选项。

例

```
指定命令行    ：optlnk file1.obj -sub=test.sub file4.obj
指定子命令    ：input file2.obj file3.obj ;这是注解。
                 library lib1.lib, &    ;指定继续行。
                 lib2.lib
```

在将子命令文件指定的选项内容展开到命令行的子命令指定位置后执行。
文件的输入顺序为 file1.obj、file2.obj、file3.obj、file4.obj。

4.2.9　　CPU 选项

表 4.15　CPU 选项卡的选项一览表

	项目	命令行格式	对话菜单	指定内容
1	SBR 地址的指定	SBr = { <SBR 地址> | User}	CPU [Specify SBR address :]	指定 8bit 绝对区域的起始地址

8bit 绝对区域地址值的指定

SBr

CPU [Specify SBR address :]

格　式　　SBr ＝ { <地址>｜User }

说　明　　指定 SBR 的地址值。
通过本选项指定地址值，能进行使用 abs8 区域的优化。如果本选项指定 user，就抑制 abs8 区域的优化。

备　考　　本选项只在 CPU 种类为 H8SX 时有效。
如果在源程序中或者在指定工具选项等时指定多个 SBR 地址，本选项就与指定无关，作为 user 指定进行处理。

4.2.10　剩余选项

表 4.16　剩余选项一览表

项目		命令行格式	对话菜单	指定内容
1	版权	LOgo NOLOgo	-	输出 不输出
2	继续指定	END	-	执行已输入的选项列，在结束处理后输入以后的选项列并继续处理
3	结束指定	EXIt	-	指定选项输入的结束
4	未参照定义符号的通知	MSg_unused		通过信息输出通知 1 次也没被参照的定义符号

版权

LOgo
NOLOgo

None (nologo is always available.)

格　式　　LOgo
　　　　　NOLOgo

说　明　　指定是否进行版权输出。
在指定 logo 选项时，输出版权显示。
在指定 nologo 选项时，抑制输出版权显示。
在省略本选项时解释为 logo。

继续处理

END

None

格　式　　END

说　明　　执行在 END 前指定的选项列。在连接处理结束后，输入在 END 后指定的选项列，继续连接处理。不能在命令行指定本选项。

例

```
input=a.obj,b.obj              ; 处理(1)
start=P,C,D/100,B/8000         ; 处理(2)
output=a.abs                   ; 处理(3)
end
input=a.abs                    ; 处理(4)
form=stype                     ; 处理(5)
output=a.mot                   ; 处理(6)
```

进行(1)～(3)的处理，输出 a.abs。
然后，进行(4)～(6)的处理，输出 a.mot。

结束处理

EXIt

None

格　式　　EXIt

说　明　　指定选项指定的结束。

不能在命令行指定本选项。

例

```
命令行指定： optlnk -sub=test.sub -nodebug
test.sub:  input=a.obj,b.obj         ; 处理(1)
           start=P,C,D/100,B/8000    ; 处理(2)
           output=a.abs              ; 处理(3)
           exit
```

进行(1)～(3)的处理，输出 a.abs。
Exit 执行后的命令行指定的 nodebug 选项无效。

第 5 章　标准库生成工具的操作方法

5.1　选项指定规则

5.1.1　命令行的格式

命令行的格式如下：

```
lbg38 [△<选项列> ...]
  <选项列>：- <选项>[=<子选项>[,...]]
```

5.2　选项解说

标准库生成工具的选项、子选项遵循 C/C++编译程序选项。与 C/C++编译程序选项的不同点如下所示。有关 C/C++编译程序选项的详细内容，请参照“**第 2 章　C/C++编译程序的操作方法**”。

命令行格式的英文大写字母表示缩写指定时的字母；下划线表示省略时的解释。

另外，用选项卡名<种类名>[项目]...表示对应综合开发环境的对话菜单。

5.2.1　追加选项

表 5.1　追加选项一览表

项目	选项	对话菜单	指定内容
1　对象头文件	Head = <sub>[,...]	Standard Library <Standard Library> [Category :]	指定生成对象的种类
	<sub>:{ ALL		全部库函数
	| RUNTIME		运行时例程
	| CTYPE		ctype.h＋运行时例程
	| MATH		math.h＋运行时例程
	| MATHF		mathf.h＋运行时例程
	| STDARG		stdarg.h＋运行时例程
	| STDIO		stdio.h＋运行时例程
	| STDLIB		stdlib.h＋运行时例程
	| STRING		string.h＋运行时例程
	| IOS		ios＋运行时例程
	| NEW		new＋运行时例程
	| COMPLEX		complex＋运行时例程
	| CPPSTRING }		string＋运行时例程
2　输出文件	OUTPut = <文件名>	Standard Library <Object> [Output file path :]	指定输出库文件名
3　可重入库	REent	Standard Library <Object> [Generate reentrant library]	生成可重入库

头文件

Head

Standard Library <Standard Library>[Category :]

格　式　　Head ＝ <sub>[,...]

```
    <sub>:{ ALL          |
            RUNTIME      |
            CTYPE        |
            MATH         |
            MATHF        |
            STDARG       |
            STDIO        |
            STDLIB       |
            STRING       |
            IOS          |
            NEW          |
            COMPLEX      |
            CPPSTRING    }
```

说　明　　用头文件名指定生成对象的种类。

有关各头文件和库函数的关系，请参照“10.3 C/C++库”。

运行时例程(runtime)总是为生成对象文件。

在省略本选项时解释为 head=all。

例　　`lbg38 -output=h8s.lib -head=mathf -cpu=2600a`

通过-cpu=2600a 编译 mathf.h 中定义的库函数和运行时例程，输出库文件 h8s.lib。

输出文件

OUTPut

Standard Library <Object>[Output file path :]

格　式　　OUTPut ＝ <文件名>

说　明　　指定输出文件名。

在省略本选项时解释为 output=stdlib.lib。

例　　`lbg38 -output=h8s.lib -optimize -speed -goptimize -cpu=2600a`

通过-optimize -speed -goptimize -cpu=2600a 编译用于全部标准库的源文件，输出库文件 h8s.lib。

可重入库

REent

`Standard Library <Object> [Generate reentrant library]`

格　式　　`REent`

说　明　　生成可重入库。但是，`rand`、`srand` 函数不可重入。

另外，不保证对 `strtok` 函数的相同字符串的连续调用。

例　　（用户程序）

```
#define  _REENTRANT
#include <stdlib.h>
```

备　考　　连接可重入库时，必须在程序内包含标准 `include` 文件前通过`#define` 语句定义（`#define _REENTRANT`）称为_REENTRANT 的宏名，或者必须在编译时通过 `define` 选项定义_REENTRANT。

5.2.2　　不可指定的选项

在 C/C++编译程序选项中不能用标准库生成工具指定的选项如表 5.2 所示。如果指定，就被忽略。

表 5.2　不可指定的选项一览表

	项目	选项	编译程序解释	内容
1	include 文件目录	Include	无	—
2	宏名的定义	DEFine	无	—
3	展开预处理器时的#line 输出抑制	NOLINe	无	—
4	信息级消息	Message NOMessage	NOMessage	不输出
5	预处理器的展开	PREProcessor	无	—
6	目标格式	Code	Code = Machinecode	输出机器语言程序
7	调试信息	DEBug NODEBug	NODEBug	不输出
8	目标文件的输出指定	OBject NOOBject	OBject	输出
9	模板示例的生成功能	Template	无	不使用模板功能
10	列表文件	List NOList	NOList	不输出
11	列表内容和格式	SHow	无	—
12	注释的嵌套	COMment	无	不使用注释的嵌套
13	MAC 寄存器的保证	MAcsave	无	不包含中断函数
14	信息级	CHAnge_message	无	—
15	C/C++语言的选择	LANg	无	遵循文件扩展名
16	版权的输出抑制	LOGo NOLOGo	NOLOGo	抑制版权的输出
17	字符串内的字符码	EUc SJis LATin1	无	不使用字符码
18	目标码内的汉字转换	OUtcode	无	不使用字符码

5.2.3　　指定选项时的注意事项

在指定选项时，必须遵循以下规则：

(1) cpu、regparam、structreg/nostructreg、longreg/nolongreg、stack、double=float、byteenum、pack、rtti=on/off、exception/noexception、bit_order= left/right、indirect=normal/extended、ptr16、sbr 选项必须指定与编译时相同的选项。

(2) 在使用#pragma global_register 时，必须通过 preinclude 选项指定（include）含#pragma global_register 声明的头文件。在综合开发环境下，必须通过标准库 <其它>[用户指定选项：]指定。

第 6 章　栈解析工具的操作方法

栈解析工具读取由优化连接编辑程序输出的栈信息文件（*.sni）或者由仿真调试程序输出的配置信息文件（*.pro），显示堆栈使用量。

对于无法输出到栈信息文件的汇编程序（由汇编程序进行汇编）的堆栈使用量，可使用编辑功能追加/修正信息。另外，汇编程序Ver.6.01能对符号指定堆栈值，并且能计算出整个系统的堆栈使用量。

作为调用信息文件（*.cal），可保存/读取有关已编辑的堆栈使用量的信息。

6.1　栈解析工具的启动

在启动栈解析工具时，必须从 Windows®的开始菜单选择“运行（R）…”，指定并执行 Call.exe。

另外，在使用综合开发环境时，必须先通过 Windows®的开始菜单选择“程序”，然后选择注册在 HEW 的“Call Walker”。也能在启动综合开发环境后从 Tools 菜单启动栈解析工具。

有关操作方法，请参照栈解析工具的帮助。

第 7 章　　环境变量

7.1　　环境变量一览表

环境变量一览表如表 7.1 所示。

表 7.1　环境变量

<table>
<tr><th></th><th>环境变量</th><th>说明</th></tr>
<tr><td>1</td><td>path</td><td>指定执行文件的保存目录。
指定格式：
PC 版　C> path= <执行文件路径名>[;<已存在的路径名>;...]
UNIX 版 C shell　%set path =(<执行文件路径名> $path)
Bourne shell　%PATH=:<执行文件路径名>[:<已存在的路径名>:...]
%export PATH</td></tr>
<tr><td>2</td><td>H38CPU</td><td>通过环境变量指定由编译程序/汇编程序的 cpu 选项指定的 CPU 种类。
<table>
<tr><th><CPU/运行模式></th><th>地址空间的位宽<*1></th><th>乘除法器指定<*2></th></tr>
<tr><td>AE5</td><td>—</td><td>—</td></tr>
<tr><td>H8SXN</td><td>—</td><td>m | d | md</td></tr>
<tr><td>H8SXM</td><td>20 | 24 (24)</td><td>m | d | md</td></tr>
<tr><td>H8SXA</td><td>20 | 24 | 28 | 32 (24)</td><td>m | d | md</td></tr>
<tr><td>H8SXX</td><td>28 | 32 (32)</td><td>m | d | md</td></tr>
<tr><td>2600n</td><td>—</td><td>—</td></tr>
<tr><td>2600a</td><td>20 | 24 | 28 | 32 (24)</td><td>—</td></tr>
<tr><td>2000n</td><td>—</td><td>—</td></tr>
<tr><td>2000a</td><td>20 | 24 | 28 | 32 (24)</td><td>—</td></tr>
<tr><td>300hn</td><td>—</td><td>—</td></tr>
<tr><td>300ha</td><td>20 | 24 (24)</td><td>—</td></tr>
<tr><td>300</td><td>—</td><td>—</td></tr>
<tr><td>300l</td><td>—</td><td>—</td></tr>
</table>
（ ( )内是省略指定时设定的默认值）
在 H38CPU 环境变量的 CPU 指定和 cpu 选项的 CPU 指定不同时，输出警告信息，优先 cpu 选项的指定。
指定格式：
PC 版　C> set H38CPU= <CPU/运行模式>[:<*1>][:<*2>]
UNIX 版 C shell　% setenv H38CPU <CPU/运行模式>[:<*1>][:<*2>]
Bourne shell　% H38CPU=<CPU/运行模式>[:<*1>][:<*2>]
% export H38CPU</td></tr>
<tr><td>3</td><td>CH38 *3</td><td>指定编译程序的 include 文件的保存目录。
系统 include 文件的检索顺序为 include 选项指定目录、CH38 指定目录。
用户 include 文件的检索顺序为当前目录、include 选项指定目录、CH38 指定目录。
如果没有指定环境变量 CH38，UNIX 版时假定为/usr/CH38，而 PC 版在省略时没有默认值。
指定格式：
PC 版　C> set CH38= <include 路径名> [;<include 路径名> ;...]
UNIX 版 C shell　%setenv CH38 <include 路径名>[:<include 路径名> :...]
Bourne shell　% CH38 = <include 路径名> [: <include 路径名>:...]
% export CH38</td></tr>
</table>

	环境变量	说明
4	CH38TMP	指定编译程序建立临时文件的目录。如果没有指定此环境变量，就在当前目录下建立临时文件。 指定格式： PC 版　C> set CH38TMP= <临时文件路径名> UNIX 版 C shell　% setenv CH38TMP <临时文件路径名> Bourne shell　% CH38TMP = <临时文件路径名> % export CH38TMP
5	CH38SBR	通过环境变量指定编译程序中的短地址基址寄存器（SBR）。设定方法以编译程序的 sbr 选项为准。 指定格式： PC 版　C> set CH38SBR = <地址> UNIX 版 C shell　% setenv CH38SBR <地址> Bourne shell　% CH38SBR = <地址> % export CH38SBR
6	HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	指定优化连接编辑程序使用的默认库名。 优先连接 library 选项指定的库。此后，如果有未解决的符号，就按照 1、2、3 的顺序检索默认库。 指定格式： PC 版　C> set HLNK_LIBRARY1= <库名 1> C> set HLNK_LIBRARY2= <库名 2> C> set HLNK_LIBRARY3= <库名 3> UNIX 版 C shell　% setenv HLNK_LIBRARY1 <库名 1> % setenv HLNK_LIBRARY2 <库名 2> % setenv HLNK_LIBRARY3 <库名 3> Bourne shell　% HLNK_LIBRARY1=<库名 1> % export HLNK_LIBRARY1 % HLNK_LIBRARY2=<库名 2> % export HLNK_LIBRARY2 % HLNK_LIBRARY3=<库名 3> % export HLNK_LIBRARY3
7	HLNK_TMP	指定优化连接编辑程序建立临时文件的目录。如果没有指定此环境变量，就在当前目录下建立临时文件。 指定格式： PC 版　C> set HLNK_TMP= <临时文件路径名> UNIX 版 C shell　% setenv HLNK_TMP <临时文件路径名> Bourne shell　% HLNK_TMP = <临时文件路径名> % export HLNK_TMP
8	HLNK_DIR *[3]	指定优化连接编辑程序的输入文件保存目录。 由 input 选项、library 选项指定的文件检索顺序为当前目录、HLNK_DIR 指定目录。 但是，对于用通配符指定的文件，只在当前目录中检索。 指定格式： PC 版　C> set HLNK_DIR= <输入文件路径名>[;<输入文件路径名> ;...] UNIX 版 C shell % setenv HLNK_DIR <输入文件路径名>[:<输入文件路径名> :...] Bourne shell % HLNK_DIR = <输入文件路径名> [:<输入文件路径名> :...] % export HLNK_DIR

【注】*3　指定多个目录时，PC 版必须用“; ”（分号）分开；
UNIX 版必须用“: ”（冒号）分开。

7.2　编译程序的隐式声明

编译程序按照选项的指定和版本进行如下的隐式#define 声明。

表 7.2　隐式声明

	选项	隐式声明
1	cpu = 300L	#define _ _300L_ _
	cpu = 300	#define _ _300_ _
	cpu = 300HN	#define _ _300HN_ _
	cpu = 300HA	#define _ _300HA_ _
	cpu = 2000N	#define _ _2000N_ _
	cpu = 2000A	#define _ _2000A_ _
	cpu = 2600N	#define _ _2600N_ _
	cpu = 2600A	#define _ _2600A_ _
	cpu = H8SXN	#define _ _H8SXN_ _
	cpu = H8SXM	#define _ _H8SXM_ _
	cpu = H8SXA	#define _ _H8SXA_ _
	cpu = H8SXX	#define _ _H8SXX_ _
	cpu = <H8SX>:M 或者 MD	#define _ _HAS_MULTIPLIER_ _
	cpu = <H8SX>:D 或者 MD	#define _ _HAS_DIVIDER_ _
	cpu = AE5	#define _ _AE5_ _
2	double=float	#define _ _FLT_ _
3	byteenum	#define _ _BENM_ _
4	cpuexpand	#define _ _CPUEX_ _
5	library=intrinsic	#define _ _INTRINSIC_LIB_ _
6	abs16	#define _ _ABS16_ _
7	—	#define _ _ADDRESS_SPACE_ _ [*1] [*4]
8	—	#define _ _DATA_ADDRESS_SIZE_ _ [*2] [*4]
9	—	#define _ _H8_ _ [*4]
10	—	#define _ _RENESAS_VERSION_ _ [*3] [*4]
11	—	#define _ _HITACHI_VERSION_ _ [*3] [*4]
12	—	#define _ _RENESAS_ _ [*4]
13	—	#define _ _HITACHI_ _ [*4]

【注】*1　定义地址空间长度（16|20|24|28|32 bit）。

*2　如下所示，将定义_ _DATA_ADDRESS_SIZE_ _的值定义为 2 或者 4：

2：（300|正常模式|中级模式）或（高级模式|最高模式）以及有 ptr16 选项指定

4：（高级模式|最高模式）以及无 ptr16 选项指定

*3　_ _RENESAS_VERSION_ _和_ _HITACHI_VERSION_ _的值参照以下：

- C 源程序中：_ _RENESAS_VERSION_ _==0xaabb

  aa: version 部分

  bb: revision 部分

- 编译程序中的定义例子

```
#define _ _RENESAS_VERSION_ _ 0x0301      // Version 3.1C 时
#define _ _RENESAS_VERSION_ _ 0x0400      // Version 4.0 时
```

*4　总是被定义。

第 8 章　　文件规格

8.1　　文件名的命名方法

如果在指定文件名时省略扩展名，就使用附加了标准文件扩展名的文件名。本开发环境中使用的标准文件扩展名如表 8.1 所示。

表 8.1　本开发环境中使用的标准文件扩展名

No.	扩展名	含义
1	c	C 源程序文件
2	cpp, cc, cp	C++源程序文件
3	h	Include 文件
4	lis, lst *1	用于 C 程序的列表文件
5	lis, lpp *1	用于 C++程序的列表文件
6	p	用于 C 程序的预处理器展开文件
7	pp	用于 C++程序的预处理器展开文件
8	src,mar	汇编源程序文件
9	exp	用于汇编程序的预处理器展开文件
10	lis	用于汇编程序的列表文件
11	obj	可再定位目标程序文件
12	rel	可再定位装入模块文件
13	abs	绝对装入模块文件
14	map	连接列表文件
15	lib	库文件
16	lbp	库列表文件
17	mot	S 型格式
18	hex	HEX 格式
19	bin	二进制文件
20	fsy	优化连接编辑程序的输出符号地址文件
21	sni	栈信息文件
22	pro	配置信息文件
23	dbg	DWARF2 格式调试信息文件
24	rti	含在扩展名为 td 的文件中定义指定的目标
25	cal	调用信息文件

【注】 *1　UNIX 版为 lis；PC 版为 lst 或者 lpp。

因为以rti_开头的文件名为系统保留名，所以不能使用。
输出到工程生成的tpldir文件夹下的文件扩展名如表8.2所示。

表 8.2　tpldir 文件夹输出文件

No.	扩展名	含义
1	td	tentative 定义的变量信息文件
2	ti	模板信息文件
3	pi	参数信息文件
4	ii	示例信息文件

文件名的命名方法的一般规则以各主机为准，请参照所使用的主机手册。

8.2　编译列表的参照方法

本节说明编译列表的内容和格式。

8.2.1　编译列表的结构

编译列表的结构和内容如表 8.3 所示。

表 8.3　编译列表的结构和内容

No.	建立的列表	内容	选项指定方法	选项省略时
1	源列表信息	源程序清单*[1]	show＝source show＝nosource	输出
		include 文件、宏展开后的源程序清单*[2]	show＝expansion show＝noexpansion	不输出
2	错误信息	编译时的错误信息	—	输出
3	符号分配信息	函数堆栈帧中的变量分配信息	show＝allocation show＝noallocation	不输出
4	目标信息	目标程序的机器语言和汇编码	show＝object show＝noobject	不输出
5	统计信息	各段的字节数、符号数信息和目标种类	show＝statistics show＝nostatistics	输出

【注】 *1　如果同时指定 noexpansion 和 object 子选项，就将源程序清单输出到目标信息中。

*2　include 文件和宏展开后的源程序清单在指定 show＝source 时有效。

8.2.2　源列表信息

源列表信息的输出格式有输出预处理前的程序格式（在指定 show=noexpansion 时）和输出预处理后的源程序格式（在指定 show=expansion 时）。输出格式分别如图 8.1 (a)、(b)所示，图 8.1 (b)中的阴影部分表示不同点。

（a）show=noexpansion的源列表信息

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1 [1] #include "header.h"
         2
         3     int sum2(void)
         4     {  int j;
         5
         6     #ifdef SMALL
         7        j=SML_INT;
         8     #else
         9        j=LRG_INT;
        10     #endif
        11
        12        return j;  /*
continue 1234567890123456789012345678901234567890123456789012345678901234567890
23456789012345678901234567890 */
        13     }
        [2]
```

（b）show=expansion的源列表信息

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1 [1] #include "header.h"
FILE NAME: header.h
         1     #define SML_INT          1
         2     #define LRG_INT        100
FILE NAME: m0260.c
         2
         3     int sum2(void)
         4     {  int j;
         5
         6     #ifdef SMALL
         7 X      j=SML_INT;
         8[3]  #else
         9  E     j=100;
        10 [4] #endif
        11
        12        return j;  /* continue123456789012345678901234567890123456789
23456789012345678901234567890 */
        13     }
        [2]
```

【注】

［1］　源程序文件名或者Include文件名

［2］　源程序或者Include文件中的行号

［3］　在指定show=expansion时，不为编译对象的#ifdef语句、#elif语句等条件编译语句的源行

［4］　在指定show=expansion时，由#define语句进行宏替换的源行

图 8.1　源列表信息的输出格式

8.2.3　　错误信息

错误信息的输出例子如图 8.2 所示。

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6----
FILE NAME: m0260.c
         1     #include "header.h"
         2
         3     extern int sum3(int);
         4
         5     sum3(int x)
         6     {
         7        int i;
         8        int j;
         9
        10        j=0;
        11        for (i=0; i<=x; i++){
        12           j+=k;                        ←———— 发生错误的行
        13        }
        14
        15        return j;
        16     }


*********** ERROR INFORMATION **********


m0260.c(12) : C2225 (E) Undeclared name "k"
    [1]   [2]     [3]  [4]          [5]

NUMBER OF ERRORS:          1 }[6]
NUMBER OF WARNINGS:        0
NUMBER OF INFORMATIONS:    0  [7]
```

【注】

［1］发生错误的源程序名，显示前10个字符
［2］发生错误的源程序的行号
［3］用于识别错误信息的序号
［4］（I） 信息级
　　（W） 警告级
　　（E） 错误级
　　（F） 致命级
［5］错误内容
［6］错误级信息、警告级信息的总数
［7］信息级消息的总数
　　（只在指定message选项时）

图 8.2　包含错误的源错误列表和错误信息

8.2.4　　符号分配信息

表示函数的参数和局部变量的分配信息。在 H8S/2600 的高级模式下编译时的符号分配信息例子如图 8.3 所示。

```
************ SOURCE LISTING ************

      Line Pi 0----+----1----+----2----+----3----+----4----+----5----+----6-
FILE NAME: m0280.c
         1     extern int h(char, char *, double );
         2
         3     int
         4     h(char a, register char *b, double c)
         5     {
         6        char    *d;
         7
         8        d= &a;
         9        h(*d,b,c);
        10        {
        11           register int i;
        12
        13           i= *d;
        14           return i;
        15        }
        16     }

******* STACK FRAME INFORMATION ********

FILE NAME: m0280.c
Function (File m0280.c   , Line     4): h
                   [1]
  Parameter Allocation
    a                                  0xfffffff7 saved from R0L
    b                             REG  ER5        saved from ER1  [2]
    c                                  0x00000008

  Level  1 (File m0280.c   , Line    5) Automatic/Register Variable Allocation
    d                                  0xfffffff2
                                                                              [3]
  Level  2 (File m0280.c   , Line   10) Automatic/Register Variable Allocation
    i                             REG  R4

Parameter Area Size     : 0x00000008 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000006 Byte(s)  [4]
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000008 Byte(s)
Total Frame Size        : 0x0000001e Byte(s)
```

【注】

[1]函数定义的文件名、行序号和函数名

[2]参数的分配位置	X saved from Y	在函数入口处将Y传递的参数复制到X时
	REG ERx	分配位置为寄存器时，表示REG
	0xffffffxx 0x000000xx	分配位置为堆栈时，表示从帧指针（ER6）的偏移量

[3]在复合语句中声明的局部变量的分配位置为堆栈时表示从ER6的偏移量，为寄存器时表示REG

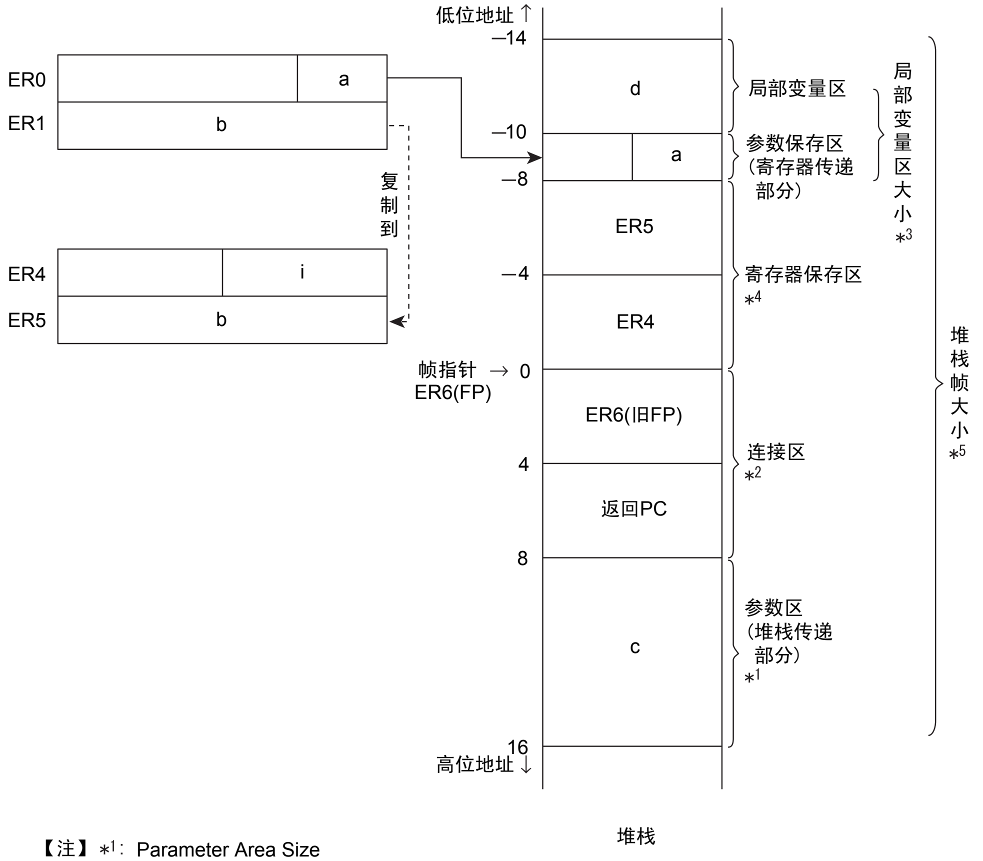
[4]在函数内使用的堆栈帧的分配信息

Parameter Area Size	：由堆栈传递的参数区和返回值设定地址区的大小
Linkage Area Size	：连接区（返回PC区＋帧指针保存区）的合计大小。但是，也有不存在帧指针保存区的情况。 在中断函数时加上CCR的保存部分，在H8SX、H8S/2600、H8S/2000时加上EXR的保存部分。
Local Variable Size	：在将函数内使用的局部变量区和由寄存器传递的参数分配到堆栈时， 为使用的参数保存区的合计大小
Temporary Size	：函数内C编译程序使用的临时区的大小
Register Save Area Size	：保存函数内使用的寄存器值的区域大小
Total Frame Size	：函数内分配的堆栈帧的合计大小

图 8.3　符号分配信息（cpu=2600a）

【注】　在指定优化选项 optimize＝1 或 H8SX 时，不输出参数分配信息和局部变量分配信息。此时显示以下的信息：
Optimize Option Specified : No Allocation Information Available

对应图 8.3 的符号分配信息的堆栈分配例子如图 8.4 所示。



【注】 *1： Parameter Area Size
*2： Linkage Area Size
*3： Local Variable Size
*4： Register Save Area Size
*5： Total Frame Size

图 8.4　堆栈分配的例子（cpu=2600a）

8.2.5　　目标信息

将源程序清单输出和不输出到目标信息时的列表例子如图 8.5、图 8.6 所示。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE               LABEL     INSTRUCTION OPERAND    COMMENT
[1]   [2]   [3]                                [4]
P                                                              ; section
         1:    extern int sum(int);
         2:         [5]
         3:    int
         4:    sum(int x)
  00000000                    _sum:                            ; function: sum
         5:    {
         6:       int i;
         7:       int j;
         8:
         9:       j=0;
        10:
        11:       for(i=0; i<=x; i++){
  00000000 1988                   SUB.W       E0,E0
  00000002 4000                   BRA         L8:8
  00000004                    L7:
  00000004 0B58                   INC.W       #1,E0
  00000006                    L8:
  00000006 1D08                   CMP.W       R0,E0
  00000008 4F00                   BLE         L7:8
        12:          j+=1;
        13:       }
        14:
        15:       return;
        16:    }
  0000000A 5470                   RTS
```

【注】

［1］　各段的段名（P、C、D、B）
［2］　来自各段前头的偏移量
［3］　各段偏移地址的内容
［4］　对应机器语言的汇编码
［5］　源程序中的行号和源列表

图 8.5　输出源程序清单时的目标信息
（show＝source, object　cpu＝2600a）

【注】在指定 show=expansion 选项时，总是为图 8.6 的目标信息。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE       C LABEL     INSTRUCTION OPERAND    COMMENT
[1]   [2]   [3]                          [4]
P                                                        ; section
                        ;*** File m0251.c   , Line 4     ; block
  00000000              _sum:                            ; function: sum
                        ;*** File m0251.c   , Line 5     ; block
                        ;*** File m0251.c   , Line 9     ; expression statement
  00000000 1911                   SUB.W       R1,R1
                        ;*** File m0251.c   , Line 10    ; expression statement
  00000002 1988                   SUB.W       E0,E0
                        ;*** File m0251.c   , Line 10    ; for
  00000004 4004                   BRA         L8:8
  00000006              L7:
                        ;*** File m0251.c   , Line 10    ; block
                        ;*** File m0251.c   , Line 11    ; expression statement
  00000006 0981                   ADD.W       E0,R1
                        ;*** File m0251.c   , Line 10    ; expression statement
  00000008 0B58                   INC.W       #1,E0
  0000000A              L8:
  0000000A 1D08                   CMP.W       R0,E0
  0000000C 4FF8                   BLE         L7:8
                        ;*** File m0251.c   , Line 13    ; return
  0000000E 0D10                   MOV.W       R1,R0
                        ;*** File m0251.c   , Line 14    ; block
  00000010 5470                   RTS
```

【注】

［1］　各段的段名（P、C、D、B）
［2］　来自各段前头的偏移量
［3］　各段偏移地址的内容
［4］　对应机器语言的汇编码

图 8.6　不输出源程序清单时的目标信息
（show＝nosource, object　cpu＝2600a）

8.2.6　　统计信息

统计信息的输出例子如图 8.7 所示。

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                               0x00000012 Byte(s)
CONSTANT SECTION(C):                               0x00000000 Byte(s)
DATA     SECTION(D):                               0x00000000 Byte(s)
BSS      SECTION(B):                               0x00000000 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000012 Byte(s)                                [1]
TOTAL CONSTANT SECTION: 0x00000000 Byte(s)
TOTAL DATA     SECTION: 0x00000000 Byte(s)
TOTAL BSS      SECTION: 0x00000000 Byte(s)

    TOTAL PROGRAM SIZE: 0x00000012 Byte(s)



** ASSEMBLER/LINKAGE EDITOR LIMITS INFORMATION **

NUMBER OF EXTERNAL REFERENCE  SYMBOLS:          0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:          1   [2]
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:            3


***** COMPILE CONDITION INFORMATION ****

COMMAND LINE: -sh=allocation -opt=0 test.c [3]
cpu         : 2600a                        [4]
```

【注】
［1］　各段的大小及其合计
［2］　目标程序的外部参照符号数、外部定义符号数、内部标号和外部标号的合计数
［3］　命令行指定的内容
［4］　CPU/运行模式

图 8.7　统计信息

【注】在指定选项 noobject 以及发生错误级、致命级的错误时，不输出统计信息。另外，在指定选项 code＝asmcode 时，必须注意：不输出统计信息的段容量信息（SECTION SIZE INFORMATION）。

8.3　汇编列表的参照方法

8.3.1　汇编列表的结构

汇编列表的结构和内容如表 8.4 所示。

表 8.4　汇编列表的结构和内容

No	建立的列表	内容	选项	选项省略时
1	源列表信息	显示有关源程序的信息。	source	输出
2	参照表信息	显示有关源程序的符号信息。	cross_reference	输出
3	段信息列表	显示有关源程序的段信息。	section	输出

【注】　全部的列表选项在指定 list 选项时有效。

8.3.2　源列表信息

输出源列表信息。源列表信息的输出例子如图 8.8 所示。

```
   1                            1              .CPU 2600A:32
   2                            2      ;
   3 00000000                   3              .SECTION AAA,CODE,ALIGN=2
   4 00000000                   4      START
   5 00000000 7A0700000000      5              MOV.L   #STACK:32,SP
   6 00000006 F800              6              MOV.B   #0:8,R0L
   7 00000008 6AA800000000      7              MOV.B   R0L,@ANS:32
   8 0000000E 7A0200001000      8              MOV.L   #DATA:32,ER2
   9                            9              .FOR.B (R1L=#1,#8,+#1)
  10 00000014 F901                 S                         MOV #1,R1L
  11 00000016 5800000A             S                         BRA _$F00002
  12 0000001A                      S                   _$F00000: .EQU $
  13 0000001A 6828             10                      MOV.B @ER2,R0L
  14 0000001C 0B02             11                      ADDS.L #1,ER2
  15 0000001E 5E000000         12                      JSR @CHANGE:24
  16                           13             .ENDF
  17 00000022                      S                   _$F00001: .EQU $
  18 00000022 8901                 S                            ADD #1,R1L
  19 00000024                      S                   _$F00002: .EQU $
  20 00000024 A908                 S                            CMP #8,R1L
  21 00000026 4FF2                 S                            BLE _$F00000
  22 00000028                      S                   _$F00003: .EQU $
  23 00000028 0180             14              SLEEP
  24 0000002A 40D4             15              BRA     START
  25                           16      ;
  26 0000002C                  17      CHANGE
  27 0000002C 6A2900000000     18              MOV.B   @ANS:32,R1L
  28                           19              .IF.B (R1L<LT>R0L)
  29 00000032 1C98                 S                         CMP R1L,R0L
  30 00000034 58F00006             S                         BLE _$I00000
  31 00000038 6AA800000000     20                      MOV.B R0L,@ANS:32
  32                           21              .ENDI
  33 0000003E                      S                   _$I00000: .EQU $
  34 0000003E                      S                   _$I00001: .EQU $
  35 0000003E 5470             22              RTS
  36                           23      ;
  37 00001000                  24              .SECTION BBB,DATA,LOCATE=H'00001000
  38 00001000                  25      DATA
  39 00001000 03020405         26              .DATA.B H'03,H'02,H'04,H'05
  40 00001004 01080607         27              .DATA.B H'01,H'08,H'06,H'07
  41                           28      ;
  42 00000000                  29              .SECTION CCC,DATA,ALIGN=2
  43 00000000                  30      ANS
  44 00000000 00000001         31              .RES.B 1
  45                           32      ;
  46 00000000                  33              .SECTION DDD,STACK,ALIGN=2
  47 00000000 00000500         34              .RES.B H'500
  48 00000500                  35      STACK
  49                           36      ;
  50 00000000                  37              .END    START
 ___ ________  _______________ ___ __ _______________________________________________
 (1)    (2)          (3)       (4)(5)               (6)
*****TOTAL ERRORS      0
*****TOTAL WARNINGS    0
```

图 8.8　源列表

源列表中的（1）～（6）的内容如下：

（1）列表行号

（2）定位计数器值

在绝对地址段时显示绝对地址，在相对地址段时显示相对地址。

（3）目标码

（4）源行号

是源文件中的源语句的行号。对于汇编程序展开的源语句，不显示行号。

（5）展开分类

是预处理器功能的源语句的分类。

展开分类有以下几种：

- I　……　include 文件
- C　……　条件汇编的满足、重复展开和带条件的重复展开
- M　……　宏展开
- S　……　结构化汇编展开

展开分类 I 还显示 include 的嵌套级。

（6）源语句

8.3.3　参照表

输出参照表信息。参照表信息的输出例子如图 8.9 所示。

```
*** CROSS REFERENCE LIST
NAME                                  SECTION        ATTR   VALUE            SEQUENCE
AAA                                   AAA            SCT    00000000       3*
ANS                                   CCC                   00000000       7       27     31
                                                                           43*
BBB                                   BBB            SCT    00001000       37*
CCC                                   CCC            SCT    00000000       42*
CHANGE                                AAA                   0000002C       15      26*
DATA                                  BBB                   00001000        8      38*
DDD                                   DDD            SCT    00000000       46*
STACK                                 DDD                   00000500        5      48*
START                                 AAA                   00000000        4*     24     50
_$F00000                              AAA            EQU    0000001A       12*     21
_$F00001                              AAA            EQU    00000022       17*
_$F00002                              AAA            EQU    00000024       11      19*
_$F00003                              AAA            EQU    00000028       22*
_$I00000                              AAA            EQU    0000003E       30      33*
_$I00001                              AAA            EQU    0000003E       34*
------------------------------------- ------------- ------  ---------- -----------------------
                 (1)                       (2)        (3)       (4)              (5)
```

图 8.9　参照表

参照表中的（1）～（5）的内容如下：

（1）符号名

（2）段名

是包含符号的段名。最多显示前 8 个字符。

（3）符号属性

是符号的属性。符号属性有以下几种：

不显示	……	标号定义
EQU	……	.EQU 定义
ASGN	……	.ASSIGN 定义
IMPT	……	外部参照
EXPT	……	外部定义
SCT	……	段名
REG	……	.REG 定义
MDEF	……	双重定义
UDEF	……	未定义

（4）符号值

是符号的值。用 8 位的 16 进制数显示。

（5）符号定义或参照的列表行号

是符号定义或参照行的列表行号。在定义行中显示星号（*）。

8.3.4　段信息列表

输出段信息。段信息的输出例子如图 8.10 所示。

```
*** SECTION DATA LIST

SECTION                               ATTRIBUTE      SIZE     START
AAA                                   REL-CODE      0000040
BBB                                   ABS-DATA      0000008   001000
CCC                                   REL-DATA      0000001
DDD                                   REL-STACK     0000500
___________________________________   __________    _______   ______
                 (1)                      (2)         (3)       (4)
```

图 8.10　段信息

段信息列表中的（1）～（4）的内容如下：

（1）段名

（2）段属性

是段的属性，显示格式种类和段属性。

（a）格式种类

ABS　……　绝对地址格式

REL　……　相对地址格式

（b）段属性

CODE　……　代码段

DATA　……　数据段

STACK　……　堆栈段

DUMMY　……　虚设段

（3）段大小

是段的大小，用 16 进制数显示。

（4）段起始地址

是绝对地址段的起始地址，相对地址段不显示。

8.4　　连接列表的参照方法

说明优化连接编辑程序输出的连接列表的内容和格式。

8.4.1　　连接列表的结构

连接列表的结构和内容如**表 8.5** 所示。

表 8.5　连接列表的结构和内容

No	建立的列表	内容	子选项	show 选项省略时*1
1	选项信息	显示由命令行或者子命令指定的选项列	—	输出
2	错误信息	显示错误信息	—	输出
3	连接映像信息	显示段名、起始/结束地址、大小、种类	—	输出
4	符号信息	按地址顺序显示静态定义符号名、地址、大小、种类 在指定 show=reference 选项时，还显示各符号的参照次数和是否进行优化	show=symbol show=reference	不输出 不输出
5	符号删除优化信息	显示通过优化删除的符号	show=symbol	不输出
6	变量存取优化对象符号信息	显示在 8bit/16bit 绝对寻址方式的参照次数	show=reference	不输出
7	函数存取优化对象符号信息	显示符号的参照次数	show=reference	不输出
8	参照信息	显示符号的参照信息	show=xreference	不输出

【注】 *1　show选项只在指定list选项时有效。

8.4.2　　选项信息

输出由命令行或者子命令文件指定的选项列。选项信息的输出例子如**图 8.11** 所示（在指定 optlnk -sub=test.sub -list –show 时）。

```
(test.sub的内容)
INPUT test.obj
```

```
*** Options ***

-sub=test.sub         }
INPUT test.obj  (2)   }  (1)
-list                 }
-show                 }
```

图 8.11　选项信息的输出例子（连接列表）

（1）按指定的顺序输出由命令行或者子命令指定的选项列。

（2）是子命令文件 test.sub 中的子命令。

8.4.3　　错误信息

输出错误信息。错误信息的输出例子如图 8.12 所示。

```
*** Error information ***

** L2310 (E) Undefined external symbol "strcmp" referred to in "test.obj"   } (1)
```

图 8.12　错误信息的输出例子（连接列表）

（1）输出错误信息。

8.4.4　　连接映像信息

按照地址顺序输出各段的起始/结束地址、大小和种类。连接映像信息的输出例子如图 8.13 所示。

```
*** Mapping List ***

SECTION                                 START      END          SIZE   ALIGN
  (1)                                    (2)       (3)           (4)    (5)
P
                                      00000000  000004d6         4d6    2
C
                                      000004d6  00000533          5d    2
D
                                      00000534  0000053c           8    2
B
                                      0000053c  00004112        3bd6    2
```

图 8.13　连接映像信息的输出例子（连接列表）

（1）显示段名。
（2）显示起始地址。
（3）显示结束地址。
（4）显示段大小。
（5）显示段的边界调整数。

8.4.5　　符号信息

在指定 show=symbol 选项时，按照地址顺序输出外部定义符号或者静态内部定义符号的地址、大小和种类。另外，在指定 show=reference 选项时，还输出各符号的参照次数和有无进行优化。符号信息的输出例子如图 8.14 所示。

```
*** Symbol List ***

SECTION=(1)
FILE=(2)                            START        END    SIZE
                                     (3)         (4)     (5)
  SYMBOL                            ADDR        SIZE    INFO      COUNTS  OPT
    (6)                              (7)         (8)     (9)       (10)   (11)

SECTION=P
FILE=test.obj
                                  00000000  00000428       428
  _main
                                  00000000         2   func ,g         0
  _malloc
                                  00000000        32   func ,l         0
FILE=mvn3
                                  00000428  00000490        68
  $MVN#3
                                  00000428         0   none ,g         0
```

图 8.14　符号信息的输出例子（连接列表）

（1）显示段名。
（2）显示文件名。
（3）显示包含在（2）的文件中的相应段的起始地址。
（4）显示包含在（2）的文件中的相应段的结束地址。
（5）显示包含在（2）的文件中的相应段的段大小。
（6）显示符号名。
（7）显示符号地址。
（8）显示符号长度。
（9）符号种类如下所示：

数据种类：	func	……	函数名
	data	……	变量名
	entry	……	入口函数名
	none	……	未设定（标号、汇编程序符号）
声明种类：	g	……	外部定义
	l	……	内部定义

（10）显示符号的参照次数。只在指定 show=reference 选项时显示。在不显示参照次数时显示“*”。
（11）有无优化如下所示：

ch	……	通过优化被更改的符号
cr	……	通过优化被生成的符号
mv	……	通过优化被移动的符号

8.4.6　　符号删除优化信息

输出通过符号删除优化（optimize=symbol_delete）被删除的符号的长度和种类。符号删除优化信息的输出例子如图 8.15 所示。

```
*** Delete Symbols ***

SYMBOL                                SIZE    INFO
  (1)                                  (2)     (3)
_Version
                                        4   data ,g
```

图 8.15　符号删除信息的输出例子（连接列表）

（1）显示删除的符号名。
（2）显示删除的符号长度。
（3）删除的符号种类如下所示：

数据种类：	func	……	函数名
	data	……	变量名
声明种类：	g	……	外部定义
	l	……	内部定义

8.4.7　　变量存取优化对象符号信息

在指定 show=reference 时，输出变量存取优化（optimize=variable_access）对象的符号长度、参照次数和有无进行优化。

将可通过 8 位绝对寻址方式或者 16 位绝对寻址方式存取的符号输出到"Variable Accessible with Abs8"、可通过 16 位绝对寻址方式存取的符号输出到"Variable Accessible with Abs16"。变量存取优化对象符号信息的输出例子如图 8.16 所示。

```
*** Variable Accessible with Abs8 ***

SYMBOL                                SIZE    COUNTS  OPTIMIZE
  (1)                                  (2)      (3)      (4)
_Char1Glob
                                        1         2   done

*** Variable Accessible with Abs16 ***

SYMBOL                                SIZE    COUNTS  OPTIMIZE
  (1)                                  (2)      (3)      (4)
_IntGlob
                                        2         2
```

图 8.16　变量存取优化对象符号信息的输出例子（连接列表）

（1）显示符号名。
（2）显示符号的长度。
（3）显示符号的参照次数。
（4）显示有无进行优化。如果已优化，就输出"done"。

8.4.8　　函数存取优化对象符号信息

在指定 show=reference 选项时，输出函数存取优化（optimize=function_call）对象的符号参照次数和有无进行优化。函数存取优化对象符号信息的输出例子如图 8.17 所示。

```
*** Function Call ***

SYMBOL                              COUNTS  OPTIMIZE
  (1)                                 (2)      (3)
_malloc
                                         5  done
_Proc0
                                         4
```

图 8.17　函数存取优化对象符号信息的输出例子（连接列表）

（1）显示符号名。
（2）显示符号的参照次数。
（3）显示有无进行优化。如果已优化，就输出"done"。

8.4.9　　参照表信息

输出符号的参照信息（参照表信息）。参照表信息的输出例子如图 8.18 所示。

```
*** Cross Reference List ***

No    Unit Name    Global.Symbol    Location   External Information
(1)      (2)           (3)            (4)              (5)
0001 a
     SECTION=P
                 _func
                                 00000100
                 _func1
                                 00000116
                 _main
                                 0000012c
                 _g
                                 00000136
     SECTION=B
                 _a
                                 00000190  0001(00000140:P)
                                           0002(00000178:P)
                                           0003(0000018c:P)
0002 b
     SECTION=P
                 _func01
                                 00000154  0001(00000148:P)
                 _func02
                                 00000166  0001(00000150:P)
0003 c
     SECTION=P
                 _func03
                                 00000184
```

图 8.18　参照表信息的输出例子(连接列表)

（1）显示 Unit 号（以目标为单位的识别号）。
（2）目标名。按连接时的输入指定顺序显示。
（3）符号名。按段升序显示。
（4）显示符号的分配地址。
　　在指定 form=rel 时，为从段起始地址开始的相对值。
（5）显示正在参照外部符号的地址。
　　输出格式如下：
　　<Unit 号>（<地址 or 段内偏移量>:<段名>）

8.5　　库列表的参照方法

本节说明优化连接编辑程序输出的库列表的内容和格式。

8.5.1　　库列表的结构

库列表的结构和内容如表 8.6 所示。

表 8.6　库列表的结构和内容

No	建立的列表	内容	子选项	show 选项省略时*1
1	选项信息	显示由命令行或者子命令指定的选项列	—	输出
2	错误信息	显示错误信息	—	输出
3	库信息	显示库信息	—	输出
4	库内模块、段、符号信息	显示库内模块	—	输出
		在指定 show=symbol 选项时，还显示模块内的符号名一览表	show=symbol	不输出
		在指定 show=section 选项时，还显示各模块内的段名和符号名一览表	show=section	不输出

【注】 *1　show选项只在指定list选项时有效。

8.5.2　　选项信息

输出由命令行或者子命令文件指定的选项列。选项信息的输出例子如图 8.19 所示（在指定 `optlnk -sub=test.sub -list –show` 时）。

```
 test.sub的内容
form   library
in     adhry.obj
output test.lib
```

```
*** Options ***

-sub=test.sub               ┐
form   library  ┐           │
in     adhry.obj├ (2)       ├ (1)
output test.lib ┘           │
-list                       │
-show                       ┘
```

图 8.19　选项信息的输出例子（库列表）

（1）按照指定顺序输出由命令行或者子命令指定的选项列。

（2）是子命令文件 `test.sub` 中的子命令。

8.5.3　错误信息

输出错误信息。错误信息的输出例子如图 8.20 所示。

```
*** Error information ***

** L1200 (W) Backed up file "main.lib" into "main.lbk"   } (1)
```

图 8.20　错误信息的输出例子（库列表）

（1）输出错误信息。

8.5.4　库信息

输出库的种类。库信息的输出例子如图 8.21 所示。

```
*** Library Information ***

LIBRARY NAME=test.lib    (1)
CPU=H8S                  (2)
ENDIAN=Big               (3)
ATTRIBUTE=system         (4)
NUMBER OF MODULE=1       (5)
```

图 8.21　库信息的输出例子（库列表）

（1）显示库名。
（2）显示 cpu 名。
（3）显示 Endian 种类。
（4）显示库文件的属性为系统库还是用户库。
（5）显示库内的模块数。

8.5.5　　库内模块、段、符号信息

输出库内的模块一览表。

在指定 show=symbol 选项时，输出模块内符号名一览表。在指定 show=section 选项时，还输出模块内段名和符号名一览表。

库内模块、段、符号信息的输出例子如图 8.22 所示。

```
*** Library List ***

MODULE     LAST UPDATE
  (1)           (2)
  SECTION
    (3)
    SYMBOL
      (4)
adhry
          29-Feb-2000 12:34:56
  P
    _main
    _Proc0
    _Proc1
  C
  D
    _Version
  B
    _IntGlob
    _CharGlob
```

图 8.22　库内模块、段、符号信息的输出例子（库列表）

（1）显示模块名。
（2）显示模块注册的日期。如果模块被更新，就显示最新的更新日期。
（3）显示模块内的段名。
（4）显示段内的符号。

第 9 章　编程

9.1　程序结构

9.1.1　段

段由 C/C++编译程序或者汇编程序输出的目标程序的执行指令和数据的各区域构成。段是分配存储器的最小单位，段有以下性质：

段属性

code	保存执行指令。
data	保存数据。
stack	堆栈区。

方式种类

相对寻址方式 …… 是能通过优化连接编辑程序进行重新分配的段。

绝对寻址方式 …… 是由地址决定的段。不能通过优化连接编辑程序进行重新分配。

初始值

定义开始执行程序时有无初始值。在同一段内有初始值的数据和没有初始值的数据不能混在一起。只要有一个初始值，就用0初始化没有初始值的区域。

写操作

表示是否可以在执行程序时进行写操作。

边界调整数

是分配段的地址调整值。优化连接编辑程序将地址调整为边界调整数的倍数地址。

9.1.2　　C/C++程序的段

C/C++程序、标准库使用的存储区种类和段的对应如表 9.1 所示。

表 9.1　存储区种类及其性质的概要

	名称	段		方式种类	初始值	边界调整数	内容
		名称	属性		写操作		
1	程序区	P*[1]	code	相对方式	有 不可	2byte	保存机器码
2	常数区	C*[1]	data	相对方式	有 不可	2byte	保存 const 型的数据
3	初始化数据区	D*[1]	data	相对方式	有 可	2byte	保存有初始值的数据
4	未初始化数据区	B*[1]	data	相对方式	无 可	2byte	保存没有初始值的数据
5	常数区 （8bit 地址空间）	$ABS8C*[1]	data	相对方式	有 不可	1byte	保存由 abs8 选项、_ _abs8 或者#pragma abs8 指定的 8 位 const 型的数据
6	初始化数据区 （8bit 地址空间）	$ABS8D*[1]	data	相对方式	有 可	1byte	保存由 abs8 选项、_ _abs8 或者#pragma abs8 指定的 8 位有初始值的数据
7	未初始化数据区 （8bit 地址空间）	$ABS8B*[1]	data	相对方式	无 可	1byte	保存由 abs8 选项、_ _abs8 或者#pragma abs8 指定的 8 位没有初始值的数据
8	常数区 （16bit 地址空间）	$ABS16C*[1]	data	相对方式	有 不可	2byte	保存由 abs16 选项、_ _abs16 或者#pragma abs16 指定的 const 型的数据
9	初始化数据区 （16bit 地址空间）	$ABS16D*[1]	data	相对方式	有 可	2byte	保存由 abs16 选项、_ _abs16 或者#pragma abs16 指定的有初始值的数据
10	未初始化数据区 （16bit 地址空间）	$ABS16B*[1]	data	相对方式	无 可	2byte	保存由 abs16 选项_ _abs16 或者#pragma abs16 指定的没有初始值的数据
11	函数地址区 （存储器间接空间）	$INDIRECT*[1]	data	相对方式	有 不可	2byte	保存由 indirect=normal 选项、_ _indirect 或者 #pragma indirect 指定的函数地址
12	函数地址区 （扩展存储器间接空间）	$EXINDIRECT*[1]	data	相对方式	有 不可	2byte	保存由 indirect=extended 选项或者_ _indirect_ex 指定的函数地址
13	函数地址区 （存储器间接空间）	$VECTxx*[1] xx: 向量号	data	绝对方式	有 不可	2byte	保存由_ _indirect、#pragma indirect、_ _indirect_ex、_ _interrupt、#pragma interrupt、_ _entry、#pragma entry 的 vect=xx 指定的函数地址

	名称	段		方式	初始值	边界	内容
		名称	属性	种类	写操作	调整数	
14	1 个字节数据区	yy$1[*2] yy:C[*1],D[*1],B[*1], $ABS16C[*1], $ABS16D[*1], $ABS16B[*1]	data	相对方式	—	1byte	处理指定 align=4 选项时的 1 个字节数据的区域 按各段生成
15	4 个字节数据区	yy$4[*2] yy:C[*1],D[*1],B[*1], $ABS16C[*1], $ABS16D[*1], $ABS16B[*1]	data	相对方式	—	4byte	处理指定 align=4 选项时的 4 个字节数据的区域 按各段生成
16	初始化数据段的地址区	C$DSEC[*3]	data	相对方式	有 不可	2byte	初始化数据区段的 ROM 地址、ROM 的最后地址和 RAM 地址
17	未初始化数据段的地址区	C$BSEC[*3]	data	相对方式	有 不可	2byte	保存未初始化数据区段的地址和最后地址
18	C++初始处理/后处理的数据区	C$INIT[*3]	data	相对方式	有 不可	2byte	对于全局类目标，保存被调用的构造函数和析构函数的地址
19	C++虚拟函数表的区域	C$VTBL[*3]	data	相对方式	有 不可	2byte	保存为了在类声明中有虚拟函数时调用虚拟函数的数据
20	堆栈区	S	stack	相对方式	无 可	2byte	执行程序时必需的区域 请参照“9.2.1 (2) 动态区的分配”
21	堆区	—	—	相对方式	无 可	—	由库函数 malloc、realloc、calloc、new 使用的区域 请参照“9.2.1 (2) 动态区的分配”
22	指定地址分配区	$ADDRESS $yy<地址> yy:C,D,B	data	绝对方式	有/无 可/不可[*4]	—	分配由#pragma address 指定的变量的区域

【注】

*1　能通过编译程序选项 section 或者扩展名#pragma section、#pragma abs8 section、#pragma abs16 section、#pragma indirect section 改换段名。

*2　给 yy 设定数据分割前的数据段名。例：C → C$1、C$4

*3　如果指定编译程序选项 section=C=zz，前缀 C 就变为 zz。

*4　初始值的有/无以及写属性的可/不可取决于 C、D、B 的段属性。

例 1：用程序例子表示 C 程序和编译程序生成段的对应。

```
int a=1;
char b;
const int c=0;
void main(){
 ...
}
```

C程序

程序区(main(){...})

常数区(c)

初始化数据区(a)

未初始化数据区(b)

编译程序生成的区域和被保存的数据

例 2：用程序例子表示 C++程序和编译程序生成段的对应。

```
class A{
  int m;
public:
  A(int p);
  ~A();
};
A a(1);
int b;
extern const char c=’a’;
int d=1;
void f(){...}
```

C++程序

程序区(f(){...})

常数区(c)

初始化数据区(d)

未初始化数据区(a,b)

初始处理/后处理的数据区
(&A::A, &A::~A)

编译程序生成的区域和被保存的数据

9.1.3　　汇编程序的段

在汇编程序中，使用".SECTION"控制指令声明段的开始、属性和方式种类。".SECTION"控制指令的声明格式如下。详细内容请参照“11.3　汇编程序控制指令”。

.SECTION <段名>[,<段属性>[,<方式种类]]

<方式种类>：相对寻址方式段：align=<边界调整数>

　　　　　　绝对寻址方式段：locate=<地址值>

例：表示汇编程序的段声明例子。

```
            .CPU 2600A
            .OUTPUT DBG
SIZE :      .EQU 8
;
            .SECTION A,CODE,ALIGN=2          …… (1)
START:
            MOV.L #CONST:32,ER0
            MOV.L #DATA:32,ER1
            MOV.L #SIZE:32,ER2
LOOP:
            CMP.L #0:32,ER2
            BEQ EXIT
            MOV.B @ER0,R3L
            MOV.B R3L,@ER1
            ADD.L #1:32,ER0
            ADD.L #1:32,ER1
            SUB.L #1:32,ER2
            BRA LOOP
EXIT:
            SLEEP
            BRA START
;
            .SECTION B,DATA,LOCATE=H'00001000  …… (2)
CONST
            .DATA.B H'01,H'02,H'03,H'04
            .DATA.B H'05,H'06,H'07,H'08
;
            .SECTION C,STACK,ALIGN=2         …… (3)
DATA
            .RES.B SIZE
;
            .END START
```

（1） 声明段名 A、边界调整数 2 和相对寻址方式的 code 段。

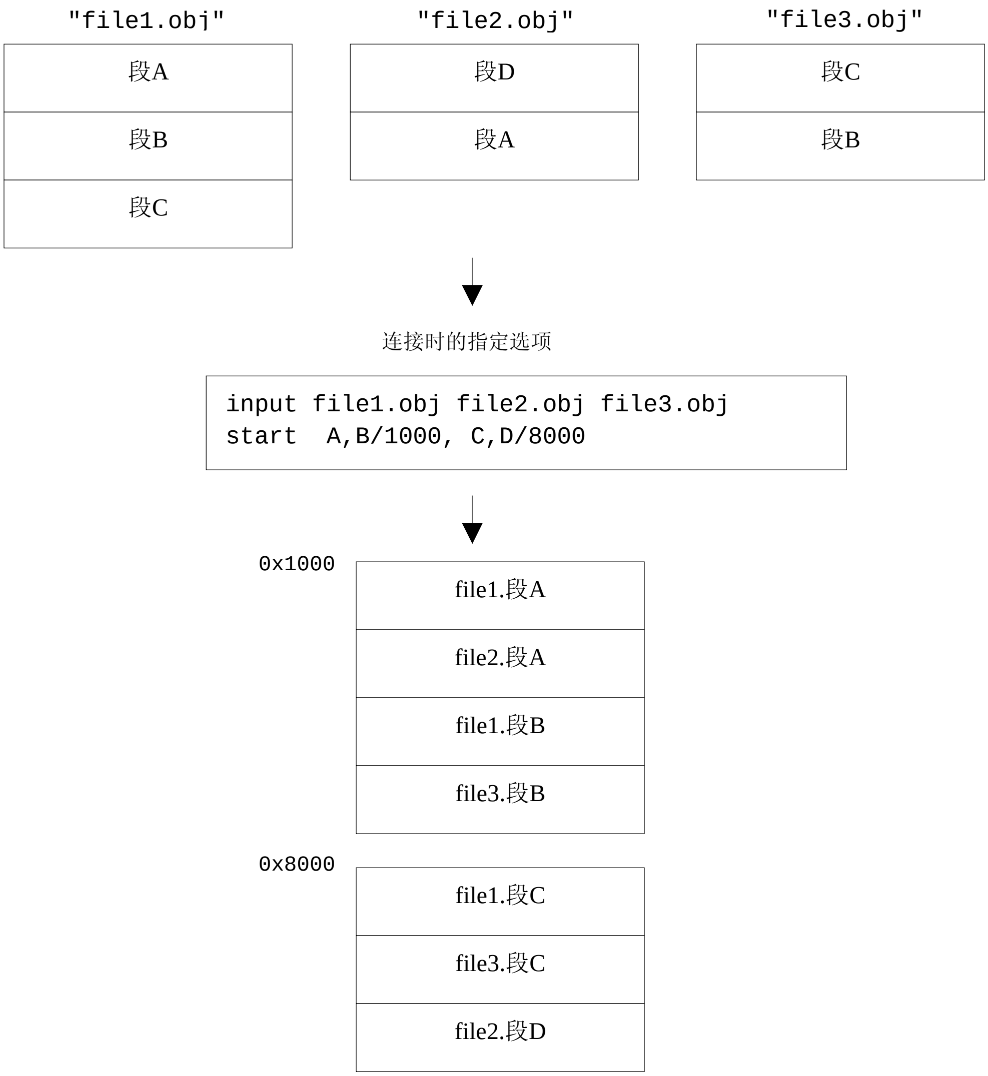
（2） 声明段名 B、分配地址 H'1000 和绝对寻址方式的 data 段。

（3） 声明段名 C、边界调整数 2 和相对寻址方式的 stack 段。

9.1.4　　段的连接

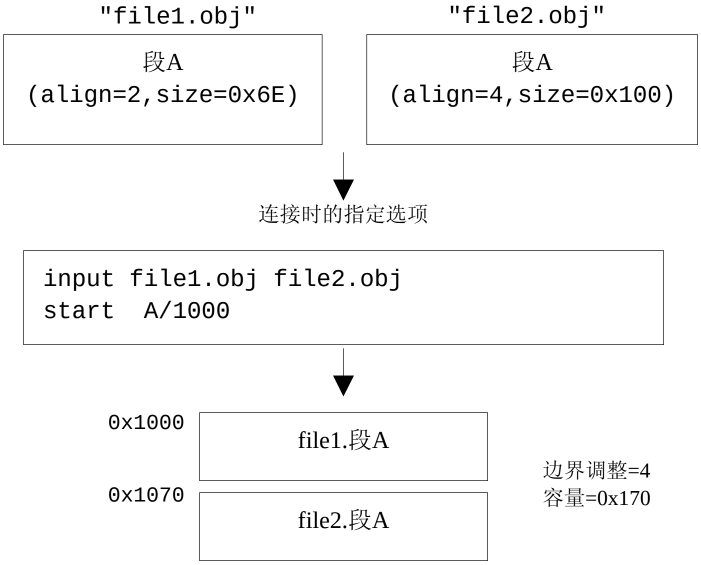
优化连接编辑程序将输入目标程序内的相同段进行连接，分配到由 `start` 选项指定的地址。

（1）按照文件的输入顺序连续分配不同文件的同名段。

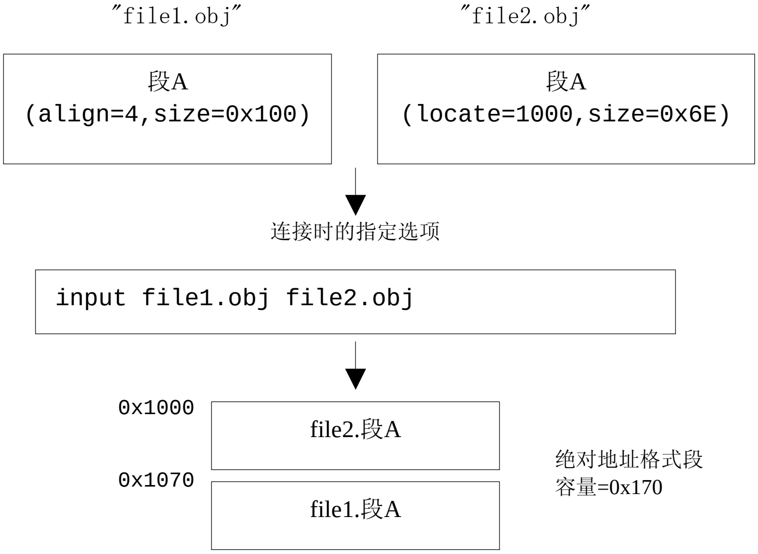


```
input file1.obj file2.obj file3.obj
start  A,B/1000, C,D/8000
```

（2）在边界调整后连接不同边界调整数的同名段。段的边界调整数以大的为准。

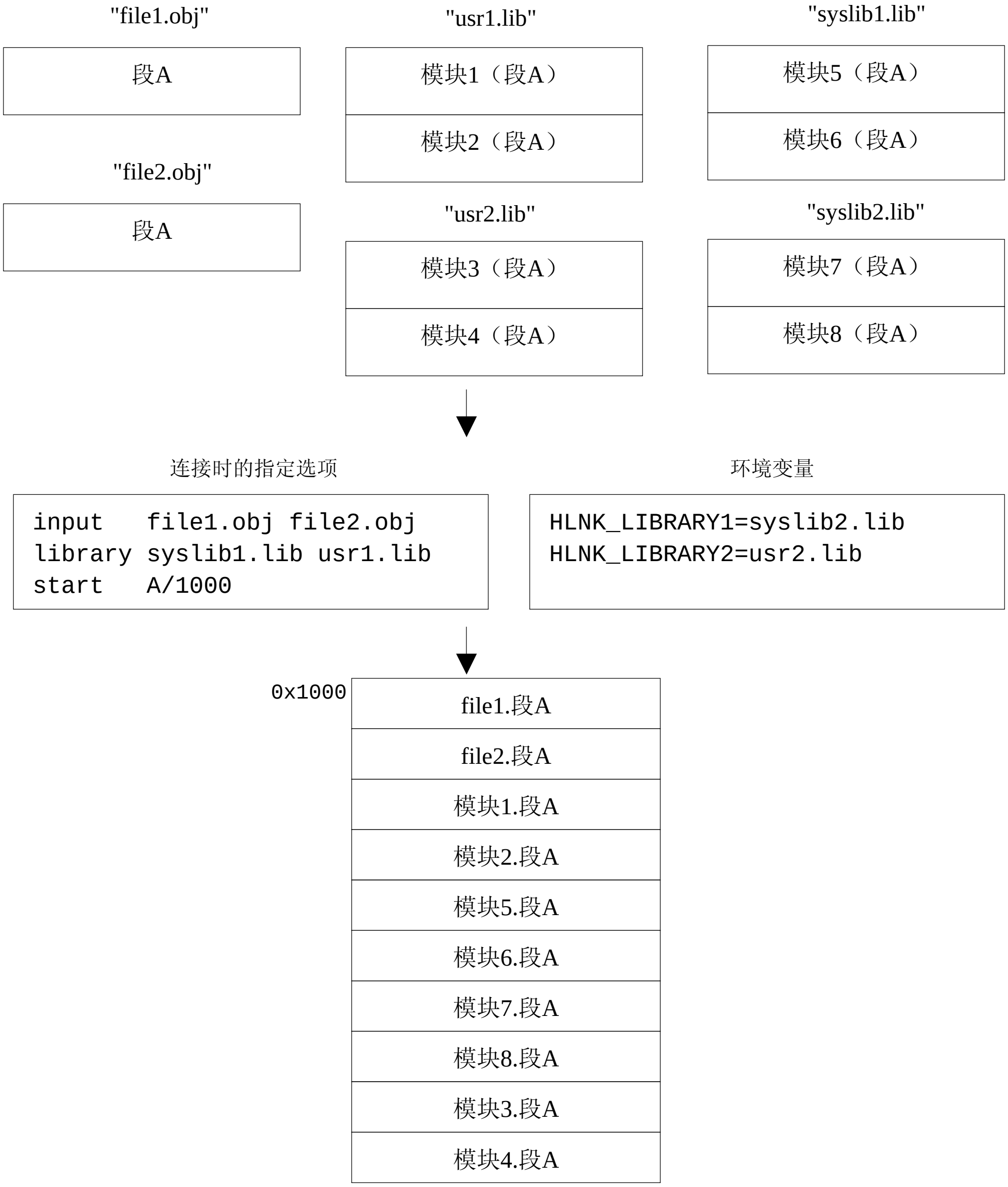


（3）在同名段中包含绝对寻址方式和相对寻址方式时，将相对寻址方式的目标连接到绝对寻址方式的目标后面。即使在指定可再定位文件（form=relocate）输出时，该段也变为绝对寻址方式段。



（4）有关同名段内目标连接顺序的规则如下：

`input` 选项或者命令行的输入文件的指定顺序
`library` 选项的用户库的指定顺序和库内模块的输入顺序
`library` 选项的系统库的指定顺序和库内模块的输入顺序
环境变量`(HLNK_LIBRARY1～3)`的库的指定顺序和库内模块的输入顺序



```
input   file1.obj file2.obj
library syslib1.lib usr1.lib
start   A/1000
```

```
HLNK_LIBRARY1=syslib2.lib
HLNK_LIBRARY2=usr2.lib
```

9.2　初始设定程序的建立

本章说明将程序嵌入到采用 H8SX、AE5、H8S/2600、H8S/2000、H8/300H 或者 H8/300 的系统的方法。

将程序嵌入到系统时，需要进行以下的准备：

- 存储器的分配
  需要将各段、堆栈区、堆区分配到系统 ROM 和 RAM 区。

- 程序执行环境的设定
  设定程序执行环境的处理有寄存器的初始设定、存储区的初始化和程序的启动。

另外，在使用输入/输出等 C/C++库函数时，需要在设定执行环境时进行库的初始化。尤其在使用输入/输出（stdio.h、ios、streambuf、istream、ostream）和存储器分配（stdlib.h、new）功能时，需要按各系统建立低级输入/输出例程和存储区分配例程。

在使用进行程序结束处理的 C 库函数（exit、atexit、abort 函数）时，需要根据用户系统另外建立这些函数。

9. 2. 1 说明决定程序存储区地址的方法，并且举例说明有关实际决定地址的优化连接编辑程序的选项指定方法。

9. 2. 2 说明执行环境的设定项目及设定程序的实例。

另外，还说明有关库函数的初始设定处理、低级例程的建立方法以及结束处理函数的建立例子。

9.2.1　　存储区的分配

为了将目标程序嵌入系统，需要决定程序使用的存储区容量，将各自的区域分配到适当的存储地址。

程序使用的存储区有静态分配区（如同与程序中的函数对应的执行指令及外部数据定义声明的数据区）和动态分配区（如同堆栈区）。以下说明各区域的分配方法：

（1）静态区的分配

（a）静态区的内容

将堆栈区和堆区以外的段分配到静态区。

将 C/C++程序的各段（程序区、常数区、初始化数据区、未初始化数据区、函数地址区、初始化数据段地址区、未初始化数据段地址区、C++初始处理/后处理的数据区和 C++虚拟函数表区）分配到静态区。

（b）容量的计算法

静态区的容量为编译程序、汇编程序生成的目标程序容量和 C/C++程序使用的库函数容量的总和。

因为在连接目标程序后，将包含库的各段的容量输出到连接列表的连接映像信息，所以能得知静态区的容量。

连接列表中连接映像信息的例子如图 9.1 所示。

```
*** Mapping List ***

SECTION                             START      END         SIZE   ALIGN
  (1)                                (2)       (3)          (4)    (5)
P
                                   00000000  000004d6       4d6   2
C
                                   000004d6  00000533        5d   2
D
                                   00000534  0000053c         8   2
B
                                   0000053c  00004112      3bd6   2
```

图 9.1　连接列表中连接映像信息的例子

将编译、汇编单位的段容量输出到编译列表中的统计信息和汇编列表中的段信息。编译列表中的统计信息例子如图 9.2 所示，汇编列表中的段信息例子如图 9.3 所示。

```
******* SECTION SIZE INFORMATION *******
PROGRAM   SECTION (P):                            0x00000080 Byte(s)
CONSTANT  SECTION (C):                            0x00000004 Byte(s)
DATA      SECTION (D):                            0x00000004 Byte(s)
BSS       SECTION (B):                            0x00000004 Byte(s)

TOTAL  PROGRAM   SECTION:  0x00000080 Byte(s)
TOTAL  CONSTANT  SECTION:  0x00000004 Byte(s)
TOTAL  DATA      SECTION:  0x00000004 Byte(s)
TOTAL  BSS       SECTION:  0x00000004 Byte(s)

TOTAL  PROGRAM  SIZE:  0x0000008C Byte(s)
```

图 9.2　编译列表中的统计信息例子

```
*** SECTION DATA LIST

SECTION                          ATTRIBUTE    SIZE             START

P                                REL-CODE    000000604
D                                REL-DATA    000000008
C                                REL-DATA    00000005D
B                                REL-DATA    000003BD6
```

图 9.3　汇编列表中的段信息例子

在不使用标准库时，文件单位的段容量的总和为静态区的容量；在使用标准库时，为各段的存储区容量加上库函数使用的存储区容量。在编译程序提供的标准库中，除了 C 语言规格规定的 C 库函数和面向嵌入式的 C++类库以外，包含程序执行时必需的进行算术运算的例程（运行时例程）。因此，必须注意：即使在源程序中不指定使用的库函数，也可能需要标准库。

能从编译程序输出的编译列表的符号分配信息得知程序使用的运行时例程。具体例子如下所示：

C程序

```
     long a,b;
     main()
    {
                a *= b;
    }
```

C编译程序输出的符号分配信息

```
******* STACK FRAME INFORMATION *******

FILE NAME:  main.c

Function (File main.c   , Line     2): main

Parameter Area Size      : 0x00000000 Byte(s)
Linkage Area Size        : 0x00000008 Byte(s)
Local Variable Size      : 0x00000000 Byte(s)
Temporary Size           : 0x00000000 Byte(s)
Register Save Area Size  : 0x00000000 Byte(s)
Total Frame Size         : 0x00000008 Byte(s)

Used Runtime Library Name
$MULL$3                                  ;运行时例程
```

（c）ROM 和 RAM 的分配

在 ROM 化程序时，根据段初始值的有无、写操作的可/不可决定是分配到 ROM 或者 RAM。

在 ROM 化 C/C++程序的各段时，如下分配到 ROM 和 RAM：

• 程序区	（段P）	→ ROM
• 常数区	（段C、$ABS8C、$ABS16C）	→ ROM
• 未初始化数据区	（段B、$ABS8B、$ABS16B）	→ RAM
• 初始化数据区	（段D、$ABS8D、$ABS16D）	→ ROM、RAM（参照（d））
• 函数地址区	（段$INDIRECT、$EXINDIRECT）	→ ROM
• 初始化数据段地址区	（段C$DSEC）	→ ROM
• 未初始化数据段地址区	（段C$BSEC）	→ ROM
• 初始处理数据区[*1]	（段C$INIT）	→ ROM
• 虚拟函数表区[*2]	（段C$VTBL）	→ ROM

–

【注】　*1　C++程序中有全局类目标时，由编译程序生成。

*2　C++程序中有虚拟函数声明时，由编译程序生成。

（d）初始化数据区的分配

如同初始化数据区，在连接时将具有初始值并能在程序执行时更改值的段保存到ROM，在程序开始执行时将其复制到RAM。因此，必需用优化连接编辑程序的rom 选项，在ROM和RAM中双重确保区域。有关指定例子，请参照“（e）存储器的分配例子和连接时的地址指定方法”。另外，在“9.2.2 （2）初始设定（PowerON_Reset）”中说明从ROM将值复制到RAM的段的初始设定。

（e）存储器的分配例子和连接时的地址指定方法

在建立绝对装入模块时，通过优化连接编辑程序的选项或者子命令指定分配到各段的区域地址。以下说明静态区的存储器分配例子和连接时的地址指定方法。

H8S/2600 高级模式中的静态区的分配例子如图 9.4 所示。

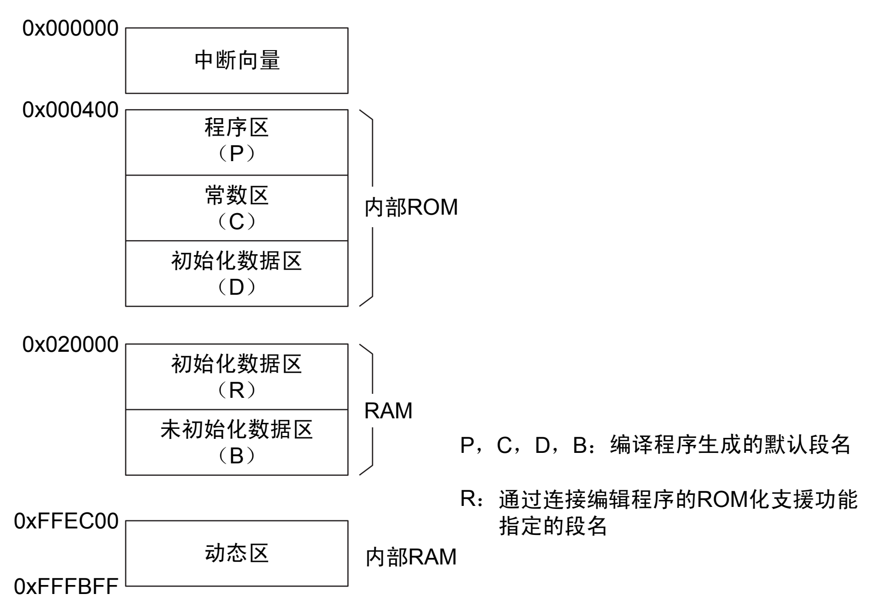


图 9.4　静态区的分配例子

如果进行如图 9.4 所示的存储器分配，就在连接时指定以下的子命令：

```
ROM△D=R                          …… ［1］
START△P,C,D/400,R,B/20000        …… ［2］
```

说明：［1］将和段名 D 相同大小的段 R 确保到输出装入模块。另外，在参照被分配到段 D 的符号时，为了使其变为段 R 中的地址，进行再定位。段 D 为 ROM 中的初始化数据段名，段 R 为 RAM 中的初始化数据段名。

［2］将段P、C、D分配到从内部ROM地址0x400开始的连续区域。另外，将段R、B分配到RAM地址0x20000开始的连续地址。

（2）动态区的分配

（a）动态区的内容

C/C++程序中使用的动态区有以下二个：

- 堆栈区
- 堆区（用于存储器分配的库函数）

（b）堆栈区容量的计算法

如果指定优化连接编辑程序的 stack 选项来输出堆栈信息文件，就能使用栈解析工具计算出 C/C++程序、标准库使用的堆栈区的最大使用量。有关栈解析工具的使用方法，请参照“**第 6 章　栈解析工具的操作方法**”。

在汇编源程序中，对用.stack 控制指令指定的标号，能通过栈解析工具计算出堆栈使用量。对除此以外的汇编程序中的标号，由于没有将堆栈使用量输出到堆栈信息文件，所以就不能通过栈解析工具计算出堆栈使用量。请参考以下的 C/C++程序的堆栈使用量计算方法，计算出汇编程序的堆栈使用量，然后加上通过栈解析工具计算出的堆栈使用量。

・C/C++程序堆栈使用量的计算方法

C/C++程序使用的堆栈区在调用函数时被分配到堆栈，在函数返回时被释放。为了计算出堆栈区的容量，首先计算每个函数的堆栈使用量，然后根据函数的调用关系计算实际的堆栈使用量。

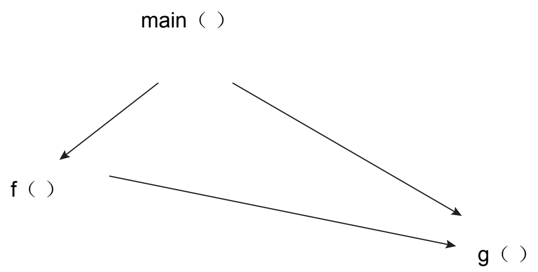
能从编译列表的符号分配信息（Total Frame Size） 得知各函数使用的堆栈区。

```
******** STACK FRAME INFORMATION ********
FILE NAME: test.c

Function (File test.c    , Line    2) : main

  Optimize Option Specified : No Allocation Information Available

Parameter Area Size        : 0x00000008 Byte(s)
Linkage Area Size          : 0x00000004 Byte(s)
Local Variable Size        : 0x00000002 Byte(s)
Temporary Size             : 0x00000000 Byte(s)
Register Save Area Size    : 0x00000004 Byte(s)
Total Frame Size           : 0x00000012 Byte(s)
```

函数使用的堆栈区容量为 Total Frame Size 的 0x12，即 18 个字节。

函数的调用关系和各函数的堆栈使用量的例子如图 9.5 所示。

此时，通过函数 f 调用函数 g 时的堆栈区容量根据表 9.2 计算。



函数名	堆栈使用量（字节）
main	18
f	32
g	24

图 9.5　函数的调用关系和堆栈使用量的例子

表 9.2　堆栈容量的计算例子

调用路径	堆栈容量的合计
main (18)→ f (32)→ g (24)	74
main (18)→ g (24)	42

←堆栈使用量
（最大值）

如此，对调用级最深的函数计算堆栈区的容量，分配其最大值（此时为 74 个字节）的堆栈区。

・有关堆栈使用量计算的注意事项

在版本为 Ver.4.0 以下或 Ver.6.0 且 CPU 不为 H8SX 时，与版本为 Ver.6.0 且 CPU 为 H8SX 时的堆栈使用量的计算方法不同。在本说明中，将版本为 Ver.4.0 以下或 Ver.6.0 且 CPU 不为 H8SX 的情况称为 A 方式；将版本为 Ver.6.0 且 CPU 为 H8SX 和版本为 Ver.6.01 且 CPU 为 H8SX 或 H8S 的情况称为 B 方式。在互相调用 A 方式编译的函数和 B 方式编译的函数时，必须注意堆栈使用量的计算。

在 A 方式和 B 方式的堆栈传递参数时的 SP（堆栈指针）处理不同。A 方式如下例[1]，在通过堆栈传递参数时利用进栈指令或者先减寻址方式（@-SP），先将 SP 递减后保存参数到堆栈。此时，如果从函数调用返回时，就在 SP 递增堆栈传递的参数大小后释放由堆栈传递的参数区，如下例[2]。在此方式中，通过堆栈传递的参数区容量根据各调用函数而不同，其容量计入被调用函数的堆栈帧大小的 Parameter Area Size，如下例[3]。

在 B 方式中，编译程序预先计算函数内使用的堆栈容量的最大值，在函数入口处确保其最大值，如下例[4]。在函数出口前 SP 保持固定的值，而在函数出口处将 SP 恢复到调用前的状态，如下例[6]。此时，在通过堆栈传递函数的参数时，SP 不变，将参数保存到 0 或者正偏移量的堆栈，如下例[5]。在此方式中。通过堆栈传递的参数区的最大值计入被调用函数的堆栈帧大小的 Temporary Size，如下例[7]。

如下所示的 CASE 1 和 CASE 4 的情况，在调用函数和被调用函数的方式相同时，从函数 g 调用函数 f 时的堆栈使用量加上各自的 Total Frame Size 后变为 12 个字节的正确值。如 CASE 2 的情况，在从 A 方式调用 B 方式时，虽然加上 Total Frame Size 后变为 8 个字节，但是由于不计入通过堆栈传递的参数区容量，所以比正确值少估计了堆栈使用量。如 CASE 3 的情况，在从 B 方式调用 A 方式时，加上 Total Frame Size 后变为 16 个字节，由于通过堆栈传递的参数区容量被二重计算，所以比正确值多估计了堆栈使用量。

为了避免过小或过大的估量，不将 A 方式和 B 方式混在一起，或者在找到从 A 方式调用 B 方式或从 B 方式调用 A 方式的位置后调整堆栈使用量。

堆栈使用量
CASE 1: 从 A 方式的函数 g 调用 A 方式的函数 f：8+4=12
CASE 2: 从 A 方式的函数 g 调用 B 方式的函数 f：4+4=8
CASE 3: 从 B 方式的函数 g 调用 A 方式的函数 f：8+8=16
CASE 4: 从 B 方式的函数 g 调用 B 方式的函数 f：8+4=12

例：　源程序　　　　　　　　A 方式　　　　　　　　　　B 方式

```
int f(struct S);         _f:                            _f:
void g(void);              SUB.W    R0,R0                 SUB.W       R0,R0
struct S{long p;} st;      RTS                            RTS
int x;                   _g:                            _g:
int f(struct S s){                                        ADD.W       #-4:16,R7    ;[4]
    return 0;              MOV.L    @_st:32,ER0           MOV.L       @_st:32,ER0
}                          PUSH.L   ER0         ;[1]      MOV.L       ER0,@SP      ;[5]
void g(void)               BSR      _f:8                  BSR         _f:8
{                          ADDS.L   #4,SP       ;[2]      MOV.W       R0,@_x:32
    x=f(st);               MOV.W    R0,@_x:32             ADDS.L      #4,SP        ;[6]
}                          RTS                            RTS
函数 f

Parameter Area Size      : 0x00000004 Byte(s) [3]  0x00000000 Byte(s)
Linkage Area Size        : 0x00000004 Byte(s)      0x00000004 Byte(s)
Local Variable Size      : 0x00000000 Byte(s)      0x00000000 Byte(s)
Temporary Size           : 0x00000000 Byte(s)      0x00000000 Byte(s)
Register Save Area Size  : 0x00000000 Byte(s)      0x00000000 Byte(s)
Total Frame Size         : 0x00000008 Byte(s)      0x00000004 Byte(s)
```

函数 g

```
Parameter Area Size     : 0x00000000 Byte(s)      0x00000000 Byte(s)
Linkage Area Size       : 0x00000004 Byte(s)      0x00000004 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)      0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)      0x00000004 Byte(s) [7]
Register Save Area Size : 0x00000000 Byte(s)      0x00000000 Byte(s)
Total Frame Size        : 0x00000004 Byte(s)      0x00000008 Byte(s)
```

（c）堆区容量的计算法

堆区中使用的存储区的容量是 C/C++程序中由存储器管理库函数（calloc、malloc、realloc、new 函数）分配的区域的总和。但是，因为存储器管理库函数在每次调用时将 4 个字节（指定 cpu＝H8SXN、H8SXM、H8SXA（有 ptr16 选项）、 H8SXX（有 ptr16 选项）、2600n、2000n、300hn、300 时）或者 8 个字节（指定 cpu＝H8SXA（无 ptr16 选项）、 H8SXX（无 ptr16 选项）、2600a、2000a、300ha 时）用作管理区，所以必须将实际确保的区域容量加上此管理区的容量。

另外，编译程序以用户指定的存储器容量（_sbrk_size）为单位管理堆区。有关_sbrk_size 的指定方法，请参照“9.2.2（5）C/C++库函数的初始设定（_INITLIB）”，确保为堆区的区域容量（HEAPSIZE）计算如下：

HEAPSIZE＝_sbrk_size × n (n≧1)

（由存储器管理库分配的区域容量）＋ 管理区容量≦HEAPSIZE

输入/输出库函数在内部处理中使用存储器管理库函数。输入/输出中分配的区域容量为

- 指定 cpu＝H8SXN、H8SXM、H8SXA（有 ptr16 选项）、H8SXX（有 ptr16 选项）、2600n、2000n、300hn、300 时：
  514 个字节 ×（同时打开的文件个数的最大值）
- 指定 cpu＝H8SXA（无 ptr16 选项）、H8SXX（无 ptr16 选项）、2600a、2000a、300ha 时：
  516 个字节 ×（同时打开的文件个数的最大值）

注意　虽然能在再次通过存储器管理库函数确保区域时重新利用由存储器管理库函数的 free 函数、delete 运算符（C++）释放的区域，但是根据重复分配，即使空区域的总容量很充足，空区域也被分割为小区域，所以有可能发生不能确保所需要的大容量区域的情况。为了避免此情况，必须尽量在程序执行开始后立即确保大容量区域。另外，必须尽量固定释放后重新利用的数据区容量。

（d）动态区的分配方法

动态区分配在 RAM 中。

对于堆栈区，通过在程序启动时的复位例程中将堆栈段的最高位地址设定到 SP（堆栈指针）决定分配的地址。通过 C/C++编译程序的_ _entry（或者#pragma entry）、#pragma stacksize，编译程序自动进行堆栈区（S 段）的生成和复位程序中的 SP 初始设定码的输出。

对于堆区，通过低级接口例程（sbrk）的初始设定决定分配的地址。请参照“9.2.2（2）初始设定（PowerON_Reset）”和“9.2.2（7）低级接口例程”。

9.2.2　执行环境的设定

本节说明有关设定执行程序时所必需的环境处理。但是，程序的执行环境因各用户系统而不同，所以必需根据用户系统的规格建立执行环境的设定程序。

程序构成的例子如图 9.6 所示。

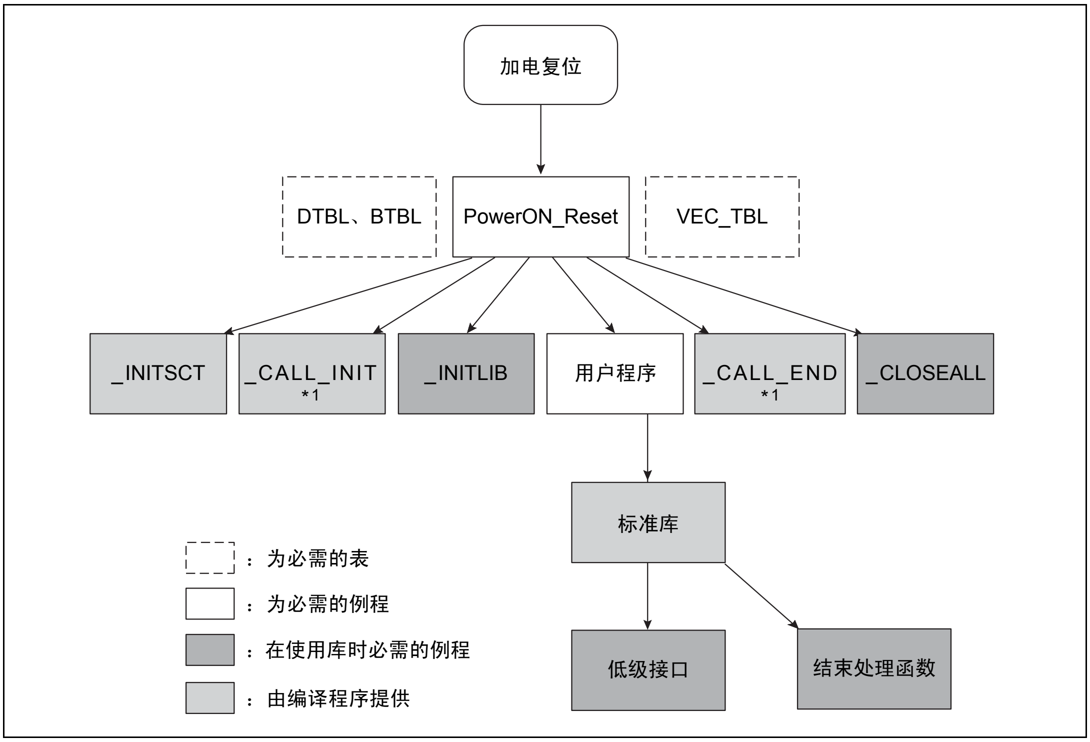


图 9.6　程序构成的例子

【注】*1　在 C++程序中有全局类目标声明时需要。

各构成例程的内容如下：

- 向量表（VEC_TBL）
  为了通过加电复位启动寄存器的初始设定程序（PowerON_Reset），设定向量表。
- 初始设定（PowerON_Reset）
  在进行寄存器的初始设定后，依次调用初始设定例程。
- 段初始化表（DTBL、BTBL）
  使用段地址运算符设定段初始化例程中使用的段的起始地址和结束地址。
- 段的初始化（_INITSCT）*1
  用零对没有设定初始值的静态变量区（未初始化数据区）进行初始化。另外，将初始化数据区的初始值从ROM复制到RAM。
- 全局类目标的初始处理（_CALL_INIT）*1*2
  对被声明为全局的类目标，调用构造函数。
- 全局类目标的后处理（_CALL_END）*1*2
  在执行main函数后，对全局类目标，调用析构函数。

- C/C++库函数的初始设定（_INITLIB）
  在使用C/C++库函数时，对初始设定所需要的内容进行初始设定。
- 文件的关闭（_CLOSEALL）
  关闭全部打开的文件。
- 低级接口例程
  低级接口例程是在使用标准输入/输出（stdio.h、ios、streambuf、istream、ostream）和存储器管理库（stdlib.h、new）时所必需的库函数和用户系统之间的接口例程。
- 结束处理例程（exit, atexit, abort）*3
  进行程序的结束处理。

【注】*1　作为标准库提供。在使用_INITSCT、_CALL_INIT、_CALL_END 时，必须 include<_h_c_lib.h>。
*2　是在 C++程序中有全局类目标的声明时所必需的处理。
*3　在使用进行程序结束处理的 C 库函数的 exit、atexit、abort 函数时，需要根据用户系统建立这些函数。在使用 C++程序或者 C 库函数 assert 宏时，必须建立 abort 函数。

以下，按照此构成说明各处理的实现方法：

（1）向量表的设定（VEC_TBL）

为了通过加电复位调用初始设定函数“PowerON_Reset”，需要将函数“PowerON_Reset”的地址设定到向量表的 0 地址。

另外，在用户系统使用中断处理和存储器间接函数调用时，需要设定中断向量和地址表。

通过用 C/C++编译程序扩展功能的_ _entry（或者#pragma entry）、_ _interrupt（或者#pragma interrupt）或_ _indirect（或者#pragma indirect）指定 vect 参数，编译程序自动生成向量表。编程的例子如下所示：

例:

```
_ _entry(vect=0) void PowerON_Reset(void)  // 将 PowerON_Reset 函数地址设定到 0 地址
{
               :
}
_ _interrupt(vect=3) void INT_NMI(void)    // 将 INT_NMI 函数地址设定到向量号 3
{
               :
}
_ _indirect(vect=4) char f(void)          // 将 f 函数地址设定到向量号 4
{
               :
}
```

（2）初始设定（PowerON_Reset）

初始设定函数进行堆栈指针（SP）和条件码寄存器（CCR）等寄存器的初始设定，在调用段的初始化例程（_INITSCT）后调用 main 函数。在 C++程序中存在全局类目标时，在 main 函数调用前后调用初始处理函数/结束处理函数的_CALL_INIT 和结束处理函数的_CALL_END 函数。

通过使用 _ _entry（或者#pragma entry），编译程序自动生成 SP 的设定。另外，使用内部函数（set_imask_ccr 等）记述条件码寄存器的设定。

将_INITSCT、_CALL_INIT 和_CALL_END 函数作为标准库提供。在使用这些函数时，必须 include <_h_c_lib.h>。

在使用 C/C++库函数时，调用进行库初始设定的“_INITLIB”和文件关闭处理的“_CLOSEALL”。

编程的例子如下所示：

例：

```
#include <machine.h>          // include<machine.h>
#include <_h_c_lib.h>         // include<_h_c_lib.h>
#pragma stacksize 0x200       // 指定 S 段（堆栈）大小
extern void PowerON_Reset(void);
extern void main(void);

#ifdef __cplusplus
extern "C" {
#endif
extern void _INITLIB(void);
extern void _CLOSEALL(void);
#ifdef __cplusplus
}
#endif

__entry(vect=0) void PowerON_Reset(void)
{                             // 给 SP 设定 S 段的最高位地址
      set_vbr(0x0);           // H8SX 需要时，初始化 VBR
      set_imask_ccr(1);       // 屏蔽中断
       _INITSCT();            // 调用段的初始化例程

#ifdef __cplusplus
        _CALL_INIT();    // 有 C++的全局类目标时，调用函数
#endif

        _INITLIB();           // 调用库的初始设定函数
        set_imask_ccr(0);     // 解除中断屏蔽
        main();
        _CLOSEALL();          // 调用文件的关闭处理函数

#ifdef __cplusplus
        _CALL_END();     // 有 C++的全局类目标时，调用函数

#endif

        sleep();
}
```

（3）段初始化表（DTBL、BTBL）

段的初始化例程（_INITSCT）用零对未初始化的数据段进行初始化，将 ROM 中的初始化数据段的数据复制到 RAM 中。在此，使用段地址运算符，将_INITSCT 函数使用的段的起始地址和结束地址设定到段初始化表。

段初始化表的段名用 C$BSEC 声明未初始化数据区，用 C$DSEC 声明初始化数据区。

编程的例子如下：

例：

```
#ifdef _ _ABS16_ _            // 将段名设定为C$DSEC
#pragma abs16 section $DSEC
#else
#pragma section $DSEC
#endif
static const struct DSEC{
    void * rom_s;             // ROM中的初始化数据段的起始地址成员
    void * rom_e;             // ROM中的初始化数据段的结束地址成员
    void * ram_s;             // RAM中的初始化数据段的起始地址成员
}DTBL[]={
    {_ _sectop (“D”), _ _secend (“D”), _ _sectop (“R”)},
    {_ _sectop (“$ABS8D”), _ _secend (“$ABS8D”), _ _sectop (“$ABS8R”)},
    {_ _sectop (“$ABS16D”), _ _secend (“$ABS16D”), _ _sectop (“$ABS16R”)}
};

#ifdef _ _ABS16_ _            // 将段名设定为C$BSEC
#pragma abs16 section $BSEC
#else
#pragma section $BSEC
#endif
static const struct BSEC{
    void * b_s;               // 未初始化数据段的起始地址成员
    void * b_e;               // 未初始化数据段的结束地址成员
}BTBL[]={
    {_ _sectop (“B”), _ _secend (“B”)},
    {_ _sectop (“$ABS8B”), _ _secend (“$ABS8B”)},
    {_ _sectop (“$ABS16B”), _ _secend (“$ABS16B”)}
};
#ifdef _ _ABS16_ _
#pragma abs16 section
#else
  #pragma section
#endif
```

【注】　必须将上述程序作为 C 语言进行编译（将文件扩展名设定为 c 或者指定 lang=c 选项）。如果作为 C++语言进行编译（将文件扩展名设定为 cpp, cc 或 cp，或者指定 lang=cpp 选项），段初始化表就作为未参照 static 数据，被编译程序删除，所以程序不能正确运行。

段初始化例程（_INITSCT）作为标准库被提供，但是与下例的程序进行同样的运行。

例：

```
static const struct DSEC{    // 前例定义的D段初始化表结构体
    void * rom_s;            // ROM中的初始化数据段的起始地址成员
    void * rom_e;            // ROM中的初始化数据段的结束地址成员
    void * ram_s;            // RAM中的初始化数据段的起始地址成员
};

static const struct BSEC{    // 前例定义的B段初始化表结构体
    void * b_s;              // 未初始化数据段的起始地址成员
    void * b_e;              // 未初始化数据段的结束地址成员
};

static void clearblock(void *b_top, void *b_end);
static void copyblock (void *d_top, void *d_end, void *r_top);

#ifdef _ _cplusplus
extern “c”                   // 连接C
#endif
void _INITSCT(void)          // 段初始化例程
{
    const struct BSEC *btbl; // B段初始化表结构体
    const struct DSEC *dtbl; // D段初始化表结构体

                             // 对未初始化的数据段进行初始化
    for( btbl = _ _sectop (“C$BSEC”);
            btbl < (struct BSEC *)_ _secend (“C$BSEC”); btbl++)
        clearblock( btbl->b_s, btbl->b_e );

                             // 将初始化数据从ROM复制到RAM
    for( dtbl = _ _sectop (“C$DSEC”);
            dtbl < (struct DSEC *)_ _secend (“C$DSEC”); dtbl++)
        copyblock( dtbl->rom_s, dtbl->rom_e, dtbl->ram_s );
}

static void clearblock(void *b_top, void *b_end)
{                            // 用零对未初化的数据段进行初始化
    char *p;
    for( p=b_top; p<(char *)b_end; p++)
        *p = 0;
}
static void copyblock(void *d_top, void *d_end, void *r_top)
{                            // 将初始化数据从ROM复制到RAM
    char *p, *q;
    for( p=r_top, q=d_top; q<(char *)d_end; p++, q++)
        *p = *q;
}
```

（4）C++全局类目标的初期设定（_CALL_INIT）

_CALL_INIT 函数调用由 C++声明为全局类目标的构造函数。由库头文件<_h_c_lib.h>提供本_CALL_INIT 函数，用以下例子进行运行说明：

例：

```
extern “C”  void _CALL_INIT(void);

typedef void (**FPP)(void);   // 函数指针型

extern “C”  void _CALL_INIT(void)
{                             // 全局类目标的初始化例程
    FPP top = (FPP)_ _sectop("C$INIT");
    FPP end = (FPP)_ _secend("C$INIT");

    while (top < end)
        (*top++)();           // 调用构造函数
}
```

（5）C/C++库函数的初始设定（_INITLIB）

在此说明 C/C++库函数的初始设定方法。

为了按照实际使用的功能进行最低限度的初始设定，请参考以下指针：

- 在使用<stdio.h>、<ios>、<streambuf>、<istream>、<ostream>各函数和 assert 宏时，需要进行标准输入/输出的初始设定（_INIT_IOLIB）。
- 在建立的低级接口例程中需要初始设定时，必须根据低级接口例程的规格进行初始设定（_INIT_LOWLEVEL）。
- 在使用 rand 函数、strtok 函数时，需要进行标准输入/输出以外的初始设定（_INIT_OTHERLIB）。

进行库初始设定的程序例子如下所示，FILE 型数据如图 9.7。

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;   // 指定堆区确保容量的最小单位。但是，本行
// 省略时，在advanced（无ptr16选项）,maximum（无ptr16选项）时为1032，
// 在normal,middle,advanced（有ptr16选项）,maximum（有ptr16选项）,300时为1028
const int _nfiles = IOSTREAM;    // 指定输入/输出文件的个数(省略时:20)
struct _iobuf _iob[IOSTREAM];
unsigned char sml_buf[IOSTREAM];
extern char *_s1ptr;

#ifdef _ _cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
   _INIT_LOWLEVEL();             // 初始设定低级接口例程
   _INIT_IOLIB();                // 初始设定输入/输出库
   _INIT_OTHERLIB();             // 初始设定rand函数和strtok函数
}
void _INIT_LOWLEVEL (void)
{                                // 必须进行低级库所必需的初始设定
}
void _INIT_IOLIB(void)
{
FILE *fp;
   for( fp = _iob; fp < _iob + _nfiles; fp++ ) // FILE型数据的初始设定
   {
      fp->_bufptr = NULL;
      fp->_bufcnt = 0;
      fp->_buflen = 0;
      fp->_bufbase = NULL;
      fp->_ioflag1 = 0;
      fp->_ioflag2 = 0;
      fp->_iofd = 0;
   }
      if(freopen("stdin*1","r",stdin)== NULL) // 打开标准输入文件
         stdin->_ioflag1 = 0xff;             // 禁止打开失败时的文件存取
      stdin->_ioflag1 |= _IOUNBUF;           // 设定为无数据缓冲*2
      if(freopen("stdout*1","w",stdout)== NULL)// 打开标准输出文件
         stdout->_ioflag1 = 0xff;            // 禁止打开失败时的文件存取
      stdout->_ioflag1 |= _IOUNBUF;          // 设定为无数据缓冲*2
      if(freopen("stderr*1","w",stderr)== NULL)// 打开标准错误文件
         stderr->_ioflag1 = 0xff;            // 禁止打开失败时的文件存取
      stderr->_ioflag1 |= _IOUNBUF;          // 设定为无数据缓冲*2
}
void _INIT_OTHERLIB(void)
{
   srand(1);                              // 使用rand函数时的初始设定
   _s1ptr=NULL;                           // 使用strtok函数时的初始设定
}
#ifdef _ _cplusplus
}
#endif
```

【注】*1　指定标准输入/输出文件的文件名。此文件名由低级接口例程“open”使用。
　　　*2　在控制台等交互设备时，置不进行缓冲的标志。

```
                    /* 在C语言中的文件型数据声明 */

struct _iobuf{
             unsigned char *_bufptr;  /* 指向缓冲器的指针  */
             long           _bufcnt;  /* 缓冲计数器        */
             unsigned char *_bufbase; /* 缓冲基址指针      */
             long            _buflen; /* 缓冲器长度        */
             char           _ioflag1; /* i/o标志           */
             char           _ioflag2; /* i/o标志           */
             char           _iofd;    /* i/o标志           */
}iob[_nfiles];
```

图 9.7　FILE 型数据

（6）文件的关闭（_CLOSEALL）

通常的文件输出预先将数据存放到存储区的缓冲区，当缓冲区满后再写到实际的外部存储装置。因此，如果不关闭文件，就有可能无法将文件的输出内容写到外部存储装置。

在嵌入式程序的情况下，通常程序没有结束。但是，在因程序错误等导致 main 函数结束时，必须将打开的文件全部关闭。

本处理在 main 函数结束时关闭打开的文件。

关闭文件的程序例子如下所示：

```
#include <stdio.h>

#ifdef _ _cplusplus
extern "C"
#endif
void _CLOSEALL(void)
{
int i;

    for( i=0; i < _nfiles; i++ )

    // 检查文件是否已打开
        if( _iob[i]._ioflag1 & (_IOREAD | _IOWRITE | _IORW ) )
            fclose( &_iob[i] ); // 关闭文件
}
```

（7）低级接口例程

在 C/C++程序中使用标准输入/输出和存储器管理库时，必须建立低级接口例程。C 库函数使用的低级接口例程一览表如**表 9.3** 所示。

表 9.3　低级接口例程一览表

名称		功能
1	open	打开文件
2	close	关闭文件
3	read	读文件
4	write	写文件
5	lseek	设定文件的读/写位置
6	sbrk	确保存储区
7	error_addr※	取得 errno 地址
8	wait_sem※	确保信标
9	signal_sem※	释放信标

【注】※使用可重入库时必需。

需要在启动程序时进行低级接口例程的初始化，其初始化必须在“9.2.2（5） C/C++库函数的初始设定（_INITLIB）”的“_INIT_LOWLEVEL”函数中进行。

以下，在说明低级输入/输出的基本思想后说明各接口例程的规格。

注意　函数名 open、close、read、write、lseek、sbrk 是低级接口例程的保留字，不能在用户程序中使用。

（a）输入/输出的基本想法

标准输入/输出库通过 FILE 型数据进行文件管理，而低级接口例程通过分配与实际文件一一对应的正整数进行文件管理，此整数称为文件序号。

open 例程对被提供的文件名分配文件序号。为了能用此序号进行文件的输入/输出，open 例程需要设定以下信息：

- 文件设备的种类（控制台、打印机和磁盘文件等）。

（对于控制台和打印机等特殊设备，需要由系统决定特别的文件名，并且由 open 例程判断）

- 在进行文件缓冲时，缓冲区的位置和容量等信息。
- 在磁盘文件时，从文件的起始位置到下次读/写位置为止的字节偏移量。

以 open 例程设定的信息为基础， 进行输入/输出（read、write 例程）和读/写位置的设定（lseek 例程）。

在进行输出文件缓冲时，close 例程必须将缓冲区的内容写到实际的文件，并且使 open 例程设定的数据区能被重新使用。

（b）低级接口例程的详细内容

本项说明建立低级接口例程的详细内容。对于各例程，说明例程调用时的接口、例程的运行以及使用注意事项。

用以下格式表示各例程的接口。另外，必须以原型声明低级接口例程，而且在 C++程序中声明时必须附加“extern "C" ”。

简单说明

(例程名)

说　明	（例程的功能概要）	
返回值	正常：	（正常结束时的返回值）
	异常：	（发生错误时的返回值）
参　数	（名称）	（含义）
	（接口中表示的参数名）	（作为参数传递的值的含义）

*int open(char *name, int mode, int flg)*

说　明

为了操作与第 1 参数的文件名对应的文件而进行准备。
在 open 例程中，为了以后的读/写，必须决定文件的种类（控制台、打印机和磁盘文件等）。每当进行读/写时必需使用 open 例程返回的文件序号存取文件的种类。
第 2 参数 mode 指定文件打开时的处理。此数据的各位的含义如下所示：

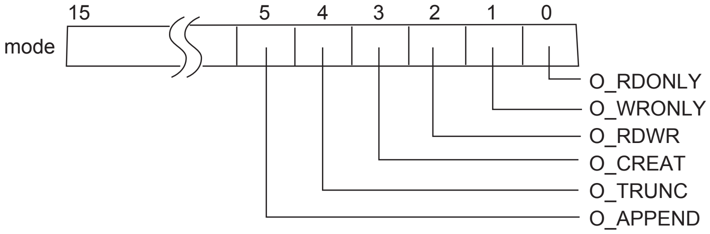


表 9.4　open 例程 mode 位的说明

Mode 位	说明
O_RDONLY（位 0）	此位为 1 时，用只读模式打开文件
O_WRONLY（位 1）	此位为 1 时，用只写模式打开文件
O_RDWR（位 2）	此位为 1 时，用读/写模式打开文件
O_CREAT（位 3）	此位为 1 时，如果文件名表示的文件不存在，就新建文件
O_TRUNC（位 4）	此位为 1 时，如果文件名表示的文件存在，就删除文件内容，并且将文件的大小更新为 0
O_APPEND（位 5）	设定下次读写时的文件中的位置 此位为 0 时：设定到文件的起始位置 此位为 1 时：设定到文件的结束位置

如果 mode 指定的文件处理和实际文件的特性发生矛盾，必须进行错误处理。如果能正常打开文件，就返回可被 read、write、lseek、close 例程使用的文件序号（正整数）。需要用低级接口例程进行文件序号与实际文件的对应管理。在打开失败时，返回-1。

返回值		
返回值	正常：	被正常打开的文件的文件序号
	异常：	-1
参　数	name	文件的文件名
	mode	指定文件打开时的处理
	flg	指定文件打开时的处理（总是为 0777）

文件的关闭

int close(int fileno)

说　明	作为参数，传递由 open 例程取得的文件序号。 为了能重新使用，必须释放 open 例程设定的文件管理信息区。另外，如果在低级接口例程中进行输出文件缓冲，就必须将缓冲区的内容写到实际文件。 在能正常关闭文件时，返回 0；否则，返回-1。	
返回值	正常：	0
	异常：	-1
参　数	fileno	关闭的文件序号

数据的读

*int read(int fileno, char *buf, unsigned int count)*

说　明	从第 1 参数（fileno）指定的文件将数据读到第 2 参数（buf）指向的区域。 由第 3 参数（count）指定读数据的字节数。 如果文件结束，就只能读取少于 count 表示的字节数。 文件的读/写位置只向前移动读到的字节数。 能正常读取时，返回实际读到的字节数；否则，返回-1。	
返回值	正常：	实际读取的字节数
	异常：	-1
参　数	fileno	读对象的文件序号
	buf	保存读数据的区域
	count	读数据的字节数

数据的写

*int write(int fileno, char *buf, unsigned int count)*

说　明	从第 2 参数（buf）指向的区域将数据写到第 1 参数（fileno）指定的文件。 第 3 参数（count）指定写数据的字节数。 在要写文件的设备（磁盘等）没有空间时，只能写少于 count 表示的字节数。建议在实际写的字节数多次连续为 0 字节时可认为磁盘已满，返回错误（-1）。 文件的读/写位置只向前移动被写的字节数。 能正常写时，返回实际写的字节数；否则，返回-1。	
返回值	正常：	实际写的字节数
	异常：	-1
参　数	fileno	写对象的文件序号
	buf	写数据的数据区
	count	写数据的字节数

文件中的位置设定

long lseek(int fileno, long offset, int base)

说　明

以字节为单位设定文件读/写时的文件中的位置。

必须根据第 3 参数（base），用以下的方法计算并设定新文件中的位置。

[1] base 为 0 时	从文件起始位置设定到 offset 字节的位置。
[2] base 为 1 时	设定到当前位置加上 offset 字节的位置。
[3] base 为 2 时	设定到文件大小加上 offset 字节的位置。

如果文件为控制台或打印机等交互设备、新的偏移量为负或者在[1]、[2]时超过文件的大小，就产生错误。

能正确设定文件位置时，返回新的读/写位置（从文件起始位置的偏移量）；否则，返回-1。

返回值	正常：	新的文件读/写位置（从文件起始位置的偏移量）（字节单位）
	异常：	-1
参　数	fileno	对象的文件序号
	offset	读/写位置的偏移量（字节单位）
	base	偏移的起点

存储区的分配

*char *sbrk(size_t size)*

说　明	作为参数，传递分配存储区的容量。 在连续调用 sbrk 例程时，必须从低位地址依次分配连续的区域。如果分配的存储区不足，就产生错误。 能正常分配时，返回分配区的起始地址；否则，返回“(char *) -1”。	
返回值	正常：	分配区的起始地址
	异常：	(char *) - 1
参　数	size	分配的数据长度

***errno** 地址的取得*

*int *errno_addr(void)*

说　明	返回当前任务的错误号的地址。 在使用由标准库生成工具建立（指定 reent 选项时）的标准库时，需要本函数。
返回值	当前任务的错误号的地址

信标的确保

int wait_sem (int semnum)

说　明	确保 semnum 指定的信标。 如果能确保，就返回 1；否则，返回 0。 在使用由标准库生成工具建立（指定 reent 选项时）的标准库时，需要本函数。	
返回值	正常： 异常：	1 0
参　数	semnum	信标 ID

信标的释放

int signal_sem (int semnum)

说　明	释放 semnum 指定的信标。 如果能释放，就返回 1；否则，返回 0。 在使用由标准库生成工具建立（指定 reent 选项时）的标准库时，需要本函数。	
返回值	正常： 异常：	1 0
参　数	semnum	信标 ID

（c）低级接口例程的建立例子

```
/**************************************************************************/
/*                              lowsrc.c:                                 */
/*－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－*/
/* H8S, H8/300 系列　仿真/调试程序　接口例程                                */
/*      －只支持标准输入/输出(stdin,stdout,stderr)－                         */
/**************************************************************************/
#include <string.h>

/* 文件序号 */

#define STDIN  0                      /* 标准输入   (控制台)        */
#define STDOUT 1                      /* 标准输出   (控制台)        */
#define STDERR 2                      /* 标准错误输出(控制台)       */

#define FLMIN  0                      /* 最小的文件序号             */
#define FLMAX  3                      /* 文件数的最大值             */

/* 文件的标志 */

#define O_RDONLY 0x0001               /* 只读                       */
#define O_WRONLY 0x0002               /* 只写                       */
#define O_RDWR   0x0004               /* 读写                       */

/* 特殊字符码 */

#define CR 0x0d                       /* 回车                       */
#define LF 0x0a                       /* 换行                       */

/* 由 sbrk 管理的区域容量 */

#if _ _DATA_ADDRESS_SIZE_ _ = = 4
#define HEAPSIZE 2064
#else
#define HEAPSIZE 2056
#endif

/**************************************************************************/
/* 参照函数的声明:                                                          */
/* 参照通过仿真/调试程序将字符输入/输出到控制台的汇编程序                        */
/**************************************************************************/

extern  void charput(char);           /* 一个字符的输出处理             */
extern  char charget(void);           /* 一个字符的输入处理             */
```

```
/**************************************************************************/
/* 静态变量的定义：                                                       */
/* 定义低级接口例程使用的静态变量                                         */
/**************************************************************************/

char flmod[FLMAX];                        /* 打开文件的模式设定位置         */

static  union {
        short  dummy ;                    /* 虚设 2 个字节边界              */
        char heap[HEAPSIZE];              /* 声明由 sbrk 管理的区域         */
 }heap_area ;

static  char   *brk=(char *)&heap_area;   /* sbrk 分配的区域的结束地址      */

/**************************************************************************/
/*   open:打开文件                                                        */
/*        返回值：文件序号（成功）                                        */
/*                -1    （失败）                                          */
/**************************************************************************/
extern  open(char *name,                  /* 文件名                         */
     int  mode,                           /* 文件模式                        */
     int  flg)                            /* 未使用                         */
{
       /* 根据文件名检查模式，返回文件序号       */
       if(strcmp(name,"stdin")==0){       /* 标准输入文件       */
             if((mode&O_RDONLY)==0)
                      return -1;
             flmod[STDIN]=mode;
             return STDIN;
       }

       else if(strcmp(name,"stdout")==0){ /* 标准输出文件       */
             if((mode&O_WRONLY)==0)
                      return -1;
             flmod[STDOUT]=mode;
             return STDOUT;
       }

       else if(strcmp(name,"stderr")==0){ /* 标准错误输出文件   */
             if((mode&O_WRONLY)==0)
                      return -1;
             flmod[STDERR]=mode;
             return STDERR;
       }

       else
             return -1;                   /* 错误               */
}
```

```
/****************************************************************************/
/*  close:关闭文件                                                          */
/*        返回值：0        （成功）                                         */
/*              -1        （失败）                                          */
/****************************************************************************/
extern  close(int fileno)                    /* 文件序号                    */
{
       if(fileno<FLMIN || FLMAX<=fileno)     /* 检查文件序号的范围          */
             return -1;

       flmod[fileno]=0;                      /*文件模式的复位               */
       return 0;
}

/****************************************************************************/
/*  read:读数据                                                             */
/*        返回值：实际读到的字符数        （成功）                          */
/*                    -1                （失败）                            */
/****************************************************************************/
extern  read(int  fileno,                    /* 文件序号                    */
     char *buf,                              /* 传送目标的缓冲区地址        */
     int  count)                             /* 读的字符数                  */
{
       int i;

       /* 根据文件序号检查模式，每输入 1 个字符后保存到缓冲区   */

       if(flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   *buf=charget();
                   if(*buf==CR)              /* 替换换行字符                */
                         *buf=LF;
                   buf++;
             }
             return count;
       }
       else

             return -1;
}
```

```
/************************************************************************ */
/*  write:写数据                                                            */
/*        返回值：实际写的字符数        （成功）                              */
/*                   -1                 （失败）                              */
/*************************************************************************/
extern  write(int  fileno,                            /* 文件序号            */
      char *buf,                                      /* 传送源的缓冲区地址   */
      int  count)                                     /* 写的字符数          */
{
       int  i;
       char c;

       /* 根据文件序号检查模式，输出每个字符                  */

       if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   c=*buf++;
                   charput(c);
             }
             return count;
       }
       else
             return -1;
}
/************************************************************************ */
/*  lseek:设定文件的读/写位置                                                */
/*        返回值：读/写位置（从文件起始位置开始的偏移量）   （成功）           */
/*                      -1                             （失败）            */
/*     (在控制台输入/输出时，不支持 lseek)                                    */
/*************************************************************************/
extern  long lseek(int  fileno,                       /* 文件序号            */
           long offset,                               /* 读/写位置           */
           int  base)                                 /* 偏移的起点          */
{
       return -1L;
}
/*************************************************************************/
/*  sbrk:写数据                                                             */
/*        返回值：分配的区域起始地址             （成功）                     */
/*                   -1                        （失败）                     */
/*************************************************************************/
extern  char  *sbrk(size_t size)                      /* 分配的区域容量       */
{
      char  *p ;
      if (brk+size>heap_area.heap+HEAPSIZE)           /* 检查空区域          */
            return (char *)-1 ;
      p=brk ;                                         /* 分配区              */
      brk += size ;                                   /* 更新结束地址         */
      return p ;
}
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                              lowlvl.nor                               |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   H8S, H8/300 系列　仿真/调试程序　接口例程                            |
;                          －输入/输出 1 个字符－                        |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX,H8S/2600,H8S/2000,H8/300H 正常模式                               |
; (cpu=H8SXN, 2600n, 2000n, 300hn)                                      |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .CPU              2600N             ; 或者 H8SXN,2000N,300HN
        .EXPORT           _charput
        .EXPORT           _charget
SIM_IO: .EQU              H'00FE            ; 指定 TRAP_ADDRESS

        .SECTION          P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   _charput: 输出 1 个字符                                              |
;             c 程序接口：   charput(char)                               |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
        MOV.B             R0L,@IO_BUF       ; 将参数设定到缓冲区
        MOV.W             #H'0102,R0        ; 设定参数和功能码
        MOV.W             #LWORD IO_BUF,R1
        MOV.W             R1,@PARM          ; 设定输入/输出缓冲区地址
        MOV.W             #LWORD PARM,R1    ; 设定参数块地址
        JSR               @SIM_IO
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   _charget: 输入 1 个字符                                              |
;             c 程序接口：   char charget(void)                          |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
        MOV.W             #H'0101,R0        ; 设定参数和功能码
        MOV.W             #LWORD IO_BUF,R1
        MOV.W             R1,@PARM          ; 设定输入/输出缓冲区地址
        MOV.W             #LWORD PARM,R1    ; 设定参数块地址
        JSR               @SIM_IO
        MOV.B             @IO_BUF,R0L
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 定义用于输入/输出的缓冲区                                              |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .SECTION          B,DATA,ALIGN=2

PARM:   .RES.W            1                 ; 参数块区域
IO_BUF: .RES.B            1                 ; 输入/输出缓冲区
        .END
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                                lowlvl.adv                                         |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   H8S, H8/300 系列   仿真/调试程序   接口例程                                      |
;                         －输入/输出 1 个字符－                                     |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX、H8S/2600、H8S/2000、H8/300H 高级模式(20 位或者 24 位地址)                    |
; (cpu=H8SXA:20|24, 2600a:20|24, 2000a:20|24, 300ha)                                |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .CPU                2600A               ; 或者 H8SXA,2000A,300HA
        .EXPORT             _charput
        .EXPORT             _charget
SIM_IO: .EQU                H'01FE              ; 指定 TRAP_ADDRESS

        .SECTION            P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   _charput: 输出 1 个字符                                                          |
;             c 程序接口：   charput(char)                                          |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
        MOV.B               R0L,@IO_BUF         ; 将参数设定到缓冲区
        MOV.W               #H'0112,R0          ; 设定参数和功能码
        MOV.L               #IO_BUF,ER1
        MOV.L               ER1,@PARM           ; 设定输入/输出缓冲区地址
        MOV.L               #PARM,ER1           ; 设定参数块地址
        JSR                 @SIM_IO
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;   _charget: 输入 1 个字符                                                          |
;             c 程序接口：   char charget(void)                                     |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
        MOV.W               #H'0111,R0          ; 设定参数和功能码
        MOV.L               #IO_BUF,ER1
        MOV.L               ER1,@PARM           ; 设定输入/输出缓冲区地址
        MOV.L               #PARM,ER1           ; 设定参数块地址
        JSR                 @SIM_IO
        MOV.B               @IO_BUF,R0L
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 定义用于输入/输出的缓冲区                                                         |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .SECTION            B,DATA,ALIGN=2
PARM:   .RES.L              1                   ; 参数块区域
IO_BUF: .RES.B              1                   ; 输入/输出缓冲区
        .END
```

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;                                lowlvl.mid                                     |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  H8S, H8/300 系列　仿真/调试程序　接口例程                                    |
;                         －输入/输出 1 个字符－                                |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; H8SX 中级模式、H8SX 高级模式/最高模式(16 位数据地址)                          |
; (cpu=H8SXM, cpu=H8SXA ptr16, cpu=H8SXX ptr16)                                 |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .CPU                 H8SXM
        .EXPORT              _charput
        .EXPORT              _charget
SIM_IO: .EQU                 H'01FE            ; 指定 TRAP_ADDRESS

        .SECTION             P,CODE,ALIGN=2
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charput: 输出 1 个字符                                                      |
;             c 程序接口：   charput(char)                                      |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charput:
        MOV.B                R0L,@IO_BUF       ; 将参数设定到缓冲区
        MOV.W                #H'0102,R0        ; 设定参数和功能码
        MOV.W                #LWORD IO_BUF,R1
        MOV.W                R1,@PARM          ; 设定输入/输出缓冲区地址
        MOV.W                #LWORD PARM,R1    ; 设定参数块地址
        JSR                  @SIM_IO
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
;  _charget: 输入 1 个字符                                                      |
;             c 程序接口：   char charget(void)                                 |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

_charget:
        MOV.W                #H'0101,R0        ; 设定参数和功能码
        MOV.W                #LWORD IO_BUF,R1
        MOV.W                R1,@PARM          ; 设定输入/输出缓冲区地址
        MOV.W                #LWORD PARM,R1    ; 设定参数块地址
        JSR                  @SIM_IO
        MOV.B                @IO_BUF,R0L
        RTS

;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 定义用于输入/输出的缓冲区                                                     |
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－
        .SECTION             B,DATA,ALIGN=2

PARM:   .RES.W               1                 ; 参数块区域
IO_BUF: .RES.B               1                 ; 输入/输出缓冲区
        .END
```

```
;-------------------------------------------
;                               lowlvl.max                              |
;-------------------------------------------
;  H8S, H8/300 系列  仿真/调试程序  接口例程                            |
;                                 －输入/输出 1 个字符－                |
;-------------------------------------------
; H8SX 最高模式、H8SX、H8S/2600、H8S/2000 高级模式(28 位或者 32 位地址)  |
; (cpu=H8SXX, H8SXA:28|32, 2600a:28|32, 2000a:28|32)                    |
;-------------------------------------------
        .CPU                H8SXX
        .EXPORT             _charput
        .EXPORT             _charget

SIM_IO: .EQU                H'01FE                ; 指定 TRAP_ADDRESS

        .SECTION            P,CODE,ALIGN=2
;-------------------------------------------
;  _charput: 输出 1 个字符                                              |
;            c 程序接口：   charput(char)                               |
;-------------------------------------------

_charput:
        MOV.B               R0L,@IO_BUF           ; 将参数设定到缓冲区
        MOV.W               #H'0122,R0            ; 设定参数和功能码
        MOV.L               #IO_BUF,ER1
        MOV.L               ER1,@PARM             ; 设定输入/输出缓冲区地址
        MOV.L               #PARM,ER1             ; 设定参数块地址
        JSR                 @SIM_IO
        RTS

;-------------------------------------------
;  _charget: 输入 1 个字符                                              |
;            c 程序接口：   char charget(void)                          |
;-------------------------------------------

_charget:
        MOV.W               #H'0121,R0            ; 设定参数和功能码
        MOV.L               #IO_BUF,ER1
        MOV.L               ER1,@PARM             ; 设定输入/输出缓冲区地址
        MOV.L               #PARM,ER1             ; 设定参数块地址
        JSR                 @SIM_IO
        MOV.B               @IO_BUF,R0L
        RTS

;-------------------------------------------
; 定义用于输入/输出的缓冲区                                             |
;-------------------------------------------
        .SECTION            B,DATA,ALIGN=2
PARM:   .RES.L              1                     ; 参数块区
IO_BUF: .RES.B              1                     ; 输入/输出缓冲区
        .END
```

```
;-------------------------------------------------------------------------
;                               lowlvl.reg                               |
;-------------------------------------------------------------------------
;  H8S, H8/300 系列  仿真/调试程序  接口例程                               |
;                             －输入/输出 1 个字符－                        |
;-------------------------------------------------------------------------
;                          H8/300 (cpu=300)                              |
;-------------------------------------------------------------------------
        .CPU                300
        .EXPORT             _charput
        .EXPORT             _charget

SIM_IO: .EQU                H'00FE               ; 指定 TRAP_ADDRESS

        .SECTION            P,CODE,ALIGN=2
;-------------------------------------------------------------------------
;  _charput: 输出 1 个字符                                                |
;            c 程序接口：  charput(char)                                  |
;-------------------------------------------------------------------------
_charput:
        MOV.B               R0L,@IO_BUF          ; 将参数设定到缓冲区
        MOV.W               #H'0102,R0           ; 设定参数和功能码
        MOV.W               #IO_BUF,R1
        MOV.W               R1,@PARM             ; 设定输入/输出缓冲区地址
        MOV.W               #PARM,R1             ; 设定参数块地址
        JSR                 @SIM_IO
        RTS

;-------------------------------------------------------------------------
;  _charget: 输入 1 个字符                                                |
;            c 程序接口：  char charget(void)                             |
;-------------------------------------------------------------------------
_charget:
        MOV.W               #H'0101,R0           ; 设定参数和功能码
        MOV.W               #IO_BUF,R1
        MOV.W               R1,@PARM             ; 设定输入/输出缓冲区地址
        MOV.W               #PARM,R1             ; 设定参数块地址
        JSR                 @SIM_IO
        MOV.B               @IO_BUF,R0L
        RTS

;-------------------------------------------------------------------------
; 定义用于输入/输出的缓冲区                                                |
;-------------------------------------------------------------------------
        .SECTION            B,DATA,ALIGN=2

PARM:   .RES.W              1                    ; 参数块区
IO_BUF: .RES.B              1                    ; 输入/输出缓冲区
        .END
```

（d）用于可重入库的低级接口例程的例子

用于可重入库的低级接口例程的例子如下所示。在使用由标准库生成工具建立（指定 reent 选项时）的库时需要。

如果 wait_sem 函数或者 signal_sem 函数返回 NG，就给 errno 设定以下内容，从库函数返回。

wait_sem	EMALRESM	malloc 的信标资源的确保失败
	ETOKRESM	strtok 的信标资源的确保失败
	EIOBRESM	_iob 的信标资源的确保失败
signal_sem	EMALFRSM	malloc 的信标资源的释放失败
	ETOKFRSM	strtok 的信标资源的释放失败
	EIOBFRSM	_iob 的信标资源的释放失败

关于中断，在确保信标后产生更高优先级的中断，如果再次进行信标的确保，就会发生死锁，所以不能用中断嵌套进行如共享资源的处理。

```
#define MALLOC_SEM              1     /* malloc 的信标 No.          */
#define STRTOK_SEM              2     /* strtok 的信标 No.          */
#define FILE_TBL_SEM            3     /* _iob 的信标 No.            */
#define SEMSIZE                 4
#define TRUE                    1
#define FALSE                   0
#define OK                      1
#define NG                      0

extern int *errno_addr(void);
extern int wait_sem(int);
extern int signal_sem(int);

int sem_errno;
int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];

/*****************************************************************************/
/*                           errno_addr: 取得 errno 地址                     */
/*                              返回值：errno 地址                           */
/*****************************************************************************/
int *errno_addr(void)
{
    /* 必须返回当前任务的 errno 地址 */
    return (&sem_errno);
}

/*****************************************************************************/
/*                           wait_sem:确保被指定的信标                       */
/*                              返回值：      OK(=1) （成功）                */
/*                                            NG(=0) （失败）                */
/*****************************************************************************/
int wait_sem(int semnum)      /* 信标 ID        */
{
    if((0 <= semnum) && (semnum < SEMSIZE)) {
         if(semaphore[semnum] == FALSE) {
             semaphore[semnum] = TRUE;
             return(OK);
         }
    }
    return(NG);
}
```

```
/*****************************************************************************/
/*                           signal_sem: 释放被指定的信标                        */
/*                                返回值:      OK(=1) （成功）                   */
/*                                             NG(=0) （失败）                   */
/*****************************************************************************/
int signal_sem(int semnum)    /* 信标 ID         */
{
    if(!force_fail_signal_sem) {
         if((0 <= semnum) && (semnum < SEMSIZE)) {
             if( semaphore[semnum] == TRUE ) {
                semaphore[semnum] = FALSE;
                return(OK);
             }
         }
    }
    return(NG);
}
```

（8）结束处理例程

（a）说明进行结束处理注册和执行（atexit）结束处理注册的库 atexit 函数的建立方法

atexit 函数将作为参数传递的函数地址注册到结束处理表中。如果注册的函数个数超过极限值（在此，能注册的个数为 32 个）或者重复注册相同函数时，就返回 NULL 的返回值。否则，就返回 NULL 以外的值（此时为注册的函数地址）。

程序例子如下所示：
例：

```
#include <stdlib.h>
typedef void *atexit_t ;

int _atexit_count=0 ;

atexit_t (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
atexit_t atexit(atexit_t (*f)(void))
{
    int i;

 for(i=0; i<_atexit_count ; i++)          // 检查是否已经注册
      if(_atexit_buf[i]==f)
         return NULL ;

 if (_atexit_count==32)                  // 检查注册数的极限值
    return NULL ;
 else{
    _atexit_buf[_atexit_count++]=f;      // 注册函数的地址
    return f;
 }
}
```

（b）程序结束（exit）例程的建立例子

说明进行程序结束处理的库 exit 函数的建立方法。由于程序结束处理因用户系统而不同，所以必须参考以下的程序例子，建立符合用户系统规格的结束处理。

exit 函数根据作为参数传递的程序结束码进行程序的结束处理，返回到程序启动时的环境。在此，将结束码设定为外部变量，然后通过在即将调用 main 函数前返回到用 setjmp 函数保存的环境来实现。为了返回到程序执行前的环境，必须在建立以下的函数“callmain”后调用函数“callmain”，以取代从初始设定函数“PowerON_Reset” 调用函数“main”。

程序例子如下所示：

```
#include <setjmp.h>
#include <stddef.h>

typedef void * atexit_t ;
extern int _atexit_count ;
extern atexit_t (*_atexit_buf[32])(void) ;
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
int main(void);
extern jmp_buf _init_env ;
int _exit_code ;

#ifdef __cplusplus
extern "C"
#endif
void exit(int code)
{
    int i;
    _exit_code=code ;                  // 将返回码设定到_exit_code
    for(i=_atexit_count-1; i>=0; i--)  // 依次执行由 atexit 函数注册的函数
        (*_atexit_buf[i])();
    _CLOSEALL();                       // 关闭所有打开的函数
    longjmp(_init_env, 1) ;            // 返回到由 setjmp 保存的环境
}
#ifdef __cplusplus
extern "C"
#endif
void callmain(void)
{
     //  用 setjmp 保存当前的环境，然后调用 main 函数
     if(!setjmp(_init_env))
         _exit_code=main();            // 在从 exit 函数返回时，结束处理
}
```

（c）异常结束（abort）例程的建立例子

在异常结束时，必须根据使用的用户系统的规格进行程序异常结束的处理。
使用 C++程序时，在以下的情况下还调用 abort 函数。

- 异常处理无法正确运行时
- 调用纯虚拟函数本身时
- `dynamic_cast` 失败时
- `typeid` 失败时
- 类数组的 `delete` 时无法取得信息时
- 在注册类目标的析构函数调用信息时发生矛盾时

在将信息输出到标准输出设备后关闭文件，然后进入无限循环，以等待复位的程序例子如下所示：

例：

```
#include <stdio.h>

#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef __cplusplus
extern "C"
#endif
void abort(void)
{
        printf(“program aborted !!\n”); // 输出信息
        _CLOSEALL();                    // 关闭文件
        while(1)    ;                   // 进入无限循环
}
```

9.3　C/C++程序和汇编程序的连接

通过支持#pragma、关键字等扩展功能和内部函数，本编译程序能用 C 语言和 C++语言记述嵌入式程序所需的全部功能。

但是，在硬件的时序要求和存储器容量的限制等严格的性能要求时，需要用汇编语言记述并与 C/C++程序连接。

在此，说明有关在连接 C/C++程序和汇编程序时该注意的事项。

- 外部名的相互参照方法
- 函数调用的接口

9.3.1　外部名的相互参照方法

C/C++程序中声明的外部名能在 C/C++程序和汇编程序之间相互参照或者更新。编译程序将以下内容作为外部名处理：

- 不为 `static` 存储类（C/C++程序）的全局变量
- 用 `extern` 存储类声明的变量名（C/C++程序）
- 没有指定为 `static` 存储类的函数名（C 程序）
- 没有指定为 `static` 存储类的非成员非直接插入的函数名（C++程序）
- 非直接插入的成员函数名（C++程序）
- 静态数据成员名（C++程序）

（1）C/C++程序中参照汇编程序外部名的方法

在汇编程序中，用“.EXPORT”控制指令声明外部定义的符号名（在前头附加下划线"_"）。

在 C/C++程序中，用“extern”声明符号名（前头无下划线"_"）。

汇编程序（定义侧）

```
     .EXPORT    _a,_b
     .SECTION
D,DATA,ALIGN=2
_a: .DATA.W   1
_b: .DATA.W   1
     .END
```

C/C++程序（参照侧）

```
extern int a,b;
f()
{
    a+=b;
}
```

（2）从汇编程序参照 C/C++程序外部（变量和 C 函数）名的方法

在 C/C++程序中，外部定义变量名（前头无下划线"_"）。

在汇编程序中，用“.IMPORT”控制指令声明外部参照的外部名（在前头附加下划线"_"）。

C/C++程序（定义侧）

```
char a,b;
```

汇编程序（参照侧）

```
   .IMPORT    _a,_b
   .SECTION P,CODE,ALIGN=2
   MOV.B      @_a,R5L
   MOV.B      R5L,@_b
   RTS
   .END
```

（3）从汇编程序参照 C++程序的外部（函数）名的方法

通过用“extern "C"”声明在汇编程序中参照的函数，能用和（2）相同的规则参照此函数 。但是，用“extern "C"”声明的函数不能多重定义。

C++程序（定义侧）

```
extern "C"
int f(int a)
{
    …
}
```

汇编程序（参照侧）

```
.IMPORT    _f
.SECTION P,CODE,ALIGN=2
    :
JSR        @_f
    :
.END
```

9.3.2　　函数调用的接口

说明在 C/C++程序和汇编程序之间互相调用函数时汇编程序侧应该遵守以下 4 个规则：

- 有关堆栈指针的规则
- 有关分配和释放堆栈帧的规则
- 有关寄存器的规则
- 有关设定和参照参数、返回值的规则

（1）有关堆栈指针的规则

不能将有效数据保存到比堆栈指针指向的地址低（0 地址的方向）的堆栈区，在中断处理中有可能被破坏。

（2）有关分配和释放堆栈帧的规则

在函数调用时（在刚执行 JSR 或者 BSR 指令后），堆栈指针指向返回地址的区域。必须在调用函数侧分配和设定比此区域高的地址的数据。

在函数返回时，必须在被调用函数侧进行返回地址区域的释放，一般使用 RTS 指令进行。在调用函数侧释放比此区域高的地址的区域（返回值地址和参数区）。

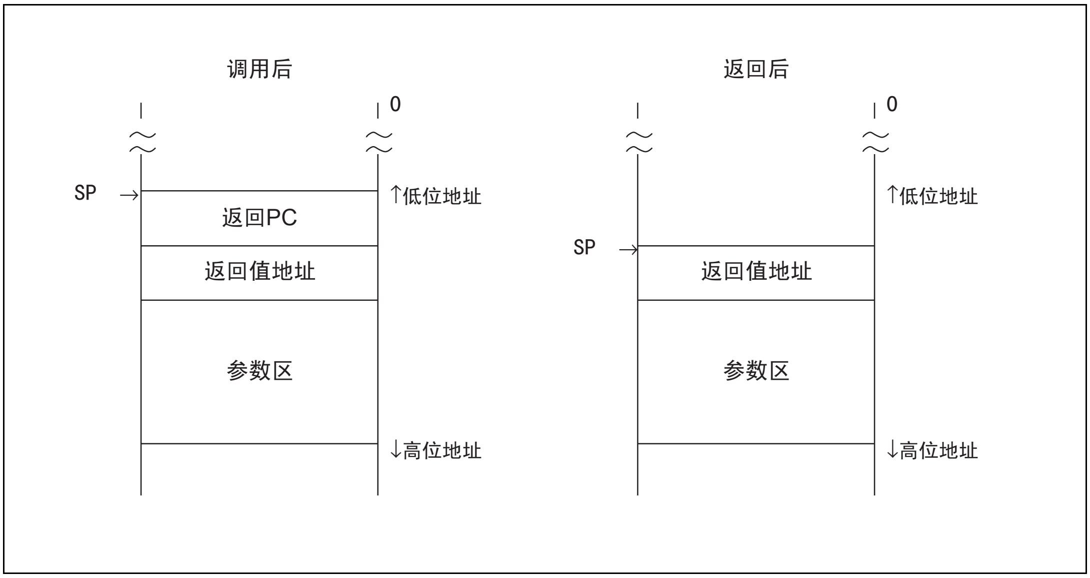


图 9.8　有关分配和释放堆栈帧的规则

（3）有关寄存器的规则

在函数调用前后，有保证值的寄存器和不保证值的寄存器。

不同 CPU 种类中的寄存器保证规则如表 9.5 所示。

表 9.5　函数调用前后的寄存器保证规则

项目	保存参数的寄存器个数	CPU 种类和对象寄存器		编程时的注意点
		H8SX H8S/2600 H8S/2000 H8/300H	H8/300	
1　不保证的寄存器 caller-save	2	ER0,ER1	R0,R1	如果在函数调用时对象寄存器中存在有效值，就在调用侧保存其有效值。被调用函数侧不进行保存而能使用。
	3	ER0～ER2	R0～R2	
2　保证的寄存器 callee-save	2	ER2～ER6	R2～R6	保存函数中使用的对象寄存器的值，返回时恢复。
	3	ER3～ER6	R3～R6	

注意　*1：能用 regparam 选项、__regparam2 或者__regparam3 指定参数保存寄存器的个数。

在 H8S/2600 高级模式中的有关寄存器保证规则的具体例子如下所示：

（a）从 C/C++程序调用汇编程序的子例程时

汇编程序（被调用侧）

```
       .EXPORT   _sub
       .SECTION P,CODE,ALIGN=2
_sub: STM.L    (ER4-ER6),@-SP
       SUB.L    #10,SP
             :
       ADD.L    #10,SP
       LDM.L    @SP+,(ER4-ER6)
       RTS
       .END
```

被调用侧（callee）保存
函数中使用的寄存器

函数处理
（ER0、ER1 不保存而能使用）

恢复保存的寄存器

C/C++程序（调用侧）

```
#ifdef _ _cplusplus
extern “C”
#endif
void sub(void);
void f(void)
{
    sub();
}
```

（b）从汇编程序调用 C 程序的子例程时

C 程序（被调用侧）

```
void sub(void)
{
    …
}
```

汇编程序（调用侧）

```
     .IMPORT   _sub
     .SECTION P,CODE,ALIGN=2
              :
     MOV.L    ER1,@(4,SP)
     MOV.L    ER0,ER6
     JSR      @_sub
               :
     RTS
     .END
```

在调用侧（caller），如果
寄存器 ER0、ER1 中存在有效值，
就将其保存到空寄存器或者堆栈。

对函数名附加"_"后进行参照

（c）从汇编程序调用 C++程序的子例程时

C++程序（被调用侧）

```
extern "C"
void sub(void)
{
    …
}
```

汇编程序（调用侧）

```
     .IMPORT    _sub
     .SECTION P,CODE,ALIGN=2
                  :
     MOV.L     ER1,@(4,SP)
     MOV.L     ER0,ER6
     JSR       @_sub
                  :
     RTS
     .END
```

在调用侧，如果寄存器 ER0,ER1 中存在有效值，就将其保存到空寄存器或堆栈。

【注】*1　用“extern “C” ”声明的函数不能多重定义。

（4）有关设定和参照参数、返回值的规则

以下说明有关设定和参照参数、返回值的方法。在函数声明中，参数和返回值的规则根据是否明确声明各个参数和返回值的型而不同。在明确声明参数和返回值的型时，使用函数的原型声明。

以下首先说明有关参数和返回值的一般规则，然后说明有关参数的分配方法和返回值的设定位置。

（a）有关参数和返回值的一般规则

- **参数的传递方法**

  必须在将参数值复制到参数分配区后调用函数。因为调用函数侧在返回后不参照参数分配区，所以即使在被调用函数侧更改参数值，也不直接影响调用侧的处理。

- **型转换的规则**

  在传递参数或者返回值时，有可能自动进行型转换。以下说明型转换的规则：

  – 返回值的型转换
    将返回值转换为该函数的返回型。
  – 声明型的参数的型转换
    将由原型声明的型的参数转换为被声明的型。
  – 没有声明型的参数的型转换
    没有由原型声明的型的参数按照以下规则进行型转换：
    - 将 char 型、unsigned char 型的参数转换为 int 型。
    - 将 float 型的参数转换为 double 型。
    - 不转换上述以外的型。

例：

(1)
```
long f();

long f()
 {    float x;
      return x;

 }
```
将返回值转换成long型。

(2)
```
void p(int,...);

f()
 {    char c;
      p(1.0, c);

 }
```
因为c没有对应型声明的参数，所以被转换成int型。

因为1.0对应int型声明的参数，所以被转换成int型。

注意　在没有用原型声明对参数的型进行声明时，为了正确传递参数，被调用侧和调用侧必须指定相同的型。否则，运行不被保证。

```
f(x)
float x;
{
        .
        .
        .
 }

main()
{
   float x;
   f(x);
 }
```
不保证运行的指定例子

```
f(float x)
{
      .
      .
      .
 }

main()
{
   float x;
   f(x);
 }
```
正确的指定例子

在不保证运行的指定例子中，因为函数“f”没有参数的原型声明，所以在函数“main”侧调用时，将参数 x 转换为 double 型。

另一方面，因为函数“f”侧将参数声明为 float 型，所以不能正确传递参数。必需用原型声明对参数的型进行声明或者将函数“f”侧的参数声明为 double 型。

正确的指定例子是用原型声明对参数的型进行声明的例子。

（b）参数分配区

有将参数分配到堆栈的参数区和分配到寄存器的情况。

各目标种类的参数分配区如图 9.9 所示，参数分配区的一般规则如表 9.6 所示。

将 C++程序的非静态函数成员的 this 指针分配到 R0 或者 ER0。

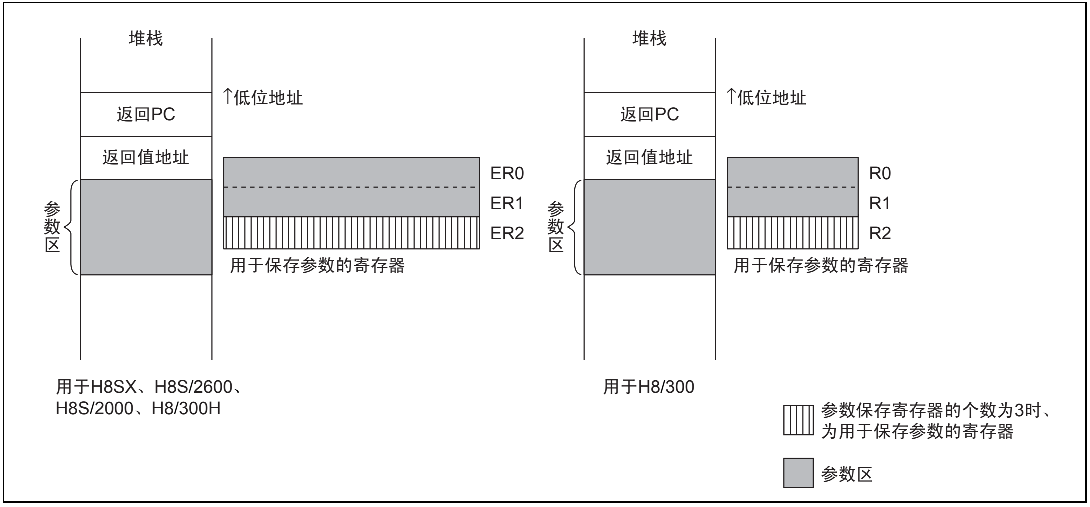


图 9.9 参数分配区

表 9.6 参数分配区的一般规则

CPU 种类	参数保存寄存器的个数	分配规则		
		分配到寄存器的参数		分配到堆栈的参数
		用于保存参数的寄存器	对象的型	
H8SX AE5 H8S/2600 H8S/2000 H8/300H	2	ER0、ER1	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、结构体（4 个字节以下）*4、指针、参照、指向数据成员的指针	[1]参数的型不为寄存器传递对象的型的参数 [2]对于具有可变个数参数的函数，用原型声明进行了声明的参数*2 [3] 由于参数的个数太多，不能分配到寄存器的参数。
	3	ER0、ER1、ER2		
H8/300	2	R0、R1	char、unsigned char、short、unsigned short、int、unsigned int、long*3、unsigned long*3、float*3、结构体（2 个字节以下）*4、结构体（4 个字节以下）*3*4、指针、参照、指向数据成员的指针	
	3	R0、R1、R2		

【注】 *1 能用 regparam 选项、__regparam2 和__regparam3 指定参数保存寄存器的个数。

*2 对于具有可变个数参数的函数，用原型声明进行声明时，将…部分及其前一个的参数分配到堆栈。

例： int f2(int,int,...);

f2(x,y,z); → 将 y、z 分配到堆栈。

*3 在指定 longreg 选项时，为对象。

*4 在指定 structreg 选项时，为对象。

（c）参数的分配

- 分配到用于参数保存的寄存器

  按照源程序的声明顺序从序号小的 LSB 侧的寄存器开始分配到用于参数保存的寄存器。用于参数保存的寄存器的分配例子如图 9.10 所示。

```
f(char a,int b)
{
      :
}
```

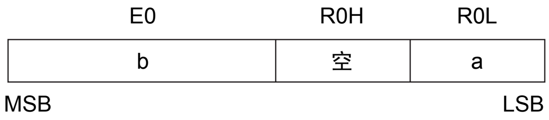


图 9.10　用于参数保存的寄存器的分配例子（H8S/2600）

- 分配到堆栈的参数区

按照源程序中指定的顺序从低位地址开始分配到堆栈的参数区。

注意　在设定结构体型、联合体型和类的型的参数时，与该型原来的边界调整无关而分配为 2 个字节边界，而且其区域使用偶数字节的区域。
这是因为 H8SX、AE5、H8S、H8/300H、H8/300 系列的堆栈指针以 2 个字节为单位进行变化。

请参照“9.3.3 参数分配的具体例子”中对各 CPU/运行模式的分配参数的具体例子。

（d）返回值的设定位置

根据函数返回值的型，有设定到寄存器和设定到存储器的情况。返回值的型和设定位置的关系请参照表9.7。

将函数的返回值设定到存储器时，返回值被设定到返回值地址指向的区域。在调用侧，除参数区以外还需确保返回值的设定区，在将该地址设定到返回值地址的区域后调用函数（请参照图9.11）。

在函数的返回值为void型时，不设定返回值。

表 9.7　返回值的型和设定位置

返回值的型	返回值的设定位置	
	用于 H8SX、AE5、H8S/2600、H8S/2000、H8/300H	用于 H8/300
char、unsigned char	寄存器（R0L）	寄存器（R0L）
short、unsigned short、int、unsigned int	寄存器（R0）	寄存器（R0）
指向函数的指针	寄存器 正常模式：（R0） 其它模式：（ER0）	寄存器（R0）
指向数据的指针、参照、指向数据成员的指针	寄存器 正常/中级模式：（R0） 高级/最高模式 并且有 ptr16 选项或者_ _ptr16 关键字：（R0）*3 高级/最高模式 并且无 ptr16 选项和_ _ptr16 关键字：（ER0）	寄存器（R0）
long、unsigned long、float	寄存器（ER0）	返回值的设定区（存储器） 寄存器（R0, R1）*1
2 个字节以下的结构体	返回值的设定区（存储器） 寄存器（R0）*2	返回值的设定区（存储器） 寄存器（R0）*2
3 个字节或者 4 个字节的结构体	返回值的设定区（存储器） 寄存器（ER0）*2	返回值的设定区（存储器） 寄存器（R0,R1）*1*2
double、long double、结构体、联合体、类和指向函数成员的指针	返回值的设定区（存储器）	返回值的设定区（存储器）

【注】 *1　指定 longreg 选项时

*2　指定 structreg 选项时

*3　ptr16 选项和_ _ptr16 关键字只在指定 H8SX 和 H8S 时有效

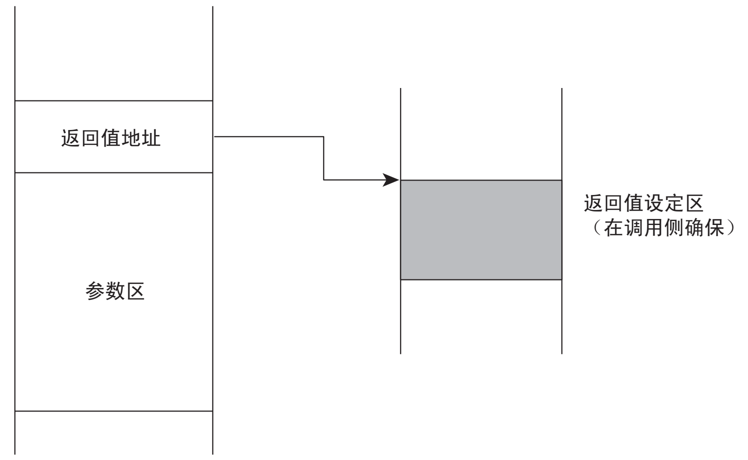


图 9.11　将返回值设定到存储器时的设定区

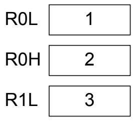
9.3.3　　参数分配的具体例子

（1）用于 H8SX、H8S/2600、H8S/2000 和 H8/300H

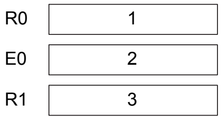
（cpu＝H8SXN、cpu＝H8SXM、cpu＝H8SXA、cpu＝H8SXX、
cpu＝2600a、cpu＝2600n、cpu＝2000a、cpu＝2000n、cpu＝300ha、cpu＝300hn）

例 1：按照声明顺序将寄存器传递对象的型的参数分配到寄存器 ER0、ER1[*1]。

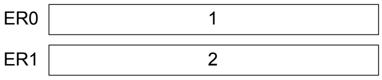
```
[1] int f(char,char,char);
                :
           f(1,2,3);
                :
```



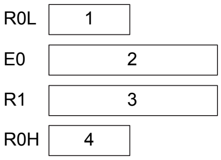
```
[2] int f(int,int,int);
                :
           f(1,2,3);
                :
```



```
[3] int f(long,long);
                :
           f(1,2);
                :
```



```
[4] int f(char,int,int,char);
                :
           f(1,2,3,4);
                :
```



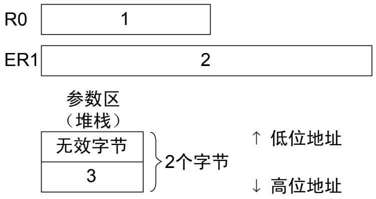
*1：参数保存寄存器的个数为 3 时，为 ER0、ER1、和 ER2

例 2：将不能分配到寄存器的参数分配到堆栈。

另外，如果参数的型为 char 型并且分配到堆栈的参数区，低位地址的字节无效。

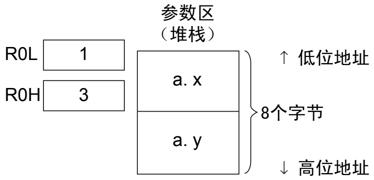
（参数保存寄存器的个数为 2 个时）

```
int f(int,long,char);
           :
        f(1,2,3);
           :
```



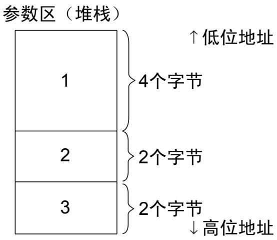
例 3：将不能分配到寄存器的型的参数分配到堆栈。

```
struct s{long x,y;}a;
int f(char,struct s,char);
      :
    f(1,a,3);
      :
```

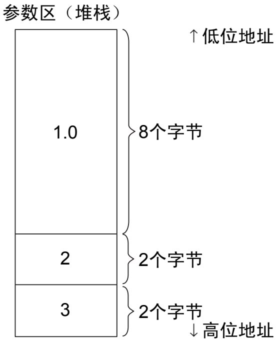


例 4：对具有可变个数参数的函数，用原型声明进行声明时，按照声明顺序将...部分参数及其前一个参数分配到堆栈。

```
[1] int f(long,...);
          :
         f(1,2,3);
          :
```

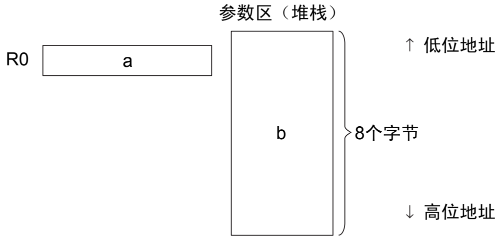


```
[2] int f(double,int,...);
          :
         f(1.0,2,3);
          :
```



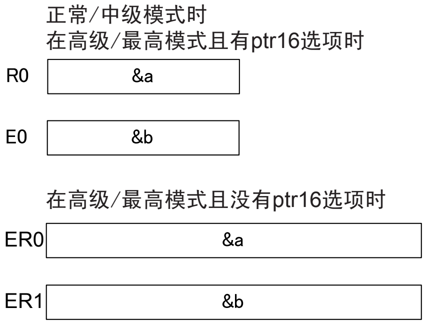
例 5：C 程序中没有原型声明时，将 char 型扩展为 int 型、float 型扩展为 double 型后进行传递。

```
int f( );
char a;
float b;
          :
        f(a,b);
          :
```

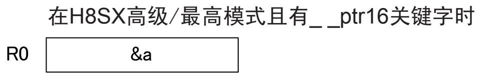


例 6：对于指向数据的指针型和 C++的参照型，在正常模式/中级模式时，分配到 2 个字节的区域；在高级模式/最高模式中并且有 ptr16 选项或有_ _ptr16 关键字时，分配到 2 个字节的区域；在高级模式/最高模式并且没有 ptr16 选项及没有_ _ptr16 关键字时，分配到 4 个字节的区域。但是，ptr16 选项和_ _ptr16 关键字在 H8SX 和 H8S 时有效。

```
int a,b;
int f(int *,int &);
          :
    f(&a,b);
          :
```

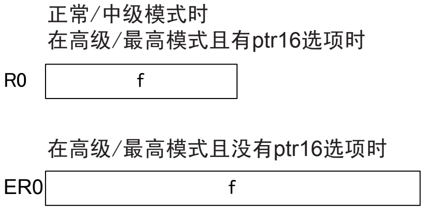


```
int g(int _ _ptr16 *);
          :
     g(&a);
          :
```

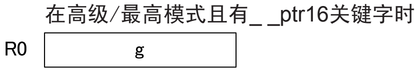


例 7：对于指向数据的指针型返回值，在正常模式/中级模式时，为 2 个字节；在高级模式/最高模式并且有 ptr16 选项或有_ _ptr16 关键字时，为 2 个字节；在高级模式/最高模式并且没有 ptr16 选项及没有_ _ptr16 关键字时，为 4 个字节。但是，ptr16 选项和_ _ptr16 关键字在 H8SX 和 H8S 时有效。

```
int *f(void);
int *p;
          :
    p = f( );
          :
```

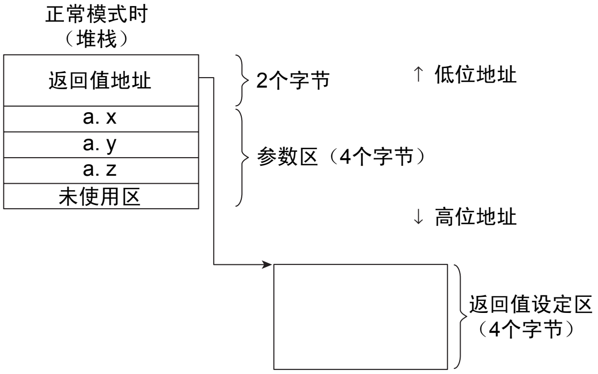


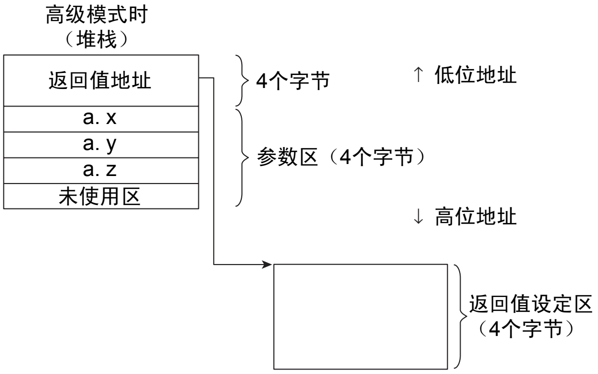
```
int _ _ptr16 *g(void);
int _ _ptr16 *q;
          :
    q = g( );
          :
```



例 8：函数的返回型超过 4 个字节或者为结构体（没有指定 structreg 时或者超过 4 个字节的结构体）时，在参数区的前面设定返回值地址。另外，结构体的长度为奇数字节时，会产生 1 个字节的未使用区。

```
struct s{char x,y,z;}a,b;
float f(struct s);
                  :
               f(a);
                  :
                  :
```

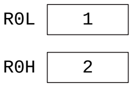




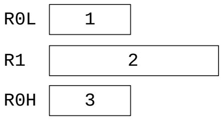
（2）用于 H8/300（cpu＝300）

例 1：按照声明顺序将寄存器传递对象的型的参数分配到寄存器 R0、R1[*1]。

```
[1] int f(char,char);
            :
          f(1,2);
            :
```



```
[2] int f(char,int,char);
            :
          f(1,2,3);
            :
```

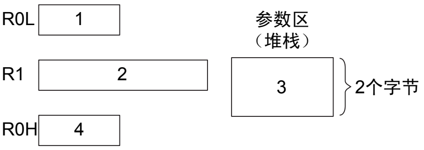


*1：参数保存寄存器个数为 3 时，为 R0、R1、R2

例 2：将不能分配到寄存器的参数分配到堆栈。

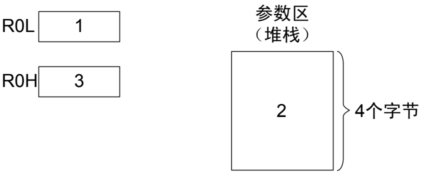
（参数保存寄存器个数为 2 个时）

```
int f(char,int,int,char);
     :
   f(1,2,3,4);
     :
```



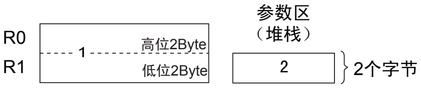
例 3：将不能分配到寄存器的型的参数分配到堆栈。

```
int f(char,long,char);
        :
      f(1,2,3);
        :
```



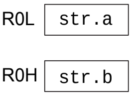
例 4：在指定 longreg 选项时，将 4 个字节的数据分配到寄存器 R0、R1。

```
int f(long,Short);
     :
   f(1,2);
     :
```



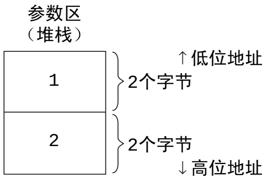
例 5：在指定 structreg 选项时，将 2 个字节以下的结构体分配到寄存器。

```
struct A{
   char a,b;
}str;

int f(struct A);
     :
   f(str)
     :
```

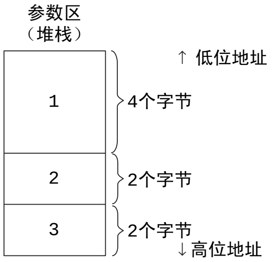


例 6：对具有可变个数参数的函数，用原型声明进行声明时，按照声明顺序将没有对应的型的参数及其前一个的参数分配到堆栈。

```
[1] int f(int,...);
          :
         f(1,2);
          :
```

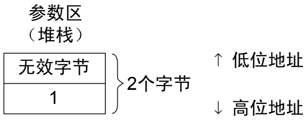


```
[2] int f(long,int,...);
          :
         f(1,2,3);
          :
```



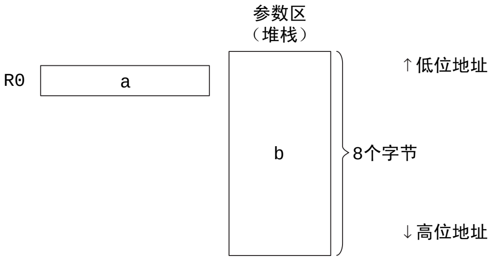
例 7：如果参数的型为 char 型并且分配到堆栈的参数区，低位地址的字节无效。

```
int f(char,...);
          :
        f(1);
          :
```



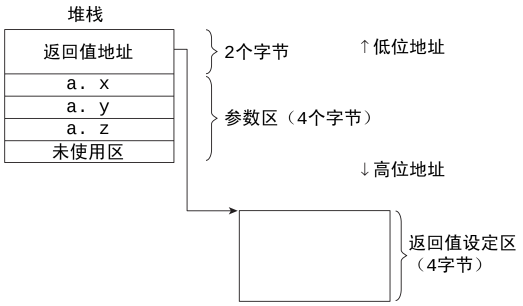
例 8：C 程序中没有原型声明时，将 char 型扩展为 int 型、float 型扩展为 double 型后进行传递。

```
int f( );
char a;
float b;
          :
        f(a,b);
          :
```



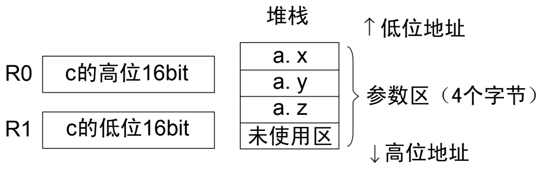
例 9：函数的返回型超过 2 个字节时，在参数区的前面设定返回值地址。另外，结构体的长度为奇数字节时，会产生 1 个字节的未使用区。

```
struct s{char x,y,z;}a,b;
float f(struct s);
     :
     f(a);
     :
     :
```



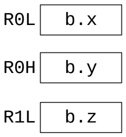
例 10：在指定 longreg 并且函数的返回型超过 2 个字节时，将返回值分配到 R0、R1。

```
struct s{char x,y,z;}a,b;
long c;
long f(struct s);
   :
c=f(a);
   :
```



例 11：在指定 structreg、longreg 并且函数的返回型为 4 个字节以下的结构体时，将返回值分配到 R0、R1。

```
struct s{char x,y,z;}a,b;
struct s f(void);
   :
b=f();
   :
```



9.3.4　　寄存器和堆栈区的使用方法

（1）用于 H8SX 的高级模式和最高模式（cpu＝H8SXA、cpu＝H8SXX）

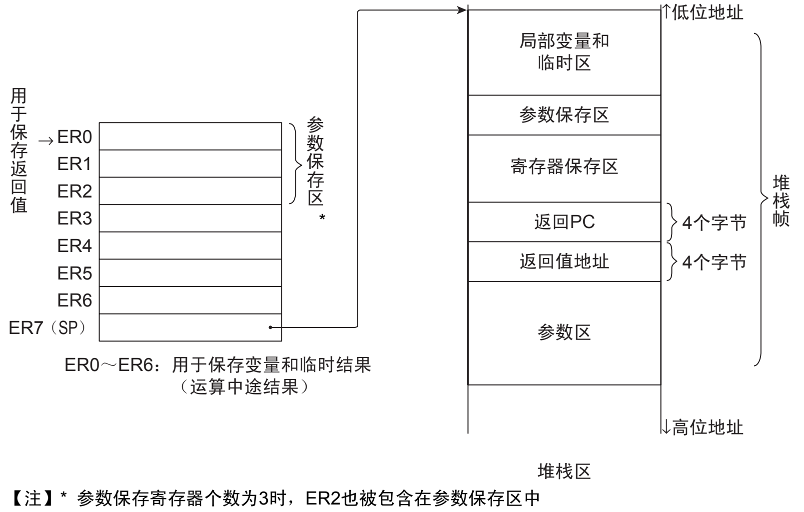


图 9.12　寄存器和堆栈区的使用方法（cpu＝H8SXA*[1]、cpu＝H8SXX*[1]）

[注]　*1：没有 ptr16 选项。

（2）用于 H8SX 的中级模式、高级模式（ptr16）和最高模式（ptr16）
（cpu＝H8SXM、cpu＝H8SXA 且 ptr16、cpu=H8SXX 且 ptr16）

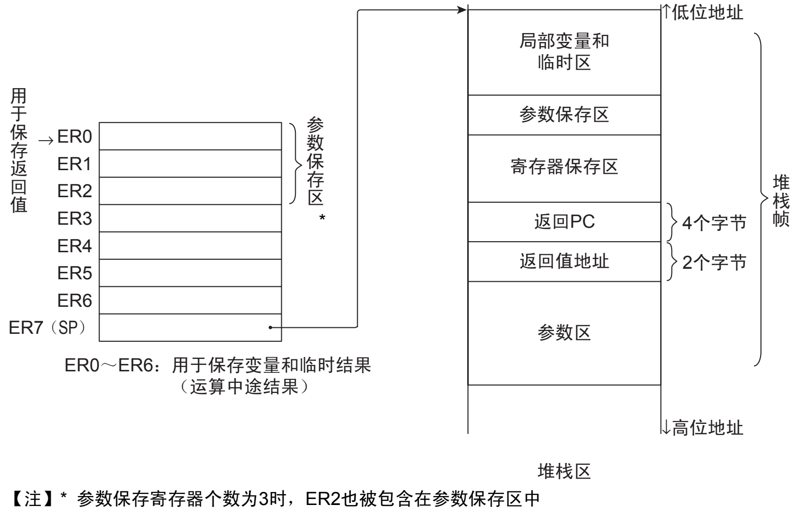
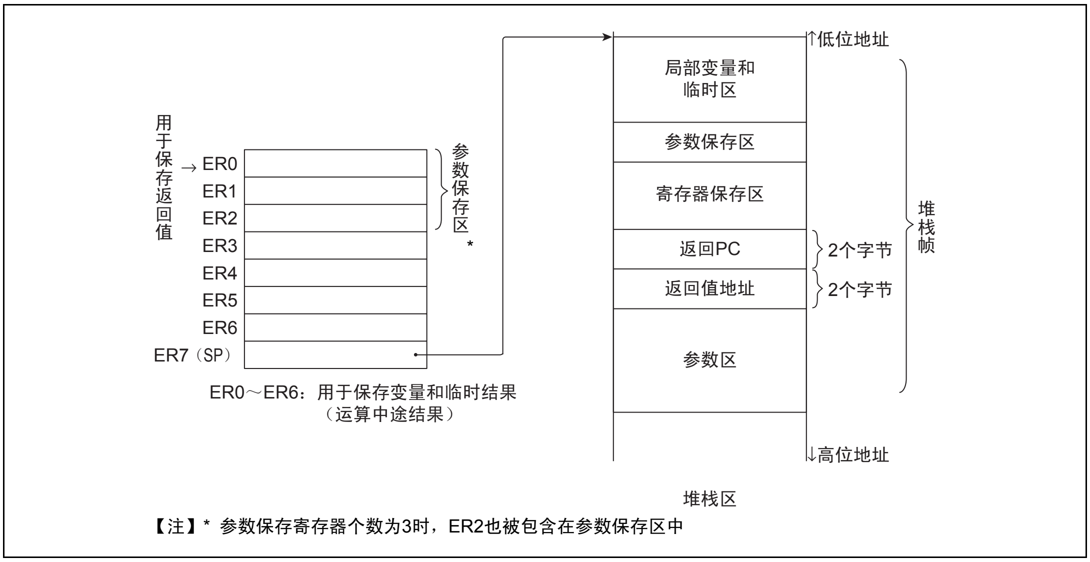


图 9.13　寄存器和堆栈区的使用方法（cpu＝H8SXM、cpu＝H8SXA*[2]、cpu＝H8SXX*[2]）

[注]　*2：有 ptr16 选项。

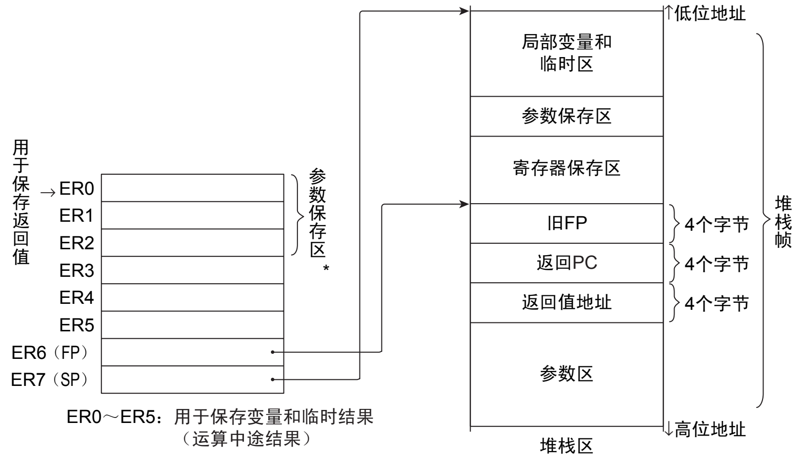
（3）用于 H8SX 的正常模式（cpu＝H8SXN）



【注】* 参数保存寄存器个数为3时，ER2也被包含在参数保存区中

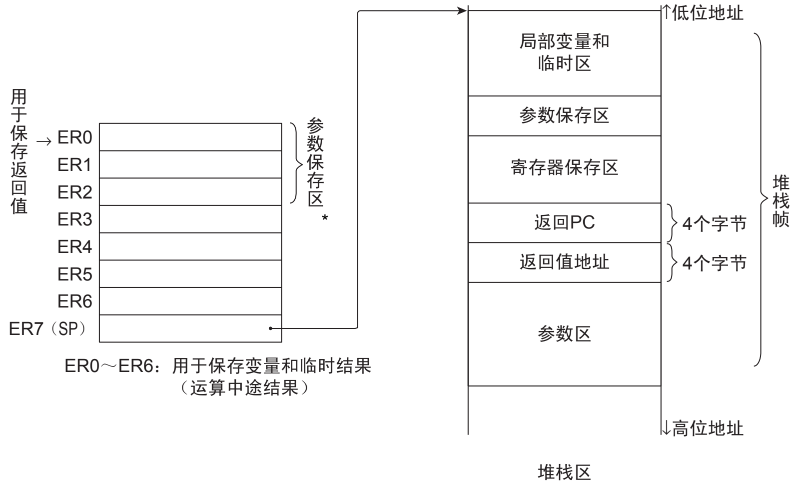
图 9.14　寄存器和堆栈区的使用方法（cpu＝H8SXN）

（4）用于 H8S/2600、H8S/2000 和 H8/300H 的高级模式
（cpu＝2600a、cpu＝2000a、cpu＝300ha）



【注】* 参数保存寄存器个数为3时，ER2也被包含在参数保存区中

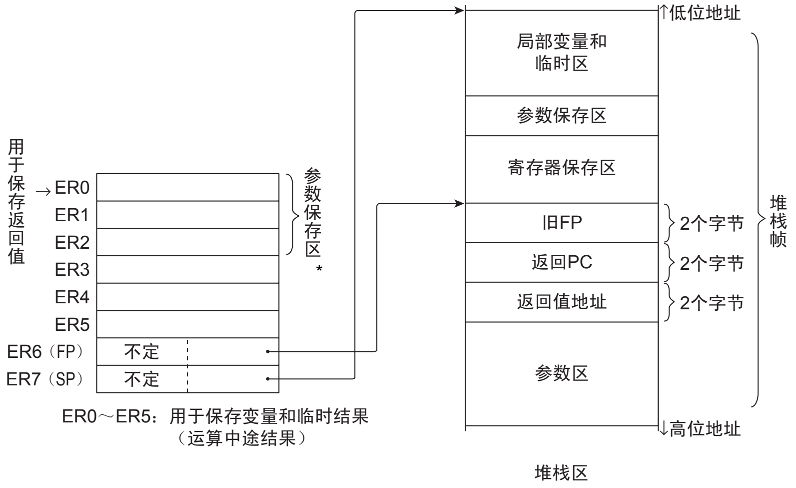
图 9.15　非优化时的寄存器和堆栈区的使用方法
（cpu＝2600a、cpu＝2000a、cpu＝300ha）



【注】* 参数保存寄存器个数为3时，ER2也被包含在参数保存区中

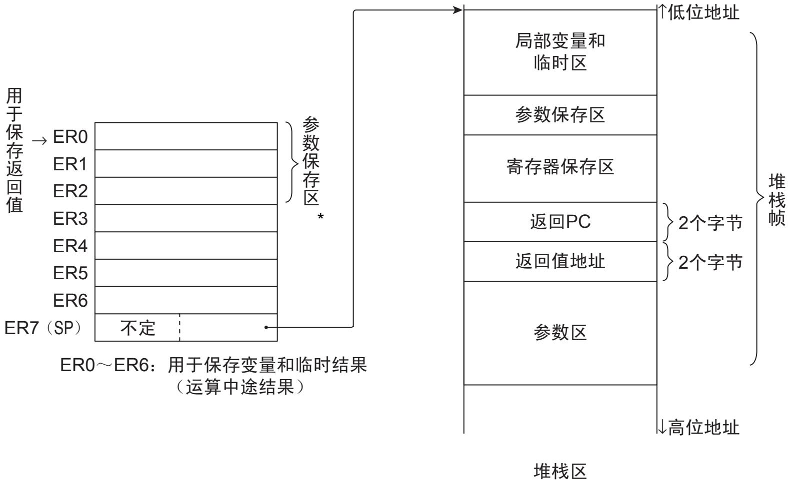
图 9.16　优化时的寄存器和堆栈区的使用方法
（cpu＝2600a、cpu＝2000a、cpu＝300ha）

（5）用于 H8S/2600、H8S/2000 和 H8/300H 的正常模式
（cpu＝2600n、cpu＝2000n、cpu＝300hn）



【注】* 参数保存寄存器个数为3时，ER2也被包含在参数保存区中

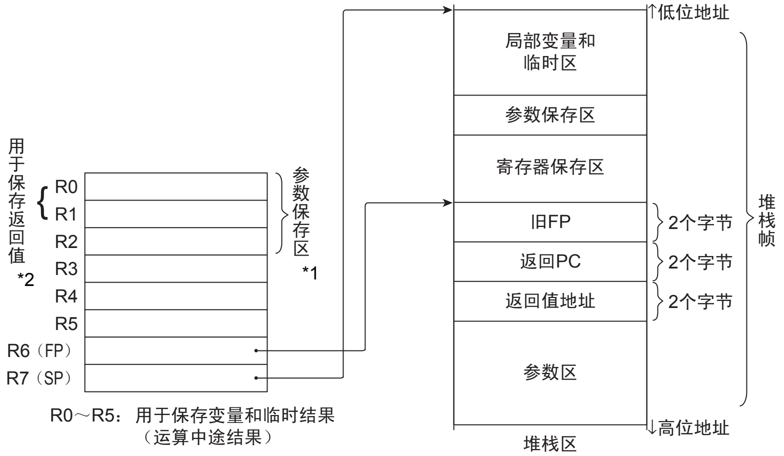
图 9.17　非优化时的寄存器和堆栈区的使用方法
（cpu＝2600n、cpu＝2000n、cpu＝300hn）



【注】* 参数保存寄存器个数为3时，ER2也被包含在参数保存区中

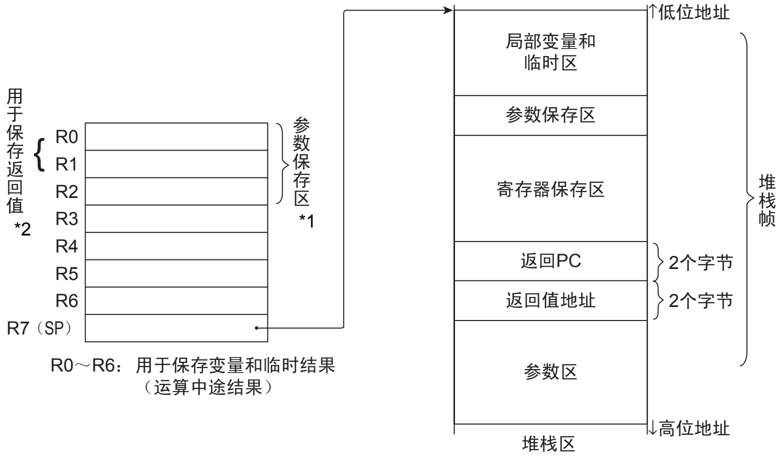
图 9.18　优化时的寄存器和堆栈区的使用方法
（cpu＝2600n、cpu＝2000n、 cpu＝300hn）

（6）用于 H8/300（cpu＝300）



【注】 *1　参数保存寄存器个数为3时，R2也被包含在参数保存区中
　　　 *2　指定longreg时，R1也被包含在返回值保存区中

图 9.19　非优化时的寄存器和堆栈区的使用方法（H8/300）



【注】 *1　参数保存寄存器个数为3时，R2也被包含在参数保存区中
　　　 *2　指定longreg时，R1也被包含在返回值保存区中

图 9.20　优化时的寄存器和堆栈区的使用方法（H8/300）

9.4　程序建立时的注意事项

本节说明编译程序中编程时的注意事项和从编译到调试的程序开发时的注意事项。

9.4.1　编程时的注意事项

（1）float 型参数的函数

声明 float 型参数的函数必须进行原型声明或者将参数的 float 型声明更改为 double 型。如果不进行原型声明而调用具有 float 型参数的函数，就不保证运行。

例：

```
void f(float); －－－－ [1]
void g( )
{
    float a;
     :
     f(a);
}

void f(float x)
{
     :
}
```

函数“f”是具有 float 型的参数。此时，必须如［1］进行原型声明。

（2）C/C++语言规格没有规定评价顺序的表达式

在 C/C++语言规格没有规定评价顺序的表达式中，如果进行因评价顺序而产生不同结果的编程，就不保证其运行。

例：

```
a[i]=a[++i];   根据先评价还是后评价赋值表达式的右边，左边i的值发生变化。
sub(++i, i);   根据先评价还是后评价函数的第1个参数，第2个参数i的值发生变化。
```

（3）因优化而可能被删除的编程

如果连续参照同一个变量或者记述不使用结果的表达式，就有可能因编译程序的优化而作为冗余码被删除。为了保证总是能存取，必须在声明时指定 volatile。

例：

```
[1] b=a;          /* 第1行的表达式有可能作为冗余码被删除。          */
     b=a;
[2] while(1)a;    /* 变量a的参照和循环语句有可能作为冗余码被删除。    */
```

（4）溢出运算和被零除

即使有溢出运算或被零除，也不输出错误信息。如果在一个常数或者常数之间的运算中发生溢出运算或被零除，就在编译时输出错误信息。但是，在 H8SX 的情况下，编译程序有可能检测不出被零除。

例：

```
void main(void)
{
    int ia;
    int ib;
    float fa;
    float fb;
    ib=32767;
    fb=3.4e+38f;

/*  在常数或者常数之间的运算时，输出对溢出和被零除的编译错误信息                  */
    ia=99999999999;             /*(W)检测常数的溢出                              */
    fa=3.5e+40f;                /*(W)检测浮点运算的溢出                          */
    ia=1/0;                     /*(E)检测被0除（H8SX、H8S除外）                  */
    fa=1.0/0.0;                 /*(W)检测浮点的被0除（H8SX、H8S除外）             */

/*  对于执行时的溢出，不输出错误信息                                              */
    ib=ib+32767;                /*  忽视运算结果的溢出                            */
    fb=fb+3.4e+38f ；           /*  忽视浮点运算结果的溢出                        */
}
```

注意　如果指定 cpuexpand 选项，就不输出溢出或下溢的错误。

（5）有关数学函数库的精度

必须注意：当 x≈1 时，acos(x)和 asin(x)函数的误差变大。

误差范围如下：

acos(1.0 – ε)中的绝对误差双精度2^{-39}（ ε = 2^{-33} ）
单精度2^{-21}（ ε = 2^{-19} ）
asin(1.0 – ε)中的绝对误差双精度2^{-39}（ ε = 2^{-28} ）
单精度2^{-21}（ ε = 2^{-16} ）

（6）对 const 型变量的写

必须注意：即使声明 const 型的变量，通过型转换将其转换为非 const 型后赋值，或者在分割编译的程序之间不进行型统一的处理，就不能通过编译程序进行对 const 型变量的写检查。

例：

```
[1] const char *p ;         /* 因为库函数strcat的第1参数为指向char型        */
          :                 /* 的指针型，                                    */
    strcat(p,"abc") ;       /* 所以有可能改写参数指向的区域。                */
```

［2］文件1

```
const int i;
```

文件2

```
extern int i;          /* 因为在文件2中没有用const型声明变量i     */
    :                  /* 所以即使在文件2中进行写操作             */
i=10;                  /* 也不会出错。                            */
```

（7）关于位操作指令的注意事项

本编译程序生成 BSET、BCLR、BNOT、BST、BIST 各位操作指令。这些指令以字节为单位读数据，在位操作后再次以字节为单位写数据。另一方面，如果读只写寄存器，就与寄存器内容无关，CPU 读到的数据为不定值。因此，在只写寄存器的位操作指令中，有可能改变操作位以外的位内容。对只写寄存器的操作例子如下所示：

例：

include文件(300x.h)的内容

```
struct S_p4ddr{
   unsigned char p7:1;
           :
   unsigned char p0:1;
};
union SS{
   unsigned char Schar;
   struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

C 源程序的内容

```
#include "300x.h"
unsigned char DDR;
//准备只写寄存器的
//备份数据
void sub(void)
{
   DDR &=~P0;
   P4DDR.Schar=DDR;
}
```

9.4.2　　用 C++编译程序编译 C 程序时的注意事项

（1）函数的原型声明

在使用函数前需要原型声明。此时，也必须声明形式参数的型。

```
extern void func1();
void g()
{
    func1(1); // C++错误
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

（2）const 目标的连接

const 目标的连接在 C 程序中为外部连接，而在 C++程序中为内部连接。另外，const 目标需要初始值。

```
const int cvalue1;
// C++错误

const int cvalue2 = 1;
// C++内部
```

```
const int cvalue1=0;
   // 设定初始值

extern const int cvalue2 = 1;
   // 与C程序相同，为外部连接
```

（3）从 void*的赋值

在 C++程序中，如果不使用明确的型转换，就不能赋值到指向其它目标型的指针（指向函数的指针和指向成员的指针除外）。

```
void func(void *ptrv, int *ptri)
{
    ptri = ptrv; // C++错误
}
```

```
void func(void *ptrv, int *ptri)
{
    ptri = (int *)ptrv; //OK
}
```

9.4.3　　程序开发时的注意事项

说明从建立到调试的程序开发时的注意事项。

（1）有关选择 CPU/运行模式的注意事项

（a）必须统一编译和汇编时指定的 CPU/运行模式。

必须统一编译和汇编时用 cpu 选项指定的 CPU/运行模式。如果将不同的 CPU/运行模式建立的目标程序连接在一起，就不能保证目标程序执行时的运行。

（b）在汇编时，必须指定和编译时指定的 CPU/运行模式相同的 CPU 种类。

在汇编 C 编译程序生成的汇编程序时，必须用 cpu 选项指定和编译时指定的 CPU/运行模式相同的 CPU 种类。

（c）在建立标准库时，必须指定和编译时指定的 CPU/运行模式相同的 CPU 种类。

在用标准库生成工具建立标准库时，必须用 cpu 选项指定和编译时指定的 CPU/运行模式相同的 CPU 种类。

（2）有关选项的注意事项

在编译时和生成库时，必须统一下述的选项。如果将不同选项建立的目标程序连接在一起，就不能保证目标程序执行时的运行。

cpu
exception/noexception
rtti = on/off
regparam
longreg/nolongreg
structreg/nostructreg
stack
double=float
byteenum
pack
bit_order = left/right
indirect = normal/extended[*1]
（能混合指定有/无 indirect 选项，但是 normal 和 extended 不能混合）
ptr16
sbr[*2]

【注】 *1 indirect = extended 只限 H8SX
*2 只限 H8SX

第 10 章　C/C++语言规格

10.1　语言规格

10.1.1　编译程序的规格

对于在语言规格中尚未规定的实现定义项目，编译程序的规格如下所示：

（1）环境

表 10.1　环境的规格

	项　目	编译程序的规格
1	对 main 函数的实际参数的含义	没规定。
2	交互输入/输出设备的结构	没规定。

（2）标识符

表 10.2　标识符的规格

	项　目	编译程序的规格
1	不与外部连接的标识符（内部名）的有效字符数	前 8189 个字符有效。
2	与外部连接的标识符（外部名）的有效字符数	前 8191 个字符有效。
3	与外部连接的标识符（外部名）的大小写字符的区别	区别大小写字符。

（3）字符

表 10.3　字符的规格

	项　目	编译程序的规格
1	源字符集和执行环境字符集的元素	都为 ASCII 字符集。但是，在字符串、字符常数中能记述 shift JIS、EUC 汉字码或者 Latin1 码。
2	在多字节字符的代码化中使用的 shift 状态	不支持 shift 状态。
3	执行程序时的字符集的字符位数	位数为 8 位。
4	字符常数和字符串中的源字符集字符及执行环境字符集字符的对应	对应相同的 ASCII 字符。
5	包含语言中尚未规定的字符或转义序列的整数字符常数的值	不支持语言规定以外的字符或转义序列。
6	包含 2 个字符以上的字符常数或者 2 个字符以上的多字节字符的宽字符常数的值	字符常数的高位 2 个字符有效。不支持宽字符常数。另外，如果指定 1 个以上的字符，就输出警告错误。
7	为了将多字节字符转换为宽字符所使用的 locale 的规格	不支持 locale。
8	char 型的值	具有和 signed char 型相同值的范围。

（4）整数

表 10.4　整数的规格

	项　目	编译程序的规格
1	整型的表现方法及其值	如表 10.5 所示。
2	将带符号整型或者无符号整型转换为相同长度的带符号整型时的值（用转换目标型无法表现结果值时）	整数值的低位 4 个字节（转换后的型为 long 型）、低位 2 个字节（转换后的型为 int/short 型）或者低位 1 个字节（转换后的值为 char 型）为转换后的值。
3	对带符号整数，按位运算的结果	为带符号的值。
4	整数除法中余数的符号	和被除数符号相同。
5	负数的带符号整型的右移结果	保持符号位。

表 10.5　整型及其值的范围

	型	值的范围	数据长度
1	char	－128～127	1 个字节
2	signed char	－128～127	1 个字节
3	unsigned char	0～255	1 个字节
4	short	－32768～32767	2 个字节
5	unsigned short	0～65535	2 个字节
6	int	－32768～32767	2 个字节
7	unsigned int	0～65535	2 个字节
8	long	－2147483648～2147483647	4 个字节
9	unsigned long	0～4294967295	4 个字节

（5）浮点

表 10.6　浮点的规格

	项　目	编译程序的规格
1	浮点型的表现方法及其值	浮点型有 float 型、double 型和 long double 型。在“10.1.3 浮点数的规格”中说明浮点型的内部表现、转换规格以及运算规格等性质。浮点型能表现的值的极限值如表 10.7 所示。
2	将整数转换为不能正确表现实际值的浮点数时的舍去方法	
3	将浮点数转换为较短的浮点数时的舍去或者舍入方法	

表 10.7　浮点数的极限值

	项　目	极限值	
		10 进制数表现　*1	16 进制数表现
1	float 型的最大值	3.4028235677973364e＋38f (3.4028234663852886e＋38f)	7f7fffff
2	float 型的正的最小值	7.0064923216240862e－46f (1.4012984643248171e－45f)	00000001
3	double long double } 型的最大值	1.7976931348623158e＋308 (1.7976931348623157e＋308)	7fefffffffffffff
4	double long double } 型的正的最小值	4.9406564584124655e－324 (4.9406564584124654e－324)	0000000000000001

【注】 *1 10 进制数表现的极限值是不为 0 或者无穷大的极限值。另外，()内表示理论值。

（6）数组和指针

表 10.8　数组和指针的规格

	项　目	编译程序的规格
1	为了保持数组大小的最大值所需要的整型（size_t）	unsigned int 型（H8/300）
		unsigned int 型（正常模式、 H8S/2000 高级模式并有 ptr16 选项、 H8S/2600 高级模式并有 ptr16 选项、 H8SX 中级模式、 H8SX 高级模式并有 ptr16 选项、 H8SX 最高模式并有 ptr16 选项）
		unsigned long 型（H8/300H 高级模式、 H8S/2000 高级模式并没有 ptr16 选项、 H8S/2600 高级模式并没有 ptr16 选项、 H8SX 高级模式并没有 ptr16 选项、 H8SX 最高模式并没有 ptr16 选项)
2	从指针型转换到整型 （指针型的长度≥整型的长度）	为指针型低位字节的值。
3	从指针型转换到整型 （指针型的长度＜整型的长度）	零扩展。
4	从整型转换到指针型 （整型的长度≥指针型的长度）	为整型低位字节的值。
5	从整型转换到指针型 （整型的长度＜指针型的长度）	零扩展。
6	为了保持相同数组中的成员指针之间的差所需要的整型（ptrdiff_t）	int 型（H8/300）
		int 型（正常模式、 H8S/2000 高级模式并有 ptr16 选项、 H8S/2600 高级模式并有 ptr16 选项、 H8SX 中级模式、 H8SX 高级模式并有 ptr16 选项、 H8SX 最高模式并有 ptr16 选项）
		long 型（H8/300H 高级模式、 H8S/2000 高级模式并没有 ptr16 选项、 H8S/2600 高级模式并没有 ptr16 选项、 H8SX 高级模式并没有 ptr16 选项、 H8SX 最高模式并没有 ptr16 选项）

（7）寄存器

表 10.9　寄存器的规格

	项　目	编译程序的规格		
1	能分配寄存器变量*[5]的寄存器	H8/300	有优化	(R3)*[1]、R4、R5、R6
			无优化	(R3)*[1]、R4、R5
		上述以外	有优化	(ER3)*[1]、ER4、ER5、ER6
			无优化	(ER3)*[1]、ER4、ER5、ER6*[4]
2	能分配到寄存器的寄存器变量*[5]的型	char、unsigned char、short、unsigned short、int、unsigned int、long*[2]、unsigned long*[2]、float*[2]、指针、参照、指向数据成员的指针、4byte 以下的结构体数据*[3]		

【注】 *1　( )内的寄存器在指定 noregexpansion 选项时，除 CPU 为 H8SX/H8S 系列以外，不分配寄存器变量。

*2　CPU 为 H8/300 系列时，不分配到寄存器。

*3　CPU 为 H8/300 系列时，可分配 2byte 以下的结构体数据。

*4　只在 CPU 为 H8SX/H8S 系列时，即使无优化，也能将寄存器变量分配到 ER6。

*5　对变量的寄存器分配，不受是否已进行了 register 存储类声明的影响。但是，在指定 enable_register 选项时，优先将进行 register 存储类声明的变量分配到寄存器。

（8）类、结构体、联合体、枚举型、位域

表 10.10　类、结构体、联合体、枚举型、位域的规格

	项　目	编译程序的规格
1	由不同型的成员存取的联合体型的成员参照	能参照，但不保证值。
2	类成员的边界调整	只有 char 型成员的类为 1 个字节调整。除此之外，为 2 个字节调整。分配方法的详细规格请参照“10.1.2(2) 复合型、类的型”。
3	int 型的位域的符号	signed int 型。
4	int 型的长度中的位域的分配顺序	从高位开始分配。*[1] *[2]
5	在 int 型的长度中已分配位域时，下次分配的位域长度超过 int 型中剩余长度时的分配方法	分配到下一个 int 型的区域。*[1]
6	位域允许的型说明符	char、unsigned char、 short、unsigned short、int、unsigned int、long、unsigned long
7	表现枚举型值的整型	int 型、unsigned char 型*[3]、char 型*[3]

【注】 *1　位域的分配方法的详细内容请参照“10.1.2（3）位域”。

*2　如果指定 bit_order=right 选项，就从低位开始分配。

*3　如果指定 byteenum 选项，就根据值为 unsigned char 型或者 char 型。

（9）限定词

表 10.11　限定词的规格

	项　目	编译程序的规格
1	对 volatile 型数据的存取种类	没有规定。

（10）声明

表 10.12　声明的规格

	项　目	编译程序的规格
1	限定基本型（算术型、结构体型、联合体型）的说明符的个数	最多能指定 16 个。

以下通过例子表示限定基本型的型个数的计算方法：

例：

(i)　int i；i是int型（基本型），限定基本型的型个数为0个。

(ii) char *f( )；f是返回指向char型（基本型）的指针型的函数型，限定基本型的型个数为2个。

（11）语句

表 10.13　语句的规格

	项　目	编译程序的规格
1	一条 switch 语句中能指定的 case 标号数	最多能指定 2147483646 个。

（12）预处理器

表 10.14　预处理器的规格

	项　目	编译程序的规格
1	条件编译的常数表达式中的单字符的字符常数和执行环境字符集的对应	预处理器语句的字符常数和执行环境字符集一致。
2	Include 文件的读取方法	用“〈”、“〉”括起来的文件从 include 选项指定的路径读取（省略时为环境变量 CH38 设定的路径）。
3	支持用双重引用符括起来的 Include 文件	支持。从当前目录读取 Include 文件。如果在当前目录中不存在该文件，就按本表 2 读取。
4	源文件的字符排列的对应（宏展开后的字符串的空格字符）	空格字符串展开为 1 个空格字符。
5	#pragma 语句的操作	请参照“10.2.1 #pragma 扩展名和关键字”。
6	_ _DATE_ _、_ _TIME_ _的值	设定编译开始时的主机定时器的值。

10.1.2　数据的内部表现

本节阐述数据型和数据的内部表现的对应。数据的内部表现由以下项目构成：

- 数据长度

  数据占有区域的大小。

- 数据的边界调整数

  是有关分配数据的地址限制，有分配到任意地址的1个字节边界调整数和分配到偶数字节的2个字节边界调整数。

- 数据范围

  表示标量（C语言）或者基本型（C++语言）值的范围。

- 数据的分配例子

  表示为复合型（C语言）或者类的型（C++语言）的元素的数据分配方法。

（1）标量型（C 语言）和基本型（C++语言）

C 语言中的标量型和 C++语言中的基本型的内部表现如表 10.15 所示：

表 10.15　标量型/基本型的内部表现

数据类型		长度（byte）	边界调整数（byte）	有/无符号	数据范围	
					最小值	最大值
1	char	1	1	有	-2^{7}(-128)	2^{7}-1(127)
2	signed char	1	1	有	-2^{7}(-128)	2^{7}-1(127)
3	unsigned char	1	1	无	0	2^{8}-1(255)
4	short	2	2	有	-2^{15}(-32768)	2^{15}-1(32767)
5	unsigned short	2	2	无	0	2^{16}-1(65535)
6	int	2	2	有	-2^{15}(-32768)	2^{15}-1(32767)
7	unsigned int	2	2	无	0	2^{16}-1(65535)
8	long	4	2	有	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
9	unsigned long	4	2	无	0	2^{32}-1 (4294967295)
10	enum（值的范围为-128～127 并且指定 byteenum 选项时）	1	1	有	-2^{7}(-128)	2^{7}-1(127)
11	enum（值的范围为 0～255 并且指定 byteenum 选项时）	1	1	无	0	2^{8}-1(255)
12	enum（上述以外）	2	2	有	-2^{15}(-32768)	2^{15}-1(32767)
13	bool *1	1	1	有	-2^{7}(-128)	2^{7}-1(127)
14	float	4	2	有	-∞	+∞
15	double, long double *2	8	2	有	-∞	+∞
16	指针（H8SX 正常模式、H8SX 中级模式、H8S/2600 正常模式、H8S/2000 正常模式、H8/300H 正常模式、H8/300）	2	2	无	0	2^{16}-1(65535)
17	指针 *3（H8/300H 高级模式）	4	2	无	0	2^{24}-1 (16777215)

	数据类型	长度（byte）	边界调整数（byte）	有/无符号	数据范围	
					最小值	最大值
18	指针 *4 （H8SX 高级模式、 H8SX 最高模式、 H8S/2600 高级模式、 H8S/2000 高级模式）	4	2	无	0	2^{32}-1 (4294967295)
19	参照 *1 （H8SX 正常模式、 H8SX 中级模式、 H8S/2600 正常模式、 H8S/2000 正常模式、 H8/300H 正常模式、 H8/300）	2	2	无	0	2^{16}-1(65535)
20	参照 *1 *3 （H8/300H 高级模式）	4	2	无	0	2^{24}-1 (16777215)
21	参照 *1 *4 （H8SX 高级模式、 H8SX 最高模式、 H8S/2600 高级模式、 H8S/2000 高级模式）	4	2	无	0	2^{32}-1 (4294967295)
22	指向数据成员的指针 *1 （H8SX 正常模式、 H8SX 中级模式、 H8S/2600 正常模式、 H8S/2000 正常模式、 H8/300H 正常模式、 H8/300）	2	2	无	0	2^{16}-1(65535)
23	指向数据成员的指针 *1 *3 （H8/300H 高级模式）	4	2	无	0	2^{24}-1 (16777215)
24	指向数据成员的指针 *1 *4 （H8SX 高级模式、 H8SX 最高模式、 H8S/2600 高级模式、 H8S/2000 高级模式）	4	2	无	0	2^{32}-1 (4294967295)
25	指向函数成员的指针 *1 *6 （H8SX 正常模式、 H8S/2600 正常模式、 H8S/2000 正常模式、 H8/300H 正常模式、 H8/300）	6	2	—	—	—
26	指向函数成员的指针 *1 *6 （H8SX 中级模式）	8	2	—	—	—
27	指向函数成员的指针 *1 *5 *6 （H8SX 高级模式、 H8SX 最高模式、 H8S/2600 高级模式、 H8S/2000 高级模式、 H8/300H 高级模式）	10	2	—	—	—

	数据类型	长度 （byte）	边界调整数 （byte）	有/无 符号	数据范围	
					最小值	最大值
28	指向虚拟函数成员的指针 *1 *6 （H8SX 正常模式、 H8S/2600 正常模式、 H8S/2000 正常模式、 H8/300H 正常模式、 H8/300）	6	2	—	—	—
29	指向虚拟函数成员的指针*1 *6 （H8SX 中级模式）	8	2	—	—	—
30	指向虚拟函数成员的指针 *1*5*6 （H8SX 高级模式、 H8SX 最高模式、 H8S/2600 高级模式、 H8S/2000 高级模式、 H8/300H 高级模式）	10	2	—	—	—

【注】 *1　只在 C++编译时有效。

*2　在指定 double=float 时，double 型的长度为 4 个字节。

*3　低位 3 个字节为地址数据，高位 1 个字节为不定值。

*4　在指定 ptr16 选项或者_ _ptr16 关键字时，长度为 2 个字节。

*5　除 H8/300H 高级模式以外，在指定 ptr16 选项时，长度为 8 个字节。

*6　用以下的类表现指向函数成员/虚拟函数成员的指针：

```
class _PMF{
  public:
    size_t delta;            // 目标的偏移值
    short index;             // 对象成员函数为虚拟函数时的虚拟函数表中的 index

    union{
        int (*_deffun)();    // 对象成员函数为非虚拟函数时的函数地址
        size_t vt_offset;    // 对象成员函数为虚拟函数时的虚拟函数表的目标中的偏移
    };
};
```

（2）复合型（C 语言）和类的型（C++语言）

本项说明 C 语言中的数组型、结构体型、联合体型以及 C++语言中的类的型的内部表现。

复合型和类的型的内部表现如**表 10.16** 所示：

表 10.16　复合型和类的型的内部表现

	数据类型	边界调整数（byte）	长度（byte）	数据的分配例子	
1	数组型	数组元素的边界调整数	数组元素的个数 ×元素长度	char a[10];	边界调整数　1byte 长度　10byte

2	结构体型	结构体成员的边界调整数中的最大值	成员长度的和
			参照“（a）结构体数据的分配方法”
				struct {
				char a,b;
				};
					边界调整数　1byte
					长度　2byte

3	联合体型	联合体成员的边界调整数中的最大值	成员长度的最大值
			参照“（b）联合体数据的分配方法”
				union {
				char a,b;
				};
					边界调整数　1byte
					长度　1byte

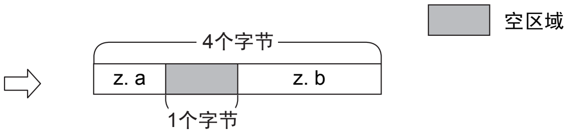
4	类的型	1）有虚拟函数:
		总是 2
		2）上述以外:
		数据成员的边界调整数中的最大值
			数据成员、指向虚拟函数表的指针、指向虚拟基类的指针的和
			参照“（c）类数据的分配方法”
				（H8S/2600 高级模式时）
				class B: public A{
				virtual void f();
				};
				class A{
				char a;
				};
					边界调整数 2byte
					长度　6byte
					边界调整数 1byte
					长度　1byte

（a）结构体数据的分配方法

- 分配结构体型的各成员时，为了符合该成员数据型的边界调整数，有可能在和前一个成员之间产生1个字节的空区域。

例：

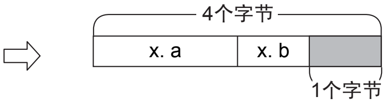
```
struct  {
    char a;
    int b;
  }z;
```



- 在结构体具有2个字节的边界调整数并且最后的成员以奇数字节结束时，作为结构体型的区域在最后还含有一个字节。

例：

```
struct  {
    int a;
    char b;
  }x;
```

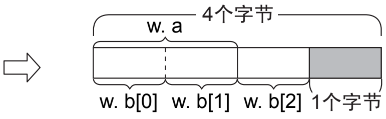


（b）联合体数据的分配方法

- 在联合体具有2个字节的边界调整数并且成员长度的最大值为奇数字节时，作为联合体型的区域在最后还含有一个字节。

例：

```
union  {
    int a;
    char b[3];
  }w;
```

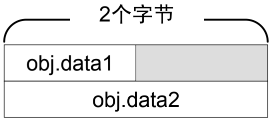


（c）类数据的分配方法

- 如果是没有基类和虚拟函数的类，就按结构体数据的分配规则分配数据成员。

例：

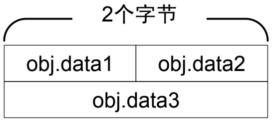
```
class A{
    char data1;
    short data2;
  public:
    A();
    int getData1(){return data1;}
}obj;
```



- 从边界调整数为1的基类派生出的类的第1个成员为1字节数据时，为了不产生空区域而分配数据成员。

例：

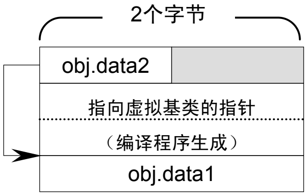
```
class A{
    char data1;
};
class B:public A{
    char data2;
    Short data3;
}obj;
```



- 类中有虚拟基类时，分配指向虚拟基类的指针。

例：

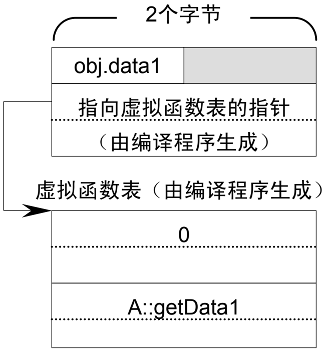
```
class A{
    short data1;
};
class B: virtual protected A{
    char data2;
}obj;
```



- 类中有虚拟函数时，编译程序生成虚拟函数表，分配指向虚拟函数表的指针。

例：

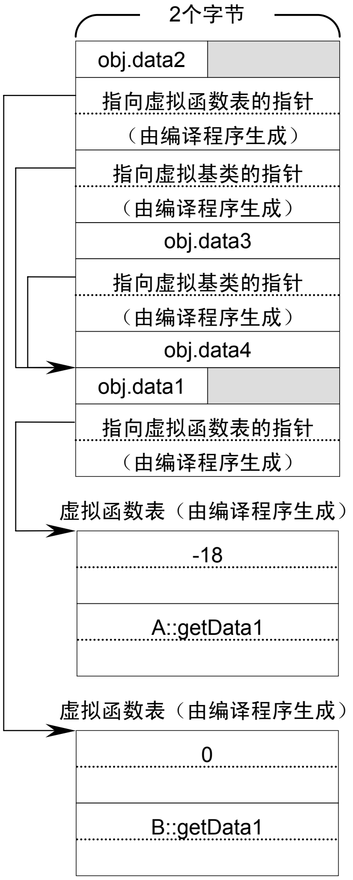
```
class A{
    char data1;
  public:
    virtual int getData1();
}obj;
```



- 有虚拟基类、基类和虚拟函数的类的例子如下所示：

  例：

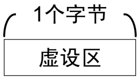
```
class A{
    char data1 ;
    virtualshort getData1();
};
class B:virtual public A{
    char data2;
    char getData2();
    short getData1();
};
class C:virtual protected A{
    int data3;
};
class D:virtual public A,public B,public C{
  public:
    int data4;
    short getData1();
}obj;
```



- 在空类的情况下，分配1个字节的虚设区。

  例：

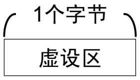
```
class A{
    void fun();
}obj;
```



- 即使在基类中有空类的空类时，虚设区也为1个字节。

  例：

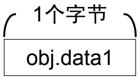
```
class A{
    void fun();
};
class B: A{
    void sub();
};
```



- 在类长度为0时，分配空类的虚设区。对于有数据成员的基类或者派生类，或者对于有虚拟函数的类，不分配虚设区。

  例：

```
class A{
    void fun();
};
class B: A{
    char data1;
}obj;
```



（3）位域

位域是通过在结构体、联合体和类中指定位宽来分配的成员。
本项说明位域特有的分配规则。

（4）位域的成员

位域成员的规格如表 10.17 所示：

表 10.17　位域成员的规格

	项　目	规格
1	位域允许的型说明符	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long
2	扩展为被声明的型时的符号处理　*1	无符号（指定 unsigned 的型）→零扩展　*2 有符号（不指定 unsigned 的型）→符号扩展

【注】 *1　使用位域的成员时，必须先将保存在位域的数据扩展为声明的型。
*2　零扩展　：扩展时给高位写 0。
符号扩展：扩展时，将位域数据的最高位解释为符号位，给高位写符号。

【注】　用带符号（signed）声明的长度为 1 位的位域数据被解释为符号。因此，能表现的值只为 0 和－1。表现 0 和 1 时，必须用无符号（unsigned）声明。

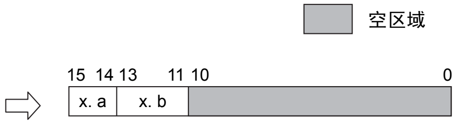
（b）位域的分配方法

根据以下 5 个规则分配位域：

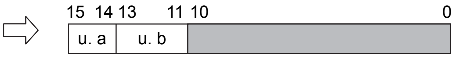
- 在区域中从左（高位侧）开始按顺序分配位域的成员。

例：

```
struct b1{
            int a:2;
            int b:3;
       }x;
```



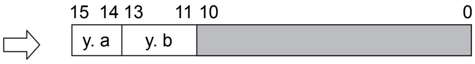
```
struct b1{
        enum E1{o,p,q} a:2;
        enum E1        b:3;
       }u;
```



- 连续有相同长度的型说明符时，尽可能分配到相同区域。

例：

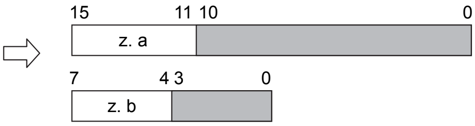
```
struct b1{
            int            a:2;
            unsigned short b:3;
       }y;
```



- 将用不同长度的型说明符声明的成员分配到下一个区域。

  例：

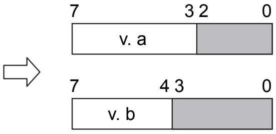
```
struct b1{
        int  a:5;
        char b:4;
    }z;
```



- 如果分配目标区域的剩余位短于下一个位域的长度，即使连续有相同长度的型说明符，剩余的区域也为未使用区域，而分配到下一个区域。

  例：

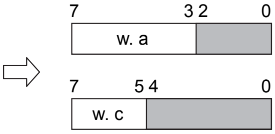
```
struct b2{
      char a:5;
      char b:4;
  }v;
```



- 如果指定位宽为0的位域成员，就将后面的成员强制分配到下一个区域。

  例：

```
struct b2{
      char a:5;
      char  :0;
      char c:3;
  }w;
```



【备考】 可从低位侧开始分配位域成员，详细内容请参照“2.2　选项解说”的 bit_order 选项或者“10.2.1　#pragma 扩展名和关键字”的#pragma bit_order。

10.1.3　浮点数的规格

（1）浮点数的内部表现

编译程序处理的浮点数的内部表现符合 IEEE 的标准格式，在此阐述 IEEE 格式的浮点数内部表现的概要。

（a）内部表现的格式

float 型用 IEEE 的单精度格式（32 位）表现；double 型和 long double 型用 IEEE 的双精度格式（64 位）表现。

（b）内部表现的结构

float 型、double 型和 long double 型的内部表现的结构如**图 10.1** 所示：

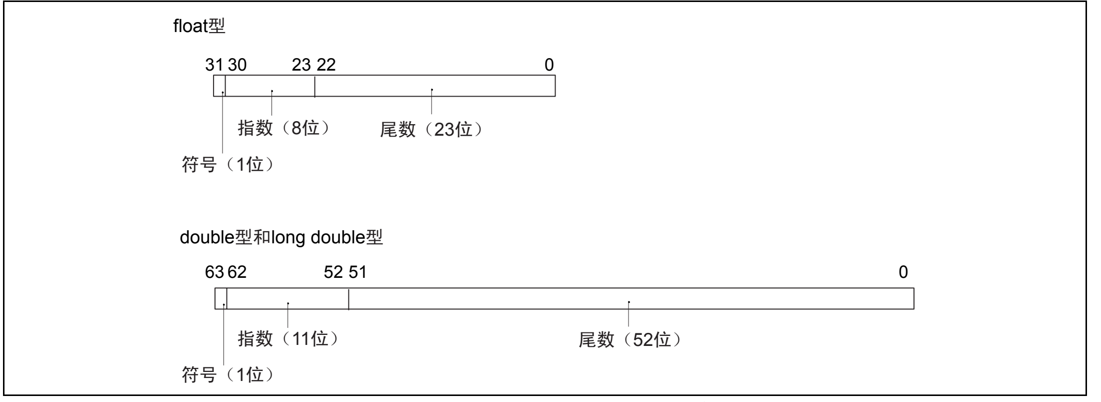


图 10.1　浮点数内部表现的结构

内部表现的各构成要素的含义如下所示：

(i)　符号

表示浮点数的符号。0表示正，1表示负。

(ii)　指数

用2的乘方表示浮点数的指数。

(iii) 尾数

对应浮点数的有效数字的数据。

（c）表现值的种类

浮点数除一般的实数值以外，还能表现无穷大等的值。浮点数表现值的种类如下所示：

(i)　正规化数

指数不为0或者所有位都不为1时，表现一般的实数值。

(ii)　非正规化数

指数为0并且尾数不为0时，表现绝对值较小的实数值。

(iii) 零

指数和尾数都为0时，表现0.0的值。

(iv) 无穷大

指数的所有位为1并且尾数为0时，表现无穷大。

(v)　非数值

指数的所有位为1并且尾数不为0时，表示得到“0.0/0.0”、“∞/∞”、“∞－∞”等不对应数值的运算结果。

决定浮点数表现值的条件如**表 10.18** 所示。

表 10.18　浮点数表现值的种类

尾数	指数		
	0	不为 0 并且所有位不为 1	所有位为 1
0	0	正规化数	无穷大
0 以外	非正规化数		非数值

【注】 非正规化数表现绝对值较小的浮点数（正规化数不能表现的范围）。但是，有效位数比正规化数少。因此，运算结果或者中途结果为非正规化数时，不保证结果的有效位数。

（2）float 型

float 型的内部表现由 1 位符号、8 位指数和 23 位尾数组成。

(i)　正规化数

符号为0（正）或者1（负），表示值的符号。

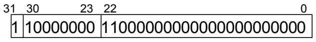
指数为1～254（2^8－2）的值。实际的指数为该值减去127，其范围为－126～127。

尾数为0～2^{23}－1的值。实际尾数假设2^{23}的位为1，后面部分被解释为有小数点的尾数。

正规化数的表现值为：

$$(-1)^{<\text{符号}>}\times 2^{<\text{指数}>-127}\times(1+<\text{尾数}>\times 2^{-23})$$

例：



符号：－

指数：10000000$_{(2)}$－127＝1　　　　(2)表示二进制数。

尾数：1.11$_{(2)}$＝1.75

值　：－1.75×2^1＝－3.5

(ii)　非正规化数

符号为0（正）或者1（负），表示值的符号。

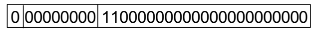
指数为0，实际指数为－126。

尾数为1～2^{23}－1，实际尾数假设2^{23}的位为0，后面部分被解释为有小数点的尾数。

非正规化数的表现值为：

$$(-1)^{<\text{符号}>}\times 2^{-126}\times(<\text{尾数}>\times 2^{-23})$$

例：



符号：＋

指数：－126　　　　(2)表示二进制数。

尾数：0.11$_{(2)}$＝0.75

值　：0.75× 2^{-126}

(iii) 零

符号为0（正）或者1（负），分别表示＋0.0、－0.0。

指数和尾数都为0。

作为值，＋0.0、－0.0都表示0.0。有关由零的符号引起的各运算中的不同功能，请参照“10.1.3（4） 浮点运算的规格”。

(iv) 无穷大

符号为0（正）或者1（负），分别表示＋∞、－∞。

指数为255（2^8－1）。

尾数为0。

(v) 非数值

指数为255（2^8－1）。

尾数为0以外的值。

【注】　没有规定有关非数值的符号和尾数（0 以外的）的值。

（3）double 型和 long double 型

double 型和 long double 型的内部表现由 1 位符号、11 位指数和 52 位尾数组成。

(i) 正规化数

符号为0（正）或者1（负），表示值的符号。

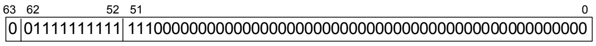
指数为1～2046（2^{11}－2）的值。实际指数为该值减去1023，其范围为－1022～1023。

尾数为0～2^{52}－1的 值。实际尾数假设2^{52}的位为1，后面部分被解释为有小数点的尾数。

正规化数的表现值为：

$$(-1)^{<\text{符号}>}\times 2^{<\text{指数}>-1023}\times(1+<\text{尾数}>\times 2^{-52})$$

例：



符号：＋

指数：1111111111 (2) －1023＝0

尾数：1.111 (2) ＝1.875

值　：1.875×2^0＝1.875

(2) 表示二进制数。

(ii) 非正规化数

符号为0（正）或者1（负），表示值的符号。

指数为0，实际指数为－1022。

尾数为1～2^{52}－1，实际尾数假设2^{52}的位为0，后面部分被解释为有小数点的尾数。

非正规化数的表现值为：

$$(-1)^{<符号>}\times 2^{-1022}\times(<尾数>\times 2^{-52})$$

例：

63	62　52	51　0
1	00000000000	1110000000000000000000000000000000000000000000000000

符号：　－

指数：　－1022

尾数：　$0.111_{(2)}$＝0.875　　　⑵表示二进制数。

值　：　0.875×2^{-1022}

(iii) 零

符号为0（正）或者1（负），分别表示＋0.0、－0.0。

指数和尾数都为0。

作为值，＋0.0、－0.0都表示0.0。有关由零的符号引起的各运算中的不同功能，请参照“10.1.3（4） 浮点运算的规格”。

(iv) 无穷大

符号为0（正）或者1（负），分别表示＋∞、－∞。

指数为2047（2^{11}－1）。

尾数为0。

(v) 非数值

指数为2047（2^{11}－1）。

尾数为0以外的值。

【注】　没有规定有关非数值的符号和尾数（0 以外的）的值。

（4）浮点运算的规格

本项说明作为 C/C++语言功能表现的浮点四则运算、以及编译时和处理 C 库函数时产生的浮点 10 进制表现和内部表现之间转换的规格。

（a）四则运算的规格

(i)　结果值的舍入方法

如果浮点四则运算结果的正确值超出内部表现的尾数有效数字，就按照以下规则进行舍入：

［1］结果值向近似该值的2个浮点数的内部表现中较近的方向舍入。

［2］结果值在近似该值的2个浮点数的正中央时，向尾数的最后位为0的方向舍入。

(ii)　溢出、下溢和无效运算时的处理

对执行时的溢出、下溢和无效运算进行以下的处理：

［1］溢出时，根据结果符号为正无穷大或者负无穷大。

［2］下溢时，根据结果符号为正零或者负零。

［3］如果将符号相反的无穷大相加、将符号相同的无穷大相减、将0和无穷大相乘、0除以0或者无穷大除以无穷大，就产生无效运算。

此时，结果为非数值。

［4］在上述任意的情况下，都设定与表示错误发生的变量errno对应的错误号。有关错误号请参照“12.3　C库函数的错误信息”。

需要检查错误时，必须根据此errno值判断错误的发生。

【注】　编译时进行常数式的运算。如果此时检测出溢出、下溢或者无效运算，就发生警告级错误。

(iii) 有关特殊值（零、无穷大、非数值）运算的注意事项：

［1］正零和负零的和为正零。

［2］同符号零的差为正零。

［3］操作数的一方或者双方包含非数值的运算结果总是为非数值。

［4］在比较运算中，正零和负零相同。

［5］操作数的一方或者双方为非数值的比较运算或者等同运算的结果对于“!=”总是为真，而对于其它总是为伪。

（b）10 进制表现和内部表现之间的转换

本项说明有关源程序中的浮点常数和内部表现之间的转换、或者通过 C 库函数的 ASCII 字符串的浮点数的 10 进制表现和内部表现之间转换的规格：

(i)　从10进制表现转换为内部表现时，首先将10进制表现转换为10进制表现的标准形式。

10进制表现的标准形式为“$\pm M\times10^{\pm N}$”，M、N的范围如下：

［1］float型的标准形式

$0\leqslant M\leqslant10^{9}-1$

$0\leqslant N\leqslant99$

［2］double型和long double型的标准形式

$0\leqslant M\leqslant10^{17}-1$

$0\leqslant N\leqslant999$

不能转换为标准形式的10进制表现为溢出或者下溢。另外，10进制表现含有多于标准形式的有效数字时，舍去低位的位数。此时，在编译时发生警告级错误，而在执行时将对应的错误号设定到变量errno。

另外，为了转换为标准形式，原来10进制表现的ASCII字符串的长度必须在511个字符以内。否则，在编译时发生错误，而在执行时将对应的错误号设定到变量errno。

从内部表现转换为10进制表现时，先转换为10进制表现的标准形式，然后根据指定格式转换为ASCII字符串。

(ii)　10进制表现的标准形式和内部表现之间的转换

10进制表现的标准形式和内部表现之间的转换在指数太大或者太小时，无法进行正确的转换。以下说明有关能正确转换的范围及其范围以外的误差的极限值。

［1］能正确转换的范围

对于以下所示的指数范围的浮点数，能正确进行“**（a）(i) 结果值的舍入方法**”所示的舍入。在此范围内不发生溢出或者下溢。

float型：　$0\leqslant M\leqslant10^{9}-1$、$0\leqslant N\leqslant13$

double型和long double型：　$0\leqslant M\leqslant10^{17}-1$、$0\leqslant N\leqslant27$

［2］误差的极限值

转换非［1］所示范围内的值时的误差和进行正确舍入时的误差的差不能超过有效数字的最小位数的0.47倍。

另外，如果超出［1］所示的范围，在转换时就有可能产生溢出或下溢。此时，在编译时发生警告级错误，而在执行时将对应的错误号设定到变量 errno。

10.1.4　运算符的评价顺序

在表达式中有多个运算符时，通过称为优先级的正数和称为用“右”或“左”表现的结合性的性质来决定这些运算符的评价顺序。

各运算符的优先级和结合性如表10.19所示。

表 10.19　运算符的优先级和结合性

优先级	运算符	结合性	适用式
1	( )　[ ]　->　.　++　-- (后缀)	左	后缀式
2	!　~　++　-- (前缀)　+　-　*　&　sizeof	右	单项式
3	(型名)	右	数据类型转换式
4	*　/　%	左	乘法式
5	+　-	左	加法式
6	<<　>>	左	移位式
7	<　<=　>　>=	左	关系式
8	==　!=	左	相等式
9	&	左	每位的 AND 式
10	^	左	每位的 XOR 式
11	|	左	每位的 OR 式
12	&&	左	逻辑 AND 运算
13	||	左	逻辑 OR 式
14	?:	左	条件式
15	=　+=　-=　*=　/=　%=　<<=　>>=　&=　|=　^=	右	赋值表达式
16	,	左	逗点式

10.2　扩展功能

编译程序支持以下的扩展功能：

- #pragma 扩展名、关键字
- 段地址运算符
- 内部函数

10.2.1　#pragma 扩展名和关键字

#pragma 扩展名和关键字的一览表如下所示：

表 10.20　存储器分配的扩展功能

	#pragma 扩展名	关键字	功能
1	#pragma stacksize	—	建立堆栈段
2	#pragma section、 #pragma abs8 section、 #pragma abs16 section、 #pragma indirect section	—	指定段的转换
3	#pragma abs8、 #pragma abs16	__abs8 __abs16	指定用短绝对寻址方式存取的变量
4	—	__near8 __near16	指定数组/结构体的地址计算长度
5	—	__ptr16	指定指针长度
6	#pragma bit_order	—	指定位域的排列顺序

表 10.21　函数的扩展功能

	#pragma 扩展名	关键字	功能
1	#pragma interrupt	__interrupt	建立中断函数
2	#pragma entry	__entry	建立入口函数
3	#pragma indirect	__indirect	指定用存储器间接进行函数调用的函数
4	—	__indirect_ex	指定用扩展存储器间接进行函数调用的函数
5	#pragma inline	__inline	指定函数的直接插入展开
6	#pragma inline_asm	—	汇编描述函数的直接插入展开
7	#pragma regsave、 #pragma noregsave	__regsave __noregsave	寄存器的保存/恢复代码的输出控制
8	—	__regparam2 __regparam3	指定用于参数的寄存器个数
9	#pragma option	—	以函数为单位指定优化选项

表 10.22　其它扩展功能

	#pragma 扩展名	关键字	功能
1	#pragma asm、 #pragma endasm	—	汇编程序嵌入功能
2	—	__asm	嵌入汇编功能
3	#pragma global_register	__global_register	分配全局变量的寄存器
4	#pragma pack 1、 #pragma pack 2、 #pragma unpack	—	指定结构体、联合体、类的边界调整数
5	—	__evenaccess	指定偶数字节存取
6	#pragma address	—	将变量分配到指定的绝对地址

【注】对于相同函数或者变量，最初指定的关键字/pragma 有效。
一旦指定属性后，就不能再指定别的属性。
另外，对于相同变量，不能将#pragma 扩展名/关键字混在一起指定。

- 错误例子

```
// 不能通过原型声明和定义指定别的关键字
_ _regsave void func(void);
_ _interrupt void func(void) {}

// 同样也不能用 pragma 指定别的属性
#pragma regsave func
_ _interrupt void func(void) {}
```

要对相同函数或者变量指定多个属性时，必须在声明/定义的阶段统一用关键字指定。

- 能正确编译的例子

```
// 能通过声明（或者定义）统一指定关键字
_ _regsave  _ _interrupt void func(void);
void func(void) {}
```

（1）有关存储器分配的扩展功能

堆栈段的建立

#pragma stacksize

格　式	`#pragma stacksize <常数>`	
说　明	建立段名为 S、长度为<常数>的堆栈段。	

例	`#pragma stacksize 100`	<代码展开例>
		`.SECTION   S, STACK`
		`.RES.W     50`

备　考	• 对长度指定的<常数>必须为偶数。 • `#pragma stacksize` 在文件中只能指定一次。	

段的转换指定

#pragma section
#pragma abs8 section
#pragma abs16 section
#pragma indirect section

格　式

```
#pragma section [{<名称> | <数值>}]
#pragma abs8 section [{<名称> | <数值>}]
#pragma abs16 section [{<名称> | <数值>}]
#pragma indirect section [{<名称> | <数值>}]
```

说　明

转换编译程序输出的段名。

默认名和转换后的段名如表 10.23 所示。

表 10.23　段转换功能和段名

对象区			指定方法	默认名	转换后
1	程序区		#pragma section <xx>	P *1	P<xx>
2	常数区			C *1	C<xx>
3	初始化数据区			D *1	D<xx>
4	未初始化数据区			B *1	B<xx>
5	8 位绝对地址	常数区	#pragma abs8 section <xx>	$ABS8C	$ABS8C<xx>
6		初始化数据区		$ABS8D	$ABS8D<xx>
7		未初始化数据区		$ABS8B	$ABS8B<xx>
8	16 位绝对地址	常数区	#pragma abs16 section <xx>	$ABS16C	$ABS16C<xx>
9		初始化数据区		$ABS16D	$ABS16D<xx>
10		未初始化数据区		$ABS16B	$ABS16B<xx>
11	存储器间接	函数地址区	#pragma indirect section <xx>	$INDIRECT $EXINDIRECT	$INDIRECT<xx> $EXINDIRECT<xx>

【注】*1　能通过 section 选项更改默认段名。
　　　　如果省略<名称>或<数值>，以后就恢复到默认段名。

例

```
#pragma section abc
int a;                  /*  将 a 分配到段 Babc       */
const int c=1;          /*  将 c 分配到段 Cabc       */
void f(void)            /*  将 f 分配到段 Pabc       */
{
    a=c;
}
#pragma section
int b;                  /*  将 b 分配到段 B          */
void g(void)            /*  将 g 分配到段 P          */
{
  b=c;
}
```

备　考

- 必须在函数定义以外的位置声明#pragma section、#pragma abs8 section、#pragma abs16 section、#pragma indirect section。
- 1 个文件中能声明的段名最多为 64 个。

指定用短绝对地址存取的变量

#pragma abs8
#pragma abs16
_ _abs8
_ _abs16

格　式

```
#pragma abs8 [(]<变量名> [,…] [)]
#pragma abs16 [(]<变量名名> [,…] [)]
_ _abs8 <型说明符> <变量名>
<型说明符> _ _abs8 <变量名>
_ _abs16 <型说明符> <变量名>
<型说明符> _ _abs16 <变量名>
```

说　明

声明分配在 8/16 位绝对地址区的变量。

- 将用#pragma abs8 和_ _abs8 声明的变量输出到段名“$ABS8C”、“$ABS8D”、“$ABS8B”，生成用 8 位绝对地址(@aa:8)存取的代码。
- 将用#pragma abs16 和_ _abs16 声明的变量输出到段名“$ABS16C”、“$ABS16D”、“$ABS16B”，生成用 16 位绝对地址(@aa:16)存取的代码。
- 有关段名的转换方法，请参照前项的#pragma abs8 section 和#pragma abs16 section。

例

```
#pragma abs8(c1)
#pragma abs16(i1)
char c1;                    /* 将 c1 分配到段名$ABS8B                  */
int i1;                     /* 将 i1 分配到段名$ABS16B、               */
char _ _abs8 c2;            /* 将 c2 分配到段名$ABS8B、                */
char _ _abs16 i2;           /* 将 i2 分配到段名$ABS16B、               */
long l;                     /* 将 l 分配到段名 B、                     */
void f(void){
  c1=c2=10;          /* 用 8 位绝对地址存取 c1,c2                      */
  i1=i2=100;         /* 用 16 位绝对地址存取 i1,i2                     */
  l=1000;            /* 用 32 位绝对地址存取 l                         */
}
```

备　考

- 以#pragma abs8、#pragma abs16 声明后的变量定义/变量声明为对象。
- 只能用#pragma abs8、_ _abs8、#pragma abs16、_ _abs16 声明的变量为分配到静态区的变量。
- 在#pragma abs8、#pragma abs16 语句的 1 行中能声明的变量数最多为 63 个。
- 在不使用#pragma abs8 section <XX> 或#pragma abs16 section <XX>时，将用#pragma abs8、_ _abs8、#pragma abs16、_ _abs16 声明的变量输出到段名“$ABS8C”、“$ABS8D”、“$ABS8B”、“$ABS16C”、“$ABS16D” 或者“$ABS16B”。连接时，必须将该段分配到 8 位/16 位绝对地址区。
- 在用#pragma abs8 声明的变量不为 1 个字节存取对象时产生错误。
  边界调整数为 1 的变量、数组、结构体为对象。

指定数组/结构体的地址计算长度

_ _near8
_ _near16

格　式

<型说明符> _ _near8 <变量名>
_ _near8 <型说明符> <变量名>
<型说明符> _ _near16 <变量名>
_ _near16 <型说明符> <变量名>

说　明

指定能用 8 位或者 16 位进行地址计算的数组/结构体。
在指定_ _near8 时，用低位 1 个字节对数组/结构体进行地址计算。
在指定_ _near16 时，用低位 2 个字节进行地址计算。

例

未指定_ _near8 时

```
struct a{
    short a1;
    short a2,a3;
};
struct a aa[10];
void f(){
   int i;
   for(i=0;i<11;i++)
      aa[i].a1 = 0;
}
  <代码展开例>
   MOV.L     #_aa,ER1
   SUB.L     ER0,ER0
Ld:
   MOV.W     R0,@ER1
   INC.W     #H'1,E0
   ADDS.L    #H'4,ER1
   INC.L     #H'2,ER1
   CMP.W     #H'B,E0
   BLT       Ld:8
   RTS
```

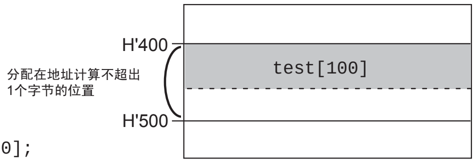
指定_ _near8 时

```
struct a{
    short a1;
    short a2,a3;
};
struct a _ _near8 aa[10];
void f(){
   int i;
   for(i=0;i<11;i++)
      aa[i].a1 = 0;
}
  <代码展开例>
   MOV.L     #_aa,ER1
   SUB.L     ER0,ER0
Ld:
   MOV.W     R0,@ER1
   INC.W     #H'1,E0
   ADD.B     #H'6,R1L
   CMP.W     #H'B,E0
   BLT       Ld:8
   RTS
```

备　考

- 必须分别将指定__near8、_ _near16 的数组/结构体分配到 8 位或者 16 位地址计算不溢出的位置。
- 如果不将指定_ _near8、_ _near16 的数组/结构体分配到正确的位置，就在连接时输出错误。
- 如果超出 8 位或者 16 位的地址边界值而分配变量，就不保证执行时的运行。必须解除_ _near8、_ _near16 的指定。

```
struct b{
   char buffer1;
   char buffer2;
};
struct b __near8 test[100];
```



2 个字节的指针长度的指定

_ _ptr16

格　式　　<型说明符> _ _ptr16 <*>

说　明　　将指针的长度指定为 2 个字节。

因为用带符号的 2 个字节指定指针值，所以需要将存取的目标分配到 16 位绝对地址区。

例

不指定_ _ptr16 时

```
_ _abs16 int a;
int *b;

func()
{
  b = &a;
}
```

<代码展开例>

```
_func:
  mov.l    #_a,er0
  mov.l    er0,@_b:32
```

指定_ _ptr16 时

```
_ _abs16 int a;
int _ _ptr16 *b;

func()
{
  b=(int _ _ptr16 *)&a;
}
```

<代码展开例>

```
_func:
  mov.l     #_a,er0
  mov.w     r0,@_b:16
```

备　考　　必须在一元运算符*前指定本关键字。

本关键字只在 H8SX 高级模式、H8SX 最高模式、H8S/2600 高级模式和 H8S/2000 高级模式时有效。

位域排列顺序的指定

#pragma bit_order

格　式　　`#pragma bit_order　[{left | right}]`

说　明　　指定位域排列顺序的转换。

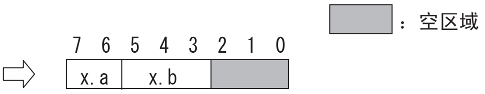
在指定 `left` 时从最高位侧开始分别分配成员;在指定 `right` 时从最低位侧开始分别分配成员。

默认设定时，解释为 `bit_order` 选项。

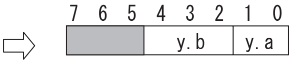
如果省略 `left` 或者 `right` 指定`#pragma bit_order`，该行的后面部分就被解释为 `bit_order` 选项。

例

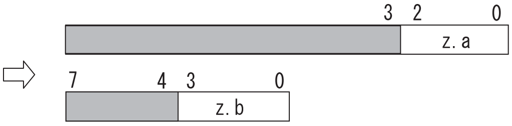
```
#pragma bit_order left
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}x;
```



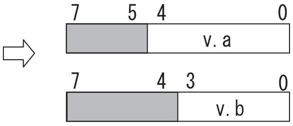
```
#pragma bit_order right
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}y;
```



```
// 为不同长度的成员时
#pragma bit_order right
typedef struct{
  unsigned int  a:3;
  unsigned char b:4;
}z;
```



```
// 超过型的长度时
#pragma bit_order right
typedef struct{
  unsigned char a:5;
  unsigned char b:4;
}v;
```



备　考　　只要不转换排列顺序，指定的位域排列顺序就有效。

也能从编译程序选项指定位域的排列顺序，详细内容请参照“2.2.2　目标选项”。

位域的详细内容请参照“10.1.2（3） 位域”。

（2）有关函数的扩展功能

中断函数的建立

#pragma interrupt
_ _interrupt

格　式　`#pragma interrupt [(]<函数名>[(中断规格)][, …][)]`
`_ _interrupt[(中断规格)] <型说明符> <函数名>`
`<型说明符> _ _interrupt[(中断规格)] <函数名>`

说　明　用`#pragma interrupt` 声明成为中断函数的函数。
中断规格一览表如表 10.24 所示。

表 10.24　中断规格一览表

项目	格式	选项	指定内容
1　堆栈转换指定	sp=	{<变量> |&<变量> |<常数> |<变量>+<常数> |&<变量>+<常数> }	用变量或者常数指定新堆栈的地址 <变量>：变量（指针型） &<变量>：变量（目标型）的地址 <常数>：常数值
2　陷阱指令返回指定	tn=	<常数>	用 TRAPA 指令指定结束 常数值（陷阱向量号）
3　中断函数结束指定	sy=	{<函数名> |<常数> |$<函数名> }	用转移到中断函数的转移指令指定结束 <函数名>：中断函数名 <常数>　：绝对地址 $<函数名>：不带下划线(_)的中断函数名
4　向量表指定	vect=	<向量号>	分配中断函数地址的向量号

- 在函数处理的前后使用通过`#pragma interrupt` 声明的函数时，保证 H8/300 时的 R0、R1、（R2 regparam=3 时）或者其它 CPU 时的 ER0、ER1、（ER2 regparam=3）寄存器，用 RTE 指令返回。
- 如果指定陷阱指令返回（tn=），就用 TRAPA 指令返回。

例：

```
extern char STK[100];
#pragma interrupt ( f(sp=STK+100, tn=2) )
                       *1       *2
_ _interrupt(sp=STK+100, tn=2) void g(void);
                 *1       *2
```

*1　将 STK+100 设定为中断函数“f”和“g”使用的堆栈指针。

*2　在结束中断函数时，通过 TRAPA #2 开始陷阱异常处理。开始陷阱异常处理时的 SP 如图 10.2，必须在陷阱例程侧使用 RTE 指令在中断函数内恢复 PC、CCR（条件码寄存器）、EXR（扩展寄存器：只支持 H8SX,H8S/2600,H8S/2000）的内容。

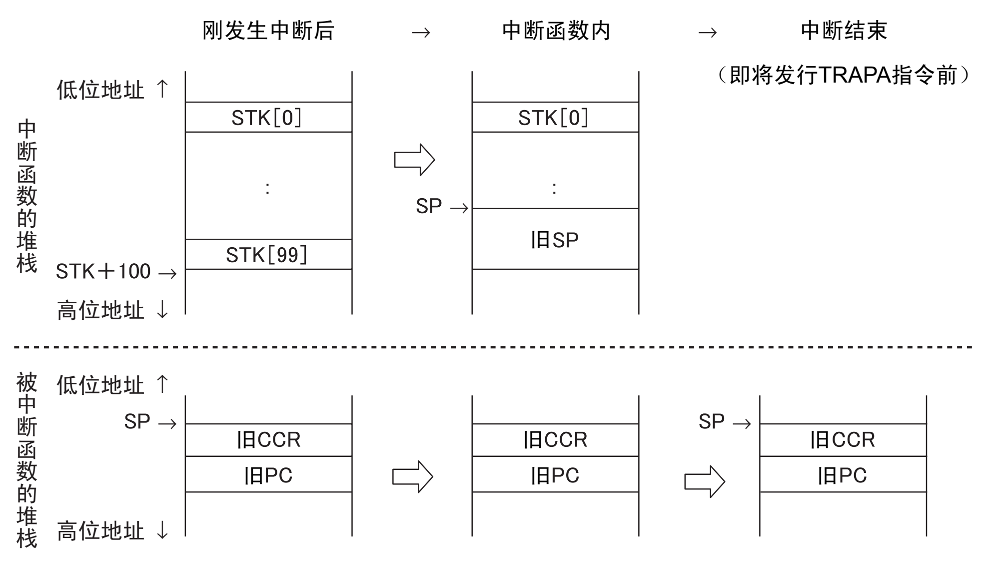


图 10.2　中断函数的堆栈使用例

- 如果指定中断函数结束（sy＝），就通过 JMP 指令转移到指定的地址。除了给结束中断函数指定的函数名指定“函数名”以外，还能指定“$函数名”。指定“$函数名”时，不给外部名为函数名的前头附加下划线（_）。
  例：
  ```
  #pragma interrupt (f1(sy=$f2))    /* 不附加下划线（_）          */
  void f2(void)                     /* 通过 JMP @f2:24 返回       */
  {
      :
  }
  ```

- 如果指定向量表（vect＝），就将该函数地址分配到指定的向量表序号。
  例：（cpu=300 时）
  ```
  #pragma interrupt (f2(vect=4))    /* 将函数 f2 的地址分配到     */
  void f1(void)                     /* 地址 8（向量号 4）         */
  {
      :
  }
  ```

- 在不指定中断规格时，作为单纯的中断函数处理。

备　考

- 以声明后的函数定义或者函数声明为对象。

```
#pragma interrupt (A::f)/* 以#pragma interrupt 声明后的         */
                        /* 函数定义或者声明为对象               */
class A{
public:
  static void f(void);  /* 将静态成员函数作为中断函数处理       */
};
void A::f(void)
{
  …
}
```

- 能定义为中断函数的函数只能是全局函数和静态成员函数。另外，函数的返回数据型只能是 void，不能指定 return 语句的返回值。如果指定，就输出错误。

```
#pragma interrupt(f1(sp=100),f2)
void f1(void)           /*   正确声明                              */
{
   …
}
int f2(void)            /* 如果函数的返回型不为 void，就出错       */
{
   …
}
```

- 不能在程序中调用声明为中断函数的函数。如果调用，就输出错误。但是，如果在没有声明中断函数的程序中调用定义为中断函数的函数，就不输出错误。此时，不能保证执行时的运行。

```
#pragma interrupt(f1)
void f1(void)
{
   …
}
int f2(void)            /* 因为函数 f1 为中断函数，所以出错 */
{
   f1();
}
```

- 对于声明为中断函数的函数，除了明确的函数调用以外，能进行函数的参照。

```
#pragma interrupt f
void f(void)
 {
   …
}
void (*VTBL)(void)={f};
                  /*   能正常编译函数调用以外的参照          */
```

- 在#pragma interrupt 语句的 1 行中能声明的函数最多为 63 个。能重复指定堆栈转换指定和陷阱指令返回指定、堆栈转换指定和中断函数结束指定。如果给中断函数指定堆栈转换指定，编译列表的符号分配信息的 Linkage Area Size 也包含用于计算旧 SP 和 SP 的 ER0（H8/300 时为 R0）的保存区容量。

入口函数的建立

#pragma entry
_ _entry

格　式

```
#pragma entry <函数名>[(<entry 规格>)]
_ _entry[(<entry 规格>)] <型说明符> <函数名>
<型说明符> _ _entry[(<entry 规格>)] <函数名>
    <entry 规格>：{sp=<常数> | vect=<向量号>}
```

说　明

将<函数名>指定的函数作为入口函数处理。

- 如果指定 sp，就在入口函数的入口处输出堆栈指针的初始设定码。将 sp 指定的<常数>用作堆栈指针的初始值。

```
例：cpu=300                           <代码展开例>
  #pragma entry INIT(sp=0x8000)        .SECTION P,CODE
  void INIT()                         _INIT:
                                       MOV.W      #H'8000,SP

      :                                        :
  }
```

- 没有指定 sp 时，就将用#pragma stacksize 建立的堆栈段的段结束地址用作堆栈指针的初始值。

```
例：cpu=300                           <代码展开例>
  #pragma stacksize 100               .SECTION S,STACK
  #pragma entry INIT                  .RES.W   50
  void INIT()                         .SECTION P,CODE
  {                                 _INIT:
       :                               MOV.W   #STARTOF S+SIZEOF S,SP
  }                                                :
```

- 如果没有 sp 指定并且也没有#pragma stacksize 声明，就生成长度为 0 的 S 段，并将 S 段的结束地址用作堆栈指针的初始值。必须在程序中进行#pragma stacksize 声明或者在连接时用 start 选项指定，以便使段 S 分配到正确的地址。

```
例：cpu=300                           <代码展开例>
  #pragma entry INIT                   .SECTION S,STACK
                                              ; 生成长度为 0 的 S 段
  void INIT()                          .SECTION P,CODE
  {                                   _INIT:
      :                                MOV.W    #STARTOF S + SIZEOF S,SP
  }                                                :
```

- 如果指定 vect，就将该函数地址分配到对应指定的向量号的地址。

```
例：cpu=300
  #pragma entry INIT(vect=0)   /* 将函数 INIT 的地址分配到地址 0 */
  void INIT()
  {                                     <代码展开例>
      :                                .SECTION  $VECT0,data,locate=0
  }                                    .DATA.W   _INIT
```

- 抑制入口函数的入口/出口的寄存器保存/恢复代码的输出。

- 如果给 CPU 种类指定 H8SX 并且通过选项或者环境变量更改 SBR 值，本扩展功能就自动设定到 SBR。

  例：cpu=H8SXA

```
//指定-SBR=0xFF00 后编译
#pragma entry INIT                    <代码展开例>
void INIT()                           .SECTION P,CODE
                                      _INIT:
    :                                 MOV.L            #H’FF00, ER3
 }                                    LDC.L            ER3,SBR
                                        :
```

备　考

- 必须在声明<函数名>前指定#pragma entry <函数名>。
- 能对声明/定义指定关键字，但是在对函数声明指定关键字时，不能指定 SP、vect。
- 在整个装入模块中只能指定 1 个入口函数。

指定用存储器间接进行函数调用的函数

#pragma indirect
_ _indirect

格　式

```
#pragma indirect [(]<函数名>[(vect=<向量号>)][,…] [) ]
<型说明符> _ _indirect[(vect=<向量号>)] <函数名>
_ _indirect[(vect=<向量号>)] <型说明符> <函数名>
```

说　明

用#pragma indirect、_ _indirect 声明存储器间接（@@aa:8）调用的函数。

- 用#pragma indirect、_ _indirect 声明的函数以“JSR @@$函数名:8”的形式调用。
  另外，如果指定 vect，就将该函数的地址分配到与指定向量对应的地址。在正常模式且 H8/300 时，向量号为 0 到 127，否则为 0 到 63。
  没有指定 vect 时，对于存储器间接调用声明的函数定义，将“$函数名”的标号和函数地址作为用于存储器间接调用的地址表，确保在段名“$INDIRECT”中。
- 段名的转换方法请参照“10.2.1（1）有关存储器分配的扩展功能”的#pragma indirect section。

例

```
(cpu=300)
_ _indirect(vect=5) char f(void);   /* 将函数 f 的地址分配到         */
char f(void)                        /* 地址 10。                    */
{
  …
}
#pragma indirect (g)
unsigned char g(void)      /* 在$indirect 段中生成$g,                */
{                          /* 保存函数 g 的地址。                     */
  …
}
void sub( )
{
  f();                     /* 用@@$f:8 存储器间接调用函数。            */
  g();                     /* 用@@$g:8 存储器间接调用函数。            */
}
```

备　考

- 以最先出现在#pragma indirect 声明后的同名的函数定义/函数声明为对象。
- 在#pragma indirect 语句的 1 行中能声明的函数最多为 63 个。
- 在正常模式且 H8/300 时，#pragma indirect、_ _indirect 中能声明的函数最多为 128，否则为 64 个。
  在连接时，必须将没有指定 vect 时生成的地址表的段($INDIRECT)分配到地址 0x0000～0x00FF。
- 通过声明#include <indirect.h>，能将运行时例程的调用码用于存储器间接调用。要选择进行存储器间接调用的运行时例程时，必须在 indirect.h 中将需要的#pragma indirect 语句以外部分变成注释行。

指定用扩展存储器间接进行函数调用的函数指定

_ _indirect_ex

格　式　<型说明符> _ _indirect_ex[(vect=<向量号>)] <函数名>
_ _indirect_ex[(vect=<向量号>)] <型说明符> <函数名>

说　明　用_ _indirect 声明扩展存储器间接（@@vec）调用的函数。
以“JSR @@$$函数名:7”的形式调用用_ _indirect_ex 声明的函数。
另外，如果指定 vect，就将该函数的地址分配到与指定向量号对应的地址。向量号为 128 到 255。
没有指定 vect 时，对于扩展存储器间接调用声明的函数定义，将“$$函数名”的标号和函数地址作为用于扩展存储器间接调用的地址表，确保在段名“$EXINDIRECT”中。

例

```
(CPU=h8sxa)
_ _indirect_ex(vect=128) char fl(void); /* 将函数 fl 的地址分配到
                                           地址 0x200。*/
char fl(void)
{
}
void subl(void)
{
    fl( );  /* 通过@@$$fl:7 扩展存储器间接调用函数 */
}
```

备　考
- 本关键字的指定只在 CPU 种类为 H8SX 时有效。
- 能用_ _indirect_ex 声明的函数最多为 128 个。
  连接时，在 H8SX 正常模式中，必须将没有指定 vect 时生成的地址表的段（$EXINDIRECT）分配到地址 0x0100～0x01FF；在 H8SX 中级模式、H8SX 高级模式和 H8SX 最高模式中，分配到地址 0x000200～0x0003FF。

函数的直接插入展开

#pragma inline
_ _inline

格　式

```
#pragma inline[(]<函数名>[,…][)]
_ _inline <型说明符> <函数名>
<型说明符> _ _inline <函数名>
```

说　明

用`#pragma inline`声明直接插入展开的函数。

如果调用由`#pragma inline`声明的函数，就不输出由 JSR、BSR 指令调用函数的代码，而将函数的代码直接展开到所调用的地址。

例

```
#pragma inline (f)          /* 将函数 f 声明为直接插入展开。          */
int a,b,c;
int f(int x,int y)
{
  return x+y;
}
void sub(void)
{
  a=f(b,c);                 /* 直接展开为 a=b+c 的代码。              */
}
```

备　考

- 以最先出现在`#pragma inline`声明后的同名的函数定义/函数声明为对象。
- 在`#pragma inline`语句的 1 行中能声明的函数最多为 63 个。
- 用`#pragma inline`、`_ _inline`声明的函数满足以下任一条件时，不进行直接插入展开。
  - 函数定义在指定`#pragma inline`、`_ _inline`前。
  - 具有可变参数。
  - 参照参数的地址。
  - 实际参数和形式参数的型不一致。
  - 超出直接插入展开的限制长度。
  - 通过展开对象函数的地址进行调用。
- 如果指定`#pragma inline`、`_ _inline`，就生成外部定义。在各源程序文件中记述了直接插入函数的实体时，必须给函数的声明指定`static`，此时不生成外部定义。对于`inline`(C++语言）指定的函数，不生成外部定义。

汇编记述函数的直接插入展开

#pragma inline_asm

格　式　　`#pragma inline_asm [(]<函数名>[,…][)]`

<函数名>:不可指定 C++成员函数或者重载函数

说　明　　直接插入展开用#pragma inline_asm 声明的汇编记述函数。

因为汇编程序嵌入式直接插入函数的参数和一般的函数调用相同，被设定到寄存器或者堆栈，所以能从 inline_asm 函数参照。必须将汇编程序嵌入式直接插入函数的返回值设定到(E)R0。

例

```
#pragma inline_asm(shlu)
extern unsigned int x;
static unsigned int shlu(unsigned int a)
{                                          ; 删除函数 shlu。
  SHLL.W    R0
  BCC      ?L1
  SUB.W     R0,R0
 ?L1:                                      ;局部标号以?开始。
}
void main(void)
{
  x = shlu(x);                             /*在 main 函数内进行直接插入展开  。*/
}
```

备　考

- 使用本功能时，必须用目标格式指定选项 code=asmcode 进行编译。
- 以出现在#pragma inline_asm 声明后的函数定义为对象。
- 必须在定义函数本体前指定#pragma inline_asm。
- 对于#pragma inline_asm 指定的函数，也生成外部定义。在各源程序文件中记述了直接插入函数的实体时，必须给函数的声明指定 static，此时不生成外部定义。
- 在汇编程序嵌入式直接插入函数中使用标号时，必须使用局部标号。局部标号的详细内容请参照“第 11 章　汇编程序的语言规格”。
- 如果汇编程序嵌入式直接插入函数中使用从 ER2 到 ER6 的寄存器，就需要在汇编程序嵌入式直接插入函数的前头和最后保存和恢复这些寄存器。
- 不能在汇编程序嵌入式直接插入函数的最后记述 RTS。
- 在使用本功能时，对于编译程序输出的汇编程序，有可能在交叉汇编程序中出现“402 ILLEGAL VALUE IN OPERAND”的错误，这是由于生成含有汇编程序嵌入位置的无转移距离的代码。必须在汇编时对交叉汇编程序指定优化选项。另外，必须对应程序，使用 JMP 指令修正编译程序输出的汇编程序，以便满足转移距离。

例：

```
（修正前）                            （修正后）
    :                                    :
  BEQ L1              →              BNE Ld
    :                                JMP L1
                                  Ld:
```

寄存器保存/恢复代码的控制功能

#pragma regsave
#pragma noregsave
_ _regsave
_ _noregsave

格　式

```
#pragma regsave [(]<函数名>[,…][)]
#pragma noregsave [(]<函数名>[,…][)]
_ _regsave <型说明符> <函数名>
<型说明符> _ _regsave <函数名>
_ _noregsave <型说明符> <函数名>
<型说明符> _ _noregsave <函数名>
```

说　明

用#pragma regsave、_ _regsave、#pragma noregsave、_ _noregsave 控制寄存器的保存/恢复代码。

- 用#pragma regsave、_ _regsave 声明的函数与函数中是否使用寄存器无关，在全部函数的入口生成保存函数调用前后保证值的寄存器的代码，在出口生成恢复代码。另外，不分配跨越函数调用的且函数调用前后保证值的寄存器。
- 用#pragma noregsave、_ _noregsave 声明的函数与函数中是否使用寄存器无关，不生成寄存器的保存/恢复代码。
- 调用#pragma noregsave、_ _noregsave 声明的函数时，不分配不跨越函数调用的保证值的寄存器。

例

（用 CPU=2600a 编译）

```
#pragma regsave (f,g)      /* 声明寄存器保存/恢复代码的生成。          */
#pragma interrupt g        /* 函数 g 为中断函数。                      */
void f(void){}             /* 保存/恢复 ER2～ER6。                     */
void g(void){}             /* 保存/恢复 ER0～ER6。                     */
```

备　考

- 以最先出现在#pragma regsave、#pragma noregsave 声明后的函数定义或者函数声明为对象。
- 在#pragma regsave/noregsave 语句的 1 行中能声明的函数最多为 63 个。
- 在将_ _noregsave 或者#pragma noregsave 指定函数的地址赋值到指向函数的指针时，通过函数指针的函数调用为一般函数调用。即有可能分配跨越指针的函数调用的保证值的寄存器，该寄存器的值有可能由于_ _noregsave 或者#pragma noregsave 指定函数而被破坏。

  例：

```
#pragma noregsave f
void (*p)(void);
int sub(void)
{
   int a=8;  // 假设将 a 分配到 R4
   p=f;
   f();      //  noregsave 函数调用（保存 R4，调用 f 后恢复 R4）
   (*p)();   //  一般函数调用（不保存/恢复 R4）
   return a;
}
```

指定用于参数的寄存器个数

_ _regparam2
_ _regparam3

格　式　　<型说明符> _ _regparam2 <函数名>
<型说明符> _ _regparam3 <函数名>

说　明　　指定用于参数的寄存器个数。
由_ _regparam2 指定的函数使用 ER0、ER1（H8/300 时为 R0、R1）；由_ _regparam3 指定的函数使用 ER0、ER1、ER2（H8/300 时为 R0、R1、R2）。

例

```
void _ _regparam2 func1(long a, int b, int c, long d);
void _ _regparam3 func2(long a, int b, int c, long d);
int long a; int b; int c; long d;
void main(void)
{                         /* cpu=2600a                    */
       ：                  /* 变量的分配情况                 */
   func1(a, b, c, d);     /* long a ： ER0                */
      ：                  /* int b  ： E1                 */
       ：                  /* int c  ： R1                 */
      ：                  /* long d ： stack              */
      ：
   func2(a, b, c, d);     /* long a ： ER0                */
      ：                  /* int b  ： E1                 */
      ：                  /* int c  ： R1                 */
}                         /* long d ： ER2                */
```

备　考　　因为本关键字不能指定在<型说明符>前，所以必须指定在函数名前。

选项的函数单位指定

#pragma option

格　式　`#pragma option [<选项串>]`

说　明　用`#pragma option`将通过选项串指定的选项设定为有效。

指定的选项适用于文件的结束或者没有<选项串>的`#pragma option`设定的部分。

如果指定 `pragma option <关键字>`，就进行关键字指定的优化。可使用的优化如表 10.25 所示，各优化选项请参照“第 2 章　C/C++编译程序的操作方法”。

表 10.25　可使用的优化选项

	选项指定方法	选项解除方法
1	case = { auto | ifthen | table }	无
2	cmncode	nocmncode
3	cpuexpand	nocpuexpand
4	macsave	nomacsave
5	regexpansion	noregexpansion
6	optimize	nooptimize
7	speed = { speed 子选项 }	无
8	sbr = {地址}	无

另外，speed 子选项如下：

表 10.26　可指定的 speed 子选项

	选项指定方法	选项解除方法
1	register	noregister
2	shift	noshift
3	loop	noloop
4	switch	noswitch
5	Inline	noinline
6	struct	nostruct
7	expression	noexpression

- 如果指定没有<选项串>的`#pragma option`，就忽视现在为止指定的`#pragma option` <选项串>，而变为命令行指定的选项。

例

```
#pragma option speed
void func(void){                    //speed 选项有效。
    :
}
#pragma option cpuexpand
void test(void){                    //speed,cpuexpand 有效。
    :
}
#pragma option
void sub1(void){                    // 按照命令行的指定。
    :
}
```

备　考　H8SX、H8S（没有指定 Legacy=v4 选项）不能指定`#pragma option speed=inline=<数值>`。

另外，如果在 H8SX 时指定`#pragma option speed=inline=<数值>`，就当作指定`#pragma option speed=inline`。

（3）其它扩展功能

汇编程序嵌入功能

#pragma asm

格　式

```
#pragma asm
     <汇编程序指令串>
#pragma endasm
```

说　明

能在用`#pragma asm`～`#pragma endasm`括起来的范围内记述汇编程序指令串。
编译程序将用`#pragma asm`～`#pragma endasm`括起来的指令串展开到编译程序生成的目标码中。

例

```
void func(void)
{
#pragma asm
     CLRMAC                        ;将 MAC 寄存器设定为 0。
#pragma endasm
        :
}
```

备　考

- 必须在编译时用 `code=asmcode` 选项指定汇编程序的输出。
  没有指定时，忽视包含`#pragma asm`、`#pragma endasm` 的汇编程序指令串。
- 编译程序不检查汇编程序指令串的语法和对编译程序生成代码的影响。另外，如果在编译时指定 `optimize=1` 选项或者 `speed` 选项，汇编程序指令串的展开内容和展开位置有可能与实际的指定不一致。使用汇编程序嵌入功能时，必须充分确认输出码和程序的运行。
- 不能嵌套指定`#pragma asm`～`#pragma endasm`。如果有嵌套，就输出错误。
- 在使用本功能时，对于编译程序输出的汇编程序，有可能在交叉汇编程序中发生“402 ILLEGAL VALUE IN OPERAND”的错误，这是由于生成含有汇编程序嵌入位置的无转移距离的代码。必须在汇编时对交叉汇编程序指定优化选项。另外，必须对应程序，使用 JMP 指令修正编译程序输出的汇编程序，以便满足转移距离。
- 通过选择语句或者重复语句指定`#pragma asm`～`#pragma endasm` 时，需要用复合语句｛｝将包含`#pragma asm`、`#pragma endasm` 的汇编程序指令串括起来。否则，就不能保证结果。

  例：

```
while(a==0)
{               …………… 必须指定。
#pragma asm
  <汇编程序指令串>
#pragma endasm
}               …………… 必须指定。
```

_ _asm

格　式

```
_ _asm{
       ...
  }
```

说　明

能在用_ _asm{ 和 }括起来的范围内记述汇编语言程序。

将用_ _asm{ 和 }括起来的范围称为_ _asm 块。_ _asm 块的语言规格如下：

（1）语法

- 编译程序将_ _asm 块作为 C/C++语言的语句处理。
  能在记述 C/C++语言的语句位置记述_ _asm 块，但是不能在函数以外的位置或者 C 语言的复合语句变量声明前记述。
- 1 行能记述 1 条指令。
- 不能将一条指令写为多行。
  在汇编程序中，如果在规定的位置指定“+”记号，就能在下一行继续记述指令，而在_ _asm 块内“+”记号无效。
- 必须给标号附加冒号(:)。
  汇编程序将从第 1 列开始写的符号当作标号，但是在_ _asm 块内能从第 1 列开始写指令。
  为了使编译程序认识标号，需要给标号末尾附加冒号(:)。
- 不能记述以“?”开始的局部标号。
- 必须用 C/C++语言格式（/* */和//）记述注解。
  不能记述汇编程序格式（;）的注解。
  注解不输出到汇编源输出或者目标列表输出。
- 除了.DATA 控制指令以外，不能记述汇编程序控制指令。另外，不支持文件 include 功能、条件汇编功能、宏功能以及结构化汇编功能。

（2）符号

（2-1）变量名

- 将静态变量名解释为地址。auto 变量名解释为对于 SP（堆栈指针）的偏移。对于变量名，不需要记述编译程序附加的前缀“_”。在下例中，x 为绝对地址，y 为对于 SP 的偏移值。

```
int x;
void func()
{
    int y;
    _ _asm {
       mov.w   @x,r0            //mov.w   @_x,r0
       mov.w   @(y,sp),r1       //mov.w   @(0,sp),r1
    }
}
```

- 将由_ _asm 块参照的 C/C++的变量分配到存储器。
- 不能参照 C++的 auto 变量和参数。

（2-2）函数名

- 只能参照 C 连接函数的函数名。
- 对于函数名，不需要记述编译程序附加的前缀“_”。

（2-3）标号

- C/C++标号和_ _asm 内的标号不能互相参照。也不能参照其它_ _asm 块的标号。
- 能记述定位计数器（$）。

（2-4）枚举符

- 能将枚举符用作常数。

（2-5）结构体成员名

- “变量名.成员名”中的静态变量解释为地址，auto 变量解释为对于 SP 的偏移。
- 如同“OFFSET 变量名.成员名”或者“OFFSET(变量名.成员名)” 指定 OFFSET 运算符时，就解释为结构体前头开始的成员偏移值。
- 不能象“变量名->成员名”、“OFFSET(变量名->成员名)”那样记述“->” 。
- 不能参照位域。
- 例：

```
struct S {
    int a;
    int b;
} x, *p;
void func()
{
    _ _asm {
        mov.w   @x.b,r0                 // mov.w   @_x+2,r0
        mov.l   @p,er1                  // mov.l   @_p,er1
        mov.w   r0,@(OFFSET(x.b),er1)   // mov.w   r0,@(2,er1)
    }
}
```

（2-6）段名

- 段名只能用作 STARTOF、SIZEOF 运算符的操作数。

（3）运算符

- 运算符只能使用以下所示的汇编语言的运算符。但是，对于逻辑异或运算符，必须用“^”取代“~”。

　一元正(+)　一元负(-)　加法(+)　减法(-)　乘法(*)　除法(/)
　一元否定(~)　逻辑与(&)　逻辑和(|)　逻辑异或(^)
　算术左移(<<)　算术右移(>>)
　段起始地址(STARTOF)　段容量(SIZEOF)
　高位字节抽出(HIGH)　低位字节抽出(LOW)
　高位字抽出(HWORD)　低位字抽出(LWORD)

（4）整数常数

- 整数常数不能使用汇编语言格式，而只能使用 C/C++语言格式。
  例如：16 进制数不能记述为 H'FF 而记述为 0xFF。

（5）字符常数

- 字符常数不能使用汇编语言格式，而使用 C/C++语言格式的字符串字面常量。例如：不用"a"而用'a'记述字符串常数。如果写"a"，就当作以空字符结束的字符串。

（6）寄存器规约

- _ _asm 块的寄存器规约和函数调用一样。即使在_ _asm 块内使用如 ER0、ER1、(ER2)的 caller-save 寄存器，也不能在_ _asm 块的入/出口进行保存/恢复。如果在_ _asm 块内使用(ER2)、ER3、ER4、ER5、ER6 等的 callee-save 寄存器，就在_ _asm 块的入/出口自动生成保存/恢复代码。前提是从_ _asm 块内的入口到出口 SP 不变化。但是，如果因函数调用暂时更改 SP，就必须在函数调用后恢复 SP。
- 即使在_ _asm 块内使用 MAC 寄存器，也不在_ _asm 块的入/出口生成保存/恢复 MAC 寄存器的代码。在_ _asm 块内更改 MAC 寄存器并且在_ _asm 块的前后保持 MAC 寄存器的值时，

必须在_ _asm 块内追加保存/恢复 MAC 寄存器的代码。在给中断函数指定 macsave 选项时，即使在_ _asm 块内改写了 MAC 寄存器，编译程序也不认为 MAC 寄存器正在被使用。

例

```
// -cpu=h8sxa
int g_x;
struct ST {
    int  a;
    char b;
    char c;
}   g_st;
enum color {BLUE, GREEN, YELLOW, RED};
                                           /* 实际的代码信息       */
void func(void)               // 为了分配局所变量而确保堆栈
{                                          /* sub.w #6,r7          */
    int x;
    int y;
    struct ST l_st;
                              // 保存在__asm 块内使用的寄存器
    __asm{                                 /* stm.l (er2-er3),@-sp */
       // y      : 局部变量、标量型、从 SP 的偏移=8
       // l_st   : 局部变量、结构体型、从 SP 的偏移=10
       mov.w    @(y,sp),r0                 /* mov.w @(8,sp)r0      */
       mov.l    #y,er1                     /* mov.l #8,er1         */
       mov.w    @(l_st.b, sp),r0           /* mov.w @(12,sp),r0    */
       mov.l    #l_st.c,er1                /* mov.l #13,er1        */
       mov.l    #OFFSET l_st.c,er0         /* mov.l #3,er0         */
       mov.l    #l_st,er2                  /* mov.l #10,er2        */
       bra      L1

    CHAR:
       .data.b  'a'                        /* .data.b H’61         */
    STRING:
       .data.w  "abc"                      /* .data.w H’6263       */
    ENUM:
       .data.w  YELLOW                     /* .data.w H’0002       */
    BOTTOM:
       .data.l  STARTOF P + SIZEOF P       /* .data.l H’00000000   */

    L1:
       // g_st   : 外部变量、结构体型
       // g_x    : 外部变量、标量型
       mov.b    #0xFF,@g_st.b              /* mov.w #H'FF,@_g_st+2  */
       mov.l    #g_st.b,er1                /* mov.w #_g_st+2,er1   */
       mov.l    #OFFSET g_st.b,er2         /* mov.w #2,er2         */
       mov.l    #g_st,er3                  /* mov.w #_g_st,er3     */
       mov.w    #func,@g_x                 /* mov.w #_func,@_g_x   */
       mov.l    #g_x,er0                   /* mov.w #_g_x,er0      */
    }                         // 恢复在__asm 块内使用的寄存器
                                           /* ldm.l @sp+,(er2-er3) */
}
```

备　考

本关键字只在指定 H8SX 或 H8S 的 CPU 时有效。
在_ _asm 块内记述的汇编程序能通过 code=machinecode 选项直接输出到目标文件。
如果在_ _asm 内更改 SP，就不能保证源级调试的运行。
在使用本功能时，对于编译程序输出的汇编程序，有可能在交叉汇编程序中发生“402 ILLEGAL VALUE IN OPERAND”的错误，这是由于生成含有汇编程序嵌入位置的无转移距离的代码。必须在汇编时对交叉汇编程序指定优化选项。另外，必须对应程序，使用 JMP 指令修正编译程序输出的汇编程序，以便满足转移距离。

全局变量的寄存器分配

#pragma global_register
_ _global_register

格　式

```
#pragma global_register [(]<变量名>=<寄存器名>[,…][)]
_ _global_register (<寄存器名>) <型说明符> <变量名>
<型说明符> _ _global_register (<寄存器名>) <变量名>
```

　　<变量名>　　：不可指定 local 变量、C++非静态的成员数据
　　<寄存器名>：ER4、ER5（H8/300 时为 R4、R5）

说　明

给<变量名>指定的全局变量分配<寄存器名>指定的寄存器。

例

```
#pragma global_register(x=R4)      /* 将外部变量 x 分配到 R4。        */
int x;
_ _global_register(R5L) char y;    /* 将外部变量 y 分配到 R5L。       */
void func1(void)
{
  x++;
}
void func2(void)
{
  y=0;
}
void func(int  a)
{
  x = a;
  func1();
  func2();
}
```

备　考

- 以#pragma global_register 声明后的变量定义/变量声明为对象。
- 对全局变量能指定整型或者指针型的变量，而不能指定浮点型的变量。
- 不能设定初始值，也不能参照地址。
- 不能保证从指定变量的（文件中没有指定寄存器）连接目标的参照。
- 不能保证在中断函数内的设定/参照。
- 不能重复指定变量、寄存器，也不能和#pragma abs8、#pragma abs16、_ _abs8、_ _abs16、_ _near8、_ _near16 重复指定。

结构体、联合体、类的边界调整数的指定

#pragma pack 1
#pragma pack 2
#pragma unpack

格　式

```
#pragma pack 1
#pragma pack 2
#pragma unpack
```

说　明

指定源程序中指定位置以后的结构体、联合体、类成员的边界调整数。

在没有指定本扩展名时或者在`#pragma unpack`指定位置后声明的结构体、联合体、类成员的边界调整数按照 `pack` 选项的指定。`#pragma pack` 扩展名和边界调整数的关系如表 10.27 所示。

表 10.27　#pragma pack 扩展名和结构体、联合体、类成员的边界调整数

扩展名/成员的型	#pragma pack 1	#pragma pack 2	#pragma unpack 或者没有指定
[unsigned]char	1	1	1
[unsigned]short、[unsigned]int、[unsigned]long、浮点型、指针型	1	2	按照 pack 选项
边界调整数为 1 的结构体、联合体、类	1	1	1
边界调整数为 2 的结构体、联合体、类	1	2	按照 pack 选项

例

```
#pragma pack 2
struct S1 {
  char a;         /*   offset:0                                   */
                  /*   gap:   1 byte                              */
  int b;          /*   offset:2                                   */
  char c;         /*   offset:4                                   */
                  /*   gap:   1 byte                              */
};
#pragma pack 1
struct S2 {
  char a;         /*   offset:0                                   */
  int b;          /*   offset:1                                   */
  char c;         /*   offset:3                                   */
};
#pragma unpack    /*   按 pack 选项的指定。假设默认值 pack=2       */
struct S3 {
  char a;         /*   offset:0                                   */
                  /*   gap:   1 byte                              */
  int b;          /*   offset:2                                   */
  char c;         /*   offset:4                                   */
                  /*   gap:   1 byte                              */
};
struct S1 s1 = {1,2,3};    /*   _s1: .data.b 1,0,0,2,3,0          */
struct S2 s2 = {1,2,3};    /*   _s2: .data.b 1,0,2,3              */
struct S3 s3 = {1,2,3};    /*   _s3: .data.b 1,0,0,2,3,0          */
void test()       /*   _test:                                     */
{                 /*     mov.w #1,R0                              */
  s1.b=1;         /*     mov.w R0,@_s1+2                          */
  s2.b=2;         /*     mov.w #2,R0      ; 以字节为单位设定/参照   */
    :             /*     mov.b R0H,@_s2+1 ; 境界調整数为 1 的成员   */
}                 /*     mov.b R0L,@_s2+2 ;                       */
```

备　考

- 也能通过 pack 选项指定结构体成员的边界调整数。如果指定选项和扩展名，就优先扩展名的指定。
- 结构体、联合体、类的边界调整数和成员中的最大边界调整数相同。详细内容请参照“**10.1.2 数据的内部表现 （2）复合型（C 语言）和类的型（C++语言）**”。
- 不能用指针存取指定#pragma pack 1 或者 pack=1 选项的结构体、联合体、类的成员（包含在使用指针的成员函数内的存取）。

例：（指定 cpu=2600a 时）

```
#pragma pack 1
struct S {
   char x;
   int y;
} s;
int *p=&s.y; // s.y 的地址有可能为奇数
void test()
{
   s.y=1;     // 能正确存取
   *p =1;     // 不能正确存取
}
```

变量存取时的字节长度指定

_ _evenaccess

格　式

```
_ _evenaccess <型说明符> <变量名>
<型说明符> _ _evenaccess <变量名>
```

说　明

对整型的变量，必须以声明的变量长度进行存储器存取。
但是在 H8/300 中，4 个字节长度的标量型变量以 2 个字节存取。有关 H8SX 请参照备考。

例

```
#define A (*(volatile unsigned short _ _evenaccess *)0xff0178)
void test(void)
{
   A &= ~0x2000 ;
}
```

未指定_ _evenaccess
（BCLR.B 中的 1 个字节存储器存取）

```
_test:
  MOV.L      #H'FF0178,ER0
  BCLR.B     #H'5,@ER0
  RTS
```

指定_ _evenaccess
（MOV.W 中的 2 个字节存储器存取）

```
_test:
  MOV.W     @H'FF0178:24,R0
  BCLR.B    #H'5,R0H
  MOV.W     R0,@H'FF0178:24
  RTS
```

备　考

- 如果对 2 个字节的计数器寄存器等进行 1 个字节的存取，没有存取的另 1 个字节就有可能为不定值。那时必须指定本关键字进行存取。
- CPU 种类为 H8SX 时，能对全部的型指定_ _evenaccess，也能对包含结构体、联合体和类的位域的成员指定。结构体、联合体和类的全体指定与各成员的分别指定相同。
- 不能对 double 型数据进行以 8 个字节为单位的存取。
- 在 H8SX 中支持小端法（little-endian）空间时，必须利用_ _evenaccess 以声明的型长度进行小端法空间的数据存取。
  另外，在 H8SX 中为了防止将大端法（big-endian）的初始值设定到小端法空间，如果对_ _evenaccess 指定的静态变量指定初始值，就出错。
  例：

```
_ _evenaccess long x=0x12345678;   /* 错误 */
void f (void)
{
  ...
}
```

- 对 CPU 类型为 H8SXN/H8SXM/H8SXA/H8SXX 并且指定 evenaccess 关键字时的结构体不能作为单纯的赋值、参数或者返回值使用。使用时，必须以成员为单位进行设定/参照。
  例：

```
typedef struct {
    int a;
    long b;
}str;

_ _evenaccess str st1;
str st2;
void func(str);
```

```
str main(void){
str temporary;
st2.a = st1.a;          /* 以各成员为单位，进行包含_ _evenaccess 指定 */
st2.b = st2.b;          /* 的结构体的单纯赋值                         */

temporary.a = st1.a;  /* 将包含_ _evenaccess 指定的结构体的数据     */
temporary.b = st1.b;  /* 赋值到没有_ _evenaccess 指定的结构体，     */
func(temporary);        /* 将该变量指定为参数                         */

return (temporary);    /* 将没有_ _evenaccess 指定的结构体           */
                        /* 指定为返回值                               */
}
```

绝对地址的指定

#pragma address

格　式　`#pragma address [(]<变量名>=<绝对地址>[,…][)]`

<绝对地址>　：实效地址　（用 C 语言的 16 进制数记述）

说　明　将指定的变量分配到<绝对地址>。

此时按编译程序指定的各变量设定段，在连接时分配到指定的绝对地址。在对连续地址指定相同段种类的变量时，就将那些变量统一到同一个段。

如果分配到 8 位/16 位短绝对区域的地址，就输出以:8/:16 存取的代码（对 8 位短绝对区域指定 2 个字节边界的变量除外）。

例

（1）

• 源程序

```
#pragma address (io=0x100)
int io;

func(void){
    io = 0;
}
```

• 输出目标

```
[未指定#pragma address 时]   [指定#pragma address 时]
 .SECTION P,CODE              .SECTION P,CODE
_func:                       _func:
 MOV.W  #0:4,@_io:32          MOV.W   #0, @_io:16
 RTS                          RTS
 .SECTION B,DATA,ALIGN=2      .SECTION  $ADDRESS$B100,DATA,LOCATE=H'100
_io:                         _io:
 .RES.W  1                    .RES.W 1
 .END                         .END
```

（2）

• 源程序

```
#pragma address (P1=0x100)
struct {
    unsigned char  BYTE;
    unsigned short WORD;
}P1;

func()
{
    P1.WORD =10;
}
```

• 输出目标

```
[未指定#pragma address 时]   [指定#pragma address 时]
 .SECTION P,CODE              .SECTION P,code
_func:                       _func:
 MOV.W  #10,@_P1+2:32         MOV.W   #10,@_P1+2:16
 RTS                          RTS
 .SECTION B,DATA,ALIGN=2      .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
```

```
_P1:                          _P1:
 .RES.W 2                      .RES.W 2
 .END                          .END
```

（3）连续分配多个相同段属性的变量时

• 源程序

```
#pragma address (io=0x100, io2=0x102)
int io;
int io2;

func(void){
    io  =0;
    io2 =0;
}
```

• 输出目标

```
[未指定#pragma address 时]  [指定#pragma address 时]
 .SECTION P,code             .SECTION P,code
_func:                      _func:
 MOV.W  #0:4,@_io2:32        MOV.W   #0,@_io2:16
 MOV.W  #0:4,@_io:32         MOV.W   #0,@_io:16
 RTS                         RTS
 .SECTION B,DATA,ALIGN=2     .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_io:                         _io:
 .RES.W 1                    .RES.W 1
_io2:                       _io2:
 .RES.W 1                    .RES.W 1
 .END                        .END
```

（4）不连续分配多个相同段属性的变量时（下例中有 2 个字节的空区域）。

• 源程序

```
#pragma address (io=0x100, io2=0x104)
int io;
int io2;

func(void){
    io = io2 =0;
}
```

• 输出目标

```
[未指定#pragma address 时]  [指定#pragma address 时]
 .SECTION P,CODE             .SECTION P,CODE
_func:                      _func:
 MOV.W  #0:4,@_io2:32        MOV.W   #0,@_io2:16
 MOV.W  #0:4,@_io:32         MOV.W   #0,@_io:16
 RTS                         RTS
 .SECTION B,DATA,ALIGN=2     .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_io:                        _io:
 .RES.W 1                    .RES.W 1
                             .SECTION $ADDRESS$B104,DATA,LOCATE=H'104
_io2:                      _io2:
 .RES.W 1                   .RES.W 1
```

```
        .END                            .END
```

备　考

- 本扩展功能只在 CPU 类型为 H8SX 和 H8S 时有效。
- 必须在声明变量前指定#pragma address。
- 如果指定复合型/类的型的成员、静态成员或者变量以外的符号名，就出错。
- 如果给边界调整数为 2 的变量或者结构体指定奇数地址，就出错。
- 如果对同一个变量多次指定#pragma address，就出错。
- 如果给不同的变量指定同一个地址或者变量的地址重叠，就出错。
- 如果对同一个变量同时指定以下的#pragma 扩展名，就出错。
  #pragma section
  #pragma abs8/abs16
  #pragma global_register
- 不能将#pragma address 指定为带初始化数据的变量。否则，就输出 C1407 (W) #pragma address ignored 的信息。

10.2.2　段地址运算符

段地址运算符

_ _sectop
_ _secend

格　式

```
_ _sectop("<段名>")
_ _secend ("<段名>")
```

说　明

参照由`_ _sectop` 指定的<段名>的起始地址。
参照由`_ _secend` 指定的<段名>的末尾+1 地址。

例

```
#include <machine.h>
#pragma section $DSEC
static const struct {
  void *rom_s;      /* ROM 中的初始化数据段的起始地址       */
  void *rom_e;      /* ROM 中的初始化数据段的结束地址       */
  void *ram_s;      /* RAM 中的初始化数据段的起始地址       */
}DTBL[]= {_ _sectop (“D”), _ _secend (“D”), _ _sectop (“R”)};

#pragma section $BSEC
static const struct {
  void *b_s;        /* 未初始化数据段的起始地址             */
  void *b_e;        /* 未初始化数据段的结束地址             */
}BTBL[]= {_ _sectop (“B”), _ _secend (“B”)};

#pragma section
#pragma stacksize 0x100    /* 声明堆栈段 S                          */
#pragma entry INIT /* 将函数 INIT 声明为入口函数                    */
void main(void);  /* 声明 main 函数                                 */
void INIT(void)   /* _INIT:                       ;入口函数开始  */
{                 /*    MOV   #STARTOF S+SIZEOF S,SP ;SP 初始设定*/
    _INITSCT();   /*    JSR   @_ _INITSCT    ;段区域的初始化 */
    main();       /*    JSR   @_main         ;调用 main 函数      */
    sleep();      /*    SLEEP                ;在低功耗状态下待机   */
}
```

段初始化方法的详细内容请参照“9.2.2　执行环境的设定”。

10.2.3　　内部函数

作为内部函数，提供不能用 C/C++语言记述的以下功能：

- 条件码寄存器的设定/参照
- 扩展寄存器的设定/参照
- 乘加运算
- 循环运算
- 特殊指令（TRAPA、SLEEP、MOVFPE、MOVTPE、TAS、EEPMOV、NOP、XCH）
- 溢出判断
- 10 进制运算

与一般函数一样，用函数调用格式记述内部函数。
但是，使用内部函数时，必须声明`#include <machine.h>`。

内部函数一览表如**表 10.28** 所示。

表 10.28　内部函数一览表

项目		规格	功能
1	条件码寄存器	void set_imask_ccr(unsigned char mask)	给中断屏蔽设定 mask 的值
2		unsigned char get_imask_ccr(void)	参照中断屏蔽
3		void set_ccr(unsigned char ccr)	设定条件码寄存器 （参数 ccr 的值→CCR）
4		unsigned char get_ccr(void)	参照条件码寄存器
5		void and_ccr(unsigned char ccr)	条件码寄存器的逻辑与 （CCR & 参数 ccr → CCR）
6		void or_ccr(unsigned char ccr)	条件码寄存器的逻辑或 （CCR | 参数 ccr → CCR）
7		void xor_ccr(unsigned char ccr)	条件码寄存器的逻辑异或 （CCR ^ 参数 ccr → CCR）
8	扩展寄存器	void set_imask_exr(unsigned char mask)	给中断屏蔽设定 mask 的值
9		unsigned char get_imask_exr(void)	参照中断屏蔽
10		void set_exr(unsigned char exr)	设定扩展寄存器 （参数 exr → EXR）
11		unsigned char get_exr(void)	参照扩展寄存器
12		void and_exr(unsigned char exr)	扩展寄存器的逻辑与 （EXR & 参数 exr → EXR）
13		void or_exr(unsigned char exr)	扩展寄存器的逻辑或 （EXR | 参数 exr → EXR）
14		void xor_exr(unsigned char exr)	扩展寄存器的逻辑异或 （EXR ^ 参数 exr → EXR）
15	向量基址寄存器	void set_vbr(void* vbr)	设定 VBR
16	乘加运算	long mac(long val,int *ptr1,int *ptr2, unsigend long count)	用 MAC 指令运算 val+Σi=0,count-1(ptr1[i]＊ptr2[i])
		long macl(long val,int *ptr1,int *ptr2, unsigned long count,unsigned long mask)	或者用环形缓冲功能运算 val+Σi=0,count-1(ptr1[i]＊*((ptr2+i)&mask))
17	64bit 乘法	long mulsu(long val1, long val2)	展开为 MULS/U 指令
		unsigned long muluu(unsigned long val1, unsigned long val2)	展开为 MULU/U 指令

项目		规格	功能
18	循环运算	char rotlc(int count,char data)	将 data 左循环 count 位
		int rotlw(int count,int data)	
		long rotll(int count,long data)	
19		char rotrc(int count,char data)	将 data 右循环 count 位
		int rotrw(int count,int data)	
		long rotrl(int count,long data)	
20	特殊指令	void trapa(unsigned int trap_no)	展开为 TRAPA #trap_no
21		void sleep(void)	展开为 SLEEP 指令
22		void movfpe(char *addr,char data) char _movfpe(char *addr)	使用 MOVFPE 指令，将*addr 设定到 data 或者返回*addr
23		void movtpe(char data,char *addr)	使用 MOVTPE 指令，将 data 设定到*addr
24		void tas(char *addr)	使用 TAS 指令，比较*addr 和 0，其结果设定到条件码，*addr 的最高位置“1”
25		void eepmov(void *dst,const void *src, unsigned char size)	使用 EEPMOV 指令，将 size 字节的数据进行*src 到*dst 的传送
		void eepmov(void *dst,const void *src, unsigned int size)	
		void eepmovb(void *dst,const void *src, unsigned char size)	
		void eepmovw(void *dst,const void *src, unsigned int size)	
		void eepmovi(void *dst,const void *src, unsigned int size)	
26		void movmdb(void *dst, const void *src, unsigned int count)	使用 movmd.b 指令，将数据进行 count 次的*src 到*dst 的传送
		void movmdw(int *dst, const int *src, unsigned int count)	使用 movmd.w 指令，将数据进行 count 次的*src 到*dst 的传送
		void movmdl(long *dst, const long *src, unsigned int count)	使用 movmd.l 指令，将数据进行 count 次的*src 到*dst 的传送
27		unsigned int movsd(char *dst, const char *src, unsigned int size)	使用 movsd 指令，将最大 size 字节的数据进行*src 到*dst 的传送。但是，在传送零数据时结束传送。
28		void nop(void)	展开为 NOP 指令
29	条件码反映运算	int ovfaddc(char dst,char src,char *rst)	将 dst + src 设定到*rst，运算结果反映在条件码中
		int ovfadduc(unsigned char dst, unsigned char src,unsigned char *rst)	
		int ovfaddw(int dst,int src,int *rst)	
		int ovfadduw(unsigned int dst, unsigned int src,unsigned int *rst)	
		int ovfaddl(long dst,long src,long *rst)	
		int ovfaddul(unsigned long dst, unsigned long src,unsigned long *rst)	

项目		规格	功能
30	条件码反映运算	int ovfsubc(char dst,char src,char *rst) int ovfsubuc(unsigned char dst, unsigned char src,unsigned char *rst) int ovfsubw(int dst,int src,int *rst) int ovfsubuw(unsigned int dst, unsigned int src,unsigned int *rst) int ovfsubl(long dst,long src,long *rst) int ovfsubul(unsigned long dst, unsigned long src,unsigned long *rst)	将 dst - src 设定到*rst，运算结果反映在条件码中
31		int ovfshalc(char dst,char *rst) int ovfshalw(int dst,int *rst) int ovfshall(long dst,long *rst)	将 dst << 1 设定到*rst，运算结果反映在条件码中（算术移位）
32		int ovfshlluc(unsigned char dst,unsigned char *rst) int ovfshlluw(unsigned int dst,unsigned int *rst) int ovfshllul(unsigned long dst,unsigned long *rst)	将 dst << 1 设定到*rst，运算结果反映在条件码中（逻辑移位）
33		int ovfnegc(char dst,char *rst) int ovfnegw(int dst,int *rst) int ovfnegl(long dst,long *rst)	将 dst 的 2 的补数设定到*rst，运算结果反映在条件码中
34	10 进制运算	void dadd(unsigned char size,char *ptr1, char *ptr2,char *rst)	将 ptr1、ptr2 作为 size 位的 10 进制数数组，进行 10 进制加法运算，结果设定到*rst
35		void dsub(unsigned char size,char *ptr1, char *ptr2,char *rst)	将 ptr1、ptr2 作为 size 位的 10 进制数数组，进行 10 进制减法运算，结果设定到*rst

中断屏蔽位的设定

void set_imask_ccr (unsigned char mask)

说　明　　给条件码寄存器（CCR）的中断屏蔽位（I）设定 mask 值（0 或者 1）。

头文件　　`<machine.h>`

参　数　　`mask`　　mask 值（0 或者 1）

例

```
#include <machine.h>          /* 必须 include <machine.h>。          */
void main(void)
{
     set_imask_ccr(0);        /* 清除中断屏蔽位。                      */
}
```

中断屏蔽位的参照

unsigned char get_imask_ccr(void)

说　明　　参照条件码寄存器（CCR）的中断屏蔽位（I）的值（0 或者 1）。

头文件　　`<machine.h>`

返回值　　条件码的中断屏蔽位的参照值

例

```
#include <machine.h>          /* 必须 include <machine.h>。          */
void main(void)
{
     if(get_imask_ccr())      /* 参照中断屏蔽位。                      */
      :
}
```

条件码寄存器的设定

void set_ccr(unsigned char ccr)

说　明	给条件码寄存器（CCR）设定 ccr 值（8 位）。	
头文件	<machine.h>	
参　数	ccr	设定值（8 位）

例

```
#include <machine.h>       /* 必须 include <machine.h>。            */
main()
{
    set_ccr(0);            /* 清除 CCR。                             */
}
```

条件码寄存器的参照

unsigned char get_ccr(void)

说　明　参照条件码寄存器（CCR）的值。

头文件　<machine.h>

返回值　条件码的参照值

例

```
#include <machine.h>       /* 必须 include <machine.h>。            */
void main(void)
{
    unsigned char a;
     a=get_ccr();          /* 参照 CCR。                              */
        :
}
```

与条件码寄存器的逻辑与

void and_ccr(unsigned char ccr)

说　明	将条件码寄存器（CCR）的值和 ccr 进行逻辑与运算，结果设定到 CCR。	
头文件	<machine.h>	
参　数	ccr	逻辑与的操作数

例

```
#include <machine.h>          /* 必须 include <machine.h>。                 */
void main(void)
{
     and_ccr(0x10);           /* 将 CCR & 0x10 设定到 CCR。                 */
}
```

与条件码寄存器的逻辑或

void or_ccr(unsigned char ccr)

说　明	将条件码寄存器（CCR）的值和 ccr 进行逻辑或运算，结果设定到 CCR。	
头文件	<machine.h>	
参　数	ccr	逻辑或的操作数

例

```
#include <machine.h>          /* 必须 include <machine.h>。                 */
void main(void)
{
     or_ccr(0x10);            /* 将 CCR | 0x10 设定到 CCR。                 */
}
```

与条件码寄存器的逻辑异或

void xor_ccr(unsigned char ccr)

说　明　　将条件码寄存器（CCR）的值和 ccr 进行逻辑异或运算，结果设定到 CCR。

头文件　　<machine.h>

参　数　　ccr　　逻辑异或的操作数

例

```
#include <machine.h>        /* 必须 include <machine.h>。              */
void main(void)
{
     xor_ccr(0x10);         /* 将 CCR ^ 0x10 设定到 CCR。              */
}
```

扩展寄存器的中断位设定

void set_imask_exr(unsigned char mask)

说　明　　给扩展寄存器（EXR）的中断屏蔽位（I2～I0）设定 mask 值（0～7）。
本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。

头文件　　<machine.h>

参　数　　mask　　mask 值

例

```
#include <machine.h>        /* 必须 include <machine.h>。              */
void main(void)
{
     set_imask_exr(0);      /* 给扩展寄存器的中断屏蔽位                */
         :                  /* 设定屏蔽级 0。                          */
}
```

扩展寄存器的中断位参照

unsigned char get_imask_exr(void)

说　明	参照扩展寄存器（EXR）的中断屏蔽位（I2～I0）的值（0～7）。 本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。
头文件	<machine.h>
返回值	扩展寄存器的中断屏蔽位的参照值

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
void main(void)
{
     if(get_imask_exr())    /* 参照扩展寄存器的中断屏蔽位。           */
         :
}
```

扩展寄存器的设定

void set_exr(unsigned char exr)

说　明	给扩展寄存器（EXR）设定 exr 值（8 位）。 本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。	
头文件	<machine.h>	
参　数	exr	设定值

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
void main(void)
{
     set_exr(0);            /* 清除扩展寄存器。                       */
}
```

扩展寄存器的参照

unsigned char get_exr(void)

说　明　　参照扩展寄存器（EXR）的值。
本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。

头文件　　`<machine.h>`

返回值　　扩展寄存器的参照值

例

```
#include <machine.h>        /* 必须 include <machine.h>。            */
void main(void)
{
unsigned char a;
     a=get_exr();           /* 参照扩展寄存器。                      */
         :
}
```

与扩展寄存器的逻辑与

void and_exr(unsigned char exr)

说　明　　将扩展寄存器（EXR）的值和 exr 进行逻辑与运算，结果设定到 EXR。
本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。

头文件　　`<machine.h>`

参　数	exr	逻辑与的操作数

例

```
#include <machine.h>        /* 必须 include <machine.h>。            */
void main(void)
{
     and_exr(0x10);         /* 将 EXR & 0x10 设定到 EXR。            */
}
```

与扩展寄存器的逻辑或

void or_exr(unsigned char exr)

说　明	将扩展寄存器（EXR）的值和 exr 进行逻辑或运算，结果设定到 EXR。 本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。
头文件	<machine.h>
参　数	exr　　逻辑或的操作数

例

```
#include <machine.h>     /* 必须 include <machine.h>。          */
void main(void)
{
     or_exr(0x10);       /* 将 EXR | 0x10 设定到 EXR。          */
}
```

与扩展寄存器的逻辑异或

void xor_exr(unsigned char exr)

说　明	将扩展寄存器（EXR）的值和 exr 进行逻辑异或运算，结果设定到 EXR。 本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。
头文件	<machine.h>
参　数	exr　　逻辑异或的操作数

例

```
#include <machine.h>     /* 必须 include <machine.h>。          */
void main(void)
{
     xor_exr(0x10);      /* 将 EXR ^ 0x10 设定到 EXR。          */
}
```

VBR 设定

void set_vbr(void vbr)*

说　明　给向量基址寄存器（VBR）设定 vbr 的值（32 位）。
本函数只在 CPU 种类为 H8SXN、H8SXM、H8SXA、H8SXX 时有效。但是，在 CPU 种类为 H8SXM 时，被指定的 vbr 值的低 16 位有效。

头文件　`<machine.h>`

参　数	vbr	设定值

例

```
#include <machine.h>       /* 必须 include <machine.h>。              */
void main(void)
{
set_vbr((void*)0x20000);  /* 给向量基址寄存器设定 0x20000。              */
}
```

MAC 指令的展开

*long mac(long val,int *ptr1,int *ptr2,unsigned long count)*
*long macl(long val,int *ptr1,int *ptr2,unsigned long count,unsigned long mask)*

说　明

展开为乘加运算的 MAC 指令。
mac 函数将 val 作为 MAC 寄存器的初始值，对 ptr1 和 ptr2 指向的 2 个字节的数据进行带符号的乘法运算，并将 4 个字节的结果数据加到 MAC 寄存器，然后 ptr1 和 ptr2 的内容都＋2。此操作重复 count 次。
macl 函数将 ptr2 的数据用作环形缓冲，进行 ptr2 与 mask 的逻辑与运算。
mac、macl 函数只在 CPU/运行模式为 H8SXN:{M|MD}、H8SXM:{M|MD}、H8SXA:{M|MD}、H8SXX:{M|MD}、2600a、2600n 时有效。

头文件

<machine.h>

返回值

乘加运算结果

参　数

val	MAC 寄存器的初始值
ptr1、ptr2	指向乘法运算数据的指针
count	循环次数
mask	用于环形缓冲的 mask 值

例

```
#include <machine.h>           /* 必须 include <machine.h>。          */
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
long l1,l2;
:
void main(void)
{
l1=mac(100,ptr1,ptr2,4);      /* 进行 l1=100+0*9+1*8+2*7+3*6 的运算。 */
l2=macl(100,ptr1,ptr2,4,~4);  /* 进行 l2=100+0*9+1*8+2*9+3*8 的运算。 */
}                             /* 将 ptr2[0], ptr2 [1]的数据            */
                              /* 重复用作环形缓冲。                    */
                              /* 为了将 ptr2 & mask 用作地址,          */
                              /* 需要将 ptr2 分配到 8 的倍数的地址。    */
```

备　考

必须将 macl 的 ptr2 指向的表进行边界调整，调整为 mask 值的补数的 2 倍。
在上述例子的情况下，必须通过连接映像确认 ptr2 是否被分配到 8 的倍数的地址。

64 位乘法结果高 32 位的取得

long mulsu(long val1, long val2)
unsigned long muluu(unsigned long val1, unsigned long val2)

说　明　　展开为进行 32bit×32bit＝64bit 乘法运算的 muls/u、mulu/u 指令。
将本内部函数的 32bit 的各参数（val1,val2）相乘，结果的高 32bit 作为运算结果返回。

头文件　　<machine.h>

参　数

val1	被乘数
val2	乘数

例

```
#include <machine.h>
long          s_val1, s_val2, s_ans;
unsigned long u_val1, u_val2, u_ans;
void f(void)
{
    s_ans = mulsu(s_val1, s_val2); /* 带符号的 32bit 乘法运算 */
    u_ans = muluu(u_val1, u_val2); /* 无符号的 32bit 乘法运算 */
}
```

备　考　　本内部函数只在 CPU 种类为 H8SXN:{M|MD}、H8SXM:{M|MD}、H8SXA:{M|MD}、H8SXX:{M|MD}时有效。

位左循环指令

char rotlc(int count,char data)
int rotlw(int count, int data)
long rotll(int count, long data)

说　明　　rotlc、rotlw、rotll 函数分别将 1 个字节、2 个字节、4 个字节的 data 向左循环 count 位，返回该结果。

头文件　　<machine.h>

返回值　　data 左循环 count 位后的结果值

参　数

count	循环位数
data	位循环对象数据

例

```
#include <machine.h>       /*  必须 include <machine.h>。                */
int i,data;
void f(void)
{
    i=rot1w(5,data);       /*  将 data 左循环 5bit。                     */
}
```

位右循环指令

char rotrc(int count, char data)
int rotrw(int count,int data)
long rotrl(int count, long data)

说　明　　rotrc、rotrw、rotrl 函数分别将 1 个字节、2 个字节、4 个字节的 data 向右循环 count 位，返回该结果。

头文件　　<machine.h>

返回值　　data 右循环 count 位后的结果值

参　数

count	循环位数
data	位循环对象数据

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
int i,data;
void f(void)
{
    i=rotrw(5,data);        /* 将 data 右循环 5bit。                  */
}
```

陷阱指令

void trapa(unsigned int trap_no)

说　明　　展开为无条件陷阱的 TRAPA #trap_no 指令。trap_no 为 0～3 的常数。
另外，本函数在 CPU/运行模式不为 300 时有效。

头文件　　<machine.h>

参　数

trap_no	对转移目标向量地址的 trap 序号

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
void f(void)
{
      :
    trapa(0);               /* 通过 trapa #0 返回。                   */
}
```

SLEEP 指令

void sleep(void)

说　明　　展开为低功耗状态指令 SLEEP。

头文件　　<machine.h>

例

```
#include <machine.h>          /* 必须 include <machine.h>。                */
void f(void)
{
        :
    sleep();                  /* 展开为 sleep 指令。                       */
}
```

E 时钟同步传送指令

void movfpe(char *addr,char data)
char _movfpe(char *addr)

说　明　　使用 E 时钟同步数据传送指令 MOVFPE，movfpe 函数在与 E 时钟同步的时序将取出的*addr 值设定到 data，_movfpe 函数将该值作为返回值返回。*addr 必须指定能用 16 位绝对地址存取的数据。

头文件　　<machine.h>

返回值

movfpe 函数	无
_movfpe 函数	传送目标数据

参　数

addr	指向传送源数据的指针
data	传送目标数据（movfpe 函数）

例

```
#include <machine.h>          /* 必须 include <machine.h>。                */
#pragma abs16 a               /* 为了能用 16 位绝对地址存取，              */
char a,data;                  /* 用#pragma abs16 声明                      */
                              /* 第一参数。                                */
void f(void)
{
    movfpe(&a,data);          /* 通过 MOVFPE 指令，将 a 传送到 data。      */
    data = _movfpe(&a);       /* 左边的二种记述，其含义相同。              */
}
```

备　考　　char _movfpe(char*addr)只在 H8SX 时有效。

E 时钟同步传送指令

*void movtpe(char data,char *addr)*

说　明　　使用 E 时钟同步数据传送指令 MOVTPE，在与 E 时钟同步的时序将 data 设定到*addr。*addr 必须指定能用 16 位绝对地址存取的数据。

头文件　　<machine.h>

参　数	data	传送源数据
	addr	指向传送目标的指针

例

```
#include <machine.h>        /* 必须 include <machine.h>。      */
#pragma abs16 a             /* 为了能用 16 位绝对地址存取，     */
char a,data;                /* 用#pragma abs16 声明            */
                            /* 第二参数。                       */
void f(void)
{
    movtpe(data,&a);        /* 在与 E 时钟同步的时序            */
}                           /* 将 data 传送到 a。               */
```

测试-与-置位指令

*void tas(char *addr)*

说　明　　使用测试-与-设定指令 TAS，将 addr 的内容和 0 比较，其结果设定到条件码寄存器（CCR），然后将 addr 的最高位置 1。

本函数只在 CPU/运行模式为 H8SXN、H8SXM、H8SXA、H8SXX、2600a、2000a、2600n、2000n 时有效。

头文件　　<machine.h>

参　数	addr	指向要进行测试-与-设定的数据的指针

例

```
#include <machine.h>        /* 必须 include <machine.h>。      */
char a;
void f(void)
{
    tas(&a);                /* 将 a-0 的结果设定到 CCR，        */
}                           /* 然后进行 a|=0x80 的运算。        */
```

块传送指令

*void eepmov(void *dst, const void *src, unsigned char size)*
*void eepmov(void *dst, const void *src ,unsigned int size)*
*void eepmovb(void *dst, const void *src, unsigned char size)*
*void eepmovw(void *dst, const void *src, unsigned int size)*

说　明　　使用块传送指令 EEPMOV，从 src 指向的地址将 size 表示的字节数的数据块传送到 dst 指向的地址。

在 eepmov 内部函数时，必须给 size 指定常数值。
在 CPU/运行模式为 300 时最大能指定的 Size 值为 255，否则为 65535。但是，因为在 256～65535 时被展开为 EEPMOV.W，所以在发生中断请求时不能使用。另外，size 为零（0）时不传送。

在 eepmovb、eepmovw 内部函数时，也能给 size 指定变量。
eepmovb 内部函数总是被展开为 EEPMOV.B。
eepmovw 内部函数总是被展开为 EEPMOV.W。

头文件　　<machine.h>

参　数

dst	指向传送目标的指针
src	指向传送源的指针
size	传送长度

例

```
#include <machine.h>          /* 必须 include <machine.h>。          */
char a[10],b[10];
void f(void)
{
    eepmov(b,a,10);           /* 使用 EEPMOV 指令，将数组 a 的数据     */
}                             /* 传送到数组 b。                        */
```

备　考　　eepmovb 内部函数和 eepmovw 内部函数只在 CPU 为 H8SX 和 H8S（没有指定 legacy=v4）时有效。

对应中断请求的块传送指令

*void eepmovi(void *dst, const void *src, unsigned int size)*

说　明　使用块传送指令 EEPMOV，从 src 指向的地址将 size 表示的字节数的数据块传送到 dst 指向的地址。被指令展开为：即使在执行 EEPMOV 指令中发生中断，也能在从中断返回后继续执行的形式。

能给 size 指定常数值和变量。
给 size 指定常数值时，能指定的最大值为 65535。
size 为零（0）时不传送。
如果给 size 指定 255 以下的常数，就输出 1 次 EEPMOV.B 指令。
如果给 size 指定 256 以上 510 以下的常数，就输出 2 次 EEPMOV.B 指令。
如果给 size 指定变量或者 512 个以上的常数，就输出如下代码。即使在执行 EEPMOV.W 指令中发生中断，也能在从中断返回后继续执行。

```
L1: EEPMOV.W
    MOV.W     R4,R4
    BNE       L1
```

头文件　`<machine.h>`

参　数

dst	指向传送目标的指针
src	指向传送源的指针
size	传送长度

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
char a[10],b[10];
void f(void)
{
    eepmovi(b,a,10);        /* 使用 EEPMOV 指令，将数组 a 的数据       */
}                           /* 传送到数组 b。                          */
```

备　考　本内部函数只在 CPU 为 H8SX 和 H8S 时有效。

用于 H8SX 的块传送指令

*void movmdb(void *dst, const void *src, unsigned int count)*
*void movmdw(int *dst, const int *src, unsigned int count)*
*void movmdl(long *dst, const long *src, unsigned int count)*

说　明	MOVMD.B、MOVMD.W、MOVMD.L 指令分别将 1 个字节、2 个字节、4 个字节的存储块，按照 count 指定的次数从 src 指向的地址传送到 dst 指向的地址。 能给 count 指定 0 到 65535 的值。但是，如果给 count 指定 0，传送次数就为 65536。
头文件	<machine.h>
参　数	src　指向传送源的指针 dst　指向传送目标的指针 count　传送次数

例

```
#include <machine.h>     /* 必须 include <machine.h>。   */
char s1[100], d1[100];
int  s2[50],  d2[50];
long s4[25],  d4[25];
void f(void)
{
    movmdb(d1, s1, 100); /* 使用 MOVMD.B 指令，将数组 s1 传送 100 个字节   */
                         /* 到数组 d1。                                   */
    movmdw(d2, s2, 50);  /* 使用 MOVMD.W 指令，将数组 s2 传送 100 个字节   */
                         /* 到数组 d2                                     */
    movmdl(d4, s4, 25);  /* 使用 MOVMD.L 指令，将数组 s4 传送 100 个字节   */
                         /* 到数组 d4                                     */
}
```

备　考　　本内部函数只在 CPU 为 H8SX 时有效。

用于 H8SX 的字符串传送指令

*unsigned int movsd(char *dst, const char *src, unsigned int size)*

说　明	通过块传送指令 movsd，进行从 src 指向的地址到 dst 指向的地址的块传送。 但是，如果将零（H'00）作为数据传送或者传送完 size 表示的字节数，就结束。返回值为 size 减去实际传送的字节数的值。 能给 size 指定 0 到 65535 的值。但是，如果给 size 指定零（0），最大传送字节数就解释为 65536。
头文件	<machine.h>
返回值	size 指定的字节数减去实际传送的字节数的值

参　数

src	指向传送源的指针
dst	指向传送目标的指针
size	最大传送字节数

例

```
#include <machine.h>            /* 必须 include <machine.h>。         */
const char *s;
char d[100];
unsigned int remain;

void f(void)
{
    remain = movsd(d, s, 100); /* 通过 MOVSD 指令将字符串 s 传送到数组 d。*/
}                              /* 最大传送 100 个字节。                */
```

备　考　　本内部函数只在 CPU 为 H8SX 时有效。

NOP 指令

void nop(void)

说　明	展开为 NOP 指令。
头文件	<machine.h>

例

```
#include <machine.h>       /* 必须 include <machine.h>。          */
int a;
void f(void)
{
    while(a) nop();        /* 当 a!=0 时执行 NOP 指令。            */
}
```

加法运算的溢出判断

*int ovfaddc(char dst,char src,char *rst)*
*int ovfaddw(int dst,int src,int *rst)*
*int ovfaddl(long dst, long src,long *rst)*
*int ovfadduc(unsigned char dst,unsigned char src,unsigned char *rst)*
*int ovfadduw(unsigned int dst,unsigned int src,unsigned int *rst)*
*int ovfaddul(unsigned long dst,unsigned long src,unsigned long *rst)*

说　明　ovfaddc、ovfaddw、ovfaddl 函数分别进行 1 个字节、2 个字节和 4 个字节 dst 和 src 的带符号加法运算。另外，ovfadduc、ovfadduw、ovfaddul 函数分别进行无符号加法运算。而且，在 rst 不为 0 时，将结果保存到 rst 指向的区域。

如果加法结果不溢出就返回 0，否则返回 0 以外的值。

只能在判断 if 语句、do 语句、while 语句、for 语句的条件表达式中指定这些函数。

另外，ovfaddl 和 ovfaddul 函数在 CPU 不为 H8/300 时有效。

头文件　<machine.h>

返回值	溢出	0 以外的值
	不溢出	0
参　数	dst,src	加法运算的操作数
	rst	结果的保存位置（rst 为 0 时不保存结果）

例

```
#include <machine.h>        /* 必须 include <machine.h>。              */
int dst,src;
void f(void)
{
    if(ovfaddw(dst,src,0)) /* 用 BVC 判断 dst + src 的结果。           */
        dst=0;
}
```

减法运算的溢出判断

*int ovfsubc(char dst,char src,char *rst)*
*int ovfsubw(int dst,int src,int *rst)*
*int ovfsubl(long dst, long src,long *rst)*
*int ovfsubuc(unsigned char dst,unsigned char src,unsigned char *rst)*
*int ovfsubuw(unsigned int dst,unsigned int src,unsigned int *rst)*
*int ovfsubul(unsigned long dst,unsigned long src,unsigned long *rst)*

说　明　ovfsubc、ovfsubw、ovfsubl 函数分别进行 1 个字节、2 个字节和 4 个字节的 dst 和 src 的带符号减法运算（dst-src）。另外，ovfsubuc、ovfsubuw、ovfsubul 函数分别进行无符号减法运算。

而且，在 rst 不为 0 时，将结果保存到 rst 指向的区域。

如果减法结果不溢出就返回 0，否则返回 0 以外的值。

只能在判断 if 语句、do 语句、while 语句、for 语句的条件表达式中指定这些函数。

另外，ovfsubl 和 ovfsubul 函数在 CPU 不为 H8/300 时有效。

头文件　<machine.h>

返回值	溢出	0 以外的值
	不溢出	0
参　数	dst,src	减法运算的操作数
	rst	结果的保存位置（rst 为 0 时不保存结果）

例

```
#include <machine.h>        /* 必须 include <machine.h>。            */
int dst,src;
void f(void)
{
    if(ovfsubw(dst,src,0)) /* 用 BVC 判断 dst – src 的结果。         */
      dst=0;
}
```

算术移位的溢出判断

*int ovfshalc(char dst,char *rst)*
*int ovfshalw(int dst,int *rst)*
*int ovfshall(long dst,long *rst)*

说　明

ovfshalc、ovfshalw 和 ovfshall 函数将 1 个字节、2 个字节、4 个字节的 dst 向左算术移 1 位，在 rst 不为 0 时，将结果保存到 rst 指向的区域。
然后，如果算术移位结果不溢出就返回 0，否则返回 0 以外的值。
只能在判断 if 语句、do 语句、while 语句、for 语句的条件表达式中指定这些函数。
另外，ovfshalw 和 ovfshall 函数在 CPU 不为 H8/300 时有效。

头文件

<machine.h>

返回值

溢出	0 以外的值
不溢出	0

参　数

dst	位移位运算的操作数
rst	结果的保存位置（rst 为 0 时不保存结果）

例

```
#include <machine.h>        /* 必须 include <machine.h>。         */
int dst;
void f(void)
{
    if(ovfshalw(dst,0))     /* 用 BVC 判断 dst<<1 的结果。         */
        dst=0;
 }
```

逻辑移位的溢出判断

*int ovfshlluc(unsigned char dst,unsigned char *rst)*
*int ovfshlluw(unsigned int dst,unsigned int *rst)*
*int ovfshllul(unsigned long dst,unsigned long *rst)*

说　明　ovfshlluc、ovfshlluw 和 ovfshllul 函数将 1 个字节、2 个字节、4 个字节的 dst 向左逻辑移 1 位，在 rst 不为 0 时，将结果保存到 rst 指向的区域。
然后，如果逻辑移位结果不溢出就返回 0，否则返回 0 以外的值。
只能在判断 if 语句、do 语句、while 语句、for 语句的条件表达式中指定这些函数。
另外，ovfshlluw 和 ovfshllul 函数在 CPU 不为 H8/300 时有效。

头文件　<machine.h>

返回值	溢出	0 以外的值
	不溢出	0
参　数	dst	位移位运算的操作数
	rst	结果的保存位置（rst 为 0 时不保存结果）

例

```
#include <machine.h>        /*  必须 include <machine.h>。              */
int dst;
void f(void)
{
    if(ovfshlluw(dst,0))    /*  用 BCC 判断 dst<<1 的结果。              */
        dst=0;
}
```

补数的溢出判断

*int ovfnegc(char dst, char *rst)*
*int ovfnegw(int dst, int *rst)*
*int ovfnegl(long dst, long *rst)*

说　明　ovfnegc、ovfnegw 和 ovfnegl 函数对 1 个字节、2 个字节、4 个字节的 dst 进行 2 的补数的计算，在 rst 不为 0 时，将结果保存到 rst 指向的区域。

然后，如果 2 的补数的结果不溢出就返回 0，否则返回 0 以外的值。

只能在判断 if 语句、do 语句、while 语句、for 语句的条件表达式中指定这些函数。

另外，ovfnegw 和 ovfnegl 函数在 CPU 不为 H8/300 时有效。

头文件　<machine.h>

返回值	溢出	0 以外的值
	不溢出	0
参　数	dst	补数运算的操作数
	rst	结果的保存位置（rst 为 0 时不保存结果）

例

```
#include <machine.h>        /* 必须 include <machine.h>。             */
int dst,rst;
void f(void)
{
    if(ovfnegw(dst,&rst))   /* 将 dst 的结果设定到 rst，根据-dst 的结果  */
        dst=0;              /* 的借位进行转移。                        */
}
```

10 进制加法

*void dadd(unsigned char size,char *ptr1,char *ptr2,char *rst)*

说　明	将从 ptr1 开始的 size 字节的数据和从 ptr2 开始的 size 字节的数据进行 10 进制加法运算，结果保存到从 rst 开始的 size 字节的区域。 size 为 1～255 的常数。
头文件	<machine.h>

参　数

size	数据长度
ptr1,ptr2	进行 10 进制加法运算的操作数
rst	结果的保存区

例

```
#include <machine.h>       /* 必须 include <machine.h>。           */
char ptr1[5]={0x01,0x23,0x45,0x67,0x89};       /* 10 进制 123456789  */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89};       /* 10 进制 123456789  */
char rst[5];
void main(void)
{
   dadd((char)5,ptr1,ptr2,rst);
                  /* 将 ptr1、ptr2 作为 10 位的 10 进制数进行加法运算。    */
}                 /* rst= 0x02,0x46,0x91,0x35,0x78                       */
```

10 进制减法

*void dsub(unsigned char size,char *ptr1,char *ptr2,char *rst)*

说　明	将从 `ptr1` 开始的 `size` 字节的数据和从 `ptr2` 开始的 `size` 字节的数据进行 10 进制减法运算，结果保存到从 `rst` 开始的 `size` 字节的区域。 `size` 为 1～255 的常数。
头文件	`<machine.h>`

参　数

size	数据长度
ptr1,ptr2	进行 10 进制减法运算的操作数
rst	结果的保存位置

例

```
#include <machine.h>       /* 必须 include <machine.h>。            */
char ptr1[5]={0x10,0x00,0x00,0x00,0x00};      /* 10 进制 1000000000  */
char ptr2[5]={0x01,0x23,0x45,0x67,0x89};      /* 10 进制 0123456789  */
char rst[5];
void main(void)
{
    dsub((char)5,ptr1,ptr2,rst);
                  /* 将 ptr1、ptr2 作为 10 位的 10 进制数进行减法运算。   */
}                 /* rst=0x08,0x76,0x54,0x32,0x11                    */
```

10.3　C/C++库

10.3.1　标准 C 库

（1）库的概要

说明 C/C++语言中能标准利用的函数的 C 库函数的规格。在此概述库的结构、本节的阅读方法和术语，以下根据库的结构说明各库函数的规格。

（a）库的种类

所谓库是指以 C/C++语言的函数形式实现输入/输出和字符串操作等标准处理的函数。另外，能通过包含对应各处理单位的标准 include 文件使用这些库。

在标准 include 文件中定义了对应库的声明和使用那些库所需的宏名。

库的种类和对应的标准 include 文件如表 10.29 所示。

表 10.29　库的种类和对应的标准 include 文件

	库的种类	内容	标准 include 文件
1	用于程序诊断的库	输出程序诊断信息的库。	＜assert.h＞
2	用于字符操作的库	进行字符操作和检验的库。	＜ctype.h＞
3	用于数值计算的库	进行三角函数等数值计算的库。	＜math.h＞ ＜mathf.h＞
4	用于程序控制转移的库	支持函数之间控制转移的库。	＜setjmp.h＞
5	用于存取可变个数的实际参数的库	对具有可变个数的实际参数的函数，支持存取该实际参数的库。	＜stdarg.h＞
6	用于输入/输出的库	进行输入/输出操作的库。 通过使用<no_float.h>，提供不支持浮点的简易输入/输出函数。	＜stdio.h＞ ＜no_float.h＞
7	用于标准处理的库	进行存储区管理等 C 程序中的标准处理的库。	＜stdlib.h＞
8	用于字符串操作的库	进行字符串比较、复制等的库。	＜string.h＞

另外，除以上标准 include 文件以外，为了提高程序建立作业的效率，只由宏名的定义构成的标准 include 文件如表 10.30 所示。

表 10.30　由宏名定义构成的标准 include 文件

	标准 include 文件	内容
1	＜stddef.h＞	定义各标准 include 文件共用的宏名。
2	＜float.h＞	定义有关浮点数内部表现的各种限制值。
3	＜limits.h＞	定义有关编译程序内部处理的各种限制值。
4	＜errno.h＞	定义在库函数中发生错误时设定给 errno 的值。

（b）库的说明格式

按标准 include 文件分类说明库的各函数。这些分类由对标准 include 文件中定义的宏名和函数声明的说明（请参照**图 10.3**）以及对各函数的说明格式（请参照**图 10.4**）构成。

标准 include 文件的说明格式如**图 10.3**、函数的说明格式如**图 10.4** 所示。

项目号　＜标准 include 文件名＞

- 说明本 include 文件所具有的全部功能的概要。
- 将本 include 文件中定义和声明的名称分类成名称种类（【型】、【常数】、【变量】、【函数】）进行说明。为宏时，在名称种类的标题（【】内）或者名称的说明位置用（宏）标明。
- 在有实现定义规格或者本 include 文件中声明的函数具有共通的注意事项时，进行补充说明。

图 10.3　标准 include 文件的说明格式

表示功能的概要。

表示库函数的型（返回值和参数）。

说　　明	说明库函数的功能。
头 文 件	声明源的标准 include 文件名。
返 回 值	正常：　库函数正常结束时的值。 异常：　库函数异常结束时的值。
参　　数	说明参数的含义。
例	说明调用步骤。
错误条件	表示在库函数的处理中不能用返回值判断的错误发生条件。在发生这样的错误时，由编译程序定义的对应错误种类的值被设定到 errno*。
备　　考	补充说明或者使用注意事项。
实现定义规格	本编译程序的处理方法。

图 10.4　函数的说明格式

【注】*　errno 为保存库函数执行中发生的错误种类的变量，详细内容请参照“10.3.1（2）＜stddef.h＞”。

（c）库函数说明所使用的术语

(i)　流输入/输出

在数据的输入/输出中，如果每次按字符调用输入/输出函数，驱动输入/输出设备或者调用 OS 功能，就会降低效率。因此，通常准备了称为缓冲的存储区，一次性地将缓冲区内的数据进行输入/输出。

一方面，从程序来看，按每个字符调用输入/输出函数的方法比较简单。

库函数通过自动管理缓冲区，在程序中无需意识缓冲区的状态而能高效率地进行以字符为单位的输入/输出。

这样，为了高效率地实现数据的输入/输出而无需意识详细的手段，就能将输入/输出作为一个数据流进行编程的功能称为流输入/输出。

(ii)　FILE 结构体和文件指针

将流输入/输出所需要的缓冲区和其它信息存储到一个结构体中，并在标准 include 文件＜stdio.h＞中定义为 FILE。

在流输入/输出中，将全部文件作为具有 FILE 结构体的数据结构的文件进行处理。这样的文件称为流文件，指向此文件结构体的指针称为文件指针，用于指定输入/输出文件。

文件指针定义为

```
FILE *fp;
```

如果用 fopen 函数等打开文件，就能得到文件指针。但是，如果打开处理失败，就返回 NULL。必须注意：如果将 NULL 指针指定为其它流输入/输出函数，该函数就异常结束。打开文件时，必须检查文件指针的值。

(iii)　函数和宏

库函数的实现方法有函数和宏两种。

函数具有和一般用户建立的函数相同的接口，在连接时被结合。

宏由#define 语句定义在与该函数关联的标准 include 文件中。

关于宏，需要注意以下几点：

- 因为宏由预处理器自动展开，所以即使用户声明了同名的函数，也无法将宏置为无效。
- 作为宏的参数，如果指定有副作用的表达式（赋值表达式、增量、减量），就不保证其效果。

例：将计算参数绝对值的 MACRO 进行如下的宏定义：

定义 `#define  MACRO(a) ((a) >= 0 ? (a) : -(a))`时，

如果 `X=MACRO(a++)`在程序内，

就展开为 `X = ((a++) >= 0 ? (a++) : -(a++))`，a 增量 2 次，而且结果值也和最初的 a 的绝对值不同。

(iv)　EOF

EOF 是在从文件输入数据的函数（getc 函数、getchar 函数、fgetc 函数等）中文件结束（End Of File）时的返回值，被定义在标准 include 文件＜stdio.h＞中。

(v)　NULL

NULL 是什么也没有指向的指针值，被定义在标准 include 文件＜stddef.h＞中。

(vi)　空字符

用字符"\0"表示 C/C++语言中字符串的结束。库函数中字符串的参数也都必须遵循此规定。将表示此字符串结束的字符"\0"称为空字符。

(vii) 返回码

在库函数中有根据返回值判断被指定的处理是否成功等结果的函数。将此时的返回值特称为返回码。

(viii)文本文件和二进制文件

大部分系统具有用于保存数据的特殊文件格式。为支持数据的保存，库函数中有文本文件和二进制文件的 2 种文件格式。

- 文本文件
  文本文件是用于保存一般文本的文件，由行的集合构成。输入文本文件时，作为行的分隔符，输入换行字符（"\n"）。另外，输出时，通过输出换行字符（"\n"），结束当前行的输出。文本文件是用于进行保存各种处理的标准文本的文件输入/输出的文件。对于文本文件，通过库函数输入/输出的字符并不一定对应文件中的物理数据的排列。

- 二进制文件
  二进制文件是由字节数据串构成的文件。通过库函数输入/输出的数据对应文件中的物理数据的排列。

(ix)　标准输入/输出文件

将通过输入/输出的库函数不进行文件打开等准备而能标准使用的文件称为标准输入/输出文件。标准输入/输出文件中有标准输入文件（stdin）、标准输出文件（stdout）和标准错误输出文件（stderr）。

- 标准输入文件（stdin）
  输入到程序的标准文件。

- 标准输出文件(stdout)
  从程序输出的标准文件。

- 标准错误输出文件(stderr)
  用于从程序输出错误信息等的标准文件。

(x)　浮点数

浮点数是近似表现实数的数。在 C/C++语言的源程序中用 10 进制数表现浮点数。但是，在计算机内部通常用 2 进制数表现浮点数。在 2 进制数的情况下，浮点数的表现如下：

$2^n \times m$（n：整数、m：2 进制小数）

在此，n 称为浮点数的指数，m 称为尾数。为了用固定的数据长度表现浮点数，通常固定 n 和 m 的位数。以下说明有关浮点数的术语：

- 基数
  是表示用几进制表现浮点数的整数值。通常，基数为2。
- 舍入
  在将精度高于浮点数的运算中途结果保存为浮点数时进行舍入。舍入有上舍入、舍去和四舍五入（在2进制小数的情况下，为0舍1入）。
- 正规化
  以$2^n \times m$的格式表现浮点数时，同一数值可能有不同的表现。
  例：
  $2^5 \times 1.0_{(2)}$　（$_{(2)}$表示 2 进制数）
  $2^6 \times 0.1_{(2)}$
  都为相同的值。

  为了确保有效位数，通常使用如前一个的起始位不为0的表现，称它为正规化浮点数，将浮点数转换为如此表现的操作称为正规化。
- 保护位
  在保持浮点数运算的中途结果时，为了进行舍入，通常需要准备比实际浮点数多1位的数据。但是，如果仅此，就在发生位数减少时无法得到正确的结果。所以可使用再增加1位以保持运算中途结果的方法，此位称为保护位。

(xi)　文件存取模式

表示在打开文件时对文件进行何种处理的字符串。字符串的种类有 12 种，如表 10.31 所示。

表 10.31　文件存取模式的种类

	存取模式	含义
1	“r”	将文本文件打开为读模式。
2	“w”	将文本文件打开为写模式。
3	“a”	将文本文件打开为追加模式。
4	“rb”	将二进制文件打开为读模式。
5	“wb”	将二进制文件打开为写模式。
6	“ab”	将二进制文件打开为追加模式。
7	“r+”	将文本文件打开为读和更新模式。
8	“w+”	将文本文件打开为写和更新模式。
9	“a+”	将文本文件打开为追加和更新模式。
10	“r+b”	将二进制文件打开为读和更新模式。
11	“w+b”	将二进制文件打开为写和更新模式。
12	“a+b”	将二进制文件打开为追加和更新模式。

(xii) 实现定义

定义因编译程序而不同。

(xiii)错误指示符和文件结束指示符

按各流文件保持错误指示符和文件结束指示符的数据。其中，错误指示符表示在输入/输出文件时是否发生错误，文件结束指示符表示输入文件是否结束。

能分别通过 ferror 函数和 feof 函数参照这些数据。

在处理流文件的函数中，只从该函数的返回值可能无法得到错误发生和文件结束的信息。在执行这样的函数后能用错误指示符和文件结束指示符调查文件的状态。

(xiv)位置指示符

对于能从如磁盘文件等文件中任意位置读写的流文件，保持表示文件中的当前读写位置的数据，称该数据为位置指示符。对于不能更改如终端设备等文件中读写位置的流文件，不使用位置指示符。

（d）使用库时的注意事项

- 在库中定义的宏内容因各编译程序而不同。
  在使用库时，如果重新定义这些宏的内容，就不能保证运行。
- 库并不在任何时候都检测错误。如果不用以下说明中所示的格式调用库函数，就不能保证运行。

（2）＜stddef.h＞

定义标准 include 文件中共用的宏名。

以下全部为实现定义：

种类	定义名	说明
型（宏）	ptrdiff_t	二个指针相减后的结果的型。
	size_t	sizeof 运算符的运算结果的型。
常数（宏）	NULL	指针什么都不指向时的值。 用相等运算符（==）与 0 比较的结果为真的值。
变量（宏）	errno	如果在处理库函数中发生错误，就将定义在各库中的错误代码设定到 errno。 在调用库函数前将 errno 置 0，在结束库函数处理后，通过检查被设定在 errno 的代码，就能查出库函数处理中发生的错误。
函数（宏）	offsetof	以字节为单位，计算从结构体成员的结构体起始位置开始的偏移值。

实现定义规格

	项目	说明	
1	宏 NULL 的值	为指向 void 型的指针型的值 0	
2	宏 ptrdiff_t 的内容	int 型	用于 H8SX 正常模式 用于 H8SX 中级模式 用于 H8SX 高级模式并有 ptr16 选项 用于 H8SX 最高模式并有 ptr16 选项 用于 H8S/2600 正常模式 用于 H8S/2000 正常模式 用于 H8S/2600 高级模式并有 ptr16 选项 用于 H8S/2000 高级模式并有 ptr16 选项 用于 H8/300H 正常模式 用于 H8/300
		long 型	用于 H8SX 高级模式并没有 ptr16 选项 用于 H8SX 最高模式并没有 ptr16 选项 用于 H8S/2600 高级模式并没有 ptr16 选项 用于 H8S/2000 高级模式并没有 ptr16 选项 用于 H8S/300H 高级模式

（3）＜assert.h＞

在程序中附加诊断功能。

种类	定义名	说明
函数（宏）	assert	在程序中附加诊断功能。

为了使＜assert.h＞定义的诊断功能无效，必须在包含＜assert.h＞前，用#define 语句定义称为 NDEBUG 的宏名（#define NDEBUG）。

【注】　如果对称为 assert 的宏名使用#undef 语句，就不能保证以后的 assert 的调用效果。

诊断

void assert(int expression)

说　明	在程序中附加诊断功能。
头文件	`<assert.h>`
参　数	`expression`　　要评价的表达式

例

```
#include  <assert.h>
int  expression;
    assert (expression);
```

备　考	当 expression 为真时，assert 宏不返回值而结束处理。当 Expression 为伪时，以编译程序定义的格式，将诊断信息输出到标准错误文件，然后调用 abort 函数。 诊断信息中包含参数的程序文本、源文件名和源行号。
实现定义规格	对于 assert(expression)，当 expression 为伪时输出信息。 ASSERTION FAILED:Δ表达式ΔFILEΔ<文件名>,lineΔ<行号>

（4）＜ctype.h＞

对字符进行种类的判断和转换。

种类	定义名	说明
函数	isalnum	判断是否为英文字母或者 10 进制数字。
	isalpha	判断是否为英文字母。
	iscntrl	判断是否为控制字符。
	isdigit	判断是否为 10 进制数字。
	isgraph	判断是否为除空格字符以外的打印字符。
	islower	判断是否为英文小写字母。
	isprint	判断是否为含有空格字符的打印字符。
	ispunct	判断是否为特殊字符。
	isspace	判断是否为空格类字符。
	isupper	判断是否为英文大写字母。
	isxdigit	判断是否为 16 进制数字。
	tolower	将英文大写字母转换为英文小写字母。
	toupper	将英文小写字母转换为英文大写字母。

在上述的函数中，如果输入参数的值不包含在能用 unsigned char 型表现的范围内并且不为 EOF 时，就不能保证该函数的运行。另外，字符种类一览表如表 10.32 所示。

表 10.32　字符种类

	字符种类	内容
1	英文大写字母	为以下 26 个字母中的任意字母。 ‘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’
2	英文小写字母	为以下 26 个字母中的任意字母。 ‘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’
3	英文字母	为英文大小写字母中的任意字母。
4	10 进制数字	为以下 10 个字符中的任意字符。 ‘0’、‘1’、‘2’、‘3’、‘4’、‘5’、‘6’、‘7’、‘8’、‘9’
5	打印字符	包含空格(‘ ’)的且显示在显示器上的字符。 对应 ASCII 码的 0x20～0x7E。
6	控制字符	打印字符以外的字符。
7	空格类字符	为以下 6 个字符中的任意字符。 空格(‘ ’)、换页(‘\f’)、换行(‘\n’)、回车(‘\r’)、水平制表符(‘\t’)、垂直制表符(‘\v’)
8	16 进制数字	为以下 22 个字符中的任意字符。 ‘0’、‘1’、‘2’、‘3’、‘4’、‘5’、‘6’、‘7’、‘8’、‘9’、 ‘A’、‘B’、‘C’、‘D’、‘E’、‘F’、‘a’、‘b’、‘c’、‘d、‘e’、‘f’
9	特殊字符	为除空格(‘ ’)、英文字母和 10 进制数字以外的任意打印字符。

实现定义的规格

	项目	编译程序的规格
1	通过 isalnum 函数、isalpha 函数、iscntrl 函数、islower 函数、isprint 函数和 isupper 函数判断的字符集	能用 unsigned char 型表现的字符集。判断的结果为真的字符如表 10.33 所示。

表 10.33　为真的字符集

	函数名	为真的字符
1	isalnum	‘0’～‘9’, ‘A’～‘Z’, ‘a’～‘z’
2	isalpha	‘A’～‘Z’, ‘a’～‘z’
3	iscntrl	‘\x00’～‘\x1f’, ‘\x7f’
4	islower	‘a’～‘z’
5	isprint	‘\x20’～‘\x7E’
6	isupper	‘A’～‘Z’

英文字母和 10 进制数字的判断

int isalnum(int c)

说　明　　判断字符是否为英文字母或者 10 进制数字。

头文件　　`<ctype.h>`

返回值　　字符 c 为英文字母或者 10 进制数字时　　：0 以外
字符 c 不为英文字母和 10 进制数字时　　：0

参　数　　c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isalnum(c);
```

英文字母的判断

int isalpha(int c)

说　明	判断字符是否为英文字母。
头文件	<ctype.h>
返回值	字符 c 为英文字母时　：0 以外 字符 c 不为英文字母时　：0
参　数	c　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isalpha(c);
```

控制字符的判断

int iscntrl(int c)

说　明	判断字符是否为控制字符。
头文件	<ctype.h>
返回值	字符 c 为控制字符时　：0 以外 字符 c 不为控制字符时　：0
参　数	c　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=iscntrl(c);
```

10 进制数字的判断

int isdigit(int c)

说　明	判断字符是否为 10 进制数字。	
头文件	<ctype.h>	
返回值	字符 c 为 10 进制数字时 字符 c 不为 10 进制数字时	：0 以外 ：0
参　数	c	要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isdigit(c);
```

非空格字符的打印字符的判断

int isgraph(int c)

说　明	判断字符是否为非空格字符(‘ ’)的任意打印字符。	
头文件	<ctype.h>	
返回值	字符 c 为非空格字符的打印字符时 字符 c 不为非空格字符的打印字符时	：0 以外 ：0
参　数	c	要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isgraph(c);
```

英文小写字母的判断

int islower(int c)

说　明	判断字符是否为英文小写字母。
头文件	<ctype.h>
返回值	字符 c 为英文小写字母时　：0 以外 字符 c 不为英文小写字母时　：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=islower(c);
```

打印字符的判断

int isprint(int c)

说　明	判断字符是否为含有空格字符(‘　’)的打印字符。
头文件	<ctype.h>
返回值	字符 c 为含有空格字符的打印字符时　：0 以外 字符 c 不为含有空格字符的打印字符时：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isprint(c);
```

特殊字符的判断

int ispunct(int c)

说　明	判断字符是否为特殊字符。
头文件	`<ctype.h>`
返回值	字符 c 为特殊字符时　：0 以外 字符 c 不为特殊字符时　：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=ispunct(c);
```

空格类字符的判断

int isspace(int c)

说　明	判断字符是否为空格类字符。
头文件	`<ctype.h>`
返回值	字符 c 为空格类字符时　：0 以外 字符 c 不为空格类字符时　：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isspace(c);
```

英文大写字母的判断

int isupper(int c)

说　明	判断字符是否为英文大写字母。
头文件	`<ctype.h>`
返回值	字符 c 为英文大写字母时　：0 以外 字符 c 不为英文大写字母时　：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=isupper(c);
```

16 进制数字的判断

int isxdigit(int c)

说　明	判断字符是否为 16 进制数字。
头文件	`<ctype.h>`
返回值	字符 c 为 16 进制数字时　：0 以外 字符 c 不为 16 进制数字时　：0
参　数	c　　要判断的字符

例

```
#include  <ctype.h>
int  c, ret;
   ret=isxdigit(c);
```

转换为英文小写字母

int tolower(int c)

说　明	将英文大写字母转换为对应的英文小写字母。
头文件	<ctype.h>
返回值	字符 c 为英文大写字母时　：与字符 c 对应的英文小写字母 字符 c 不为英文大写字母时　：字符 c
参　数	c　要转换的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=tolower(c);
```

转换为英文大写字母

int toupper(int c)

说　明	将英文小写字母转换为对应的英文大写字母。
头文件	<ctype.h>
返回值	字符 c 为英文小写字母时　：与字符 c 对应的英文大写字母 字符 c 不为英文小写字母时　：字符 c
参　数	c　要转换的字符

例

```
#include  <ctype.h>
int  c, ret;
    ret=toupper(c);
```

（5）＜float.h＞

定义有关浮点数内部表现的各种限制值。

以下全部为实现定义：

种类	定义名	定义值	说明
常数（宏）	FLT_RADIX	2	指数表现中的基数。
	FLT_ROUNDS	1	表示是否舍入加法运算结果。本宏定义的含义如下： ・舍入运算结果时　：正值 ・舍去运算结果时　：0 ・不特别规定时　　：－1 舍入和舍去方法为实现定义。
	FLT_GUARD	1	表示在乘法运算结果中是否使用保护位。本宏定义的含义如下： ・使用保护位　：1 ・不使用保护位：0
	FLT_NORMALIZE	1	表示是否将浮点数的值进行正规化处理。本宏定义的含义如下： ・正规化　：1 ・不正规化：0
	FLT_MAX	3.4028235677973364e+38F	作为浮点数值，float 型能表现的最大值。
	DBL_MAX	1.7976931348623158e+308	作为浮点数值，double 型能表现的最大值。
	LDBL_MAX	1.7976931348623158e+308	作为浮点数值，long double 型能表现的最大值。
	FLT_MAX_EXP	127	作为浮点数值，float 型能表现的基数乘方的最大值。
	DBL_MAX_EXP	1023	作为浮点数值，double 型能表现的基数乘方的最大值。
	LDBL_MAX_EXP	1023	作为浮点数值，long double 型能表现的基数乘方的最大值。
	FLT_MAX_10_EXP	38	作为浮点数值，float 型能表现的 10 的乘方的最大值。
	DBL_MAX_10_EXP	308	作为浮点数值，double 型能表现的 10 的乘方的最大值。
	LDBL_MAX_10_EXP	308	作为浮点数值，long double 型能表现的 10 的乘方的最大值。
	FLT_MIN	1.175494351e-38F	作为浮点数值，float 型能表现的正值的最小值。
	DBL_MIN	2.2250738585072014e-308	作为浮点数值，double 型能表现的正值的最小值。
	LDBL_MIN	2.2250738585072014e-308	作为浮点数值，long double 型能表现的正值的最小值。
	FLT_MIN_EXP	-149	作为正值，float 型能表现的浮点数值的基数乘方的最小值。
	DBL_MIN_EXP	-1074	作为正值，double 型能表现的浮点数值的基数乘方的最小值。

种类	定义名	定义值	说明
常数（宏）	LDBL_MIN_EXP	-1074	作为正值，long double 型能表现的浮点数值的基数乘方的最小值。
	FLT_MIN_10_EXP	-44	作为正值，float 型能表现的浮点数值的 10 的乘方的最小值。
	DBL_MIN_10_EXP	-323	作为正值，double 型能表现的浮点数值的 10 的乘方的最小值。
	LDBL_MIN_10_EXP	-323	作为正值，long double 型能表现的浮点数值的 10 的乘方的最小值。
	FLT_DIG	6	float 型浮点数值的 10 进制精度的最大位数。
	DBL_DIG	15	double 型浮点数值的 10 进制精度的最大位数。
	LDBL_DIG	15	long double 型浮点数值的 10 进制精度的最大位数。
	FLT_MANT_DIG	24	将 float 型的浮点数值按基数表现时的尾数的最大位数。
	DBL_MANT_DIG	53	将 double 型的浮点数值按基数表现时的尾数的最大位数。
	LDBL_MANT_DIG	53	将 long double 型的浮点数值按基数表现时的尾数的最大位数。
	FLT_EXP_DIG	8	将 float 型的浮点数值按基数表现时的指数的最大位数。
	DBL_EXP_DIG	11	将 double 型的浮点数值按基数表现时的指数的最大位数。
	LDBL_EXP_DIG	11	将 long double 型的浮点数值按基数表现时的指数的最大位数。
	FLT_POS_EPS	5.9604648328104311e-8F	表示在 float 型中 1.0＋x≠1.0 的最小浮点数值 x。
	DBL_POS_EPS	1.1102230246251567e-16	表示在 double 型中 1.0＋x≠1.0 的最小浮点数值 x。
	LDBL_POS_EPS	1.1102230246251567e-16	表示在 long double 型中 1.0＋x≠1.0 的最小浮点数值 x。
	FLT_NEG_EPS	2.9802324164052156e-8F	表示在 float 型中 1.0－x≠1.0 的最小浮点数值 x。
	DBL_NEG_EPS	5.5511151231257834e-17	表示在 double 型中 1.0－x≠1.0 的最小浮点数值 x。
	LDBL_NEG_EPS	5.5511151231257834e-17	表示在 long double 型中 1.0－x≠1.0 的最小浮点数值 x。
	FLT_POS_EPS_EXP	-23	表示在 float 型中 1.0＋(基数)n≠1.0 的最小整数 n。
	DBL_POS_EPS_EXP	-52	表示在 double 型中 1.0＋(基数)n≠1.0 的最小整数 n。
	LDBL_POS_EPS_EXP	-52	表示在 long double 型中 1.0＋(基数)n≠1.0 的最小整数 n。

种类	定义名	定义值	说明
常数（宏）	FLT_NEG_EPS_EXP	-24	表示在 float 型中 1.0－(基数)n≠1.0 的最小整数 n。
	DBL_NEG_EPS_EXP	-53	表示在 double 型中 1.0－(基数)n≠1.0 的最小整数 n。
	LDBL_NEG_EPS_EXP	-53	表示在 long double 型中 1.0－(基数)n≠1.0 的最小整数 n。

（6）＜limits.h＞

定义有关整型数据内部表现的各种限制值。

以下全部为实现定义：

种类	定义名	定义值	说明
常数（宏）	CHAR_BIT	8	表示 char 型由几位构成。
	CHAR_MAX	127	作为值，char 型变量具有的最大值。
	CHAR_MIN	-128	作为值，char 型变量具有的最小值。
	SCHAR_MAX	127	作为值，signed char 型变量具有的最大值。
	SCHAR_MIN	-128	作为值，signed char 型变量具有的最小值。
	UCHAR_MAX	255u	作为值，unsigned char 型变量具有的最大值。
	SHRT_MAX	32767	作为值，short 型变量具有的最大值。
	SHRT_MIN	-32768	作为值，short 型变量具有的最小值。
	USHRT_MAX	65535u	作为值，unsigned short 型变量具有的最大值。
	INT_MAX	32767	作为值，int 型变量具有的最大值。
	INT_MIN	-32768	作为值，int 型变量具有的最小值。
	UINT_MAX	65535u	作为值，unsigned int 型变量具有的最大值。
	LONG_MAX	2147483647	作为值，long 型变量具有的最大值。
	LONG_MIN	-2147483647L-1L	作为值，long 型变量具有的最小值。
	ULONG_MAX	4294967295u	作为值，unsigned long 型变量具有的最大值。

（7）＜errno.h＞

定义库函数中发生错误时给 errno 设定的值。

以下全部为实现定义：

种类	定义名	说明
变量（宏）	errno	int 型变量。在库函数中发生错误时，设定错误号。
常数（宏）	ERANGE	请参照“12.3　C 库函数的错误信息”。
	EDOM	
	EDIV	
	ESTRN	
	PTRERR	
	ECBASE	
	ETLN	
	EEXP	
	EEXPN	
	EFLOATO	
	EFLOATU	
	EDBLO	
	EDBLU	
	ELDBLO	
	ELDBLU	
	NOTOPN	
	EBADF	
	ECSPEC	

（8）＜math.h＞

计算各种数值。

以下常数（宏）全部为实现定义：

种类	定义名	说明
常数（宏）	EDOM	当输入到函数的参数值超出函数中定义的值的范围时，为设定在 errno 中的值。
	ERANGE	当函数计算结果不能表示为 double 型的值或者发生溢出/下溢时，为设定在 errno 中的值。
	HUGE_VAL	当函数的计算结果溢出时，为函数的返回值。
函数	acos	计算浮点数的反余弦。
	asin	计算浮点数的反正弦。
	atan	计算浮点数的反正切。
	atan2	计算浮点数之间相除后的结果的反正切。
	cos	计算浮点数的弧度值的余弦。
	sin	计算浮点数的弧度值的正弦。
	tan	计算浮点数的弧度值的正切。
	cosh	计算浮点数的双曲余弦。
	sinh	计算浮点数的双曲正弦。
	tanh	计算浮点数的双曲正切。
	exp	计算浮点数的指数函数。
	frexp	将浮点数分为[0.5, 1.0)的值和 2 的乘方的积。
	ldexp	计算浮点数和 2 的乘方的积。
	log	计算浮点数的自然对数。
	log10	计算以 10 为底的浮点数的对数。
	modf	将浮点数分为整数和小数。
	pow	计算浮点数的乘方。
	sqrt	计算浮点数的正平方根。
	ceil	求上舍入浮点数的小数部分后的整数值。
	fabs	计算浮点数的绝对值。
	floor	求舍去浮点数的小数部分后的整数值。
	fmod	计算浮点数之间相除后的结果的余数。

以下说明发生错误时的运行：

（1）定义域错误

当输入到函数的参数值超出函数中定义的值范围时，发生定义域错误。此时，将EDOM的值设定到errno。另外，函数的返回值为实现定义。

（2）范围错误

当函数中的计算结果不能表示为double型的值时，发生范围错误。此时，将ERANGE的值设定到errno。另外，当计算结果溢出时，将和正确进行计算时相同符号的HUGE_VAL值作为返回值返回。当计算结果下溢时，将0作为返回值返回。

【注】 1. 在可能因调用<math.h>的函数而发生定义域错误时，直接使用结果值很危险，必须检查errno后再使用。

例：

```
          .
          .
          .
1    x=asin(a);
2    if (errno==EDOM)
             printf ("error\n");
4    else
5            printf ("result is : %lf\n", x);
          .
          .
          .
```

在第 1 行，使用 asin 函数求反正弦值。此时，如果参数 a 的值超出 asin 函数的定义域[-1.0, 1.0］的范围，就将 EDOM 的值设定到 errno。在第 2 行，判断是否发生定义域错误。如果发生定义域错误，就在第 3 行输出 error。否则，就在第 5 行输出反正弦值。

2. 是否发生范围错误根据编译程序的浮点数内部表现格式而不同。例如：在采用能将无穷大作为值表现的内部表现格式时，能不让范围错误发生而实现＜math.h＞的库函数。

实现定义规格

	项目	编译程序的规格
1	数学函数的输入参数值超出范围时的数学函数的返回值	返回非数值。非数值的格式请参照“10.1.3 浮点数的规格”。
2	如果在数学函数中发生下溢错误，是否将宏“ERANGE”的值设定到“errno”	不设定。
3	当 fmod 函数中第 2 个实际参数的值为 0 时，是否为范围错误	为范围错误。

反余弦

double acos(double d)

说　明　　计算浮点数的反余弦。

头文件　　`<math.h>`

返回值　　正常：　d 的反余弦值
　　　　　异常：　定义域错误时，返回值为非数值。

参　数　　d　　　　　　要求反余弦的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=acos(d);
```

错误条件　　当 d 的值超出[－1.0，1.0]的范围时，为定义域错误。

备　考　　acos 函数返回值的范围为[0，π]。

反正弦

double asin(double d)

说　明　　计算浮点数的反正弦。

头文件　　`<math.h>`

返回值　　正常：　d 的反正弦值
　　　　　异常：　定义域错误时，返回值为非数值。

参　数　　d　　　　　　要求反正弦的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=asin(d);
```

错误条件　　当 d 的值超出[－1.0，1.0]的范围时，为定义域错误。

备　考　　asin 函数返回值的范围为[－π/2，π/2]。

反正切

double atan(double d)

说　明	计算浮点数的反正切。	
头文件	`<math.h>`	
返回值	d 的反正切值	
参　数	d	要求反正切的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=atan(d);
```

备　考　atan 函数返回值的范围为(－π/2, π/2)。

除法运算后的反正切

double atan2(double y, double x)

说　明　　计算浮点数之间相除后的结果的反正切。

头文件　　`<math.h>`

返回值　　正常：　x 除 y 的反正切值
异常：　定义域错误时，返回值为非数值。

参　数

x	除数
y	被除数

例

```
#include  <math.h>
double  x, y, ret;
    ret=atan2(y, x);
```

错误条件　　当 x 和 y 的值都是 0.0 时，为定义域错误。

备　考　　atan2 函数返回值的范围为(－π， π］。atan2 函数表示的含义如图 10.5 所示。atan2 函数的结果是求通过点(x， y)和原点的直线与 x 轴的角度。当 y＝0.0 并且 x 为负时，结果为π；当 x＝0.0 时，根据 y 值的正负，结果为±π/2。

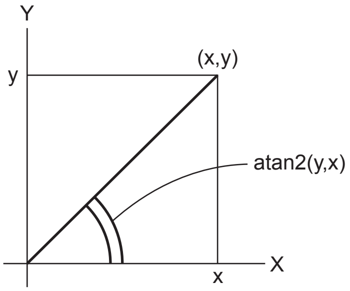


图 10.5　atan2 函数的含义

余弦

double cos(double d)

说　明	计算浮点数的弧度值的余弦。
头文件	`<math.h>`
返回值	d 的余弦值
参　数	d　　要求余弦的弧度值

例

```
#include  <math.h>
double  d, ret;
    ret=cos(d);
```

正弦

double sin(double d)

说　明	计算浮点数的弧度值的正弦。
头文件	`<math.h>`
返回值	d 的正弦值
参　数	d　　要求正弦的弧度值

例

```
#include  <math.h>
double  d, ret;
    ret=sin(d);
```

正切

double tan(double d)

说　明	计算浮点数的弧度值的正切。
头文件	`<math.h>`
返回值	d 的正切值
参　数	d　要求正切的弧度值

例

```
#include  <math.h>
double  d, ret;
    ret=tan(d);
```

双曲余弦

double cosh(double d)

说　明	计算浮点数的双曲余弦。
头文件	`<math.h>`
返回值	d 的双曲余弦值
参　数	d　要求双曲余弦的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=cosh(d);
```

双曲正弦

double sinh(double d)

说　明	计算浮点数的双曲正弦。	
头文件	`<math.h>`	
返回值	d 的双曲正弦值	
参　数	d	要求双曲正弦的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=sinh(d);
```

双曲正切

double tanh(double d)

说　明	计算浮点数的双曲正切。	
头文件	`<math.h>`	
返回值	d 的双曲正切值	
参　数	d	要求双曲正切的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=tanh(d);
```

指数函数

double exp(double d)

说　明　　计算浮点数的指数函数。

头文件　　`<math.h>`

返回值　　d 的指数函数值

参　数

d	要求指数函数的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=exp(d);
```

将浮点数分为尾数和指数

*double frexp(double value, int *e)*

说　明　　将浮点数分为[0.5, 1.0)的值和 2 的乘方的积。

头文件　　`<math.h>`

返回值　　value 为 0.0：0.0

value 不为 0.0：用 $\mathtt{ret*2}^{\mathtt{e}\,\text{指示区的值}}$ = value 定义的 ret 值

参　数

value	要分为[0.5, 1.0)的值和 2 的乘方的积的浮点数
e	指向保存 2 的乘方值的存储区的指针

例

```
#include  <math.h>
double  ret, value;
int  *e;
    ret=frexp(value, e);
```

备　考　　frexp 函数将 value 分为[0.5, 1.0)的值和 2 的乘方的积。将分解后的结果的 2 的乘方值设定到 e 指向的区域。

返回值 ret 的范围为[0.5, 1.0)或者 0.0。

如果 value 为 0.0，e 指向的 int 型存储区的内容和 ret 值为 0.0。

将尾数和指数转换为浮点数

double ldexp(double e, int f)

说　明　　计算浮点数和 2 的乘方的积。

头文件　　`<math.h>`

返回值　　$e*2^{f}$运算结果的值

参　数

e	要求 2 的乘方值的浮点数
f	2 的乘方值

例

```
#include  <math.h>
double  ret, e;
int  f;
    ret=ldexp(e, f);
```

自然对数

double log(double d)

说　明　　计算浮点数的自然对数。

头文件　　`<math.h>`

返回值

正常：　d 的自然对数值
异常：　定义域错误时，返回值为非数值。

参　数

d	要求自然对数的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=log(d);
```

错误条件

当 d 的值是负值时，为定义域错误。
当 d 的值是 0.0 时，为范围错误。

常用对数

double log10(double d)

说　明	计算以 10 为底的浮点数的对数。
头文件	`<math.h>`
返回值	正常：　d 是以 10 为底的对数值 异常：　当定义域错误时，返回值为非数值。
参　数	d　　要求以 10 为底的对数的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=log10(d);
```

错误条件

当 d 的值是负值时，为定义域错误。
当 d 的值是 0.0 时，为范围错误。

将浮点数分为整数和小数

*double modf(double a, double *b)*

说　明	将浮点数分为整数和小数。
头文件	`<math.h>`
返回值	a 的小数
参　数	a　　要分为整数和小数的浮点数 b　　指向保存整数的存储区的指针

例

```
#include  <math.h>
double  a, *b, ret;
    ret=modf(a, b);
```

乘方

double pow(double x, double y)

说　明	计算浮点数的乘方。
头文件	`<math.h>`
返回值	正常：　x 的 y 乘方的值 异常：　定义域错误时，返回值为非数值。
参　数	x　　被乘方的值 y　　乘方的值

例

```
#include  <math.h>
double  x, y, ret;
    ret=pow(x, y);
```

错误条件　当 x 的值为 0.0 并且 y 的值在 0.0 以下，或者 x 的值为负并且 y 的值不为整数时，为定义域错误。

平方根

double sqrt(double d)

说　明	计算浮点数的正平方根。
头文件	`<math.h>`
返回值	正常：　d 的正平方根的值 异常：　定义域错误时，返回值为非数值。
参　数	d　　要求正平方根的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=sqrt(d);
```

错误条件　当 d 的值是负值时，为定义域错误。

上舍入

double ceil(double d)

说　明	求上舍入浮点数的小数部分后的整数值。
头文件	`<math.h>`
返回值	上舍入 d 的小数部分后的整数值
参　数	d　　要上舍入小数部分的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=ceil(d);
```

备　考　ceil 函数将大于或者等于 d 的最小整数值作为 double 型的值返回。因此，当 d 的值是负值时，返回舍去小数部分后的值。

绝对值

double fabs(double d)

说　明	计算浮点数的绝对值。
头文件	`<math.h>`
返回值	d 的绝对值
参　数	d　　要求绝对值的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=fabs(d);
```

舍去

double floor(double d)

说　明	求舍去浮点数小数部分后的整数值。
头文件	`<math.h>`
返回值	舍去 d 的小数部分后的整数值
参　数	d　　要舍去小数部分的浮点数

例

```
#include  <math.h>
double  d, ret;
    ret=floor(d);
```

备　考　floor 函数将不超出 d 的范围的最大整数值作为 double 型的值返回。因此，当 d 的值为负值时，返回上舍入小数部分后的值。

余数

double fmod(double x, double y)

说　明	计算浮点数之间相除后的结果的余数。
头文件	`<math.h>`
返回值	y 的值为 0.0：x y 的值不为 0.0：y 除 x 的结果的余数

参　数		
	x	被除数
	y	除数

例

```
#include  <math.h>
double  x, y, ret;
    ret=fmod(x, y);
```

备　考　在 fmod 函数中，参数 x 和 y、返回值 ret 有如下所示的关系：

　x=y*i+ret（i 为整数值）

另外，返回值 ret 的符号与 x 的符号相同。

在无法表现 x/y 的商时，不能保证结果的值。

（9）＜mathf.h＞

进行各种数值计算。

在<mathf.h>中声明了 ANSI 规格规定以外的单精度格式的数学函数和进行了宏定义。

各函数接受 float 型的参数，返回 float 型的值。

以下常数（宏）全部为实现定义：

种类	定义名	说明
常数（宏）	EDOM	当输入到函数的参数值超出函数中定义的值的范围时，为设定在 errno 中的值。
	ERANGE	当函数计算结果不能表示为 float 型的值或者发生溢出/下溢时，为设定在 errno 中的值。
	HUGE_VALF	当函数计算的结果溢出时，为函数的返回值。
函数	acosf	计算浮点数的反余弦。
	asinf	计算浮点数的反正弦
	atanf	计算浮点数的反正切。
	atan2f	计算浮点数之间相除后的结果的反正切。
	cosf	计算浮点数的弧度值的余弦。
	sinf	计算浮点数的弧度值的正弦。
	tanf	计算浮点数的弧度值的正切。
	coshf	计算浮点数的双曲余弦。
	sinhf	计算浮点数的双曲正弦。
	tanhf	计算浮点数的双曲正切。
	expf	计算浮点数的指数函数。
	frexpf	将浮点数分为[0.5, 1.0]的值和 2 的乘方的积。
	ldexpf	计算浮点数和 2 的乘方的积。
	logf	计算浮点数的自然对数。
	log10f	计算以 10 为底的浮点数的对数。
	modff	将浮点数分为整数和小数。
	powf	计算浮点数的乘方。
	sqrtf	计算浮点数的正平方根。
	ceilf	求上舍入浮点数的小数部分后的整数值。
	fabsf	计算浮点数的绝对值。
	floorf	求舍去浮点数的小数部分后的整数值。
	fmodf	计算浮点数之间相除后结果的余数。

以下说明发生错误时的运行：

（1）定义域错误

当输入到函数的参数值超出函数中定义的值范围时，发生定义域错误。此时，将EDOM的值设定到errno。另外，函数的返回值为实现定义。

（2）范围错误

当函数中的计算结果不能表示为float型的值时，发生范围错误。此时，将ERANGE的值设定到errno。另外，当计算结果溢出时，将和正确进行计算时相同符号的HUGE_VALF值作为返回值返回。当计算结果下溢时，将0作为返回值返回。

【注】1. 在可能因调用<mathf.h>的函数而发生定义域错误时，直接使用结果值很危险，必须检查errno后再使用。

例：

```
          .
          .
          .
1   x=asinf(a);
2   if (errno==EDOM)
            printf ("error\n");
4   else
5           printf ("result is : %f\n", x);
          .
          .
          .
```

在第 1 行，使用 asinf 函数求反正弦值。此时，如果参数 a 的值超出 asinf 函数的定义域［-1.0, 1.0］的范围，就将 EDOM 的值设定到 errno。在第 2 行，判断是否发生定义域错误。如果发生定义域错误，就在第 3 行输出 error。否则，就在第 5 行输出反正弦值。

2. 是否发生范围错误根据编译程序的浮点数内部表现格式而不同。例如：在采用能将无穷大作为值表现的内部表现格式时，能不让范围错误发生而实现＜mathf.h＞的库函数。

实现定义规格

	项目	编译程序的规格
1	数学函数的输入参数值超出范围时的数学函数的返回值	返回非数值。非数值的格式请参照“10.1.3 浮点数的规格”。
2	如果在数学函数中发生下溢错误，是否将宏“ERANGE”的值设定到“errno”	不设定。
3	当 fmod 函数中第 2 个实际参数的值为 0 时，是否为范围错误	为范围错误。

反余弦

float acosf(float f)

说　明	计算浮点数的反余弦。
头文件	`<mathf.h>`
返回值	正常：　f 的反余弦值 异常：　定义域错误时，返回值为非数值。
参　数	f　　　　　　　要求反余弦的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=acosf(f);
```

错误条件	当 f 的值超出[－1.0, 1.0]的范围时，为定义域错误。
备　考	acosf 函数返回值的范围为[0, π]。

反正弦

float asinf(float f)

说　明	计算浮点数的反正弦。
头文件	`<mathf.h>`
返回值	正常：　f 的反正弦值 异常：　定义域错误时，返回值为非数值。
参　数	f　　　　　　　要求反正弦的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=asinf(f);
```

错误条件	当 f 的值超出[-1.0, 1.0]的范围时，为定义域错误。
备　考	asinf 函数返回值的范围为[-π/2, π/2]。

反正切

float atanf(float f)

说　明	计算浮点数的反正切。	
头文件	`<mathf.h>`	
返回值	`f` 的反正切值	
参　数	`f`	要求反正切的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=atanf(f);
```

备　考　`atanf` 函数返回值的范围为(－π/2, π/2)。

除法运算后的反正切

float atan2f(float y, float x)

说　明	计算浮点数之间相除后的结果的反正切。
头文件	`<mathf.h>`
返回值	正常：　x 除 y 的反正切值 异常：　定义域错误时，返回值为非数值。
参　数	x　除数 y　被除数

例

```
#include  <mathf.h>
float  x, y, ret;
    ret=atan2f(y, x);
```

错误条件　当 x 和 y 的值都为 0.0 时，为定义域错误。

备　考　atan2f 函数返回值的范围为(－π，π]。atan2f 函数表示的含义如图 10.6 所示。atan2f 函数的结果是求通过点(x，y)和原点的直线与 x 轴的角度。当 y＝0.0 并且 x 为负时，结果为π；当 x＝0.0 时，根据 y 的值的正负，结果为±π/2。

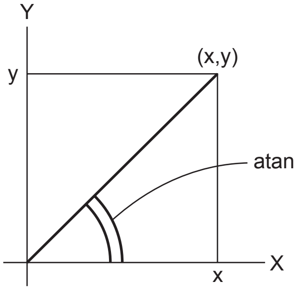


图 10.6　atan2f 函数的含义

余弦

float cosf(float f)

说　明	计算浮点数的弧度值的余弦。
头文件	`<mathf.h>`
返回值	f 的余弦值
参　数	f　　　　　　要求余弦的弧度值

例

```
#include  <mathf.h>
float  f, ret;
    ret=cosf(f);
```

正弦

float sinf(float f)

说　明	计算浮点数的弧度值的正弦。
头文件	`<mathf.h>`
返回值	f 的正弦值
参　数	f　　　　　　要求正弦的弧度值

例

```
#include  <mathf.h>
float  f, ret;
    ret=sinf(f);
```

正切

float tanf(float f)

说　明	计算浮点数的弧度值的正切。
头文件	`<mathf.h>`
返回值	f 的正切值
参　数	f　　　　　　要求正切的弧度值

例

```
#include  <mathf.h>
float  f, ret;
    ret=tanf(f);
```

双曲余弦

float coshf(float f)

说　明	计算浮点数的双曲余弦。
头文件	`<mathf.h>`
返回值	f 的双曲余弦值
参　数	f　　　　　　要求双曲余弦的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=coshf(f);
```

双曲正弦

float sinhf(float f)

说　明	计算浮点数的双曲正弦。	
头文件	`<mathf.h>`	
返回值	f 的双曲正弦值	
参　数	f	要求双曲正弦的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=sinhf(f);
```

双曲正切

float tanhf(float f)

说　明	计算浮点数的双曲正切。	
头文件	`<mathf.h>`	
返回值	f 的双曲正切值	
参　数	f	要求双曲正切的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=tanhf(f);
```

指数函数

float expf(float f)

说　明　　计算浮点数的指数函数。

头文件　　`<mathf.h>`

返回值　　f 的指数函数值

参　数　　f　　　　要求指数函数的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=expf(f);
```

将浮点数分为尾数和指数

float frexpf(float value, float ret)

说　明　　将浮点数分为[0.5, 1.0)的值和 2 的乘方的积。

头文件　　`<mathf.h>`

返回值　　正常：　value 为 0.0：0.0

value 不为 0.0：ret*$2^{e\ \text{指示区的值}}$=用 value 定义的 ret 值

参　数　　value　　　　要分解为[0.5, 1.0)的值和 2 的乘方积的浮点数

e　　　　指向保存 2 的乘方值的存储区的指针

例

```
#include  <mathf.h>
float  ret, value;
int  *e;
    ret=frexpf(value, e);
```

备　考　　frexpf 函数将 value 分为[0.5, 1.0)的值和 2 的乘方的积。将分解后的结果的 2 的乘方值设定到 e 指向的区域。

返回值 ret 的范围为[0.5, 1.0)或者 0.0。

如果 value 为 0.0，e 指向的 int 型存储区的内容和 ret 的值为 0.0。

将尾数和指数转换为浮点数

float ldexpf(float ret, int f)

说　明　　计算浮点数和 2 的乘方的积。

头文件　　`<mathf.h>`

返回值　　$e*2^{f}$ 运算结果的值

参　数

e	要求 2 的乘方值的浮点数
f	2 的乘方值

例

```
#include  <mathf.h>
float  ret, e;
int  f;
    ret=ldexpf(e, f);
```

自然对数

float logf(float f)

说　明　　计算浮点数的自然对数。

头文件　　`<mathf.h>`

返回值　　正常：　f 的自然对数值
　　　　　异常：　定义域错误时，返回值为非数值。

参　数

f	要求自然对数的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=logf(f);
```

错误条件　　当 f 的值是负值时，为定义域错误。
　　　　　　当 f 的值为 0.0，为范围错误。

常用对数

float log10f(float f)

说　明	计算以 10 为底的浮点数的对数。
头文件	`<mathf.h>`
返回值	正常：　f 是以 10 为底的对数值 异常：　定义域错误时，返回值为非数值。
参　数	f　　要求以 10 为底的对数的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=log10f(f);
```

错误条件　当 f 的值是负值时，为定义域错误。
当 f 的值为 0.0 时，为范围错误。

将浮点数分为整数和小数

*float modff(float a, float *b)*

说　明	将浮点数分为整数和小数。
头文件	`<mathf.h>`
返回值	a 的小数

参　数		
	a	要分为整数和小数的浮点数
	b	指向保存整数的存储区的指针

例

```
#include  <mathf.h>
float  a, *b, ret;
    ret=modff(a, b);
```

乘方

float powf(float x, float y)

说　明	计算浮点数的乘方。
头文件	`<mathf.h>`
返回值	正常：　x 的 y 乘方的值 异常：　定义域错误时，返回值为非数值。
参　数	x　　被乘方的值 y　　乘方的值
例	

```
#include  <mathf.h>
float  x, y, ret;
    ret=powf(x, y);
```

错误条件	当 x 的值为 0.0 并且 y 的值在 0.0 以下，或者 x 的值为负值并且 y 的值不为整数值时，为定义域错误。

平方根

float sqrtf(float f)

说　明	计算浮点数的正平方根。
头文件	`<mathf.h>`
返回值	正常：　f 的正平方根的值 异常：　定义域错误时，返回值为非数值。
参　数	f　　要求正平方根的浮点数
例	

```
#include  <mathf.h>
float  f, ret;
    ret=sqrtf(f);
```

错误条件	当 f 的值是负值时，为定义域错误。

上舍入

float ceilf(float f)

说　明	求上舍入浮点数的小数部分后的整数值。	
头文件	`<mathf.h>`	
返回值	上舍入 f 的小数部分后的整数值	
参　数	f	要上舍入小数部分的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=ceilf(f);
```

备　考　ceilf 函数将大于或者等于 f 的最小整数值作为 float 型的值返回。因此，当 f 的值是负值时，返回舍去小数部分后的值。

绝对值

float fabsf(float f)

说　明	计算浮点数的绝对值。	
头文件	`<mathf.h>`	
返回值	f 的绝对值	
参　数	f	要求绝对值的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=fabsf(f);
```

舍去

float floorf(float f)

说　明	求舍去浮点数的小数部分后的整数值。
头文件	`<mathf.h>`
返回值	舍去 f 的小数部分后的整数值
参　数	f　要舍去小数部分的浮点数

例

```
#include  <mathf.h>
float  f, ret;
    ret=floorf(f);
```

备　考　floorf 函数将没有超出 f 值范围的最大整数值作为 float 型的值返回。因此，当 f 的值是负值时，返回上舍入小数部分后的值。

余数

float fmodf(float x, float y)

说　明	计算浮点数之间相除后的结果的余数。
头文件	`<mathf.h>`
返回值	y 的值为 0.0：x y 的值不为 0.0：y 除 x 的结果的余数
参　数	x　被除数 y　除数

例

```
#include  <mathf.h>
float  x, y, ret;
    ret=fmodf(x, y);
```

备　考　在 fmodf 函数中，参数 x 和 y、返回值 ret 有如下所示的关系：

　x=y*i+ret（i 为整数值）

另外，返回值 ret 的符号与 x 的符号相同。

在无法表现 x/y 的商时，不能保证结果的值。

（10）<setjmp.h>

支持函数之间的控制移动。

以下的宏为实现定义。

种类	定义名	说明
型（宏）	jmp_buf	是与为了预先保存可进行函数间控制移动的信息的存储区对应的型名。
函数	setjmp	将由当前执行中的函数 jmp_buf 定义的执行环境保存到指定的存储区。
	longjmp	恢复由 setjmp 函数保存的函数执行环境，将控制移动到调用 setjmp 函数的程序位置。

setjmp 函数保存当前函数的执行环境。此后，能通过调用 longjmp 函数返回到调用 setjmp 函数的程序位置。

使用 setjmp、longjmp 函数支持函数间的控制移动的例子如下所示：

```
 1    #include <stdio.h>
 2    #include <setjmp.h>
 3    jmp_buf env;
 4    void main( )
 5    {
 6
 7
 8            if (setjmp(env)!=0){
 9                    printf("return from longjmp\n");
10                    exit(0);
11            }
12            sub( );
13    }
14
15    void sub( )
16    {
17            printf("subroutine is running \n");
18            longjmp(env, 1);
19    }
```

【说明】

在第 8 行调用 setjmp 函数。此时，将调用 setjmp 函数的环境保存到 jmp_buf 型变量 env。此时的返回值为 0，所以接着调用下一个函数 sub。

通过在函数 sub 中被调用的 longjmp 函数，恢复保存在变量 env 的环境。其结果是程序犹如从第 8 行的 setjmp 函数返回。但是，此时的返回值为 longjmp 函数的第 2 个参数指定的值(1)。

其结果是执行第 9 行以后的内容。

全局 goto 的跳转目标的设定

int setjmp(jmp_buf env)

说　明	将当前执行中的函数的执行环境保存到指定的存储区。
头文件	`<setjmp.h>`
返回值	调用 setjmp 函数时：0 从 longjmp 函数返回时：0 以外
参　数	env　　指向要保存执行环境的存储区的指针

例

```
#include  <setjmp.h>
int  ret;
jmp_buf  env;
    ret=setjmp(env);
```

备　考

在 longjmp 函数中使用通过 setjmp 函数保存的执行环境。

调用 setjmp 函数时的返回值为 0，但是从 longjmp 函数返回时的返回值为 longjmp 函数指定的第 2 个参数的值。

从复杂的表达式中调用 setjmp 函数时，有可能丢失一部分表达式评价中途结果等当前执行环境，所以只能在 setjmp 函数结果和常数表达式比较的情况下使用 setjmp 函数，尽量不要在复杂的表达式中调用。

全局 goto

void longjmp(jmp_buf env, int ret)

说　明	恢复由 setjmp 函数保存的函数的执行环境，并且将控制移动到调用 setjmp 函数的程序位置。
头文件	`<setjmp.h>`
参　数	env　　指向要保存执行环境的存储区的指针 ret　　返回给 setjmp 函数的返回码

例

```
#include  <setjmp.h>
int  ret;
jmp_buf  env;
    longjmp(env, ret);
```

备　考

longjmp 函数从 env 指定的存储区恢复在同一个程序中最后被调用的 setjmp 函数所保存的函数的执行环境，并且将控制移动到调用该 setjmp 函数的程序位置。此时。longjmp 函数的 ret 作为 setjmp 函数的返回值返回。但是，在 ret 为 0 时，返回给 setjmp 函数的返回值为 1。

在不调用 setjmp 函数时或者在调用 setjmp 函数的函数已执行 return 语句时，不能保证 longjmp 函数的运行。

（11）＜stdarg.h＞

对于具有可变个数的参数的函数，能参照该参数。

以下全部为实现定义。

种类	定义名	说明
型（宏）	va_list	是为了参照可变个数的参数而通过 va_start、va_arg 和 va_end 宏共用的变量的型。
函数（宏）	va_start	为了参照可变个数的参数，进行初始设定处理。
	va_arg	对于具有可变个数参数的函数，能参照当前参照中的参数的下一个参数。
	va_end	对于具有可变个数的参数的函数，结束参数的参照。

使用本标准 include 文件定义的宏的程序例子如下所示：

```
 1    #include <stdio.h>
 2    #include <stdarg.h>
 3
 4    extern void prlist(int count, ...);
 5
 6    void main( )
 7    {
 8            prlist(1, 1);
 9            prlist(3, 4, 5, 6);
10            prlist(5, 1, 2, 3, 4, 5);
11    }
12
13    void prlist(int count, ...)
14    {
15            va_list ap;
16            int i;
17
18            va_start(ap, count);
19            for(i=0; i<count; i++)
20                    printf("%d", va_arg(ap, int));
21            putchar('\n');
22            va_end(ap);
23    }
```

【说明】

在此例中，第 1 个参数指定输出数据的个数，实现只输出后面参数的个数的函数 prlist。

在第 18 行用 va_start 对可变个数的参数的参照进行初始化。然后，每输出一个参数时，就根据 va_arg 宏参照下一个参数（第 20 行）。va_arg 宏将参数的型名（此时为 int 型） 指定为第 2 个参数。

如果结束参数的参照，就调用 va_end 宏（第 22 行）。

开始可变个数的参数的取出

void va_start(va_list ap, parmN)

说　明　　为了参照可变个数的参数，进行初始设定处理。

头文件　　`<stdarg.h>`

参　数

ap	要对可变个数的参数进行存取的变量
parmN	最右侧参数的标识符

例

```
#include  <stdarg.h>
void func(int count,...){
    va_list       ap;
    va_start(ap, count);
}
```

备　考　　va_start 宏对 va_arg 和 va_end 宏使用的 ap 进行初始化。
另外，给 parmN 指定外部函数定义中的参数排列的最右侧参数的标识符，即“, ...”前一个的标识符。
为了参照没有名称的可变个数的参数，必须最先执行 va_start 宏调用。

可变个数的参数的取出

type va_arg(va_list ap, type)

说　明　　对具有可变个数的参数的函数，可参照当前参照中的参数的下一个参数。

头文件　　`<stdarg.h>`

返回值　　参数值

参　数

ap	要对可变个数的参数进行存取的变量
type	存取的参数的型

例

```
#include  <stdarg.h>
va_list  ap;
int  ret;
    ret=va_arg(ap, int);
```

备　考　　将被 va_start 宏初始化的 va_list 型变量指定为第 1 个参数。
每次使用 va_arg 时更新 ap 的值，其结果，依次将可变个数的参数作为本宏的返回值返回。
在调用步骤的 type 处必须指定参照的参数的型。
ap 必须和被 va_start 初始化的 ap 相同。
在函数的参数中，对于 type 的型，指定 char 型、unsigned char 型、short 型、unsigned short 型和 float 型时，如果指定因型转换而改变长度的型，就不能正确地参照参数。如果指定这样的型，就不能保证运行。

结束可变个数的参数的取出

void va_end(va_list ap)

说　明	对具有可变个数的参数的函数，结束参数的参照。
头文件	`<stdarg.h>`
参　数	`ap`　　要参照可变个数的参数的变量

例

```
#include  <stdarg.h>
va_list  ap;
    va_end(ap);
```

备　考

ap 必须和被 va_star 初始化的 ap 相同。
另外，如果在从函数 return 前不调用 va_end 宏，就不能保证该函数的运行。

（12）＜stdio.h＞

进行有关流输入/输出文件的输入/输出处理。

以下常数（宏）全部为实现定义。

种类	定义名	说明
常数（宏）	FILE	预先保存指向流输入/输出处理所需的缓冲区的指针和错误指示符、结束指示符等各种控制信息的结构体的型。
	_IOFBF	作为缓冲区的使用方法，输入/输出处理使用全部缓冲区。
	_IOLBF	作为缓冲区的使用方法，输入/输出处理以行为单位使用缓冲区。
	_IONBF	作为缓冲区的使用方法，输入/输出处理不使用缓冲区。
	BUFSIZ	输入/输出处理所需的缓冲区的大小。
	EOF	表示文件的结尾（end_of_file）。
	L_tmpnam *[1]	为了保存由 tmpnam 函数生成的临时文件名的字符串所需的数组长度。
	SEEK_CUR	表示将文件的当前读写位置从当前位置移动到指定的偏移位置。
	SEEK_END	表示将文件的当前读写位置从文件结束位置移动到指定的偏移位置。
	SEEK_SET	表示将文件的当前读写位置从文件起始位置移动到指定的偏移位置。
	SYS_OPEN *[1]	在执行时能同时打开的文件数量。
	TMP_MAX *[1]	由 tmpnam 函数生成的固有文件名的个数的最大值。
	stderr	指向标准错误文件的文件指针。
	stdin	指向标准输入文件的文件指针。
	stdout	指向标准输出文件的文件指针。
函数	fclose	关闭流输入/输出文件。
	fflush	将流输入/输出文件的缓冲区内容输出到文件。
	fopen	用指定的文件名打开流输入/输出文件。
	freopen	关闭当前打开的流输入/输出文件，用指定的文件名重新打开新的文件。
	setbuf	在用户程序侧定义和设定流输入/输出缓冲区。
	setvbuf	在用户程序侧定义和设定流输入/输出缓冲区。
	fprintf	根据格式将数据输出到流输入/输出文件。
	fscanf	从流输入/输出文件输入数据后，根据格式转换。

【注】 *1 在本实现定义中，没被定义。

种类	定义名	说明
函数	printf	根据格式转换数据后，输出到标准输出文件（stdout）。
	scanf	从标准输入文件（stdin） 输入数据后，根据格式转换。
	sprintf	根据格式转换数据后，输出到指定的区域。
	sscanf	从指定的存储区输入数据后，根据格式转换。
	vfprintf	根据格式将可变个数的参数列表输出到指定的流输入/输出文件。
	vprintf	根据格式将可变个数的参数列表输出到标准输出文件。
	vsprintf	根据格式将可变个数的参数列表输出到指定的存储区。
	fgetc	从流输入/输出文件输入 1 个字符。
	fgets	从流输入/输出文件输入字符串。
	fputc	将 1 个字符输出到流输入/输出文件。
	fputs	将字符串输出到流输入/输出文件。
	getc	（宏）从流输入/输出文件输入 1 个字符。
	getchar	（宏）从标准输入文件输入 1 个字符。
	gets	从标准输入文件输入字符串。
	putc	（宏）将 1 个字符输出到流输入/输出文件。
	putchar	（宏）将 1 个字符输出到标准输出文件。
	puts	将字符串输出到标准输出文件。
	ungetc	给流输入/输出文件返回 1 个字符。
	fread	将数据从流输入/输出文件输入到指定的存储区。
	fwrite	将数据从存储区输出到流输入/输出文件。
	fseek	移动流输入/输出文件的当前读写位置。
	ftell	取得流输入/输出文件的当前读写位置。
	rewind	将流输入/输出文件的当前读写位置移动到文件的起始位置。
	clearerr	清除流输入/输出文件的错误状态。
	feof	判断流输入/输出文件是否结束。
	ferror	判断流输入/输出文件是否为错误状态。
	perror	给标准错误文件（stderr）输出对应错误号的错误信息。

实现定义规格

	项目	
1	是否需要表示输入文本最后行结束的换行字符	没规定。根据低级接口例程的规格。
2	在读取时，是否读取换行字符前的空格字符	
3	给写在二进制文件中的数据附加空字符数	
4	追加模式中的文件位置指示符的初始值	
5	在文本文件输出后，是否会丢失以后的文件数据	
6	文件缓冲规格	
7	是否存在长度为 0 的文件	
8	合法文件名的结构规则	
9	是否能同时打开相同的文件	
10	fprintf 函数中的%p 格式转换的输出格式	为 16 进制数输出
11	fscanf 函数中的%p 格式转换的输入格式 fscanf 函数中的转换字符“－”的含义	为 16 进制数输入。 如果“－”不在最初、最后或者不在“^”的后面，就表示前一个字符和后一个字符的范围。
12	由 fgetpos、ftell 函数设定的 errno 的值	不支持 fgetpos 函数。 对于 ftell 函数，没规定。根据低级接口例程的规格。
13	perror 函数生成的信息输出格式	信息的输出格式如(a)所示。
14	长度为 0 时的 calloc、malloc、realloc 函数的运行	分配 0 字节的区域。

(a) perror 函数的输出格式

<字符串>：<与设定在 error 中的错误号对应的错误信息>。

(b) 用 printf 函数或者 fprintf 函数显示浮点的无穷大和非数值时的格式如表 10.34 所示。

表 10.34　无穷大和非数值的显示格式

	值	显示格式
1	正无穷大	＋＋＋＋＋＋
2	负无穷大	－－－－－－
3	非数值	＊＊＊＊＊＊

对流输入/输出文件进行一连的输入/输出处理的程序例子如下所示：

```
 1   #include <stdio.h>
 2
 3   void main( )
 4   {
 5          int c;
 6          FILE *ifp, *ofp;
 7
 8          if ((ifp=fopen("INPUT.DAT","r"))==NULL){
 9                 fprintf(stderr,"cannot open input file\n");
10                 exit(1);
11          }
12          if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
13                 fprintf(stderr,"cannot open output file\n");
14                 exit(1);
15          }
16          while ((c=getc(ifp))!=EOF)
17                 putc(c, ofp);
18          fclose(ifp);
19          fclose(ofp);
20   }
```

【说明】

这是将文件 INPUT.DAT 的内容复制到文件 OUTPUT.DAT 的程序。

通过第 8 行的 fopen 函数和第 12 行的 fopen 函数分别打开文件 INPUT.DAT 和 OUTPUT.DAT。如果打开失败，NULL 就作为 fopen 函数的返回值被返回，所以输出错误信息后结束程序。

在 fopen 函数正常结束时，返回指向保存被打开的文件信息的数据（FILE 型）的指针，所以将它们分别设定到变量 ifp 和 ofp。

如果打开成功，就使用这些 FILE 型的数据进行输入/输出。

如果文件处理结束，就用 fclose 函数关闭文件。

文件的关闭

int fclose(FILE *fp)

说　明	关闭流输入/输出文件。
头文件	`<stdio.h>`
返回值	正常：　0 异常：　0 以外
参　数	fp　　　　　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fclose(fp);
```

备　考

fclose 函数关闭文件指针 fp 指向的流输入/输出文件。如果流输入/输出文件的输出文件处于打开状态并且在缓冲区中存在未输出的数据，fclose 函数就将它输出到文件后关闭。

另外，在系统自动分配输入/输出缓冲区时被释放。

缓冲区的转储清除

int fflush(FILE *fp)

说　明	将流输入/输出文件的缓冲区内容输出到文件。
头文件	`<stdio.h>`
返回值	正常：　0 异常：　0 以外
参　数	fp　　　　　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fflush(fp);
```

备　考

在流输入/输出文件的输出文件处于打开状态时，fflush 函数将文件指针 fp 指向的流输入/输出文件缓冲区中的未输出内容输出到文件。另外，在输入文件处于打开状态时，ungetc 函数的指定无效。

文件的打开

*FILE *fopen(const char *fname, const char *mode)*

说　明	用指定的文件名打开流输入/输出文件。
头文件	`<stdio.h>`
返回值	正常：　指向被打开的文件信息的文件指针 异常：　NULL
参　数	fname　　指向表示文件名的字符串的指针 mode　　指向表示文件存取模式的字符串的指针

例

```
#include  <stdio.h>
FILE  *ret;
const char  *fname, *mode;
    ret=fopen(fname, mode);
```

备　考

fopen 函数将 fname 指向的字符串作为文件名打开流输入/输出的文件。

在用写模式或者追加模式打开不存在的文件时，建立新文件。在用写模式打开已存在的文件时，删除以前写入的文件内容，从文件的开始位置进行写。

对于用追加模式打开的文件，从该文件的结束位置进行写处理。对于用更新模式打开的文件，能对此文件进行输入和输出二种处理。

但是，在输出处理后，如果不执行 fflush、fseek 或者 rewind 函数，就不能继续输入处理。同样地，在输入处理后，如果不执行 fflush、fseek 或者 rewind 函数，就不能继续输出处理。

另外，也有在表示文件存取模式的字符串后附加指示打开方法的字符。

文件的重新打开

*FILE *freopen(const char *fname, const char *mode, FILE *fp)*

说　明	关闭当前打开的流输入/输出文件，用指定的文件名重新打开新文件。
头文件	`<stdio.h>`
返回值	正常：　`fp` 异常：　`NULL`

参　数

`fname`	指向表示新文件名的字符串的指针
`mode`	指向表示文件存取模式的字符串的指针
`fp`	当前打开的流输入/输出文件的文件指针

例

```
#include  <stdio.h>
const char  *fname, *mode;
FILE  *ret, *fp;
    ret=freopen(fname, mode, fp);
```

备　考

`freopen` 函数首先关闭文件指针 `fp` 指向的流输入/输出文件（即使在不能正确进行此关闭处理时，也进行如下处理）。

接着，重新使用该 `fp` 指向的 `FILE` 结构体，将文件名 `fname` 表示的文件打开为流输入/输出文件。

`freopen` 函数在同时打开的文件数被限制时等有效。

`freopen` 函数通常返回和 `fp` 相同的值。但是，如果发生错误，就返回 `NULL`。

缓冲区的设定

*void setbuf(FILE *fp, char buf[BUFSIZ])*

说　明　　在用户程序中定义和设定流输入/输出缓冲区。

头文件　　`<stdio.h>`

参　数

	`fp`	文件指针
	`buf`	指向缓冲区的指针

例

```
#include  <stdio.h>
FILE  *fp;
char  buf[BUFSIZ];
    setbuf(fp, buf);
```

备　考　　对于文件指针 `fp` 指向的流输入/输出文件，`setbuf` 函数将 `buf` 指向的存储区定义为输入/输出缓冲区。作为结果，使用容量为 `BUFSIZ` 的缓冲区进行输入/输出处理。

缓冲控制

*int setvbuf(FILE *fp, char *buf, int type, size_t size)*

说　明	在用户程序中定义和设定流输入/输出缓冲区。
头文件	`<stdio.h>`
返回值	正常：　0 异常：　0 以外

参　数

fp	文件指针
buf	指向缓冲区的指针
type	缓冲区的管理方式
size	缓冲区的容量

例

```
#include  <stdio.h>
FILE  *fp;
char  *buf;
int  type, ret;
size_t  size;
    ret=setvbuf(fp, buf, type, size);
```

备　考

对文件指针 fp 指向的流输入/输出文件，setvbuf 函数将 buf 指向的存储区定义为输入/输出缓冲区。

此缓冲区有以下三种使用方法：

(a) 给 type 指定_IOFBF 时

使用缓冲区进行全部输入/输出处理。

(b) 给 type 指定_IOLBF 时

以行为单位使用缓冲区进行输入/输出处理。即在写有换行字符时、缓冲区满时或者请求输入时，从缓冲区取出输入/输出数据。

(c) 给 type 指定_IONBF 时

不使用缓冲区进行输入/输出处理。

setvbuf 函数通常返回 0。但是，在给 type 或者 size 设定非法值时或者在缓冲区的使用方法等请求不被接受时，就返回 0 以外的值。

不能在关闭被打开的流输入/输出文件前释放缓冲区。另外，必须在打开流输入/输出文件后并且在进行输入/输出处理之前使用 setvbuf 函数。

带格式的文件输出

*int fprintf(FILE *fp, const char *control [, arg] …)*

说　明　　根据格式将数据输出到流输入/输出文件。

头文件　　`<stdio.h>`

返回值

正常：　转换并输出的字符数
异常：　负值

参　数

fp	文件指针
control	指向表示格式的字符串的指针
arg, …	根据格式输出的数据的排列

例

```
#include  <stdio.h>
FILE  *fp;
const char  *control="%s";
int  ret;
char buffer[]="Hello World\n";
    ret=fprintf(fp, control, buffer);
```

备　考

`fprintf` 函数根据 `control` 指向的格式字符串，对参数 `arg` 进行转换和编辑，然后输出到文件指针 `fp` 指向的流输入/输出文件。

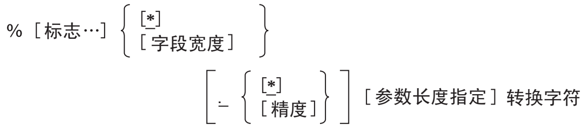
`fprintf` 函数通常返回转换和输出的数据个数。但是，如果发生错误，就返回负值。

格式规格如下：

【格式概要】

表示格式的字符串由 2 种字符串构成：

- 通常字符
  输出表示转换规格的字符串以外的字符。
- 转换规格
  转换规格以%开头的字符串指定后面参数的转换方法。转换规格的格式遵循以下规则：



此转换规格在没有实际输出参数时，不能保证其运行。另外，如果实际输出参数的个数比转换规格多，就忽视多余的全部参数。

【转换规格的说明】

(a)　标志

指定对带符号等输出数据的限定。能指定的标志种类和含义如表 10.35 所示。

表 10.35　标志的种类和含义

	项目	
1	－	在转换数据的字符数小于指定的字段宽度时，在字段内向左靠紧输出该数据。
2	＋	转换为带符号的数据时，根据该数据的符号，给转换数据的前头附加正号或者负号。
3	空格	在带符号的数据转换中，如果不给转换数据的前头附加符号，就给该数据的前头附加空格。 和“＋”一起使用时，忽视本标志。
4	＃	根据表 10.37 说明的转换种类，对转换后的数据进行限定。 1. c、d、l、s 和 u 转换时 忽视本标志。 2. o 转换时 在转换数据的前头附加 0。 3. x（或者 X）转换时 在转换数据的前头附加 0x（或者 0X）。 4. e、E、f、g 和 G 转换时 即使在转换数据中没有小数，也输出小数点。另外在 g 或者 G 转换时，删除转换后的数据后面附加的 0。

(b) 字段宽度

用任意的 10 进制数指定输出转换数据的字符数。

在转换数据的字符数小于字段宽度时，在该数据的前面附加空格（但是，如果指定‘－’标志，就在数据后面附加空格），使该数据的字符数和字段宽度相等。

如果转换数据的字符数大于字段宽度，就将字段宽度扩展到能输出转换结果的宽度。

另外，如果字段宽度指定以 0 开头，就在输出数据的前头附加字符“0”而不是空格。

(c) 精度

根据表 10.37 说明的转换种类，指定转换数据的精度。

在句点(.)后以连续 10 进制整数的格式指定精度。如果省略 10 进制整数，就假设指定为 0。

在指定的精度和指定的字段宽度产生矛盾时，字段宽度的指定无效。

各转换种类和精度指定的含义如下所示：

- d、i、o、u、x 和 X 转换时
  表示转换数据的最小位数。
- e、E 和 f 转换时
  表示转换数据的小数的位数。
- g 和 G 转换时
  表示转换数据的最大有效位数。
- s 转换时
  表示打印的最大字符数。

(d) 参数的长度指定

在 d、i、o、u、x、X、e、E、f、g 和 G 转换时（参照表 10.37），指定转换数据的长度（short 型、long 型和 long double 型）。在其它转换时，忽视本指定。长度指定的种类及其含义如表 10.36 所示。

表 10.36　参数长度指定的种类及其含义

种类		含义
1	h	在 d、i、o、u、x 和 X 转换中，指定转换数据为 short 型或者 unsigned short 型。
2	l	在 d、i、o、u、x 和 X 转换中，指定转换数据为 long 型、unsigned long 型或者 double 型。
3	L	在 e、E、f、g 和 G 转换中，指定转换数据为 long double 型。

(e) 转换字符

指定将转换数据转换为何种格式。

如果转换数据为结构体或者数组型或者指向这些型的指针，除了通过 s 转换进行字符数组转换和通过 p 转换进行指针转换，就不能保证其运行。

转换字符和转换方式如表 10.37 所示。如果将此表中未记述的英文字符指定为转换字符，就不能保证其运行。另外，指定除此以外的字符时的运行因编译程序而不同。

表10.37　转换字符和转换方式

	转换字符	转换种类	转换方式	转换对象的数据的型	有关精度的注意事项
1	d	d转换	将int型数据转换为带符号10进制数的字符串。d转换和i转换为相同规格。	int型	精度的指定表示最少输出的字符数。如果转换后的字符数小于精度值，就在字符串的前头附加0。在省略精度时，假设为1。另外，对于0值的数据，如果给精度指定0，就不输出任何内容。
2	i	i转换		int型	
3	o	o转换	将int型数据转换为无符号8进制数的字符串。	int型	
4	u	u转换	将int型数据转换为无符号10进制数的字符串。	int型	
5	x	x转换	将int型数据转换为无符号16进制数。16进制字符用a，b，c，d，e或者f。	int型	
6	X	X转换	将int型数据转换为无符号16进制数。16进制字符使用A，B，C，D，E或者F。	int型	
7	f	f转换	将double型数据转换为“［－］ddd.ddd”格式的10进制数的字符串。	double型	精度的指定表示小数的位数。存在小数时，必定在小数点前输出1位数字。在省略精度时，假设为6。另外，如果给精度指定0，就不输出小数点和小数。输出的数据被舍入。
8	e	e转换	将double型数据转换为“［－］d.ddde±dd”格式的10进制数的字符串。至少输出2位指数。	double型	精度的指定表示小数点的位数。转换字符在小数点前输出1位数字、在小数点后输出和精度相等位的数字。在省略精度时，假设为6。另外，如果给精度指定0，就不输出小数。输出的数据被舍入。
9	E	E转换	将double型数据转换为“［－］d.dddE±dd”格式的10进制数的字符串。至少输出2位指数。	double型	
10	g	g转换（或者G转换）	从指定转换值和有效位数的精度值决定是以f转换格式还是以e转换（或者E转换）格式输出double型数据。如果被转换的数据指数小于－4或者大于指定有效位数的精度，就转换为e转换（或者E转换）格式。	double型	精度的指定表示转换数据的最大有效位数。
11	G			double型	
12	c	c转换	将int型数据作为unsigned char型数据，转换为对应该数据的字符。	int型	精度的指定无效。

	转换字符	转换种类	转换方式	转换对象的数据的型	有关精度的注意事项
13	s	s 转换	对于由 char 型的指针型数据指向的字符串，输出表示字符串结束的空字符之前的字符或者由精度指定的字符数（但是，不输出空字符。另外，在转换字符串中不包含空格、水平制表符、换行字符）。	指向 char 型的指针型	精度的指定表示输出的字符数。如果省略精度，就输出由数据指向的字符串的空字符之前的字符（但是，不输出空字符。另外，在转换字符串中不包含空格、水平制表符、换行字符）。
14	p	p 转换	将数据视为指针，转换为各编译程序定义的能打印的字符串。	指向 void 型的指针	精度的指定无效。
15	n	不进行数据转换。	将数据视为指向 int 型的指针型，将以前输出的数据字符数设定到此数据指向的存储区。	指向 int 型的指针型	
16	%	此转换不进行数据的转换。	输出%。	无	

(f) 对字段宽度或者精度的*指定

能将*指定为字段宽度或者精度指定的值。此时，将与此转换规格对应的参数的值用作字段宽度或者精度指定的值。此参数有负的字段宽度时，就解释为给正的字段宽度指定标志“－”。另外，有负的精度时，解释为省略精度。

带格式的文件输入

*int fscanf(FILE *fp, const char *control [, ptr] …)*

说　明　从流输入/输出文件输入数据，并且根据格式转换。

头文件　`<stdio.h>`

返回值

正常：　输入转换成功的数据个数

异常：　在转换输入数据前结束输入数据时：EOF

参　数

fp	文件指针
control	指向表示格式字符串的指针
ptr	指向保存输入数据的存储区的指针

例

```
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret,buffer[10];
    ret=fscanf(fp, control, buffer);
```

备　考

fscanf 函数从文件指针 fp 指向的流输入/输出文件输入数据，按 control 指向的格式字符串进行转换和编辑，其结果保存到 ptr 指向的存储区。用于输入数据的格式规格如下所示：

【格式概要】

表示格式的字符串由以下 3 种字符串构成：

- 空格字符

  如果指定空格(‘ ’)、水平制表符(‘\t’)或者换行字符(‘\n’)，就进行输入数据的跳读处理，直到下一个字符不为空格类字符。
- 通常字符

  如果指定既不是以上的空格字符也不是%开头的字符，就输入 1 个字符数据。在此输入的字符必须和表示格式的字符串中指定的字符一致。
- 转换规格

  转换规格以%开头的字符串指定输入数据的转换方法和转换数据的保存方法（保存到格式字符串后的参数所指向的区域）。转换规格的格式遵循以下规则：

  % [*] [字段宽度] [转换后的数据长度] 转换字符

  对于格式中的转换规格，如果没有指向保存输入数据的存储区的指针时，就不能保证其运行。另外，在格式结束后，如果还有指向保存输入数据的存储区的指针时，就忽视该指针。

【转换规格的说明】

- *指定

  抑制将输入数据保存到参数指向的存储区。
- 字段宽度

  用 10 进制数指定输入数据的最大字符数。
- 转换后的数据长度

  在 d、i、o、u、x、X、e、E、f 转换时（参照表 10.39），指定转换后的数据长度（short 型、long 型、long double 型）。在其它转换时，忽视本指定。长度指定的种类及其含义如表 10.38 所示。

表 10.38　转换后的数据长度的指定种类及其含义

	种类	含义
1	h	在 d，i，o，u，x 和 X 转换中，将转换后的数据指定为 short 型。
2	l	在 d，i，o，u，x 和 X 转换中，将转换后的数据指定为 long 型。 在 e，E 和 f 转换中，将转换后的数据指定为 double 型。
3	L	在 e，E 和 f 转换中，将转换后的数据指定为 long double 型。

- 转换字符
  按各转换字符指定的转换种类转换输入数据。但是，如果读到的字符为空格类字符或转换对象不允许的字符或者超过指定的字段宽度，就结束处理。

表 10.39　转换字符和转换内容

	转换字符	转换种类	转换方式	对应的参数的数据型
1	d	d 转换	将 10 进制数的字符串转换为整型数据。	整型
2	i	i 转换	将前头带符号的 10 进制数字符串或者最后带 u（U）或 l（L）的 10 进制数字符串转换为整型数据。另外，以 0x（或者 0X）开头的字符串解释为 16 进制数，将其字符串转换为 int 型数据；以 0 开头的字符串解释为 8 进制数，将其字符串转换为 int 型数据。	整型
3	o	o 转换	将 8 进制数的字符串转换为整型数据。	整型
4	u	u 转换	将无符号的 10 进制数的字符串转换为整型数据。	整型
5	x	x 转换	将 16 进制数的字符串转换为整型数据。	整型
6	X	X 转换	x 转换和 X 转换的含义相同。	
7	s	s 转换	将空格、水平制表符、换行字符之前的内容作为一个字符串进行转换。在字符串的最后附加空字符（设定转换数据的字符串需要能保存包含空字符的容量）。	字符型
8	c	c 转换	输入 1 个字符。此时，即使输入字符为空格类字符，也不进行跳读处理。如果只读取空格类字符以外的字符，就必须指定％1s。另外，如果指定字段宽度，就读取其指定宽度的字符。因此，此时保存转换数据的存储区需要被指定的容量。	char 型
9	e	e 转换	将表示浮点数的字符串转换为浮点型数据。e 转换和 E 转换或者 g 转换和 G 转换的含义相同。输入格式为能用 strtod 函数表现的浮点数。	浮点型
10	E	E 转换		
11	f	f 转换		
12	g	g 转换		
13	G	G 转换		
14	p	p 转换	将在 fprintf 函数中由 P 转换的格式的字符串转换为指针型数据。	指向 void 型的指针型
15	n	不进行数据转换。	不输入数据，设定以前输入的数据字符数。	整型
16	［	［转换	指定由中括号（［］）括起来的字符集。此字符集定义构成字符串的字符集。如果字符集的第一个字符不为＾，就将此字符集中不存在的字符出现之前的内容作为一个字符串输入。如果第一个字符为“＾”，就将此字符集中的字符（“＾”除外）出现之前的内容作为一个字符串输入。在输入字符串的最后自动附加空字符（设定转换数据的字符串需要能保存包含空字符的容量）。	字符型
17	％	不进行数据转换。	读％。	无

如果转换字符不为表 10.39 中的英文字符时，就不保证其运行。另外，如果为其它字符，其运行为实现定义。

带格式的输出

*int printf(const char *control [, arg] …)*

说　明	根据格式转换数据，并且输出到标准输出文件（stdout）。
头文件	<stdio.h>
返回值	正常：　转换并输出的字符数 异常：　负值
参　数	control　　　　指向表示格式的字符串的指针 arg　　　　根据格式输出的数据

例

```
#include  <stdio.h>
char  *s;
const char  *control="%s";
int  ret;
char buffer[]="Hellow World\n";
    ret=sprintf(fp, control, buffer);
```

备　考

printf 函数根据 control 指向的格式字符串，对参数 arg 进行转换和编辑，然后输出到标准输出文件（stdout）。
格式规格的详细内容请参照 fprintf()。

带格式的输入

*int scanf(const char *control [, ptr] …)*

说　明	从标准输入文件（stdin）输入数据，并且根据格式进行转换。
头文件	<stdio.h>
返回值	正常：　输入转换成功的数据个数 异常：　EOF
参　数	control　　　　指向表示格式的字符串的指针 ptr　　　　指向保存输入转换数据的存储区的指针

例

```
#include  <stdio.h>
const char  *control="%d";
int  ret,buffer[10];
    ret=scanf(control, buffer);
```

备　考

scanf 函数从标准输入文件（stdin） 输入数据，按 control 指向的格式字符串，对该输入数据进行转换和编辑，其结果保存到 ptr 指向的存储区。
scanf 函数将输入转换成功的数据个数作为返回值返回。如果在最先转换前结束标准输入文件，就返回 EOF。
格式规格的详细内容请参照 fscanf()。
对于%e 转换，double 型时用 l 指定，long double 型时用 L 指定。默认型为 float 型。

带格式的字符串输出

int sprintf(char *s, const char *control [, arg] …)

说　明　　根据格式转换数据，并且输出到指定的区域。

头文件　　`<stdio.h>`

返回值　　转换的字符数

参　数

s	指向输出数据的存储区的指针
control	指向表示格式的字符串的指针
arg	根据格式输出的数据

例

```
#include  <stdio.h>
char  *s;
const char  *control="%s";
int  ret;
char buffer[]="Hellow World\n";
    ret=sprintf(fp, control, buffer);
```

备　考　　`sprintf` 函数按 `control` 指向的格式字符串，对参数 `arg` 进行转换和编辑，然后输出到 `s` 指向的存储区。

在转换和输出的字符列的最后附加空字符。此空字符不包含在作为返回值输出的字符数中。

格式规格的详细内容请参照 `fprintf()`。

带格式的字符串输入

int sscanf(const char *s, const char *control [, ptr] …)

说　明　　从指定的存储区输入数据，并且根据格式转换。

头文件　　`<stdio.h>`

返回值

正常：　输入转换成功的数据个数

异常：　EOF

参　数

s	有输入数据的存储区
control	指向表示格式的字符串的指针
ptr	指向保存输入转换数据的存储区的指针

例

```
#include  <stdio.h>
const char  *s, *control="%d";
int  ret,buffer[10];
    ret=sscanf(s, control, buffer);
```

备　考　　`sscanf` 函数从 `s` 指向的存储区输入数据，按 `control` 指向的格式字符串，对该输入数据进行转换和编辑，其结果保存到 `ptr` 指向的存储区。

`sscanf` 函数返回输入转换成功的数据个数。另外，如果在最先转换前结束数据的输入，就返回 EOF。

格式规格的详细内容请参照 `fscanf()`。

可变个数参数的文件输出

*int vfprintf(FILE *fp, const char *control, va_list arg)*

说　明　　根据格式将可变个数的参数列表输出到指定的流输入/输出文件。

头文件　　`<stdio.h>`

返回值　　正常：　转换并输出的字符数
　　　　　异常：　负值

参　数	`fp`	文件指针
	`control`	指向表示格式的字符串的指针
	`arg`	参数列表

例

```
#include  <stdarg.h>
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
     ret=vfprintf(fp, control, ap);
   va_end(ap);
}
```

备　考　　`vfprintf` 函数根据 `control` 指向的格式字符串，按顺序对可变个数的参数列表进行转换和编辑，然后输出到 `fp` 指向的流输入/输出文件。

`vfprintf` 函数返回转换和输出的数据个数。但是，如果发生输出错误，就返回负值。

另外，`vfprintf` 函数不调用 `va_end` 宏。

格式规格的详细内容请参照 `fprintf()`。

必须通过 `va_start` 和 `va_arg` 宏对表示参数列表的 `arg` 进行初始化。

可变个数参数的输出

*int vprintf(const char *control, va_list arg)*

说　明	根据格式将可变个数的参数列表输出到标准输出文件（stdout）。
头文件	<stdio.h>
返回值	正常：　转换并输出的字符数 异常：　负值

参　数

control	指向表示格式的字符串的指针
arg	参数列表

例

```
#include  <stdarg.h>
#include  <stdio.h>
FILE  *fp;
const char  *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vprintf(control, ap);
   va_end(ap);
}
```

备　考

vprintf 函数根据 control 指向的格式字符串，按顺序对可变个数的参数列表进行转换和编辑，然后输出到标准输出文件。

vprintf 函数返回转换和输出的数据个数。但是，如果发生输出错误，就返回负值。

另外、vprintf 函数不调用 va_end 宏。

格式规格的详细内容请参照 fprintf()。

必须通过 va_start 和 va_arg 宏对表示参数列表的 arg 进行初始化。

可变个数参数的字符串输出

*int vsprintf(char *s, const char *control, va_list arg)*

说　明	根据格式将可变个数的参数列表输出到指定的存储区。
头文件	`<stdio.h>`
返回值	正常：　转换的字符数 异常：　负值

参　数

`s`	指向输出数据的存储区的指针
`control`	指向表示格式的字符串的指针
`arg`	参数列表

例

```
#include  <stdarg.h>
#include  <stdio.h>
#define NUM 128
char str[NUM];
int ret;

void prlist(int count,...){
    va_list      ap;
    int          i;
    char         *s=str;
    va_start(ap, count);
    for (i=0;i<count;i++){
         ret=vsprintf(s,"%d",ap);
         va_arg(ap,int);
         s+=ret;
    }
}
```

备　考

`vsprintf` 函数根据 `control` 指向的格式字符串，按顺序对可变个数的参数列表进行转换和编辑，然后输出到 `s` 指向的存储区。

在转换输出的字符串的最后附加空字符。此空字符不包含在作为返回值输出的字符数中。

格式规格详细内容请参照 `fprintf()`。

必须通过 `va_start` 和 `va_arg` 宏对表示参数列表的 `arg` 进行初始化。

从文件输入 1 个字符

*int fgetc(FILE *fp)*

说　明	从流输入/输出文件输入 1 个字符。
头文件	`<stdio.h>`
返回值	正常：　文件结束时：EOF 　　　　文件未结束时：输入的字符 异常：　EOF
参　数	fp　　　　　　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=fgetc(fp);
```

错误条件	如果发生读错误，就设定该文件的错误指示符。
备　考	fgetc 函数从文件指针 fp 指向的流输入/输出文件输入 1 个字符。 fgetc 函数通常返回输入的 1 个字符。但是，在文件结束或者发生错误时，返回 EOF。另外，在文件结束时，对该文件设定文件结束指示符。

从文件输入字符串

*char *fgets(char *s, int n, FILE *fp)*

说　明	从流输入/输出文件输入字符串。		
头文件	<stdio.h>		
返回值	正常：	文件结束时：NULL 文件未结束时：s	
	异常：	NULL	
参　数	s		指向输入字符串的存储区的指针
	n		输入字符串的存储区的字节数
	fp		文件指针

例

```
#include  <stdio.h>
char  *s, *ret;
int  n;
FILE  *fp;
    ret=fgets(s, n, fp);
```

备　考

fgets 函数从文件指针 fp 指向的流输入/输出文件将字符串输入到指针 s 指向的存储区。

fgets 函数输入 n-1 个字符、换行字符前的字符或者文件结束前的字符，在输入字符串的最后附加空字符。

fgets 函数通常返回指向输入字符串的存储区的指针 s。但是，在文件结束或者发生错误时，返回 NULL。

文件结束时，s 指向的存储区的内容不变。但是，在发生错误时，不能保证 s 指向的存储区的内容。

将 1 个字符输出到文件

*int fputc(int c, FILE *fp)*

说　明	将 1 个字符输出到流输入/输出文件。
头文件	`<stdio.h>`
返回值	正常：　输出的字符 异常：　EOF

参　数

c	要输出的字符
fp	文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  c, ret;
    ret=fputc(c, fp);
```

错误条件　　如果发生写错误，就对该文件设定错误指示符。

备　考

fputc 函数将字符 c 输出到文件指针 fp 指向的流输入/输出文件。
fputc 函数通常返回输出的字符 c。但是，发生错误时，返回 EOF。

将字符串输出到文件

*int fputs(const char *s, FILE *fp)*

说　明	将字符串输出到流输入/输出文件。
头文件	`<stdio.h>`
返回值	正常：　0 异常：　0 以外

参　数

s	指向要输出的字符串的指针
fp	文件指针

例

```
#include  <stdio.h>
const char  *s;
int  ret;
FILE  *fp;
    ret=fputs(s, fp);
```

备　考

fputs 函数将 s 指向的空字符前的字符串输出到文件指针 fp 指向的流输入/输出文件。此时，不输出表示字符串结束的空字符。

fputs 函数通常返回 0。但是，发生错误时，返回 0 以外的值。

从文件输入 1 个字符

int getc(FILE *fp)

说　明　从流输入/输出文件输入 1 个字符。

头文件　`<stdio.h>`

返回值

正常：　文件结束时：EOF
　　　　文件未结束时：输入的字符
异常：　EOF

参　数

fp	文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=getc(fp);
```

错误条件　如果发生读错误，就对该文件设定错误指示符。

备　考

getc 函数从文件指针 fp 指向的流输入/输出文件输入 1 个字符。

getc 函数通常返回输入的 1 个字符。但是，在文件结束或者发生错误时，返回 EOF。另外，文件结束时，对该文件设定文件结束指示符。

1 个字符的输入

int getchar()

说　明　从标准输入文件（stdin）输入 1 个字符。

头文件　`<stdio.h>`

返回值

正常：　文件结束时：EOF
　　　　文件未结束时：输入的字符
异常：　EOF

例

```
#include  <stdio.h>
int  ret;
    ret=getchar();
```

错误条件　如果发生读错误，就对该文件设定错误指示符。

备　考

getchar 函数从标准输入文件（stdin）输入 1 个字符。

getchar 函数通常返回输入的 1 个字符。但是，在文件结束或者发生错误时，返回 EOF。另外，文件结束时，对该文件设定文件结束指示符。

字符串的输入

*char *gets(char *s)*

说　明	从标准输入文件（stdin）输入字符串。	
头文件	<stdio.h>	
返回值	正常：　文件结束时：NULL 　　　　文件未结束时：s 异常：　NULL	
参　数	s	指向输入字符串的存储区的指针

例

```
#include  <stdio.h>
char  *ret, *s;
    ret=gets(s);
```

备　考

gets 函数从标准输入文件（stdin）将字符串输入到以 s 开头的存储区。
gets 函数在文件结束或者换行字符之前输入字符，代替换行字符附加空字符。
gets 函数通常返回指向输入字符串的存储区的指针 s。但是，在标准输入文件结束或者发生错误时，返回 NULL。
结束标准输入文件时，s 指向的存储区的内容不变。但是，发生错误时，不能保证 s 指向的存储区的内容。

将 1 个字符输出到文件

*int putc(int c, FILE *fp)*

说　明	将 1 个字符输出到流输入/输出文件。	
头文件	<stdio.h>	
返回值	正常：　输出的字符 异常：　EOF	
参　数	c	要输出的字符
	fp	文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  c, ret;
    ret=putc(c, fp);
```

错误条件　　如果发生写错误，就对该文件设定错误指示符。

备　考

putc 函数将字符 c 输出到文件指针 fp 指向的流输入/输出文件。
putc 函数通常返回输出的字符 c。但是，发生错误时，返回 EOF。

1 个字符的输出

int putchar(int c)

说　明	将 1 个字符输出到标准输出文件（stdout）。
头文件	<stdio.h>
返回值	正常：　输出的字符 异常：　EOF
参　数	c　　要输出的字符

例

```
#include  <stdio.h>
int  c, ret;
    ret=putchar(c);
```

错误条件　如果发生写错误，就对该文件设定错误指示符。

备　考　putchar 函数将字符 c 输出到标准输出文件（stdout）。putchar 宏通常返回输出的字符 c。但是，发生错误时，返回 EOF。

字符串的输出

*int puts(const char *s)*

说　明	将字符串输出到标准输出文件（stdout）。
头文件	<stdio.h>
返回值	正常：　0 异常：　0 以外
参　数	s　　指向输出字符串的指针

例

```
#include  <stdio.h>
const char  *s;
int  ret;
    ret=puts(s);
```

备　考　puts 函数将 s 指向的字符串输出到标准输出文件（stdout）。此时，不输出表示字符串结束的字符，而输出换行字符。

puts 函数通常返回 0。但是，发生错误时，返回 0 以外的值。

将 1 个字符返回到文件

*int ungetc(int c, FILE *fp)*

说　明	将 1 个字符返回到流输入/输出文件。
头文件	`<stdio.h>`
返回值	正常：　返回的字符 异常：　EOF
参　数	c　　要返回的字符 fp　　文件指针

例

```
#include  <stdio.h>
int  c, ret;
FILE  *fp;
    ret=ungetc(c, fp);
```

备　考

ungetc 函数将字符 c 返回到文件指针 fp 指向的流输入/输出文件。
在此，如果不调用 fflush、fseek、rewind 函数，被返回的字符就为下一个输入数据。
ungetc 函数通常返回已返回的字符 c。但是，发生错误时，返回 EOF。
对于 ungetc 函数，如果不执行 fflush、fseek、rewind 函数，就不能保证调用 2 次以上时的运行。另外，如果执行 ungetc 函数，文件的当前位置指示符就回移一个位置。但是，如果此位置指示符已处于文件的起始位置，就能不保证位置指示符。

读文件

*size_t fread(void *ptr, size_t size, size_t n, FILE *fp)*

说　明　　从流输入/输出文件将数据输入到指定的存储区。

头文件　　`<stdio.h>`

返回值　　`size` 或者 `n` 为 `0` 时：`0`
`size` 和 `n` 都不为 `0` 时：输入成功的成员个数

参　数

`ptr`	指向输入数据的存储区的指针
`size`	1 个成员的字节数
`n`	输入的成员个数
`fp`	文件指针

例

```
#include  <stdio.h>
void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp;
    ret=fread(ptr, size, n, fp);
```

备　考　　`fread` 函数将 `size` 指定的字节数作为 `1` 个成员，从文件指针 `fp` 指向的流输入/输出文件输入 `n` 个成员的数据到 `ptr` 指向的存储区。此时，对于文件的位置指示符向前移了输入数据的字节数。

`fread` 函数返回实际输入成功的成员个数，所以通常和 `n` 的值相同。但是，在文件结束或者发生错误时，返回在那时输入成功的成员个数，所以返回值小于 `n`，可使用 `ferror`、`feof` 函数来区分是文件结束还是发生错误。

`Size` 或者 `n` 为 `0` 时，返回值为 `0`，`ptr` 指向的存储区内容不变。另外，如果发生错误或者只能输入部分成员，就不保证该文件的位置指示符。

写文件

*size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp)*

说　明	从存储区将数据输出到流输入/输出文件。	
头文件	`<stdio.h>`	
返回值	输出成功的成员个数	
参　数	`ptr`	指向保存输出数据的存储区的指针
	`size`	1 个成员的字节数
	`n`	输出的成员个数
	`fp`	文件指针

例

```
#include  <stdio.h>
const void  *ptr;
size_t  size;
size_t  n, ret;
FILE  *fp;
    ret=fwrite(ptr, size, n, fp);
```

备　考

`fwrite` 函数将 `size` 指定的字节数作为 1 个成员，从 `ptr` 指向的存储区输出 `n` 个成员的数据到文件指针 `fp` 指向的流输入/输出文件。此时，对于文件的位置指示符向前移了输出数据的字节数。

`fwrite` 函数返回实际输出成功的成员个数，所以通常和 `n` 的值相同。但是，在发生错误时，返回那时输出成功的成员个数，所以返回值小于 `n`。

在发生错误时，不能保证该文件的位置指示符。

文件读写位置的移动

*int fseek(FILE *fp, long offset, int type)*

说　明　　移动流输入/输出文件的当前读写位置。

头文件　　`<stdio.h>`

返回值　　正常：　0
　　　　　异常：　0 以外

参　数

`fp`	文件指针
`offset`	从偏移种类指定位置开始的偏移
`type`	偏移种类

例

```
#include  <stdio.h>
FILE  *fp;
long  offset;
int  type, ret;
    ret=fseek(fp, offset, type);
```

备　考　　`fseek` 函数将文件指针 `fp` 指向的流输入/输出文件的当前读写位置从偏移种类 `type` 指定的位置移动到 `offset`（字节）指定的位置。

偏移种类如表 10.40 所示。
`fseek` 函数通常返回 `0`，但是出现异常时，返回 `0` 以外的值。

表 10.40　偏移种类

	偏移种类	含义
1	SEEK_SET	从文件的起始位置移动到 offset（字节）指定的位置。此时，offset 指定的值必须为 0 或正。
2	SEEK_CUR	从文件的当前位置移动到 offset（字节）指定的位置。此时，如果 offset 指定的值为正，就向文件的后方移动；如果为负，就向文件的起始处移动。
3	SEEK_END	从文件结束位置移动到 offset（字节）指定的位置。此时，offset 指定的值必须为 0 或负。

文本文件时，偏移种类必须为 SEEK_SET，并且 `offset` 必须为 `0` 或者为对该文件用 `ftell` 函数返回的值。另外，必须注意：根据 `fseek` 函数的调用，`ungetc` 函数的效果被取消。

文件读写位置的取得

*long ftell(FILE *fp)*

说　明	取得流输入/输出文件的当前读写位置。
头文件	`<stdio.h>`
返回值	当前位置指示符的位置（文本文件） 从文件起始位置到当前位置的字节数（二进制文件）
参　数	`fp`　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
long  ret;
    ret=ftell(fp);
```

备　考

`ftell` 函数取得文件指针 `fp` 指向的流输入/输出文件的当前读写位置。
二进制文件时，`ftell` 函数返回从文件起始位置到当前位置的字节数。但是，在文本文件时，作为位置指示符的位置，返回值为能由 `fseek` 函数使用的实现定义值。
在对文本文件 2 次使用 `ftell` 函数后，其返回值的差并非表示实际文件中的差异。

移动到文件起始位置

*void rewind(FILE *fp)*

说　明	将流输入/输出文件的当前读写位置移动到文件的起始位置。
头文件	`<stdio.h>`
参　数	`fp`　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
    rewind(fp);
```

备　考

`rewind` 函数将文件指针 `fp` 指向的流输入/输出文件的当前读写位置移动到文件的起始位置。
另外，`rewind` 函数清除该文件的结束指示符和错误指示符。
必须注意：根据 `rewind` 函数的调用，`ungetc` 函数的效果被取消。

错误状态的清除

void clearerr(FILE *fp)

说　明　清除流输入/输出文件的错误状态。

头文件　`<stdio.h>`

参　数　`fp`　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
    clearerr(fp);
```

备　考　`clearerr` 函数清除文件指针 `fp` 指向的流输入/输出文件的错误指示符和结束指示符。

文件结束的判断

int feof(FILE *fp)

说　明　判断流输入/输出文件是否结束。

头文件　`<stdio.h>`

返回值　文件结束时：0 以外
文件未结束时：0

参　数　`fp`　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=feof(fp);
```

备　考　`feof` 函数判断文件指针 `fp` 指向的流输入/输出文件是否结束。
`feof` 函数对指定的流输入/输出文件的文件结束指示符进行调查，如果结束指示符被设定，就返回 0 以外的值，表示文件结束；如果没有被设定，就返回 0，表示文件未结束。

文件错误状态的判断

*int ferror(FILE *fp)*

说　明	判断流输入/输出文件是否为错误状态。
头文件	`<stdio.h>`
返回值	文件为错误状态时：0 以外 文件不为错误状态时：0
参　数	fp　　文件指针

例

```
#include  <stdio.h>
FILE  *fp;
int  ret;
    ret=ferror(fp);
```

备　考　ferror 函数判断文件指针 fp 指向的流输入/输出文件是否为错误状态。

ferror 函数对指定的流输入/输出文件的错误指示符进行调查，如果错误指示符被设定，就返回 0 以外的值，表示文件为错误状态；如果没被设定，就返回 0，表示文件不为错误状态。

错误信息的输出

*void perror(const char *s)*

说　明	将对应错误号的错误信息输出到标准错误文件（stderr）。
头文件	`<stdio.h>`
参　数	s　　指向错误信息的指针

例

```
#include  <stdio.h>
const char  *s;
    perror(s);
```

备　考　perror 函数将对应 errno 的 s 指向的错误信息输出到标准错误文件（stderr）。

如果 s 不为 NULL 并且 s 指向的字符串不为空字符，输出的信息就以 s 指向的字符串、冒号、空格、实现定义的错误信息、换行字符的格式输出。

（13）＜no_float.h＞

提供不支持浮点转换（%f、%e、%E、%g 或者%G）的简易输入/输出函数。在输入/输出不需要浮点转换的文件时，能减少 ROM 容量。

【函数】

种类	定义名	说明
函数	fprintf	根据格式将数据输出到流输入/输出文件。
	fscanf	从流输入/输出文件输入数据后，根据格式转换。
	printf	根据格式转换数据后，输出到标准输出文件（stdout）。
	scanf	从标准输入文件（stdin）输入数据后，根据格式转换。
	sprintf	根据格式转换数据后，输出到指定的区域。
	sscanf	从指定的存储区输入数据后，根据格式转换。
	vfprintf	根据格式将可变个数的参数列表输出到指定的流输入/输出文件。
	vprintf	根据格式将可变个数的参数列表输出到标准输出文件。
	vsprintf	根据格式将可变个数的参数列表输出到指定的存储区。

使用本函数时，必须在包含<stdio.h>前包含<no_float.h>。

例：

```
#include <no_float.h>
#include <stdio.h>
void main(void)
{
    printf(“Hello\n”);
}
```

【注】 指定#include <no_float.h>时，如果用本函数指定浮点数，就不能保证执行时的运行。

（14）＜stdlib.h＞

定义进行 C 程序中的标准处理的函数。

以下宏是实现定义。

种类	定义名	说明
型（宏）	onexit_t	由 onexit 函数注册的函数返回型和 onexit 函数返回值的型。
	div_t	div 函数返回值的结构体的型。
	ldiv_t	ldiv 函数返回值的结构体的型。
常数（宏）	RAND_MAX	rand 函数中生成的伪随机数的整数最大值。
函数	atof	将表现数的字符串转换为 double 型的浮点数值。
	atoi	将表现 10 进制数的字符串转换为 int 型的整数值。
	atol	将表现 10 进制数的字符串转换 long 型的整数值。
	strtod	将表现数的字符串转换为 double 型的浮点数值。
	strtol	将表现数的字符串转换为 long 型的整数值。
	rand	生成从 0 到 RAND_MAX 之间的伪随机数的整数。
	srand	设定由 rand 函数生成的伪随机数串的初始值。
	calloc	分配存储区，用 0 初始化所有被分配的存储区。
	free	释放被指定的存储区。
	malloc	分配存储区。
	realloc	将存储区的容量更改为指定的容量。
	bsearch	进行 2 分检索。
	qsort	进行排序。
	abs	取得 int 型整数的绝对值。
	div	计算 int 型整数除法运算的商和余数。
	labs	取得 long 型整数的绝对值。
	ldiv	计算 long 型整数除法运算的商和余数。

将字符串转换为 double 型

*double atof(const char *nptr)*

说　明	将表现数的字符串转换为 `double` 型的浮点数值。	
头文件	`<stdlib.h>`	
返回值	被转换的 `double` 型的浮点数值	
参　数	`nptr`	要转换的表现数的字符串的指针

例

```
#include  <stdlib.h>
const char  *nptr;
double  ret;
    ret=atof(nptr);
```

备　考

对不符合浮点数格式的最初字符之前的字符串进行转换。
即使发生溢出等错误， `atof` 函数也不设定 `errno`。另外，如果发生错误，就不能保证结果值。
如果有可能发生转换错误，可使用 `strtod` 函数。

将字符串转换为 int 型

*int atoi(const char *nptr)*

说　明	将表现 10 进制数的字符串转换为 `int` 型的整数值。	
头文件	`<stdlib.h>`	
返回值	被转换的 `int` 型的整数值	
参　数	`nptr`	要转换的表现数的字符串的指针

例

```
#include  <stdlib.h>
const char  *nptr;
int  ret;
    ret=atoi(nptr);
```

备　考

对不符合 10 进制数格式的最初字符之前的字符串进行转换。
即使发生溢出等错误，`atoi` 函数也不设定 `errno`。另外，如果发生错误，就不能保证结果值。
如果有可能发生转换错误，可使用 `strtol` 函数。

将字符串转换为 long 型

*long atol(const char *nptr)*

说　明	将表现 10 进制数的字符串转换为 `long` 型的整数值。
头文件	`<stdlib.h>`
返回值	被转换的 `long` 型的整数值
参　数	`nptr`　　　要转换的表现数的字符串的指针

例

```
#include  <stdlib.h>
const char  *nptr;
long  ret;
    ret=atol(nptr);
```

备　考

对不符合 10 进制数格式的最初字符之前的字符串进行转换。
即使发生溢出等错误，`atol` 函数也不设定 `errno`。另外，如果发生错误，就不能保证结果值。
如果有可能在转换时发生错误，可使用 `strtol` 函数。

将字符串转换为 double 型

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

说　明	将表现数的字符串转换为 `double` 型的浮点数值。	
头文件	`<stdlib.h>`	
返回值	正常：	`nptr` 指向的字符串不以构成浮点数的字符开始时：0 `nptr` 指向的字符串以构成浮点数的字符开始时：被转换的 `double` 型的浮点数值
	异常：	转换后的值溢出时：具有和转换的字符串相同符号的 `HUGE_VAL` 转换后的值下溢时：0
参　数	`nptr`	指向要转换的表现数的字符串的指针
	`endptr`	指向存储区的指针，该存储区保存不构成浮点数的最初字符的指针

例

```
#include  <stdlib.h>
const char  *nptr;
char  **endptr;
double  ret;
    ret=strtod(nptr, endptr);
```

错误条件　转换后的值发生溢出/下溢时，给 `errno` 设定 `ERANGE`。

备　考　`strtod` 函数根据“**10.1.3（4）浮点运算的规格**”的规则，将从最初数字或者小数点开始到不构成浮点数的字符前的字符串转换为 `double` 型的浮点数。但是，既不出现指数也不出现小数点时，就假设小数点在字符串的最后数字的后面。将指向不构成浮点数的最初字符的指针设定到 `endptr` 指向的区域。如果字符串的开始字符为不构成浮点数的字符，就将 `nptr` 的值设定到 `endptr`，但是在 `endptr` 为 `NULL` 时，不进行此设定。

将字符串转换为 long 型

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

说　明	将表现数的字符串转换为 long 型的整数值。
头文件	<stdlib.h>
返回值	正常：　nptr 指向的字符串不以构成整数的字符开始时：0 　　　　nptr 指向的字符串以构成整数的字符开始时：被转换的 long 型的整数值 异常：　转换后的值溢出时：LONG_MAX 或者 LONG_MIN（取决于转换的字符串的符号）

参　数

nptr	指向要转换的表现数的字符串的指针
endptr	指向存储区的指针，该存储区保存不构成整数的最初字符的指针
base	转换基数（0 或者 2～36）

例

```
#include  <stdlib.h>
long  ret;
const char  *nptr;
char  **endptr;
int  base;
    ret=strtol(nptr, endptr, base);
```

错误条件　转换后的值发生溢出时，给 errno 设定 ERANGE。

备　考

strtol 函数将从最初数字到不构成整数的最初字符之前的字符串转换为 long 型的整数值。将指向不构成整数的最初字符的指针设定到 endptr 指向的存储区。如果字符串的开始字符为不构成整数的字符，就将 nptr 的值设定到 endptr，但是在 endptr 为 NULL 时，不进行此设定。

base 为 0 时，根据“10.1.1（4）整数”的规则进行转换。base 的值在 2 到 36 之间时，表示转换时的基数。要转换的字符串中的 a（或者 A）到 z（或者 Z）的字符对应 10 到 35 的值。如果在要转换的字符串中存在大于或者等于 base 值的字符，就结束转换处理。另外，转换时忽视符号后的 0，也忽视 base 为 16 时的 0x（或者 0X）。

伪随机数的生成

int rand()

说　明　　生成 0 到 RAND_MAX 之间的伪随机数的整数。

头文件　　`<stdlib.h>`

返回值　　伪随机数的整数值

例

```
#include  <stdlib.h>
int  ret;
    ret=rand();
```

伪随机数串的初始设定

void srand(unsigned int seed)

说　明　　设定 rand 函数生成的伪随机数串的初始值。

头文件　　`<stdlib.h>`

参　数		
	seed	伪随机数串生成的初始值

例

```
#include  <stdlib.h>
unsigned  int seed;
    srand(seed);
```

备　考　　srand 函数设定 rand 函数生成伪随机数串的初始值。因此，rand 函数正在生成伪随机数时，如果再次使用 srand 函数设定相同的初始值，就重复生成伪随机数串。

如果在 srand 函数前调用 rand 函数，就将伪随机数串生成的初始值设定为 1。

带初始化存储区的分配

*void *calloc(size_t nelem, size_t elsize)*

说　明	分配存储区，用 0 初始化全部被分配的存储区。
头文件	`<stdlib.h>`
返回值	正常：　被分配的存储区的起始地址 异常：　不能分配存储区时或者某个参数为 0 时：NULL
参　数	nelem　　元素的个数 elsize　　一个元素占用的字节数

例

```
#include  <stdlib.h>
size_t  nelem, elsize;
void  *ret;
    ret=calloc(nelem, elsize);
```

备　考　将以 elsize 字节为单位的存储区分配给 nelem 个存储区，并且用 0 初始化被分配存储区的全部位。

存储区的释放

*void free(void *ptr)*

说　明	释放被指定的存储区。
头文件	`<stdlib.h>`
参　数	ptr　　要释放的存储区地址

例

```
#include  <stdlib.h>
void  *ptr;
    free(ptr);
```

备　考　释放后的 ptr 指向的存储区能重新分配使用。如果 ptr 为 NULL，就不进行任何操作。
对于不是由 calloc、malloc、realloc 函数分配的存储区或者已由 free、realloc 函数释放的存储区，不保证其运行。另外，也不保证已释放的存储区的参照运行。

存储区的分配

*void *malloc(size_t size)*

说　明	分配存储区。
头文件	`<stdlib.h>`
返回值	正常：　被分配的存储区的起始地址 异常：　不能分配存储区时或者 size 为 0 时：NULL
参　数	size　　要分配的存储区的字节数

例

```
#include  <stdlib.h>
size_t  size;
void  *ret;
    ret=malloc(size);
```

备　考　只分配 size 表示的字节长度的存储区。

存储区分配容量的变更

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

说　明	将存储区容量更改为指定的容量。
头文件	`<stdlib.h>`
返回值	正常：　更改后的存储区的起始地址 异常：　不能分配存储区时或者 size 为 0 时：NULL
参　数	ptr　　要更改的存储区的起始地址 size　　更改后的存储区的字节数

例

```
#include  <stdlib.h>
size_t  size;
void  *ptr, *ret;
    ret=realloc(ptr, size);
```

备　考　将 ptr 指向的存储区容量更改为 size 表示的字节长度的存储区。如果新分配的存储区容量小于更改前的存储区容量，新分配的存储区的内容不变。

在 ptr 不是指向 calloc、malloc、realloc 函数分配的存储区的指针时或者 ptr 是指向已被 free、realloc 函数释放的存储区的指针时，不保证其运行。

二分检索

*void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))*

说　明　　进行二分检索。

头文件　　`<stdlib.h>`

返回值　　检索到一致的成员时：指向一致成员的指针
没有检索到一致的成员：NULL

参　数

key	指向要检索的数据的指针
base	指向检索对象表的指针
nmemb	检索对象的成员个数
size	检索对象的成员字节数
compar	指向比较函数的指针

例

```
#include  <stdlib.h>
const void  *key, *base;
size_t  nmemb, size;
int  (*compar)(const void *, const void *);
void  *ret;
    ret=bsearch(key, base, nmemb, size, compar);
```

备　考　　根据二分检索法，在 base 指向的表中检索和 key 指向的数据一致的成员。比较函数必须取得指向二个比较数据的指针 p1（第 1 参数）、p2（第 2 参数），并且根据以下的规格返回结果：

```
*p1  <  *p2 时    返回负值。
*p1  ==  *p2 时   返回 0。
*p1  >  *p2 时    返回正值。
```

检索对象的各成员需要按升序排列。

排序

*void qsort(const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))*

说　明　　进行排序。

头文件　　`<stdlib.h>`

参　数

base	指向排序对象表的指针
nmemb	排序对象的成员个数
size	排序对象的成员字节数
compar	指向比较函数的指针

例

```
#include  <stdlib.h>
const void  *base;
size_t  nmemb, size;
int  (*compar)(const void *, const void *);
    qsort(base, nmemb, size, compar);
```

备　考　　对 base 指向的表的数据进行排序。通过指向比较函数的指针指定数据的排列顺序。此函数必须取得指向二个比较数据的指针 p1（第 1 参数）、p2（第 2 参数），并且根据以下的规格返回结果：

*p1　<　*p2 时	返回负值。
*p1　==　*p2 时	返回 0。
*p1　>　*p2 时	返回正值。

绝对值

int abs(int i)

说　明　　取得绝对值。

头文件　　`<stdlib.h>`

返回值　　i 的绝对值

参　数

i	要取得绝对值的整数

例

```
#include  <stdlib.h>
int  i, ret;
    ret=abs(i);
```

备　考　　在 i 绝对值不能表现为 int 型整数值时，不能保证其运行。

商和余数

div_t div(int numer, int denom)

说　明　　计算 `int` 型整数除法运算的商和余数。

头文件　　`<stdlib.h>`

返回值　　`denom` 除 `numer` 的结果的商和余数。

参　数

numer	被除数
denom	除数

例

```
#include  <stdlib.h>
int  numer, denom;
div_t  ret;
    ret=div(numer, denom);
```

绝对值

long labs(long j)

说　明　　取得 `long` 型整数的绝对值。

头文件　　`<stdlib.h>`

返回值　　`j` 的绝对值

参　数

j	要取得绝对值的整数

例

```
#include  <stdlib.h>
long  j;
long  ret;
    ret=labs(j);
```

备　考　　在 `j` 绝对值的结果不能表现为 `long` 型的整数值时，不能保证其运行。

商和余数

ldiv_t ldiv(long numer, long denom)

说　明	计算 `long` 型整数除法运算的商和余数。	
头文件	`<stdlib.h>`	
返回值	`denom` 除 `numer` 的结果的商和余数。	
参　数	`numer`	被除数
	`denom`	除数

例

```
#include  <stdlib.h>
long  numer, denom;
ldiv_t  ret;
    ret=ldiv(numer, denom);
```

（15）＜string.h＞

定义字符数组的操作所需的各种函数。

种类	定义名	说明
函数	memcpy	将指定容量的复制源存储区的内容复制到复制目标存储区。
	strcpy	包含空字符，将复制源的字符串内容复制到复制目标存储区。
	strncpy	将指定字符数的复制源的字符串复制到复制目标存储区。
	strcat	将字符串连接到字符串的后面。
	strncat	将指定字符数的字符串连接到字符串的后面。
	memcmp	比较二个指定存储区的内容。
	strcmp	比较二个指定的字符串。
	strncmp	对二个指定的字符串进行指定字符数的比较。
	memchr	在指定的存储区中，检索指定字符最初出现的位置。
	strchr	在指定的字符串中，检索指定字符最初出现的位置。
	strcspn	从头开始调查指定的字符串，取得从头开始的且不属于另外指定的字符串中的字符的字符数。
	strpbrk	在指定的字符串中，检索另外指定的字符串中的字符最初出现的位置。
	strrchr	在指定的字符串中，检索指定字符最后出现的位置。
	strspn	从头开始调查指定的字符串，取得从头开始的且属于另外指定的字符串中的字符的连续字符数。
	strstr	在指定的字符串中，检索另外指定的字符串最初出现的位置。
	strtok	将指定的字符串分成若干个字句。
	memset	从指定存储区的起始位置开始设定指定字符数的指定字符。
	strerror	设定错误信息。
	strlen	计算字符串的长度。
	memmove	将指定容量的复制源存储区的内容复制到复制目标存储区。即使复制源和复制目标的存储区重叠，也能正确复制。

使用本标准 inlcude 文件定义的函数时，需要注意以下二个事项：

（1）必须注意：在复制字符串时，如果复制目标区域小于复制源区域，就不能保证运行。

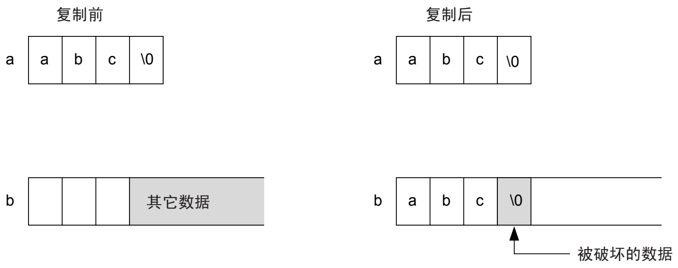
实现定义规格

	项目	C 编译程序的规格
1	strerror 函数返回的错误信息的内容	请参照“12.3 C 库函数的错误信息”。

例：

```
 char  a[ ]="abc";
 char  b[3];
       :
       :
     strcpy(b, a);
```

此时，数组 a 的长度（包含空字符）为 4 个字节。因此，如果通过 strcpy 函数进行复制，就将改写数组 b 区域以外的数据。



（2）必须注意：在复制字符串时，如果复制源区域和复制目标区域重叠，就不能保证正确的运行。

例：

```
 int a [ ]="a";
       .
       .
       .
 strcpy(&a[1], a);
       .
       .
       .
```

此时，在读取复制源的字符串的空字符前，字符‘a’被盖写于空字符，因此，复制源字符串数据后面的数据被改写。

存储区的复制

*void *memcpy(void *s1, const void *s2, size_t n)*

说　明	将指定容量的复制源存储区的内容复制到复制目标存储区。	
头文件	`<string.h>`	
返回值	s1 的值	
参　数	s1	指向复制目标存储区的指针
	s2	指向复制源存储区的指针
	n	复制的字符数

例

```
#include  <string.h>
void  *ret, *s1;
const void  *s2;
size_t  n;
    ret=memcpy(s1, s2, n);
```

字符串的复制

*char *strcpy(char *s1, const char *s2)*

说　明	包含空字符，将复制源字符串的内容复制到复制目标存储区。	
头文件	`<string.h>`	
返回值	s1 的值	
参　数	s1	指向复制目标存储区的指针
	s2	指向复制源字符串的指针

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strcpy(s1, s2);
```

字符串的复制

*char *strncpy(char *s1, const char *s2, size_t n)*

说　明	将指定字符数的复制源字符串复制到复制目标存储区。	
头文件	`<string.h>`	
返回值	s1 的值	
参　数	s1	指向复制目标存储区的指针
	s2	指向复制源字符串的指针
	n	复制的字符数

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
size_t  n;
    ret=strncpy(s1, s2, n);
```

备　考　从 s2 指定的字符串复制 n 个字符到 s1 指定的存储区。在 s2 指定的字符串长度短于 n 个字符时，附加空字符，使其长度增加到 n 个字符。

在 s2 指定的字符串长度长于 n 个字符时，被复制到 s1 中的字符串不以空字符结束。

字符串的连接

*char *strcat(char *s1, const char *s2)*

说　明	在字符串的后面连接字符串。	
头文件	`<string.h>`	
返回值	s1 的值	
参　数	s1	指向被连接字符串的指针
	s2	指向连接字符串的指针

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strcat(s1, s2);
```

备　考　将 s2 指定的字符串连接到 s1 指定的字符串的最后。此时，s1 字符串的结束空字符被 s2 的起始字符替换，并且 s2 字符串结束的空字符也被复制。

字符串的连接

*char *strncat(char *s1, const char *s2, size_t n)*

说　明	将指定字符数的字符串连接到字符串。
头文件	<string.h>
返回值	s1 的值

参　数

s1	指向被连接字符串的指针
s2	指向连接字符串的指针
n	连接字符数

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
size_t  n;
    ret=strncat(s1, s2, n);
```

备　考

从 s2 指定的字符串最多连接 n 个字符到 s1 指定的字符串的最后。s1 指向的字符串的结束空字符被 s2 的起始字符替换。

另外，在连接后的字符串的最后一定附加空字符。

存储区的比较

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

说　明	比较二个指定存储区的内容。
头文件	<string.h>

返回值

s1 指定的存储区	>	s2 指定的存储区时：正值
s1 指定的存储区	==	s2 指定的存储区时：0
s1 指定的存储区	<	s2 指定的存储区时：负值

参　数

s1	指向被比较存储区的指针
s2	指向比较存储区的指针
n	比较存储区的字符数

例

```
#include  <string.h>
const void  *s1, *s2;
size_t  n;
int  ret;
    ret=memcmp(s1, s2, n);
```

备　考

将 s1 指定的存储区和 s2 指定的存储区进行最初 n 个字符内容的比较。

此比较为实现定义。

字符串的比较

int strcmp(const char *s1, const char *s2)

说　明　　比较二个指定字符串的内容。

头文件　　`<string.h>`

返回值

s1 指定的字符串	>	s2 指定的字符串时：正值
s1 指定的字符串	==	s2 指定的字符串时：0
s1 指定的字符串	<	s2 指定的字符串时：负值

参　数

s1	指向被比较字符串的指针
s2	指向比较字符串的指针

例

```
#include  <string.h>
const char  *s1, *s2;
int  ret;
    ret=strcmp(s1, s2);
```

备　考　　比较 s1 指定的字符串和 s2 指定的字符串的内容，将其结果设定为返回值。
此比较为实现定义。

字符串的比较

int strncmp(const char *s1, const char *s2, size_t n)

说　明　　对二个指定字符串进行指定字符数的比较。

头文件　　`<string.h>`

返回值

s1 指定的字符串	>	s2 指定的字符串时：正值
s1 指定的字符串	==	s2 指定的字符串时：0
s1 指定的字符串	<	s2 指定的字符串时：负值

参　数

s1	指向被比较字符串的指针
s2	指向比较字符串的指针
n	比较字符数的最大值

例

```
#include  <string.h>
const char  *s1, *s2;
size_t  n;
int  ret;
    ret=strncmp(s1, s2, n);
```

备　考　　在最初的 n 个字符的范围内，比较 s1 指定的字符串和 s2 指定的字符串的内容。
此比较为实现定义。

存储区内的字符检索

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

说　明	在指定的存储区中，检索指定字符最初出现的位置。
头文件	`<string.h>`
返回值	找到检索字符时：指向找到的字符的指针 没有找到检索字符时：NULL

参　数

s	指向检索存储区的指针
c	检索字符
n	进行检索的字符数

例

```
#include  <string.h>
const void  *s;
int  c;
size_t  n;
void  *ret;
    ret=memchr(s, c, n);
```

备　考　作为返回值，返回指向 c 字符在 s 指定的存储区的前头 n 字符中最初出现的位置的指针。

最初的字符位置

*char *strchr(const char *s, int c)*

说　明	在指定的字符串中，检索指定字符最初出现的位置。
头文件	`<string.h>`
返回值	找到检索字符时：指向找到的字符的指针 没有找到检索字符时：NULL

参　数

s	指向检索字符串的指针
c	检索字符

例

```
#include  <string.h>
const char  *s;
int  c;
char  *ret;
    ret=strchr(s, c);
```

备　考　作为返回值，返回指向 c 字符在 s 指定的字符串中最初出现的位置的指针。
作为检索对象，也包含表示 s 指定的字符串结束的空字符。

指定字符集最初出现前的字符数

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

说　明　　从头开始调查指定的字符串，取得由另外指定的字符串中没有的字符构成的最初字符串的字符数。

头文件　　`<string.h>`

返回值　　在 s1 中，由 s2 指定字符以外的字符构成的最初字符串的字符数

参　数

s1	指向被调查字符串的指针
s2	指向用于调查 s1 的字符串的指针

例

```
#include  <string.h>
const char  *s1, *s2;
size_t  ret;
    ret=strcspn(s1, s2);
```

备　考　　从头开始调查 s1 指定的字符串，取得由 s2 指定字符以外的字符构成的最初字符串的字符数，将该字符串长度作为返回值返回。
表示 s2 字符串结束的空字符不作为 s2 字符串的一部分。

指定字符集最初出现的位置

*char *strpbrk(const char *s1, const char *s2)*

说　明　　在指定的字符串内，检索另外指定的字符串中的字符最初出现的位置。

头文件　　`<string.h>`

返回值　　找到检索结果时：指向找到的字符的指针
没有找到检索结果时：NULL

参　数

s1	指向检索字符串的指针
s2	指向表示 s1 内检索字符的字符串的指针

例

```
#include  <string.h>
const char  *s1, *s2;
char  *ret;
    ret=strpbrk(s1, s2);
```

备　考　　在 s1 指定的字符串中，检索 s2 指定的字符串中的一个字符最初出现的位置，将该指针作为返回值返回。

最后的字符位置

*char *strrchr(const char *s, int c)*

说　明　　在指定的字符串中，检索指定字符最后出现的位置。

头文件　　`<string.h>`

返回值　　找到检索结果时：指向找到的字符的指针
没有找到检索结果时：NULL

参　数

s	指向被检索字符串的指针
c	检索字符

例

```
#include  <string.h>
const char  *s;
int  c;
char  *ret;
    ret=strrchr(s, c);
```

备　考　　作为返回值，返回指向 c 指定的字符在 s 指定的字符串中最后出现的位置的指针。
作为检索对象，也包含表示 s 指定的字符串结束的空字符。

指定字符集连续部分的长度

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

说　明　　从头开始调查指定的字符串，取得由另外指定的字符串中的字符构成的最初字符串的字符数。

头文件　　`<string.h>`

返回值　　在 s1 中，由 s2 指定的字符构成的最初字符串的字符数

参　数

s1	指向被调查字符串的指针
s2	指向用于调查 s1 的字符串的指针

例

```
#include  <string.h>
const char  *s1, *s2;
size_t  ret;
    ret=strspn(s1, s2);
```

备　考　　从头开始调查 s1 指定的字符串，取得由 s2 指定的字符构成的最初字符串的字符数，将该字符串长度作为返回值返回。

最初的字符串位置

*char *strstr(const char *s1, const char *s2)*

说　明	在指定的字符串中，检索另外指定的字符串最初出现的位置。
头文件	`<string.h>`
返回值	找到检索结果时：指向找到的字符的指针 没有找到检索结果时：NULL

参　数

s1	指向被检索字符串的指针
s2	指向检索字符串的指针

例

```
#include  <string.h>
const char  *s1, *s2;
char  *ret;
    ret=strstr(s1, s2);
```

备　考　　在 s1 指定的字符串中，检索 s2 指定的字符串最初出现的位置，将该指针作为返回值返回。

字句的分开

*char *strtok(char *s1, const char *s2)*

说　明　　将指定的字符串分成若干个字句。

头文件　　`<string.h>`

返回值　　分成字句时：指向已分成的字句起始位置的指针
没有分成字句时：NULL

参　数

s1	指向被分成若干个字句的字符串的指针
s2	指向用于分开字符串的指针

例

```
#include  <string.h>
char  *s1, *ret;
const char  *s2;
    ret=strtok(s1, s2);
```

备　考　　为了分开字符串而连续调用 strtok 函数。

(a) 最初调用时

用 s2 指定的字符串中的字符将 s1 指定的字符串从起始位置开始分成字句。如果分成字句，就将指向该字句起始位置的指针作为返回值返回；否则，返回 NULL。

(b) 第 2 次以后调用时

用 s2 指定的字符串中的字符将从以前被分成字句的下一个字符开始的字符串分成字句。如果分成字句，就将指向该字句起始位置的指针作为返回值返回；否则，返回 NULL。

第 2 次以后调用时，给第 1 参数指定 NULL。另外，每次调用时的 s2 指定的字符串可不同。在被分成字句的最后附加空字符。

strtok 函数的使用例如下所示：

例：

```
1    #include  <string.h>
2    static  char  s1[ ]="a@b, @c/@d";
3    char  *ret;
4
5    ret = strtok(s1, "@");
6    ret = strtok(NULL, ",@");
7    ret = strtok(NULL, "/@");
8    ret = strtok(NULL, "@");
```

【说明】

此例为通过 strtok 函数将字符串““a@b,@c/@d””分成 a, b, c, d 字句的程序。

在第 2 行，给字符串 s1 设定初始值字符串“a@b, @c/@d”。

在第 5 行，为了将“@”作为分割字符分开字句，调用 strtok 函数。其结果，得到了被返回的指向字符‘a’的指针，并对字符‘a’的下一个最初的分割字符“@”埋入空字符。结果字符串“a”被分开。

然后，为了将同一个字符串依次分开字句，给第 1 参数指定 NULL 并调用 strtok 函数。结果，字符串依次被分成“b”、“c”、“d”。

字符的重复

*void *memset(void *s, int c, size_t n)*

说　明　　从指定的存储区的起始位置开始设定指定字符数的指定字符。

头文件　　`<string.h>`

返回值　　s 的值

参　数

s	指向被设定字符的存储区的指针
c	设定字符
n	设定字符数

例

```
#include  <string.h>
void  *s, *ret;
int  c;
size_t  n;
    ret=memset(s, c, n);
```

备　考　　给 s 指定的存储区设定 n 个字符的字符 c。

错误信息字符串

*char *strerror(int s)*

说　明　　指定错误号，返回对应的错误信息。

头文件　　`<string.h>`

返回值　　指向与错误号对应的错误信息（字符串）的指针

参　数

s	错误号

例

```
#include  <string.h>
char  *ret;
int  s;
    ret=strerror(s);
```

备　考　　作为返回值，返回指向与错误号 s 对应的错误信息的指针。
错误信息的内容为实现定义。
如果修正作为返回值返回的错误信息，就不能保证运行。

字符串的长度

*size_t strlen(const char *s)*

说　明	计算字符串的长度。	
头文件	`<string.h>`	
返回值	字符串的字符数	
参　数	s	指向长度计算对象的字符串的指针

例

```
#include  <string.h>
const char  *s;
size_t  ret;
    ret=strlen(s);
```

备　考　表示 s 指定的字符串结束的空字符不作为字符串长度计算的范围。

存储区的移动

*void *memmove(void *s1, const void *s2, size_t n)*

说　明　将指定容量的复制源存储区的内容复制到复制目标存储区。
另外，即使复制源和复制目标的存储区有重叠部分，因为在重写复制源的重叠部分前进行复制，所以也能正确地复制。

头文件	`<string.h>`	
返回值	s1 的值	
参　数	s1	指向复制目标的存储区的指针
	s2	指向复制源的存储区的指针
	n	复制的字符数

例

```
#include  <string.h>
void  *ret, *s1;
const void  *s2;
size_t  n;
    ret=memmove(s1, s2, n);
```

10.3.2　　嵌入式 C++类库

（1）库的概要

以下说明在 C++程序中能标准利用的嵌入式 C++类库的规格。在此说明有关类库的种类和对应的标准 include 文件。以下按库的结构，说明有关各类库的规格。

- 库的种类

类库的种类和对应的标准 include 文件如表 10.41 所示。

表 10.41　类库的种类和标准 inlcude 文件的对应

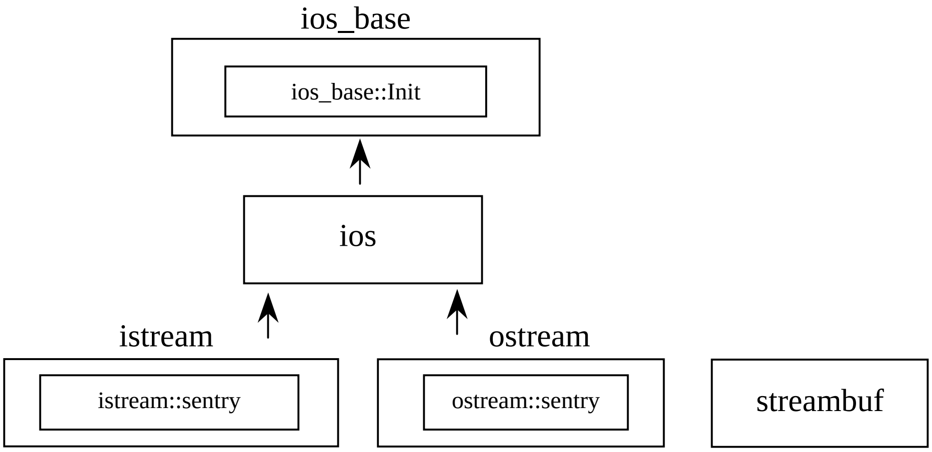
	库的种类	内容	标准 include 文件
1	流输入/输出类库	进行输入/输出操作的库。	<ios>, <streambuf>, <istream>,<ostream>, <iostream>,<iomanip>
2	存储器操作库	进行存储器确保/释放的库。	<new>
3	复数计算类库	进行复数数据运算的库。	<complex>
4	字符串操作类库	进行字符串操作的库。	<string>

（2）流输入/输出类库

与流输入/输出类库对应的头文件如下：

- <ios>
  定义进行输入/输出格式设定、输入/输出状态管理的数据成员和函数成员。除了 ios 类，还定义 Init 类和 ios_base 类。
- <streambuf>
  定义有关流缓冲区的函数。
- <istream>
  定义从输入流的输入函数。
- <ostream>
  定义对输出流的输出函数。
- <iostream>
  定义输入/输出函数。
- <iomanip>
  定义具有参数的操纵符。

这些类的派生关系如下所示。箭头表示从派生类参照基类。另外，streambuf 类没有派生关系。



流输入/输出类库中共用的型名如下所示：

种类	定义名	说明
型	streamoff	用 long 型定义的型。
	streamsize	用 size_t 型定义的型。
	int_type	用 int 型定义的型。
	pos_type	用 long 型定义的型。
	off_type	用 long 型定义的型。

（a）ios_base::Init 类

种类	定义名	说明
变量	init_cnt	对流输入/输出目标数进行计数的静态数据成员。 需要通过低级接口初始化成 0。
函数	init()	构造函数。
	~init()	析构函数。

ios_base::Init::Init()

类 Init 的构造函数。

使 init_cnt 递增。

ios_base::Init::~Init()

类 Init 的析构函数。

使 init_cnt 递减。

（b）ios_base 类

种类	定义名	说明
型	fmtflags	表示格式控制信息的型。
	iostate	表示流缓冲区的输入/输出状态的型。
	openmode	表示文件打开模式的型。
	seekdir	表示流缓冲区的查找状态的型。
变量	fmtfl	格式标志。
	wide	字段宽度。
	prec	输出时的精度(小数的位数)。
	fillch	填充字符。
函数	void_ec2p_init_base()	初始化。
	void_ec2p_copy_base(ios_base&ios_base_dt)	复制 ios_base_dt。
	ios_base()	构造函数。
	~ios_base()	析构函数。
	fmtflags flags() const	参照格式标志（fmtfl）。
	fmtflags flags(fmtflags fmtflg)	将 fmtflg&格式标志（fmtfl）设定为格式标志（fmtfl）。
	fmtflags setf(fmtflags fmtflg)	将 fmtflg 设定为格式标志（fmtfl）。
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	将 mask&fmtflg 设定为格式标志（fmtfl）。
	void unsetf(fmtflags mask)	将~mask&格式标志（fmtfl）设定为格式标志（fmtfl）。
	char fill() const	参照填充字符（fillch）。
	char fill(char ch)	将 ch 设定为填充字符（fillch）。
	int precision() const	参照精度（prec）。
	streamsize precision(streamsize preci)	将 preci 设定为精度（prec）。
	streamsize width() const	参照字段宽度（wide）。
	streamsize width(streamsize wd)	将 wd 设定为字段宽度（wide）。

ios_base::fmtflags

定义有关输入/输出的格式控制信息。

fmtflags 的各位屏蔽的定义如下所示：

```
const ios_base::fmtflags ios_base::boolalpha   = 0x0000;
const ios_base::fmtflags ios_base::skipws      = 0x0001;
const ios_base::fmtflags ios_base::unitbuf     = 0x0002;
const ios_base::fmtflags ios_base::uppercase   = 0x0004;
const ios_base::fmtflags ios_base::showbase    = 0x0008;
const ios_base::fmtflags ios_base::showpoint   = 0x0010;
const ios_base::fmtflags ios_base::showpos     = 0x0020;
const ios_base::fmtflags ios_base::left        = 0x0040;
const ios_base::fmtflags ios_base::right       = 0x0080;
const ios_base::fmtflags ios_base::internal    = 0x0100;
const ios_base::fmtflags ios_base::adjustfield = 0x01c0;
const ios_base::fmtflags ios_base::dec         = 0x0200;
const ios_base::fmtflags ios_base::oct         = 0x0400;
const ios_base::fmtflags ios_base::hex         = 0x0800;
const ios_base::fmtflags ios_base::basefield   = 0x0e00;
const ios_base::fmtflags ios_base::scientific  = 0x1000;
const ios_base::fmtflags ios_base::fixed       = 0x2000;
const ios_base::fmtflags ios_base::floatfield  = 0x3000;
const ios_base::fmtflags ios_base::_fmtmask    = 0x3fff;
```

ios_base::iostate

定义流缓冲区的输入/输出状态。

iostate 的各位屏蔽的定义如下所示：

```
const ios_base::iostate ios_base::goodbit      = 0x0;
const ios_base::iostate ios_base::eofbit       = 0x1;
const ios_base::iostate ios_base::failbit      = 0x2;
const ios_base::iostate ios_base::badbit       = 0x4;
const ios_base::iostate ios_base::_statemask   = 0x7;
```

ios_base::openmode

定义文件的打开模式。

openmode 的各位屏蔽的定义如下所示：

```
const ios_base::openmode ios_base::in          = 0x01;  打开用于输入的文件。
const ios_base::openmode ios_base::out         = 0x02;  打开用于输出的文件。
const ios_base::openmode ios_base::ate         = 0x04;  打开后只查找一次 eof。
const ios_base::openmode ios_base::app         = 0x08;  每次写时都查找 eof。
const ios_base::openmode ios_base::trunc       = 0x10;  用重写模式打开文件。
const ios_base::openmode ios_base::binary      = 0x20;  用二进制模式打开文件。
```

ios_base::seekdir

定义流缓冲区的查找状态。

决定用于继续输入或输出的流内的位置。

seekdir 的各位屏蔽的定义如下所示：

const ios_base::seekdir ios_base::beg = 0x0;

const ios_base::seekdir ios_base::cur = 0x1;

const ios_base::seekdir ios_base::end = 0x2;

void ios_base::_ec2p_init_base()

用以下的值进行初始设定：

fmtfl = skipws | dec;

wide = 0;

prec = 6;

fillch = ‘ ‘;

void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)

复制 ios_base_dt。

ios_base::ios_base()

类 ios_base 的构造函数。

调用 Init::Init()。

ios_base::~ios_base()

类 ios_base 的析构函数。

ios_base::fmtflags ios_base::flags() const

参照格式标志（fmtfl）。

返回值为格式标志（fmtfl）。

ios_base::fmtflags ios_base::flags(fmtflags fmtflg)

将 fmtflg&格式标志（fmtfl）设定为格式标志（fmtfl）。

返回值为设定前的格式标志（fmtfl）。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg)

将 fmtflg 设定为格式标志（fmtfl）。

返回值为设定前的格式标志（fmtfl）。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg, fmtflags mask)

将 mask&fmtflg 的值设定为格式标志（fmtfl）。

返回值为设定前的格式标志（fmtfl）。

void ios_base::unsetf(fmtflags mask)

将~mask&格式标志（fmtfl） 设定为格式标志（fmtfl）。

char ios_base::fill() const

参照填充字符（fillch）。

返回值为填充字符（fillch）。

char ios_base::fill(char ch)

将 ch 设定为填充字符。

返回值为设定前的填充字符（fillch）。

int ios_base::precision() const

参照精度（prec）。

返回值为精度（prec）。

streamsize ios_base::precision(streamsize preci)

将 preci 设定为精度（prec）。

返回值为设定前的精度（prec）。

streamsize ios_base::width() const

参照字段宽度（wide）。

返回值为字段宽度（wide）。

streamsize ios_base::width(streamsize wd)

将 wd 设定为字段宽度（wide）。

返回值为设定前的字段宽度（wide）。

（c）ios 类

种类	定义名	说明
变量	sb	指向 streambuf 目标的指针。
	tiestr	指向 ostream 目标的指针。
	state	指向 streambuf 的状态标志。
函数	ios() ios(streambuf* sbptr)	构造函数。
	void init(streambuf* sbptr)	进行初始设定。
	virtual ~ios()	析构函数。
	operator void*() const	判断有无错误（!state&(badbit | failbit)）。
	bool operator!() const	判断有无错误（state&(badbit | failbit)）。
	iostate rdstate() const	参照状态标志（state）。
	void clear(iostate st = goodbit)	清除指定状态（st）以外的状态标志（state）。
	void setstate(iostate st)	将 st 设定为状态标志(state)。
	bool good() const	判断有无错误（state==goodbit）。
	bool eof() const	判断是否为输入流的最后（state&eofbit）。
	bool bad() const	判断有无错误（state&badbit）。
	bool fail() const	判断输入文本是否与要求的模式不一致（state&(badbit | failbit)）。
	ostream* tie() const	参照指向 ostream 目标的指针（tiestr）。
	ostream* tie(ostream* tstrptr)	将 tstrptr 设定为指向 ostream 目标的指针（tiestr）。
	streambuf* rdbuf() const	参照指向 streambuf 目标的指针（sb）。
	streambuf* rdbuf(streambuf* sbptr)	将 sbptr 设定为指向 streambuf 目标的指针（sb）。
	ios& copyfmt(const ios& rhs)	复制 rhs 的状态标志（state）。

ios::ios()

类 ios 的构造函数。

调用 init(0)，将初始值设定为其成员目标。

ios::ios(streambuf* sbptr)

类 ios 的构造函数。

调用 init(sbptr)，将初始值设定为其成员目标。

void ios::init(streambuf* sbptr)

将 sbptr 设定为 sb。

将 state、tiestr 置 0。

virtual ios::~ios()

类 ios 的析构函数。

ios::operator void*() const
判断有无错误（!state&(badbit | failbit)）。
返回值如下：
有错误时：false
无错误时：true

bool ios::operator!() const
判断有无错误（state&(badbit | failbit)）。
返回值如下：
有错误时：true
无错误时：false

iostate ios::rdstate() const
参照状态标志（state）。
返回值为状态标志（state）。

void ios::clear(iostate st = goodbit)
清除除指定状态（st）的状态标志（state）。
指向 streambuf 目标的指针（sb）为 0 时，将 badbit 设定为状态标志（state）。

void ios::setstate(iostate st)
将 st 设定为状态标志（state）。

bool ios::good() const
判断有无错误（state==goodbit）。
返回值如下：
有错误时：false
无错误时：true

bool ios::eof() const
判断是否为输入流的最后（state&eofbit）。
返回值如下：
为输入流的最后时　：true
不为输入流的最后时：false

bool ios::bad() const
判断有无错误（state&badbit）。
返回值如下：
有错误时：true
无错误时：false

bool ios::fail() const
判断输入文本是否与要求的模式不一致（state&(badbit | failbit)）。
返回值如下：
不一致时：true
一致时　：false

ostream* ios::tie() const

参照指向 ostream 目标的指针（tiestr）。
返回值为指向 ostream 目标的指针（tiestr）。

ostream* ios::tie(ostream* tstrptr)

将 tstrptr 设定为指向 ostream 目标的指针（tiestr）。
返回值为设定前的指向 ostream 目标的指针（tiestr）。

streambuf* ios::rdbuf() const

参照指向 streambuf 目标的指针（sb）。
返回值为指向 streambuf 目标的指针（sb）。

streambuf* ios::rdbuf(streambuf* sbptr)

将 sbptr 设定为指向 streambuf 目标的指针（sb）。
返回值为设定前的指向 streambuf 目标的指针（sb）。

ios& ios::copyfmt(const ios& rhs)

复制 rhs 的状态标志（state）。
返回值为*this。

（d）ios 类操纵符

种类	定义名	说明
函数	ios_base& boolalpha(ios_base& str)	设定为 bool 型的格式。
	ios_base& noboolalpha(ios_base& str)	清除 bool 型的格式。
	ios_base& showbase(ios_base& str)	设定为基数显示前缀模式。
	ios_base& showpoint(ios_base& str)	设定为小数点生成模式。
	ios_base& noshowpoint(ios_base& str)	清除小数点生成模式。
	ios_base& showpos(ios_base& str)	设定为+记号生成模式。
	ios_base& noshowpos(ios_base& str)	清除+记号生成模式。
	ios_base& skipws(ios_base& str)	设定为空格跳读模式。
	ios_base& noskipws(ios_base& str)	清除空格跳读模式。
	ios_base& uppercase(ios_base& str)	设定为大写字母转换模式。
	ios_base& nouppercase(ios_base& str)	清除大写字母转换模式。
	ios_base& internal(ios_base& str)	设定为内部补充模式。
	ios_base& left(ios_base& str)	设定为左侧补充模式。
	ios_base& right(ios_base& str)	设定为右侧补充模式。
	ios_base& dec(ios_base& str)	设定为 10 进制模式。
	ios_base& hex(ios_base& str)	设定为 16 进制模式。
	ios_base& oct(ios_base& str)	设定为 8 进制模式。
	ios_base& fixed(ios_base& str)	设定为固定小数点模式。
	ios_base& scientific(ios_base& str)	设定为科学记数法模式。

ios_base& boolalpha(ios_base& str)

设定为 bool 型的格式。
返回值为 str。

ios_base& noboolalpha(ios_base& str)

清除 bool 型的格式。
返回值为 str。

ios_base& showbase(ios_base& str)

设定为数据开头的基数显示模式。
16 进制数时，在行的开头附加 0x。10 进制数时，直接输出。
8 进制数时，在行的开头附加 0。
返回值为 str。

ios_base& noshowbase(ios_base& str)

清除数据开头的基数显示模式。
返回值为 str。

ios_base& showpoint(ios_base& str)

设定为输出小数点的模式。
没有指定精度时，用 6 位小数显示。
返回值为 str。

ios_base& noshowpoint(ios_base& str)

清除输出小数点的模式。

返回值为 str。

ios_base& showpos(ios_base& str)

设定为+记号生成输出模式（给正数附加+号）。

返回值为 str。

ios_base& noshowpos(ios_base& str)

清除+记号生成输出模式。

返回值为 str。

ios_base& skipws(ios_base& str)

设定为空格跳读输入模式（跳读连续空格）。

返回值为 str。

ios_base& noskipws(ios_base& str)

清除空格跳读输入模式。

返回值为 str。

ios_base& uppercase(ios_base& str)

设定为大写字符转换输出模式。

16 进制的基数表现为大写字母 0X，数值本身也为大写字母。

浮点的指数表现也为大写字母 E。

返回值为 str。

ios_base& nouppercase(ios_base& str)

清除大写字母转换输出模式。

返回值为 str。

ios_base& internal(ios_base& str)

在字段宽度（wide）范围内输出时，按如下顺序输出：

符号、基数
填充字符（fill）
数值

返回值为 str。

ios_base& left(ios_base& str)

在字段宽度（wide）范围内输出时向左对齐。

返回值为 str。

ios_base& right(ios_base& str)

字段宽度（wide）范围内输出时向右对齐。

返回值为 str。

ios_base& dec(ios_base& str)

将转换基数设定为 10 进制模式。

返回值为 str。

ios_base& hex(ios_base& str)

将转换基数设定为 16 进制模式。

返回值为 str。

ios_base& oct(ios_base& str)

将转换基数设定为 8 进制模式。

返回值为 str。

ios_base& fixed(ios_base& str)

设定为固定小数点输出模式。

返回值为 str。

ios_base& scientific(ios_base& str)

设定为科学记数法输出模式（指数记数法）。

返回值为 str。

（e）streambuf 类

种类	定义名	说明
常数	eof	表示文件结束。
变量	_B_cnt_ptr	指向缓冲区有效数据长度的指针。
	B_beg_ptr	指向缓冲区基址指针的指针。
	_B_len_ptr	指向缓冲区的长度的指针。
	B_next_ptr	指向缓冲区的下一个读位置的指针。
	B_end_ptr	指向缓冲区结束位置的指针。
	B_beg_pptr	指向控制缓冲区的起始位置的指针。
	B_next_pptr	指向缓冲区的下一个读位置的指针。
	C_flg_ptr	指向文件输入/输出控制标志的指针。
函数	char* _ec2p_getflag() const	参照文件输入/输出控制标志的指针。
	char*& _ec2p_gnptr()	参照指向缓冲区的下一个读位置的指针。
	char*& _ec2p_pnptr()	参照指向缓冲区的下一个写位置的指针。
	void _ec2p_bcntplus()	缓冲区的有效数据长度递增。
	void _ec2p_bcntminus()	缓冲区的有效数据长度递减。
	void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)	设定 streambuf 的指针。
	streambuf()	构造函数。
	virtual ~streambuf()	析构函数。
	streambuf* pubsetbuf(char* s, streamsize n)	确保流输入/输出缓冲区。此函数调用 setbuf(s,n)[*1]。
	pos_type pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	用 way 指定的方法移动输入/输出流的读写位置。此函数调用 seekoff(off,way,which)[*1]。
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	求从流的起始位置到当前位置的偏移量。此函数调用 seekpos(sp,which)[*1]。
	int pubsync()	转储清除输出流。此函数调用 sync()[*1]。
	streamsize in_avail()	求从输入流的最后位置到当前位置的偏移量。
	int_type snextc()	读下一个字符。
	int_type sbumpc()	读一个字符，并将指针设定到下一个。
	int_type sgetc()	读一个字符。

种类	定义名	说明
函数	int sgetn(char* s, streamsize n)	将 n 个字符设定到 s 指向的存储区。
	int_type sputbackc(char c)	回放读位置。
	int sungetc()	回放读位置。
	int sputc(char c)	插入字符 c。
	int_type sputn(const char* s, streamsize n)	插入 s 指向的 n 个字符。
	char* eback() const	求输入流的头指针。
	char* gptr() const	求输入流的 next 指针。
	char* egptr() const	求输入流的尾指针。
	void gbump(int n)	将输入流的 next 指针移动 n。
	void setg(char* gbeg, char* gnext, char* gend)	赋值输入流的各指针。
	char* pbase() const	求输出流的头指针。
	char* pptr() const	求输出流的 next 指针。
	char* epptr() const	求输出流的尾指针。
	void pbump(int n)	将输出流的 next 指针移动 n。
	void setp(char* pbeg, char* pend)	设定输出流的各指针。
	virtual streambuf* setbuf(char* s, streamsize n) *1	按派生的各类，执行个别定义的运算。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	更改流位置。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	更改流位置。
	virtual int sync() *1	转储清除输出流。
	virtual int showmanyc() *1	求输入流的有效字符数。
	virtual streamsize xsgetn(char* s, streamsize n)	将 n 个字符设定到 s 指向的存储区。
	virtual int_type underflow() *1	不移动流位置，读一个字符。
	virtual int_type uflow() *1	读 next 指针的一个字符。
	virtual int_type pbackfail(int_type c = eof) *1	回放 c 表示的字符。
	virtual streamsize xsputn(const char* s, streamsize n)	插入 s 指向的 n 个字符。
	virtual int_type overflow(int_type c = eof) *1	将 c 插入到输出流。

【注】 *1　此类不定义处理。

streambuf::streambuf()

构造函数。

用以下的值进行初始化。

_B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr =_B_len_ptr = 0

B_beg_pptr = &B_beg_ptr

B_next_pptr = &B_next_ptr

virtual streambuf::~streambuf()

析构函数。

streambuf* streambuf::pubsetbuf(char* s, streamsize n)

确保流输入/输出缓冲区。

此函数调用 setbuf(s,n)。

返回值为 setbuf(s,n)。

pos_type streambuf::pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))

用 way 指定的方法移动输入/输出流的读写位置。

此函数调用 seekoff(off,way,which)。

返回值为重新设定的流位置。

pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))

求从流的起始位置到当前位置的偏移量。

只从当前的流指针移动 sp。

此函数调用 seekpos(sp,which)。

返回值为从起始位置的偏移量。

int streambuf::pubsync()

转储清除输出流。

此函数调用 sync()。

返回值为 0。

streamsize streambuf::in_avail()

求从输入流的最后位置到当前位置的偏移量。

返回值如下：

读位置有效时：从最后位置到当前位置的偏移量

读位置无效时：0（调用 showmanyc()）

int_type streambuf::snextc()

读一个字符。如果读到的字符不为 eof，就读下一个字符。

返回值如下：

不为 eof 时：读到的字符

为 eof 时　：eof

int_type streambuf::sbumpc()

读一个字符，并将指针设定到下一个。

返回值如下：

读位置有效时：读到的字符

读位置无效时：eof

int_type streambuf::sgetc()

读一个字符。

返回值如下：

读位置有效时：读到的字符

读位置无效时：eof

int streambuf::sgetn(char* s, streamsize n)

将 n 个字符设定到 s 指向的存储区。

如果在字符串中检测出 eof，就结束设定。

返回值为设定的字符数。

int_type streambuf::sputbackc(char c)

在读位置正常并且读位置的回放数据与 c 相同时，就回放读位置。

返回值如下：

能回放时　：c 的值

不能回放时：eof

int streambuf::sungetc()

读位置正常时，回放读位置。

返回值如下：

能回放时　：回放的值

不能回放时：eof

int streambuf::sputc(char c)

插入字符 c。

返回值如下：

写位置正确时　：c 的值

写位置不正确时：eof

int_type streambuf::sputn(const char* s, streamsize n)

插入 s 指向的 n 个字符。

缓冲区小于 n 时，只插入缓冲区容量个的字符。

返回值为插入的字符数。

char* streambuf::eback() const

求输入流的头指针。

返回值为的头指针。

char* streambuf::gptr() const

求输入流的 next 指针。

返回值为 next 指针。

char* streambuf::egptr() const

求输入流的尾指针。

返回值为的尾指针。

void streambuf::gbump(int n)

将输入流的 next 指针移动 n。

void streambuf::setg(char* gbeg, char* gnext, char* gend)

给输入流的各指针进行以下的设定：

*B_beg_pptr = gbeg;
*B_next_pptr = gnext;
B_end_ptr = gend;
*_B_cnt_ptr = gend-gnext;
*_B_len_ptr = gend-gbeg;

char* streambuf::pbase() const

求输出流的头指针。

返回值为的头指针。

char* streambuf::pptr() const

求输出流的 next 指针。

返回值为 next 指针。

char* streambuf::epptr() const

求输出流的尾指针。

返回值为尾指针。

void streambuf::pbump(int n)

将输出流的 next 指针移动 n。

void streambuf::setp(char* pbeg, char* pend)

给输出流的各指针进行以下的设定：

*B_beg_pptr = pbeg;
*B_next_pptr = pbeg;
B_end_ptr = pend;
*_B_cnt_ptr=pend-pbeg;
*_B_len_ptr=pend-pbeg;

virtual streambuf* streambuf::setbuf(char* s, streamsize n)

按从 streambuf 派生的各类，执行个别定义的运算。

返回值为*this。此类不定义处理。

virtual pos_type streambuf::seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode)(ios_base::in | ios_base::out))

更改流位置。

返回值为(-1)。此类不定义处理。

virtual pos_type streambuf::seekpos(pos_type off, ios_base::openmode = (ios_base::openmode)(ios_base::in | ios_base::out))
更改流位置。
返回值为(-1)。此类不定义处理。

virtual int streambuf::sync()
转储清除输出流。
返回值为 0。此类不定义处理。

virtual int streambuf::showmanyc()
求输入流的有效字符数。
返回值为 0。此类不定义处理。

virtual streamsize streambuf::xsgetn(char* s, streamsize n)
将 n 个字符设定到 s 指向的存储区。
缓冲区小于 n 时，只设定缓冲区容量个的字符。
返回值为输入的字符数。

virtual int_type streambuf::underflow ()
不移动流位置，读一个字符。
返回值为 eof。此类不定义处理。

virtual int_type streambuf::uflow()
读 next 指针的一个字符。
返回值为 eof。此类不定义处理。

virtual int_type streambuf::pbackfail(int_type c = eof)
回放 c 表示的字符。
返回值为 eof。此类不定义处理。

virtual streamsize streambuf::xsputn(const char* s, streamsize n)
插入 s 指向的 n 个字符。
缓冲区小于 n 时，只插入缓冲区容量个的字符。
返回值为插入的字符数。

virtual int_type streambuf::overflow(int_type c = eof)
将 c 插入到输出流。
返回值为 eof。此类不定义处理。

（f）istream::sentry 类

种类	定义名	说明
变量	ok_	是否为可输入状态。
函数	sentry(istream& is, bool noskipws = false)	构造函数。
	~sentry()	析构函数。
	operator bool()	参照 ok_。

istream::sentry::sentry(istream& is, bool noskipws = _false)

内部类 sentry 的构造函数。
good()不为 0 时，能进行带格式或者不带格式输入。
tie()不为 0 时，转储清除关联的输出流。

istream::sentry::~sentry()

内部类 sentry 的析构函数。

istream::sentry::operator bool()

参照 ok_。
返回值为 ok_。

（g）istream 类

种类	定义名	说明
变量	chcount	由最后调用的输入函数抽出的字符数。
函数	int _ec2p_getistr(char* str, unsigned int dig, int mode)	以 dig 表示的基数转换 str。
	istream(streambuf* sb)	构造函数。
	virtual ~istream()	析构函数。
	istream& operator>>(bool& n)	将抽出的字符保存到 n。
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(float& n)	
	istream& operator>>(double& n)	
	istream& operator>>(long double& n)	
	istream& operator>>(void*& p)	转换为指向 void 的指针，并保存到 p。
	istream& operator>>(streambuf* sb)	抽出字符，保存到 sb 指向的存储区。
	streamsize gcount() const	求 chcount（抽出的字符数）。
	int_type get()	抽出字符。
	istream& get(char& c)	抽出字符，保存到 c。
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	抽出长度为 n-1 的字符串，保存到 s 指向的存储区。如果在字符串内检测出‘delim’，就结束输入。
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	
	istream& get(streambuf& sb)	抽出字符串，保存到 sb 指向的存储区。
	istream& get(streambuf& sb, char delim)	抽出字符串，保存到 sb 指向的存储区。如果在中途检测出字符‘delim’，就结束输入。
	istream& getline(char* s, streamsize n)	抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	

种类	定义名	说明
函数	istream& getline(char* s, streamsize n, char delim)	抽出长度为 n-1 的字符串，保存到 s 指向的存储区。如果在中途检测出字符‘delim’，就结束输入。
	istream& getline(signed char* s, streamsize n, char delim)	
	istream& getline(unsigned char* s, streamsize n, char delim)	
	istream& ignore(streamsize n = 1, int_type delim = streambuf::eof)	跳读 n 个字符。如果在中途检测出字符‘delim’，就中止跳读处理。
	int_type peek()	查找下一个可取得的输入字符。
	istream& read(char* s, streamsize n)	抽出长度为 n 的字符串，保存到 s 指向的存储区。
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	抽出长度为 n 的字符串，保存到 s 指向的存储区。
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome(unsigned char* s, streamsize n)	
	istream& putback(char c)	将字符返回到输入流。
	istream& unget()	返回输入流的位置。
	int sync()	调查是否有输入流。此函数调用 streambuf::pubsync()。
	pos_type tellg()	调查输入流的位置。此函数调用 streambuf::pubseekoff(0,cur,in)。
	istream& seekg(pos_type pos)	只从当前的流指针移动 pos。此函数调用 streambuf::pubseekpos(pos)。
	istream& seekg(off_type off, ios_base::seekdir dir)	用 dir 指定的方法移动输入流的读位置。此函数调用 streambuf::pubseekoff(off,dir)。

int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)

以 dig 表示的基数转换 str。

返回值为转换的基数。

istream::istream(streambuf* sb)

类 istream 构造函数。

调用 ios::init(sb)。

进行 chcount=0 的设定。

virtual istream::~istream ()

类 istream 的析构函数。

istream& istream::operator>>(bool& n)
istream& istream::operator>>(short& n)
istream& istream::operator>>(unsigned short& n)
istream& istream::operator>>(int& n)
istream& istream::operator>>(unsigned int& n)
istream& istream::operator>>(long& n)
istream& istream::operator>>(unsigned long& n)
istream& istream::operator>>(float& n)
istream& istream::operator>>(double& n)
istream& istream::operator>>(long double& n)

将抽出的字符保存到 n。
返回值为*this。

istream& istream::operator>>(void*& p)

将抽出的字符转换为 void*型，保存到 p 指向的存储区。
返回值为*this。

istream& istream::operator>>(streambuf* sb)

抽出字符，保存到 sb 指向的存储区。
没有抽出字符时，调用 setstate(failbit)。
返回值为*this。

streamsize istream::gcount() const

请参照 chcount（抽出的字符数）。
返回值为 chcount。

int_type istream::get()

抽出字符。
返回值如下：

能抽出时　：抽出的字符
不能抽出时：调用 setstate(failbit)，streambuf::eof

istream& istream::get (char& c)
istream& istream::get(signed char& c)
istream& istream::get(unsigned char& c)

抽出字符，保存到 c。抽出的字符为 streambuf::eof 时，设定 failbit。
返回值为*this。

istream& istream::get(char* s, streamsize n)
istream& istream::get(signed char* s, streamsize n)
istream& istream::get(unsigned char* s, streamsize n)

抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::get(char* s, streamsize n, char delim)
istream& istream::get(signed char* s, streamsize n, char delim)
istream& istream::get(unsigned char* s, streamsize n, char delim)

抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
如果在字符串内检测出‘delim’，就结束。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::get(streambuf& sb)

抽出字符串，保存到 sb 指向的存储区。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::get(streambuf& sb, char delim)

抽出字符串，保存到 sb 指向的存储区。
如果在中途检测出字符‘delim’，就结束。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::getline(char* s, streamsize n)
istream& istream::getline(signed char* s, streamsize n)
istream& istream::getline(unsigned char* s, streamsize n)

抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::getline(char* s, streamsize n, char delim)
istream& istream::getline(signed char* s, streamsize n, char delim)
istream& istream::getline(unsigned char* s, streamsize n, char delim)

抽出长度为 n-1 的字符串，保存到 s 指向的存储区。
如果在中途检测出字符‘delim’，就结束。
ok_==false 或者抽出的字符数为 0 时，设定 failbit。
返回值为*this。

istream& istream::ignore(streamsize n = 1, int_type delim = streambuf::eof)

跳读 n 个字符。
如果在中途检测出字符‘delim’，就中止跳读处理。
返回值为*this。

int_type istream::peek()

查找下一个可取得的输入字符。
返回值如下：

ok_==false 时：streambuf::eof
ok_!=false 时 ：rdbuf()->sgetc()

istream& istream::read(char* s, streamsize n)
istream& istream::read(signed char* s, streamsize n)
istream& istream::read(unsigned char* s, streamsize n)

ok_!=false 时，抽出长度为 n 的字符串，保存到 s 指向的存储区。
抽出的字符数与 n 不同时，设定 eofbit。
返回值为*this。

streamsize istream::readsome(char* s, streamsize n)
streamsize istream::readsome (signed char* s, streamsize n)
streamsize istream::readsome (unsigned char* s, streamsize n)

抽出长度为 n 的字符串，保存到 s 指向的存储区。
如果字符数大于流容量，就保存流容量的字符数。
返回值为抽出的字符数。

istream& istream::putback(char c)

将字符 c 返回到输入流。回放的字符为 streambuf::eof 时，设定 badbit。
返回值为*this。

istream& istream::unget()

返回一个输入流的指针。
抽出的字符为 streambuf::eof 时，设定 badbit。
返回值为*this。

int istream::sync()

调查是否有输入流。
此函数调用 streambuf::pubsync()。
返回值如下：
　无输入流时：streambuf::eof
　有输入流时：0

pos_type istream::tellg()

查找输入流的位置。
此函数调用 streambuf::pubseekoff(0,cur,in)。
返回值如下：
　从流起始位置的偏移量
　但是，在输入处理发生错误时为-1

istream& istream::seekg(pos_type pos)

只从当前的流指针移动 pos。
此函数调用 streambuf::pubseekpos(pos)。
返回值为*this。

istream& istream::seekg(off_type off, ios_base::seekdir dir)

用 dir 指定的方法移动输入流的读位置。

此函数调用 streambuf::pubseekoff(off,dir)。

在输入处理发生错误时，不进行处理。

返回值为*this。

（h）istream 类操纵符

种类	定义名	说明
函数	istream& ws(istream& is)	跳读空格字符。

istream& ws(istream& is)

跳读空格类字符。

返回值为 is。

（i）istream 成员外函数

种类	定义名	说明
函数	istream& operator>>(istream& in, char* s)	抽出字符串，保存到 s 指向的存储区。
	istream& operator>>(istream& in, signed char* s)	
	istream& operator>>(istream& in, unsigned char* s)	
	istream& operator>>(istream& in, char& c)	抽出字符，保存到 c。
	istream& operator>>(istream& in, singed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

istream& operator>>(istream& in, char* s)
istream& operator>>(istream& in, signed char* s)
istream& operator>>(istream& in, unsigned char* s)

抽出字符串，保存到 s 指向的存储区。

在保存（字段宽度-1）个的字符、输入流中出现 streambuf::eof 或者下一个能输入的字符 c 为 isspace(c)=1 时，结束处理。保存字符数为 0 时，设定 failbit。

返回值为 in。

istream& operator>>(istream& in, char& c)
istream& operator>>(istream& in, singed char& c)
istream& operator>>(istream& in, unsigned char& c)

抽出字符，保存到 c。

没有抽出输入时，设定 failbit。

返回值为 in。

（j）ostream::sentry 类

种类	定义名	说明
变量	ok_	是否为可输出状态。
	_ _ec2p_os	指向 ostream 目标的指针。
函数	sentry(ostream& os)	构造函数。
	~sentry()	析构函数。
	operator bool()	参照 ok_。

ostream::sentry::sentry (ostream& os)

内部类 sentry 的构造函数。

如果 good()不为 0 并且 tie()不为 0，就调用 flush()。给__ec2p_os 设定 os。

ostream::sentry::~sentry ()

内部类 sentry 的析构函数。

如果_ _ec2p_os->flags() & ios_base::unitbuf 为真，就调用 flush()。

ostream::sentry::operator bool ()

参照 ok_。

返回值为 ok_。

（k）ostream 类

种类	定义名	说明
函数	ostream(streambuf* sbptr)	构造函数。
	virtual ~ostream()	析构函数。
	ostream& operator<<(bool n) ostream& operator<<(short n) ostream& operator<<(unsigned short n) ostream& operator<<(int n) ostream& operator<<(unsigned int n) ostream& operator<<(long n) ostream& operator<<(unsigned long n) ostream& operator<<(float n) ostream& operator<<(double n) ostream& operator<<(long double n) ostream& operator<<(void* n)	将 n 插入到输出流。
	ostream& operator<<(streambuf* sbptr)	将 sbptr 的输出行插入到输出流。
	ostream& put(char c)	将字符 c 插入到输出流。
	ostream& write(const char* s, streamsize n) ostream& write(const signed char* s, streamsize n) ostream& write(const unsigned char* s, streamsize n)	将 s 的 n 个字符插入到输出流。
	ostream& flush()	转储清除输出流。此函数调用 streambuf::pubsync()。
	pos_type tellp()	求当前写位置。此函数调用 streambuf::pubseekoff(0,cur,out)。
	ostream& seekp(pos_type pos)	求从流的起始位置到当前位置的偏移量。从当前流指针移动 pos。此函数调用 streambuf::pubseekpos(pos)。
	ostream& seekp(off_type off, seekdir dir)	以 dir 为基准，只将流的写位置移动 off。此函数调用 streambuf::pubseekoff(off,dir)。

ostream::ostream(streambuf* sbptr)

构造函数。

调用 ios (sbptr)。

virtual ostream::~ostream()

析构函数。

ostream& ostream::operator<<(bool n)
ostream& ostream::operator<<(short n)
ostream& ostream::operator<<(unsigned short n)
ostream& ostream::operator<<(int n)
ostream& ostream::operator<<(unsigned int n)
ostream& ostream::operator<<(long n)
ostream& ostream::operator<<(unsigned long n)
ostream& ostream::operator<<(float n)
ostream& ostream::operator<<(double n)
ostream& ostream::operator<<(long double n)
ostream& ostream::operator<<(void* n)

sentry::ok_==true 时，将 n 插入到输出流。

sentry::ok_==false 时，设定 failbit。

返回值为*this。

ostream& ostream::operator<<(streambuf* sbptr)

sentry::ok_==true 时，将 sbptr 的输出行插入到输出流。

sentry::ok_==false 时，设定 failbit。

返回值为*this。

ostream& ostream::put(char c)

sentry::ok_==true 并且 rdbuf()->sputc(c)!=streambuf::eof 时，将 c 插入到输出流。

上述以外的情况时，设定 failbit。

返回值为*this。

ostream& ostream::write(const char* s, streamsize n)
ostream& ostream::write(const signed char* s, streamsize n)
ostream& ostream::write(const unsigned char* s, streamsize n)

sentry::ok_==true 并且 rdbuf()->sputn(s, n)==n 时，将 s 的 n 个字符插入到输出流。

上述以外的情况时，设定 badbit。

返回值为*this。

ostream& ostream::flush()

转储清除输出流。

此函数调用 streambuf::pubsync()。

返回值为*this。

pos_type ostream::tellp()

求当前写位置。

此函数调用 streambuf::pubseekoff(0,cur,out)。

返回值如下：

当前流位置

但是，在处理中发生错误时为-1

ostream& ostream::seekp(pos_type pos)

没有错误时，求从流的起始位置到当前位置的偏移量。

另外，只从当前流指针移动 pos。

此函数调用 streambuf::pubseekpos(pos)。

返回值为*this。

ostream& ostream::seekp(off_type off, seekdir dir)

没有错误时，以 dir 为基准将流位置移动 off。

此函数调用 streambuf::pubseekoff(off,dir)。

返回值为*this。

（l）ostream 类操纵符

种类	定义名	说明
函数	ostream& endl(ostream& os)	插入换行，转储清除输出流。
	ostream& ends(ostream& os)	插入空码。
	ostream& flush(ostream& os)	转储清除输出流。

ostream& endl(ostream& os)

将换行字符插入到流。

转储清除输出流。此函数调用 flush()。

返回值为 os。

ostream& ends(ostream& os)

给输出流插入空码。

返回值为 os。

ostream& flush(ostream& os)

转储清除输出流。此函数调用 streambuf::sync()。

返返回值为 os。

（m）ostream 成员外函数

种类	定义名	说明
函数	ostream& operator<<(ostream& os, char s)	将 s 插入到输出流。
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const singed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

ostream& operator<<(ostream& os, char s)
ostream& operator<<(ostream& os, signed char s)
ostream& operator<<(ostream& os, unsigned char s)
ostream& operator<<(ostream& os, const char* s)
ostream& operator<<(ostream& os, const singed char* s)
ostream& operator<<(ostream& os, const unsigned char* s)

sentry::ok_= =true 并且没有错误时，将 s 插入到输出流。
上述以外的情况时，设定 failbit。
返回值为 os。

（n）smanip 类操纵符

种类	定义名	说明
函数	smanip resetiosflags(ios_base::fmtflags mask)	清除由 mask 值指定的标志。
	smanip setiosflags(ios_base::fmtflags mask)	设定格式标志（fmtfl）。
	smanip setbase(int base)	设定输出时使用的基数。
	smanip setfill(char c)	设定填充字符（fillch）。
	smanip setprecision(int n)	设定精度（prec）。
	smanip setw(int n)	设定字段宽度（wide）。

smanip resetiosflags(ios_base::fmtflags mask)
清除由 mask 值指定的标志。
返回值为输入/输出对象的目标。

smanip setiosflags(ios_base::fmtflags mask)
设定格式标志（fmtfl）。
返回值为输入/输出对象的目标。

smanip setbase(int base)
设定输出时使用的基数。
返回值为输入/输出对象的目标。

smanip setfill(char c)
设定填充字符（fillch）。
返回值为输入/输出对象的目标。

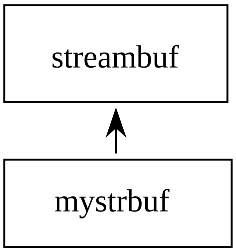
smanip setprecision(int n)
设定精度（prec）。
返回值为输入/输出对象的目标。

smanip setw(int n)
设定字段宽度（wide）。
返回值为输入/输出对象的目标。

（o）EC++输入/输出库的使用例子

初始化 istream、ostream 的目标时，通过用指向 mystrbuf 类目标的指针取代 streambuf，输入/输出流就能成为可使用状态。

类的派生关系如下。箭头表示从派生类参照基类。



种类	定义名	说明
变量	_file_Ptr	文件指针。
函数	mystrbuf()	构造函数。进行 streambuf 缓冲区的初始化。
	mystrbuf(void* ptr)	（同上）
	virtual ~mystrbuf()	析构函数。
	void* myfptr() const	返回指向 FILE 型结构体的指针。
	mystrbuf* open(const char* filename, int mode)	指定文件名和模式，打开文件。
	mystrbuf* close()	关闭文件。
	virtual streambuf* setbuf(char* s, streamsize n)	确保流输入/输出缓冲区。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	更改流指针的位置。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	更改流指针的位置。
	virtual int sync()	转储清除流。
	virtual int showmanyc()	返回输入流的有效字符数。
	virtual int_type underflow()	不移动流位置，读一个字符。
	virtual int_type pbackfail(int_type c = streambuf::eof)	回放 c 表示的字符。
	virtual int_type overflow(int_type c = streambuf::eof)	插入 c 表示的字符。
	void _Init(_f_type* fp)	初始处理。

例：

```
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
void main(void)
{
    mystrbuf myfin(stdin);
    mystrbuf myfout(stdout);
    istream mycin(&myfin);
    ostream mycout(&myfout);

    int i;
    short s;
    long l;
    char c;
    string str;

    mycin >> i >> s >> l >> c >> str;
    mycout << "This is EC++ Liblary." << endl
            << i << s << l << c << str << endl;
    return;
}
```

（3）存储器管理库

与存储器管理库对应的头文件如下：

- <new>

  定义进行存储器的确保/释放的函数。

  通过给_ec2p_new_handler 变量设定异常处理函数的地址，在存储器确保失败时，也能执行异常处理。

  _ec2p_new_handler 为 static 变量，初始值为 NULL。通过使用此处理程序，会失去可重入性。

  异常处理函数所需的运行：

  - 建立和返回可分配的区域。
  - 不能建立和返回时的运行没有规定。

种类	定义名	说明
型	new_handler	指向返回 void 型的函数的指针型。
变量	_ec2p_new_handler	指向异常处理函数的指针。
函数	void* operator new(size_t size)	确保 size 字节的区域。
	void* operator new[](size_t size)	确保 size 字节的数组区。
	void* operator new(size_t size, void* ptr)	将 ptr 指定的区域分配为存储区。
	void* operator new[](size_t size, void* ptr)	将 ptr 指定的区域分配为数组区。
	void operator delete(void* ptr)	释放区域。
	void operator delete[](void* ptr)	释放数组区。
	new_handler set_new_handler(new_handler new_P)	给_ec2p_new_handler 设定异常处理函数地址（new_P）。

void* operator new(size_t size)

分配 size 字节的区域。

如果区域分配失败并且 new_handler 被设定，就调用 new_handler。

返回值如下：

区域确保成功时：指向 void 型的指针

区域确保失败时：NULL

void* operator new [](size_t size)

确保 size 字节的数组区。

如果区域分配失败并且 new_handler 被设定，就调用 new_handler。

返回值如下：

区域确保成功时：指向 void 型的指针

区域确保失败时：NULL

void* operator new(size_t size, void* ptr)

将 ptr 指定的区域分配为存储区。

返回值为 ptr。

void* operator new[](size_t size, void* ptr)

将 ptr 指定的区域分配为数组区。

返回值为 ptr。

void operator delete(void* ptr)

释放 ptr 指定的存储区。ptr 为 NULL 时，不进行任何操作。

void operator delete [](void* ptr)

释放 ptr 指定数组区。ptr 为 NULL 时，不进行任何操作。

new_handler set_new_handler(new_handler new_P)

给_ec2p_new_handler 设定 new_P。

返回值为_ec2p_new_handler。

（4）复数计算类库

与复数计算类库对应的头文件如下：

- <complex>

  定义 float_complex 类和 double_complex 类。

  这些类没有派生关系。

（a）float_complex 类

种类	定义名	说明
型	value_type	float 型。
变量	_re	定义 float 精度的实部。
	_im	定义 float 精度的虚部。
函数	float_complex(float re = 0.0f, float im = 0.0f)	构造函数。
	float_complex(const double_complex& rhs)	
	float real() const	求实部（_re）。
	float imag() const	求虚部（_im）。
	float_complex& operator=(float rhs)	将 rhs 复制到实部。虚部设定 0.0f。
	float_complex& operator+=(float rhs)	实部加上 rhs，结果保存到*this。
	float_complex& operator-=(float rhs)	实部减去 rhs，结果保存到*this。
	float_complex& operator*=(float rhs)	乘 rhs，结果保存到*this。
	float_complex& operator/=(float rhs)	除 rhs，结果保存到*this。
	float_complex& operator=(const float_complex& rhs)	复制 rhs。
	float_complex& operator+=(const float_complex& rhs)	加上 rhs，结果保存到*this。
	float_complex& operator-=(const float_complex& rhs)	减去 rhs，结果保存到*this。
	float_complex& operator*=(const float_complex& rhs)	乘 rhs，结果保存到*this。
	float_complex& operator/=(const float_complex& rhs)	除 rhs，结果保存到*this。

float_complex::float_complex (float re = 0.0f, float im = 0.0f)

类 float_complex 的构造函数。

用以下的值进行初始化：

_re = re;

_im = im;

float_complex::float_complex(const double_complex& rhs)

类 float_complex 的构造函数。

用以下的值进行初始化：

_re = (float)rhs.real();

_im = (float)rhs.imag();

float float_complex::real () const
求实部。
返回值为 this->_re。

float float_complex::imag () const
求虚部。
返回值为 this->_im。

float_complex& float_complex::operator= (float rhs)
将 rhs 复制到实部（_re），虚部（_im） 设定 0.0f。
返回值为*this。

float_complex& float_complex::operator+= (float rhs)
实部（_re）加上 rhs，结果保存到实部，虚部（_im）的值不变。
返回值为*this。

float_complex& float_complex::operator-= (float rhs)
实部（_re）减去 rhs，结果保存到实部（_re），虚部（_im）的值不变。
返回值为*this。

float_complex& float_complex::operator*= (float rhs)
乘 rhs，结果保存到*this。
（_re=_re*rhs, _im=_im*rhs）
返回值为*this。

float_complex& float_complex::operator/= (float rhs)
除 rhs，结果保存到*this。
（_re=_re/rhs, _im=_im/rhs）
返回值为*this。

float_complex& float_complex::operator= (const float_complex& rhs)
复制 rhs。
返回值为*this。

float_complex& float_complex::operator+= (const float_complex& rhs)
加上 rhs，结果保存到*this。
返回值为*this。

float_complex& float_complex::operator-= (const float_complex& rhs)
减去 rhs，结果保存到*this。
返回值为*this。

float_complex& float_complex::operator*= (const float_complex& rhs)
乘 rhs，结果保存到*this。
返回值为*this。

float_complex& float_complex::operator/= (const float_complex& rhs)

除 rhs，结果保存到*this。

返回值为*this。

（b）float_complex 成员外函数

种类	定义名	说明
函数	float_complex operator+(const float_complex& lhs)	进行 lhs 的一元＋运算。
	float_complex operator+(const float_complex& lhs, const float_complex& rhs)	lhs 和 rhs 相加，结果保存到 lhs。
	float_complex operator+(const float_complex& lhs, const float& rhs)	
	float_complex operator+(const float& lhs, const float_complex& rhs)	
	float_complex operator-(const float_complex& lhs)	进行 lhs 的一元－运算。
	float_complex operator-(const float_complex& lhs, const float_complex& rhs)	lhs 减去 rhs，结果保存到 lhs。
	float_complex operator-(const float_complex& lhs, const float& rhs)	
	float_complex operator-(const float& lhs, const float_complex& rhs)	
	float_complex operator*(const float_complex& lhs, const float_complex& rhs)	lhs 和 rhs 相乘，结果保存到 lhs。
	float_complex operator*(const float_complex& lhs, const float& rhs)	
	float_complex operator*(const float& lhs, const float_complex& rhs)	
	float_complex operator/(const float_complex& lhs, const float_complex& rhs)	rhs 除 lhs，结果保存到 lhs。
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	
	bool operator==(const float_complex& lhs, const float_complex& rhs)	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	bool operator==(const float_complex& lhs, const float& rhs)	
	bool operator==(const float& lhs, const float_complex& rhs)	

种类	定义名	说明
函数	bool operator!=(const float_complex& lhs, const float_complex& rhs)	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	bool operator!=(const float_complex& lhs, const float& rhs)	
	bool operator!=(const float& lhs, const float_complex& rhs)	
	istream& operator>>(istream& is, float_complex& x)	输入 u、(u)或者(u,v)（u：实部，v：虚部）格式的 x。
	ostream& operator<<(ostream& os, const float_complex& x)	以 u、(u)或者 (u,v)（u：实部，v：虚部）格式输出 x。
	float real(const float_complex& x)	求实部。
	float imag(const float_complex& x)	求虚部。
	float abs(const float_comlex& x)	求绝对值。
	float arg(const float_complex& x)	求相位角。
	float norm(const float_complex& x)	求 2 次方的绝对值。
	float_complex conj(const float_complex& x)	求共轭复数。
	float_complex polar(const float& rho, const float& theta)	求对应大小为 rho、相位角为 theta 的复数的 float_complex 值。
	float_complex cos(const float_complex& x)	求复数余弦。
	float_complex cosh(const float_complex& x)	求复数双曲余弦。
	float_complex exp(const float_complex& x)	求指数函数。
	float_complex log(const float_complex& x)	求自然对数。
	float_complex log10(const float_complex& x)	求常用对数。
	float_complex pow(const float_complex& x, int y)	求 x 的 y 次方。
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	
	float_complex pow(const float& x, const float_complex& y)	
	float_complex sin(const float_complex& x)	求复数正弦。
	float_complex sinh(const float_complex& x)	求复数双曲正弦。
	float_complex sqrt(const float_complex& x)	求右半空间范围内的平方根。
	float_complex tan(const float_complex& x)	求复数正切。
	float_complex tanh(const float_complex& x)	求复数双曲正切。

float_complex operator+ (const float_complex& lhs)
进行 lhs 的一元＋运算。
返回值为 lhs。

float_complex operator+(const float_complex& lhs, const float_complex& rhs)
float_complex operator+(const float_complex& lhs, const float& rhs)
float_complex operator+(const float& lhs, const float_complex& rhs)
lhs 和 rhs 相加，结果保存到 lhs。
返回值为 float_complex(lhs)+=rhs。

float_complex operator-(const float_complex& lhs)
进行 lhs 的一元－运算。
返回值为 float_complex(-lhs.real(),-lhs.imag())。

float_complex operator-(const float_complex& lhs, const float_complex& rhs)
float_complex operator-(const float_complex& lhs, const float& rhs)
float_complex operator-(const float& lhs, const float_complex& rhs)
lhs 减去 rhs，结果保存到 lhs。
返回值为 float_complex(lhs)-=rhs。

float_complex operator*(const float_complex& lhs, const float_complex& rhs)
float_complex operator*(const float_complex& lhs, const float& rhs)
float_complex operator*(const float& lhs, const float_complex& rhs)
lhs 和 rhs 相乘，结果保存到 lhs。
返回值为 float_complex(lhs)*=rhs。

float_complex operator/(const float_complex& lhs, const float_complex& rhs)
float_complex operator/(const float_complex& lhs, const float& rhs)
float_complex operator/(const float& lhs, const float_complex& rhs)
rhs 除 lhs，结果保存到 lhs。
返回值为 float_complex(lhs)/=rhs。

bool operator==(const float_complex& lhs, const float_complex& rhs)
bool operator==(const float_complex& lhs, const float& rhs)
bool operator==(const float& lhs, const float_complex& rhs)
将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。float 型参数时，将虚部假设为 float 型的 0.0f。
返回值为 lhs.real()= =rhs.real() && lhs.imag()= =rhs.imag()。

bool operator!=(const float_complex& lhs, const float_complex& rhs)
bool operator!=(const float_complex& lhs, const float& rhs)
bool operator!=(const float& lhs, const float_complex& rhs)
将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。float 型参数时，将虚部假设为 float 型的 0.0f。
返回值为 lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()。

istream& operator>>(istream& is, float_complex& x)

输入 u、(u)或者(u,v)（u 为实部，v 为虚部）格式的 x。将输入值转换为 float_complex。
如果输入 u、(u)、(u,v)格式以外的数据，就调用 is.setstate(ios_base::failbit)。
返回值为 is。

ostream& operator<<(ostream& os, const float_complex& x)

将 x 输出到 os。
输出格式为 u、(u)或者(u,v)（u 为实部，v 为虚部）。
返回值为 os。

float real(const float_complex& x)

求实部。
返回值为 x.real()。

float imag(const float_complex& x)

求虚部。
返回值为 x.imag()。

float abs(const float_complex& x)

求绝对值。
返回值为|x.real()| + |x.imag()|。

float arg(const float_complex& x)

求位相角。
返回值为 atan2f(x.imag() , x.real())。

float norm(const float_complex& x)

求 2 次方的绝对值。
返回值为 x.real()^2+ x.imag()^2。

float_complex conj(const float_complex& x)

求共轭复数。
返回值为 float_complex(x.real(), (-1)*x.imag())。

float_complex polar(const float& rho, const float& theta)

求对应大小为 rho、位相角（偏角）为 theta 复数的 float_complex 的值。
返回值为 float_complex(rho*cosf(theta), rho*sinf(theta))。

float_complex cos(const float_complex& x)

求复数余弦。
返回值为 float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))。

float_complex cosh(const float_complex& x)

求复数双曲余弦。
返回值为 cos(float_complex((-1)*x.imag(), x.real()))。

float_complex exp(const float_complex& x)

求指数函数。

返回值为 expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())。

float_complex log(const float_complex& x)

求（以 e 为底）自然对数。

返回值为 float_complex(logf(abs(x)), arg(x))。

float_complex log10(const float_complex& x)

求（以 10 为底）常用对数。

返回值为 float_complex(log10f(abs(x)), arg(x)/logf(10))。

float_complex pow(const float_complex& x, int y)
float_complex pow(const float_complex& x, const float& y)
float_complex pow(const float_complex& x, const float_complex& y)
float_complex pow(const float& x, const float_complex& y)

求 x 的 y 次方。

pow(0,0)时，为定义域错误。

返回值如下：

float_complex pow (const float_complex& x,const float_complex& y)：exp(y* logf(x))

上述以外：exp(y*log(x))

float_complex sin(const float_complex& x)

求复数正弦。

返回值为 float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))。

float_complex sinh(const float_complex& x)

求复数双曲正弦。

返回值为 float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))。

float_complex sqrt(const float_complex& x)

求右半空间范围内的平方根。

返回值为 float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))。

float_complex tan(const float_complex& x)

求复数正切。

返回值为 sin(x) / cos(x)。

float_complex tanh(const float_complex& x)

求复数双曲正切。

返回值为 sinh(x) / cosh(x)。

（c）double_complex 类

种类	定义名	说明
型	value_type	double 型。
变量	_re	定义 double 精度的实部。
	_im	定义 double 精度的虚部。
函数	double_complex(double re = 0.0, double im = 0.0)	构造函数。
	double_complex(const float_complex&)	
	double real() const	求实部。
	double imag() const	求虚部。
	double_complex& operator=(double rhs)	将 rhs 复制到实部，将虚部设定为 0.0。
	double_complex& operator+=(double rhs)	实部加上 rhs，结果保存到*this。
	double_complex& operator-=(double rhs)	实部减去 rhs，结果保存到*this。
	double_complex& operator*=(double rhs)	乘 rhs，结果保存到*this。
	double_complex& operator/=(double rhs)	被 rhs 除，结果保存到*this。
	double_complex& operator=(const double_complex& rhs)	复制 rhs。
	double_complex& operator+=(const double_complex& rhs)	减去 rhs，结果保存到*this。
	double_complex& operator-=(const double_complex& rhs)	加上 rhs，结果保存到*this。
	double_complex& operator*=(const double_complex& rhs)	乘 rhs，结果保存到*this。
	double_complex& operator/=(const double_complex& rhs)	被 rhs 除，结果保存到*this。

double_complex::double_complex(double re = 0.0, double im = 0.0)

类 double_complex 的构造函数。

用以下的值进行初始化。

_re = re;

_im = im;

double_complex::double_complex(const float_complex&)

类 double_complex 的构造函数。

用以下的值进行初始化。

_re = (double)rhs.real();

_im = (double)rhs.imag();

double double_complex::real () const

求实部。

返回值为 this->_re。

double double_complex::imag () const

求虚部。

返回值为 this->_im。

double_complex& double_complex::operator=(double rhs)
将 rhs 复制到实部（_re），将虚部（_im）设定为 0.0。
返回值为*this。

double_complex& double_complex::operator+=(double rhs)
实部（_re）加上 rhs，结果保存到实部（_re），虚部（_im）的值不变。
返回值为*this。

double_complex& double_complex::operator-=(double rhs)
实部（_re）减去 rhs，结果保存到实部（_re），虚部（_im）的值不变。
返回值为*this。

double_complex& double_complex::operator*=(double rhs)
乘 rhs，结果保存到*this。
（_re=_re*rhs, _im=_im*rhs）
返回值为*this。

double_complex& double_complex::operator/=(double rhs)
被 rhs 除，结果保存到*this。
（_re=_re/rhs, _im=_im/rhs）
返回值为*this。

double_complex& double_complex::operator=(const double_complex& rhs)
复制 rhs。
返回值为*this。

double_complex& double_complex::operator+=(const double_complex& rhs)
加上 rhs，结果保存到*this。
返回值为*this。

double_complex& double_complex::operator-=(const double_complex& rhs)
减去 rhs，结果保存到*this。
返回值为*this。

double_complex& double_complex::operator*=(const double_complex& rhs)
乘 rhs，结果保存到*this。
返回值为*this。

double_complex& double_complex::operator/=(const double_complex& rhs)
被 rhs 除，结果保存到*this。
返回值为*this。

（d）double_complex 成员外函数

种类	定义名	说明
函数	double_complex operator+(const double_complex& lhs)	进行 lhs 的一元＋运算。
	double_complex operator+(const double_complex& lhs, const double_complex& rhs)	lhs 和 rhs 相加，结果保存到 lhs。
	double_complex operator+(const double_complex& lhs, const double& rhs)	
	double_complex operator+(const double& lhs, const double_complex& rhs)	
	double_complex operator-(const double_complex& lhs)	进行 lhs 的一元－运算。
	double_complex operator-(const double_complex& lhs, const double_complex& rhs)	lhs 减去 rhs，结果保存到 lhs。
	double_complex operator-(const double_complex& lhs, const double& rhs)	
	double_complex operator-(const double& lhs, const double_complex& rhs)	
	double_complex operator*(const double_complex& lhs, const double_complex& rhs)	lhs 和 rhs 相乘，结果保存到 lhs。
	double_complex operator*(const double_complex& lhs, const double& rhs)	
	double_complex operator*(const double& lhs, const double_complex& rhs)	
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	rhs 除 lhs，结果保存到 lhs。
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	
	bool operator==(const double_complex& lhs, const double_complex& rhs)	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	

种类	定义名	说明
函数	bool operator!=(const double_complex& lhs, const double_complex& rhs) bool operator!=(const double_complex& lhs, const double& rhs) bool operator!=(const double& lhs, const double_complex& rhs)	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	istream& operator>>(istream& is, double_complex& x)	输入 u、(u)或者(u,v)（u：实部、v：虚部）格式的 x。
	ostream& operator<<(ostream& os, double_complex& x)	以 u、(u)或者 (u,v)（u：实部、v：虚部）格式输出 x。
	double real(const double_complex& x)	求实部。
	double imag(const double_complex& x)	求虚部。
	double abs(const double_comlex& x)	求绝对值。
	double arg(const double_complex& x)	求相位角。
	double norm(const double_complex& x)	求 2 次方的绝对值。
	double_complex conj(const double_complex& x)	求共轭复数。
	double_complex polar(const double& rho, const double& theta)	求对应大小为 rho、相位角为 theta 的复数的 double_complex 值。
	double_complex cos(const double_complex& x)	求复数余弦。
	double_complex cosh(const double_complex& x)	求复数双曲余弦。
	double_complex exp(const double_complex& x)	求指数函数。
	double_complex log(const double_complex& x)	求自然对数。
	double_complex log10(const double_complex& x)	求常用对数。
	double_complex pow(const double_complex& x, int y) double_complex pow(const double_complex& x, const double& y) double_complex pow(const double_complex& x, const double_complex& y) double_complex pow(const double& x, const double_complex& y)	求 x 的 y 次方。
	double_complex sin(const double_complex& x)	求复数正弦。
	double_complex sinh(const double_complex& x)	求复数双曲正弦。
	double_complex sqrt(const double_complex& x)	求右半空间范围的平方根。
	double_complex tan(const double_complex& x)	求复数正切。
	double_complex tanh(const double_complex& x)	求复数双曲正切。

double_complex operator+(const double_complex& lhs)

进行 lhs 的一元＋运算。

返回值为 lhs。

double_complex operator+(const double_complex& lhs, const double_complex& rhs)
double_complex operator+(const double_complex& lhs, const double& rhs)
double_complex operator+(const double& lhs, const double_complex& rhs)

lhs 加上 rhs，结果保存到 lhs。

返回值为 double_complex(lhs)+=rhs。

double_complex operator-(const double_complex& lhs)

进行 lhs 的一元－运算。

返回值为 double_complex(-lhs.real(), -lhs.imag())。

double_complex operator-(const double_complex& lhs, const double_complex& rhs)
double_complex operator-(const double_complex& lhs, const double& rhs)
double_complex operator-(const double& lhs, const double_complex& rhs)

lhs 减去 rhs，结果保存到 lhs。

返回值为 double_complex(lhs)-=rhs。

double_complex operator*(const double_complex& lhs, const double_complex& rhs)
double_complex operator*(const double_complex& lhs, const double& rhs)
double_complex operator*(const double& lhs, const double_complex& rhs)

lhs 乘 rhs，结果保存到 lhs。

返回值为 double_complex(lhs)*=rhs。

double_complex operator/(const double_complex& lhs, const double_complex& rhs)
double_complex operator/(const double_complex& lhs, const double& rhs)
double_complex operator/(const double& lhs, const double_complex& rhs)

rhs 除 lhs，结果保存到 lhs。

返回值为 double_complex(lhs)/=rhs。

bool operator==(const double_complex& lhs, const double_complex& rhs)
bool operator==(const double_complex& lhs, const double& rhs)
bool operator==(const double& lhs, const double_complex& rhs)

将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。double 型参数时，将虚部假设为 double 型的 0.0。

返回值为 lhs.real()= =rhs.real() && lhs.imag()= =rhs.imag()。

bool operator!=(const double_complex& lhs, const double_complex& rhs)
bool operator!=(const double_complex& lhs, const double& rhs)
bool operator!=(const double& lhs, const double_complex& rhs)

将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。double 型参数时，将虚部假设为 double 型的 0.0。

返回值为 lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()。

istream& operator>>(istream& is, double_complex& x)

输入 u、(u)或者(u,v)（u 为实部、v 为虚部）格式的复数 x。将输入值转换为 double_complex。
如果输入 u、(u)、(u,v)格式以外的数据，就调用 is.setstate(ios_base::failbit)。
返回值为 is。

ostream& operator<<(ostream& os, const double_complex& x)

将 x 输出到 os。
输出格式为 u、(u)或者(u,v)（u 为实部、v 为虚部）。
返回值为 os。

double real(const double_complex& x)

求实部。
返回值为 x.real()。

double imag(const double_complex& x)

求虚部。
返回值为 x.imag()。

double abs(const double_complex& x)

求绝对值。
返回值为|x.real()| + |x.imag()|。

double arg(const double_complex& x)

求相位角。
返回值为 atan2(x.imag() , x.real())。

double norm(const double_complex& x)

求 2 次方的绝对值。
返回值为 x.real()^2+ x.imag()^2。

double_complex conj(const double_complex& x)

求共轭复数。
返回值为 double_complex(x.real(), (-1)*x.imag())。

double_complex polar(const double& rho, const double& theta)

求对应大小为 rho、相位角（偏角）为 theta 的复数的 double_complex 值。
返回值为 double_complex(rho*cos(theta), rho*sin(theta))。

double_complex cos(const double_complex& x)

求复数余弦。
返回值为 double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag()))。

double_complex cosh(const double_complex& x)

求复数双曲余弦。
返回值为 cos(double_complex((-1)*x.imag(), x.real()))。

double_complex exp(const double_complex& x)
求指数函数。
返回值为 exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())。

double_complex log(const double_complex& x)
求（以 e 为底）自然对数。
返回值为 double_complex(log(abs(x)), arg(x))。

double_complex log10 (const double_complex& x)
求（以 10 为底）常用对数。
返回值为 double_complex(log10(abs(x)), arg(x)/log(10))。

double_complex pow(const double_complex& x, int y)
double_complex pow(const double_complex& x, const double& y)
double_complex pow(const double_complex& x, const double_complex& y)
double_complex pow(const double& x, const double_complex& y)
求 x 的 y 次方。
pow(0,0)时，为定义域错误。
返回值为 exp(y*log(x))。

double_complex sin(const double_complex& x)
求复数正弦。
返回值为 double_complex(sin(x.real())*cosh(x.imag())或者 cos(x.real())*sinh(x.imag())) 。

double_complex sinh(const double_complex& x)
求复数双曲正弦。
返回值为 double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))。

double_complex sqrt(const double_complex& x)
求右半空间范围的平方根。
返回值为 double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))。

double_complex tan(const double_complex& x)
求复数正切。
返回值为 sin(x) / cos(x)。

double_complex tanh(const double_complex& x)
求复数双曲正切。
返回值为 sinh(x) / cosh(x)。

（5）字符串操作类库

与字符串操作类库对应的头文件如下：

- <string>
  定义 string 类。

本类没有派生关系。

（a）string 类

种类	定义名	说明
型	iterator	char*型。
	const_iterator	const char*型。
常数	npos	字符串的最大长度（UINT_MAX 字符）。
变量	s_ptr	指向目标保存字符串的区域的指针。
	s_len	目标保存的字符串长度。
	s_res	确保目标保存字符串的区域的容量。
函数	string(void)	构造函数。
	string(const string& str, size_t pos = 0, size_t n = npos)	
	string(const char* str, size_t n)	
	string(const char* str)	
	string(size_t n, char c)	
	~string()	析构函数。
	string& operator=(const string& str)	赋值 str。
	string& operator=(const char* str)	赋值 str。
	string& operator=(char c)	赋值 c。
	iterator begin()	求字符串的头指针。
	const_iterator begin() const	
	iterator end()	求字符串的尾指针。
	const_iterator end() const	
	size_t size() const	求被保存的字符串的长度。
	size_t length() const	
	size_t max_size() const	求被确保的区域的容量。
	void resize(size_t n, char c)	将可保存的字符串长度更改为 n。
	void resize(size_t n)	将可保存的字符串长度更改为 n。
	size_t capacity() const	求被确保的区域的容量。
	void reserve(size_t res_arg = 0)	重新分配区域。
	void clear()	清除被保存的字符串。
	bool empty() const	检查被保存的字符串长度是否为 0。
	const char& operator[](size_t pos) const	参照 s_ptr[pos]。
	char& operator[](size_t pos)	
	const char& at(size_t pos) const	
	char& at(size_t pos)	

种类	定义名	说明
函数	string& operator+=(const string& str)	追加 str 字符串。
	string& operator+=(const char* str)	追加 str 字符串。
	string& operator+=(char c)	追加 c 字符。
	string& append(const string& str)	追加 str 字符串。
	string& append(const char* str)	
	string& append(const string& str, size_t pos, size_t n)	给目标位置 pos 追加 str 字符串的 n 个字符。
	string& append(const char* str, size_t n)	追加字符串 str 的 n 个字符。
	string& append(size_t n, char c)	追加 n 个字符 c。
	string& assign(const string& str)	赋值 str 字符串。
	string& assign(const char* str)	
	string& assign(const string& str, size_t pos, size_t n)	给位置 pos 赋值字符串 str 的 n 个字符。
	string& assign(const char* str, size_t n)	赋值字符串 str 的 n 个字符。
	string& assign(size_t n, char c)	赋值 n 个字符 c。
	string& insert(size_t pos1, const string& str)	将 str 字符串插入到位置 pos1。
	string& insert(size_t pos1, const string& str, size_t pos2, size_t n)	从 str 字符串的位置 pos2 将 n 个字符插入到位置 pos1。
	string& insert(size_t pos, const char* str, size_t n)	将字符串 str 的 n 个字符插入到 pos 位置。
	string& insert(size_t pos, const char* str)	将字符串 str 插入到 pos 位置。
	string& insert(size_t pos, size_t n, char c)	将 n 个字符 c 的字符串插入到位置 pos。
	iterator insert(iterator p, char c = char())	将字符 c 插入到 p 指向的字符串的前面。
	void insert(iterator p, size_t n, char c)	将 n 个字符 c 插入到 p 指向的字符的前面。
	string& erase(size_t pos = 0, size_t n = npos)	从位置 pos 删除 n 个字符。
	iterator erase(iterator position)	删除由 position 参照的字符。
	iterator erase(iterator first, iterator last)	删除范围[first, last]内的字符。
	string& replace(size_t pos1, size_t n1, const string& str)	用 str 字符串替换从位置 pos1 开始的 n1 个字符的字符串。
	string& replace(size_t pos1, size_t n1, const char* str)	

种类	定义名	说明
函数	string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	用从 str 的位置 pos2 开始的 n2 个字符的字符串替换从位置 pos1 开始的 n1 个字符的字符串。
	string& replace(size_t pos, size_t n1, const char* str, size_t n2)	用 str 的 n2 个字符的字符串替换从位置 pos 开始的 n1 个字符的字符串。
	string& replace(size_t pos, size_t n1, size_t n2, char c)	用 n2 个字符 c 替换从位置 pos 开始的 n1 个字符的字符串。
	string& replace(iterator i1, iterator i2, const string& str)	用 str 字符串替换从位置 i1 到 i2 的字符串。
	string& replace(iterator i1, iterator i2, const char* str)	
	string& replace(iterator i1, iterator i2, const char* str, size_t n)	用 str 字符串的 n 个字符替换从位置 i1 到 i2 的字符串。
	string& replace(iterator i1, iterator i2, size_t n, char c)	用 n 个字符 c 替换从位置 i1 到 i2 的字符串。
	size_t copy(char* str, size_t n, size_t pos = 0) const	将字符串 str 的 n 个字符的字符串复制到位置 pos。
	void swap(string& str)	与 str 的字符串交换。
	const char* c_str() const	参照指向保存字符串的区域的指针。
	const char* data() const	
	size_t find(const string& str, size_t pos = 0) const	检索在位置 pos 以后与 str 的字符串相同的字符串最初出现的位置。
	size_t find(const char* str, size_t pos = 0) const	
	size_t find(const char* str, size_t pos, size_t n) const	检索在位置 pos 以后与 str 的 n 个字符相同的字符串最初出现的位置。

种类	定义名	说明
函数	size_t find(char c, size_t pos = 0) const	检索在位置pos以后字符c最初出现的位置。
	size_t rfind(const string& str, size_t pos = npos) const	检索在位置pos以前与str的字符串相同的字符串最后出现的位置。
	size_t rfind(const char* str, size_t pos = npos) const	
	size_t rfind(const char* str, size_t pos, size_t n) const	检索在位置pos以前与str的n个字符相同的字符串最后出现的位置。
	size_t rfind(char c, size_t pos = npos) const	检索在位置pos以前字符c最后出现的位置。
	size_t find_first_of(const string& str, size_t pos = 0) const	检索在位置pos以后包含在字符串str中的任意字符最初出现的位置。
	size_t find_first_of(const char* str, size_t pos = 0) const	
	size_t find_first_of(const char* str, size_t pos, size_t n) const	检索在位置pos以后包含在字符串str的n个字符中的任意字符最初出现的位置。
	size_t find_first_of(char c, size_t pos = 0) const	检索在位置pos以后字符c最初出现的位置。
	size_t find_last_of(const string& str, size_t pos = npos) const	检索在位置pos以前包含在字符串str中的任意字符最后出现的位置。
	size_t find_last_of(const char* str, size_t pos = npos) const	
	size_t find_last_of(const char* str, size_t pos, size_t n) const	检索在位置pos以前包含在字符串str的n个字符中的任意字符最后出现的位置。
	size_t find_last_of(char c, size_t pos = npos) const	检索在位置pos以前字符c最后出现的位置。
	size_t find_first_not_of(const string& str, size_t pos = 0) const	检索在位置pos以后与str中的任意字符不同的字符最初出现的位置。
	size_t find_first_not_of(const char* str, size_t pos = 0) const	
	size_t find_first_not_of(const char* str, size_t pos, size_t n)	检索在位置pos以后与从str起始位置到n个字符之间的任意字符不同的字符最初出现的位置。
	size_t find_first_not_of(char c, size_t pos = 0) const	检索在位置pos以后与字符c不同的字符最初出现的位置。

种类	定义名	说明
函数	size_t find_last_not_of(const string& str, size_t pos = npos) const	检索在位置 pos 以前与 str 中的任意字符不同的字符最后出现的位置。
	size_t find_last_not_of(const char* str, size_t pos = npos) const	
	size_t find_last_not_of(const char* str, size_t pos, size_t n) const	检索在位置 pos 以前与从 str 起始位置到 n 个字符之间的任意字符不同的字符最后出现的位置。
	size_t find_last_not_of(char c, size_t pos = npos) const	检索在位置 pos 以前与字符 c 不同的字符最后出现的位置。
	string substr(size_t pos = 0, size_t n = npos) const	对被保存的字符串，生成具有范围[pos,n]的字符串的目标。
	int compare(const string& str) const	将字符串和 str 字符串进行比较。
	int compare(size_t pos1, size_t n1, const string& str) const	将从位置 pos1 开始的 n1 个字符的字符串和 str 进行比较。
	int compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	将从位置 pos1 开始的 n1 个字符的字符串和从 str 的位置 pos2 开始的 n2 个字符的字符串进行比较。
	int compare(const char* str) const	和 str 比较。
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	将从位置 pos1 开始的 n1 个字符的字符串和 str 的 n2 个字符的字符串进行比较。

string::string(void)
进行以下设定：
s_ptr = 0;
s_len = 0;
s_res = 1;

string::string(const string& str, size_t pos = 0, size_t n = npos)
复制 str。但是，s_len 为 n 和 s_len 中较小一方的值。

string::string(const char* str, size_t n)
进行以下设定：
s_ptr = str;
s_len = n;
s_res = n+1;

string::string(const char* str)
进行以下设定：
s_ptr = str;
s_len = str 的字符串长度;
s_res = str 的字符串长度+1;

string::string(size_t n, char c)
进行以下设定：
s_ptr =n 个字符 c 的字符串
s_len = n
s_res = n+1;

string::~string()
类 string 的析构函数。
释放保存字符串的区域。

string& string::operator=(const string& str)
赋值 str 的数据。
返回值为*this。

string& string::operator=(const char* str)
从 str 生成 string 目标，赋值该数据。
返回值为*this。

string& string::operator=(char c)
从 c 生成 string 目标，赋值该数据。
返回值为*this。

string::iterator string::begin()
string::const_iterator string::begin() const
求字符串的头指针。
返回值为字符串的头指针。

string::iterator string::end()
string::const_iterator string::end() const
求字符串的尾指针。
返回值为字符串的尾指针。

size_t string::size() const
size_t string::length() const
求被保存的字符串的长度。
返回值为被保存的字符串的长度。

size_t string::max_size() const
求被确保的区域的容量。
返回值为被确保的区域的容量。

void string::resize(size_t n, char c)
将目标能保存的字符串长度更改为 n。
n<=size()时，替换成长度为 n 的原字符串。
n>size()时，替换成在原字符串的后面填入 c 的字符串（使其长度为 n）。
需要 n<=max_size()。
n>max_size()时，作为 n=max_size()计算。

void string::resize(size_t n)
将目标能保存的字符串长度更改为 n。
n<=size()时，替换成长度为 n 的原字符串。
需要 n<=max_size()。

size_t string::capacity() const
求被确保的区域的容量。
返回值为被确保的区域的容量。

void string::reserve(size_t res_arg = 0)
重新分配存储区。
在 reserve()后，capacity()大于或者等于 reserve()的参数。
如果重新分配，就将全部参照、指针和参照此数列中的元素的 iterator 设定为无效。

void string::clear()
清除被保存的字符串。

bool string::empty() const
检查被保存的字符串长度是否为 0。
返回值如下：

被保存的字符串长度为 0 时	：true
被保存的字符串长度不为 0 时	：false

const char& string::operator[] (size_t pos) const
char& string::operator[](size_t pos)
const char& string::at(size_t pos) const
char& string::at(size_t pos)
请参照 s_ptr[pos]。
返回值如下：
n< s_len 时　：s_ptr [pos]
n>= s_len 时：’\0’

string& string::operator+=(const string& str)
追加 str 保存的字符串。
返回值为*this。

string& string::operator+=(const char* str)
从 str 生成 string 目标，追加该字符串。
返回值为*this。

string& string::operator+=(char c)
从 c 生成 string 目标，追加该字符串。
返回值为*this。

string& string::append(const string& str)
string& string::append(const char* str)
将 str 的字符串追加到目标。
返回值为*this。

string& string::append(const string& str, size_t pos, size_t n)
给目标的位置 pos 追加 str 字符串的 n 个字符。
返回值为*this。

string& string::append(const char* str, size_t n)
追加字符串 str 的 n 个字符。
返回值为*this。

string& string::append(size_t n, char c)
追加 n 个字符 c。
返回值为*this。

string& string::assign(const string& str)
string& string::assign(const char* str)
赋值 str 字符串。
返回值为*this。

string& string::assign(const string& str, size_t pos, size_t n)
给位置 pos 赋值字符串 str 的 n 个字符。
返回值为*this。

string& string::assign(const char* str, size_t n)
　赋值字符串 str 的 n 个字符。
　返回值为*this。

string& string::assign(size_t n, char c)
　赋值 n 个字符 c。
　返回值为*this。

string& string::insert(size_t pos1, const string& str)
　将 str 字符串插入到位置 pos1。
　返回值为*this。

string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)
　从 str 字符串的 pos2 位置将 n 个字符插入到位置 pos1。
　返回值为*this。

string& string::insert(size_t pos, const char* str, size_t n)
　将字符串 str 的 n 个字符插入到 pos 位置。
　返回值为*this。

string& string::insert(size_t pos, const char* str)
　将字符串 str 插入到 pos 位置。
　返回值为*this。

string& string::insert(size_t pos, size_t n, char c)
　将 n 个字符 c 的字符串插入到位置 pos。
　返回值为*this。

string::iterator string::insert(iterator p, char c = char())
　在 p 指定的字符串前插入字符 c。
　返回值为插入的字符。

void string::insert(iterator p, size_t n, char c)
　在 p 指定的字符前插入 n 个字符 c。

string& string::erase(size_t pos = 0, size_t n = npos)
　从位置 pos 删除 n 个字符。
　返回值为*this。

iterator string::erase(iterator position)
　删除由 position 参照的字符。
　返回值如下：
　　　有删除元素的下一个 iterator 时：删除元素的下一个 iterator
　　　无删除元素的下一个 iterator 时：end()

iterator string::erase(iterator first, iterator last)
删除范围[first, last]中的字符。
返回值如下：
有 last 的下一个 iterator 时：last 的下一个 iterator
无 last 的下一个 iterator 时：end()

string& string::replace(size_t pos1, size_t n1, const string& str)
string& string::replace(size_t pos1, size_t n1, const char* str)
用 str 字符串替换从位置 pos1 开始的 n1 个字符的字符串。
返回值为*this。

string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)
用从 str 的位置 pos2 开始的 n2 个字符的字符串替换从位置 pos1 开始的 n1 个字符的字符串。
返回值为*this。

string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)
用 str 的 n2 个字符的字符串替换从位置 pos 开始的 n1 个字符的字符串。
返回值为*this。

string& string::replace(size_t pos, size_t n1, size_t n2, char c)
用 n2 个的字符 c 替换从位置 pos 开始的 n1 个字符的字符串。
返回值为*this。

string& string::replace(iterator i1, iterator i2, const string& str)
string& string::replace(iterator i1, iterator i2, const char* str)
用 str 字符串替换从位置 i1 到 i2 的字符串。
返回值为*this。

string& string::replace(iterator i1, iterator i2, const char* str, size_t n)
用 str 的 n 个字符的字符串替换从位置 i1 到 i2 的字符串。
返回值为*this。

string& string::replace(iterator i1, iterator i2, size_t n, char c)
用 n 个字符 c 替换从位置 i1 到 i2 的字符串。
返回值为*this。

size_t string::copy(char* str, size_t n, size_t pos = 0) const
将字符串 str 的 n 个字符的字符串复制到位置 pos。
返回值为 rlen。

void string::swap(string& str)
与 str 的字符串交换。

const char* string::c_str() const
const char* string::data() const
参照指向保存字符串的区域的指针。
返回值为 s_ptr。

size_t string::find(const string& str, size_t pos = 0) const
size_t string::find(const char* str, size_t pos = 0) const
检索在位置 pos 以后与 str 字符串相同的字符串最初出现的位置。
返回值为字符串偏移量。

size_t string::find(const char* str, size_t pos, size_t n) const
检索在位置 pos 以后与 str 的 n 个字符相同的字符串最初出现的位置。
返回值为字符串的偏移量。

size_t string::find(char c, size_t pos = 0) const
检索在位置 pos 以后字符 c 最初出现的位置。
返回值为字符串的偏移量。

size_t string::rfind(const string& str, size_t pos = npos) const
size_t string::rfind(const char* str, size_t pos = npos) const
检索在位置 pos 以前与 str 字符串相同的字符串最后出现的位置。
返回值为字符串的偏移量。

size_t string::rfind(const char* str, size_t pos, size_t n) const
检索在位置 pos 以前与字符串 str 的 n 个字符相同的字符串最后出现的位置。
返回值为字符串的偏移量。

size_t string::rfind(char c, size_t pos = npos) const
检索在位置 pos 以前字符 c 最后出现的位置。
返回值为字符串的偏移量。

size_t string::find_first_of(const string& str, size_t pos = 0) const
size_t string::find_first_of(const char* str, size_t pos = 0) const
检索在位置 pos 以后包含在字符串 str 中的任意字符最初出现的位置。
返回值为字符串的偏移量。

size_t string::find_first_of(const char* str, size_t pos, size_t n) const
检索在位置 pos 以后包含在字符串 str 的 n 个字符中的任意字符最初出现的位置。
返回值为字符串的偏移量。

size_t string::find_first_of(char c, size_t pos = 0) const
检索在位置 pos 以后字符 c 最初出现的位置。
返回值为字符串的偏移量。

size_t string::find_last_of(const string& str, size_t pos = npos) const
size_t string::find_last_of(const char* str, size_t pos = npos) const
检索在位置 pos 以前包含在字符串 str 中的任意字符最后出现的位置。
返回值为字符串的偏移量。

size_t string::find_last_of(const char* str, size_t pos, size_t n) const
　检索在位置 pos 以前包含在字符串 str 的 n 个字符中的任意字符最后出现的位置。
　返回值为字符串的偏移量。

size_t string::find_last_of(char c, size_t pos = npos) const
　检索在位置 pos 以前字符 c 最后出现的位置。
　返回值为字符串的偏移量。

size_t string::find_first_not_of(const string& str, size_t pos = 0) const
size_t string::find_first_not_of(const char* str, size_t pos = 0) const
　检索在位置 pos 以后与 str 中的任意字符不同的字符最初出现的位置。
　返回值为字符串的偏移量。

size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const
　检索在位置 pos 以后与从 str 起始位置到 n 个字符之间的任意字符不同的字符最初出现的位置。
　返回值为字符串的偏移量。

size_t string::find_first_not_of(char c, size_t pos = 0) const
　检索在位置 pos 以后与字符 c 不同的字符最初出现的位置。
　返回值为字符串的偏移量。

size_t string::find_last_not_of (const string& str, size_t pos = npos) const
size_t string::find_last_not_of(const char* str, size_t pos = npos) const
　检索在位置 pos 以前与 str 中的任意字符不同的字符最后出现的位置。
　返回值为字符串的偏移量。

size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const
　检索在位置 pos 以前与从 str 起始位置到 n 个字符之间的任意字符不同的字符最后出现的位置。
　返回值为字符串的偏移量。

size_t string::find_last_not_of(char c, size_t pos = npos) const
　检索在位置 pos 以前与字符 c 不同的字符最后出现的位置。
　返回值为字符串的偏移量。

string string::substr(size_t pos = 0, size_t n = npos) const
　对被保存的字符串，生成具有范围[pos,n]的字符串的目标。
　返回值为具有范围[pos,n]的字符串的目标。

int string::compare(const string& str) const
　将字符串和 str 字符串进行比较。
　返回值如下：
　　　字符串相同时：0
　　　字符串不同时：this->s_len>str.s_len 时为 1
　　　　　　　　　　this->s_len < str.s_len 时为-1

int string::compare(size_t pos1, size_t n1, const string& str) const

将从位置 pos1 开始的 n1 个字符的字符串和 str 进行比较。

返回值如下：

字符串相同时：0
字符串不同时：this->s_len>str.s_len 时为 1
this->s_len < str.s_len 时为-1

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

将从位置 pos1 开始的 n1 个字符的字符串和从 str 的位置 pos2 开始的 n2 个字符的字符串进行比较。

返回值如下：

字符串相同时：0
字符串不同时：this->s_len>str.s_len 时为 1
this->s_len < str.s_len 时为-1

int string::compare(const char* str) const

和 str 比较。

返回值如下：

字符串相同时：0
字符串不同时：this->s_len>str.s_len 时为 1
this->s_len < str.s_len 时为-1

int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const

将从位置 pos1 开始的 n1 个字符的字符串和 str 的 n2 个字符的字符串进行比较。

返回值如下：

字符串相同时：0
字符串不同时：this->s_len>str.s_len 时为 1
this->s_len < str.s_len 时为-1

（b）string 类操纵符

种类	定义名	说明
函数	string operator+(const string& lhs, const string& rhs)	将 rhs 的字符串（或者字符）追加到 lhs 的字符串（或者字符），在生成目标后保存该字符串。
	string operator+(const char* lhs, const string& rhs)	
	string operator+(char lhs, const string& rhs)	
	string operator+(const string& lhs, const char* rhs)	
	string operator+(const string& lhs, char rhs)	
	bool operator==(const string& lhs, const string& rhs)	将 lhs 的字符串和 rhs 的字符串进行比较。
	bool operator==(const char* lhs, const string& rhs)	
	bool operator==(const string& lhs, const char* rhs)	
	bool operator!=(const string& lhs, const string& rhs)	将 lhs 的字符串和 rhs 的字符串进行比较。
	bool operator!=(const char* lhs, const string& rhs)	
	bool operator!=(const string& lhs, const char* rhs)	
	bool operator<(const string& lhs, const string& rhs)	将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
	bool operator<(const char* lhs, const string& rhs)	
	bool operator<(const string& lhs, const char* rhs)	
	bool operator>(const string& lhs, const string& rhs)	将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
	bool operator>(const char* lhs, const string& rhs)	
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	
	bool operator>=(const string& lhs, const string& rhs)	将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
	bool operator>=(const char* lhs, const string& rhs)	
	bool operator>=(const string& lhs, const char* rhs)	
	void swap(string& lhs, string& rhs)	交换 lhs 的字符串和 rhs 的字符串。
	istream& operator>>(istream& is, string& str)	将字符串抽出到 str。
	ostream& operator<<(ostream& os, const string& str)	插入字符串。
	istream& getline(istream& is, string& str, char delim)	从 is 抽出字符串，附加到 str。如果在中途检测出字符‘delim’，就结束输入。
	istream& getline(istream& is, string& str)	从 is 抽出字符串，附加到 str。如果在中途检测出换行字符，就结束输入。

string operator+(const string& lhs, const string& rhs)
string operator+(const char* lhs, const string& rhs)
string operator+(char lhs, const string& rhs)
string operator+(const string& lhs, const char* rhs)
string operator+(const string& lhs, char rhs)

将 rhs 的字符串（或者字符）追加到 lhs 的字符串（或者字符），在生成目标后保存该字符串。
返回值为保存被连接的字符串的目标。

bool operator==(const string& lhs, const string& rhs)
bool operator==(const char* lhs, const string& rhs)
bool operator==(const string& lhs, const char* rhs)

将 lhs 的字符串和 rhs 的字符串进行比较。
返回值如下：
字符串相同时：true
字符串不同时：false

bool operator!=(const string& lhs, const string& rhs)
bool operator!=(const char* lhs, const string& rhs)
bool operator!=(const string& lhs, const char* rhs)

将 lhs 的字符串和 rhs 的字符串进行比较。
返回值如下：
字符串相同时：false
字符串不同时：true

bool operator<(const string& lhs, const string& rhs)
bool operator<(const char* lhs, const string& rhs)
bool operator<(const string& lhs, const char* rhs)

将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
返回值如下：
lhs.s_len < rhs.s_len 时　：true
lhs.s_len >= rhs.s_len 时：false

bool operator>(const string& lhs, const string& rhs)
bool operator>(const char* lhs, const string& rhs)
bool operator>(const string& lhs, const char* rhs)

将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
返回值如下：
lhs.s_len > rhs.s_len 时　：true
lhs.s_len <= rhs.s_len 时：false

bool operator<=(const string& lhs, const string& rhs)
bool operator<=(const char* lhs, const string& rhs)
bool operator<=(const string& lhs, const char* rhs)

将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
返回值如下：
lhs.s_len <= rhs.s_len 时：true
lhs.s_len > rhs.s_len 时　：false

bool operator>=(const string& lhs, const string& rhs)
bool operator>=(const char* lhs, const string& rhs)
bool operator>=(const string& lhs, const char* rhs)
将 lhs 的字符串长度和 rhs 的字符串长度进行比较。
返回值如下：
lhs.s_len >= rhs.s_len 时：true
lhs.s_len < rhs.s_len 时 ：false

void swap(string& lhs, string& rhs)
交换 lhs 字符串和 rhs 字符串。

istream& operator>>(istream& is, string& str)
将字符串抽出到 str。
返回值为 is。

ostream& operator<<(ostream& os, const string& str)
插入字符串。
返回值为 os。

istream& getline(istream& is, string& str, char delim)
从 is 抽出字符串，附加到 str。
如果在中途检测出字符‘delim’，就结束输入。
返回值为 is。

istream& getline (istream& is, string& str)
从 is 抽出字符串，附加到 str。
如果在中途检测出换行字符，就结束输入。
返回值为 is。

10.3.3　可重入库

可重入库一览表如表10.42所示。在表中，用△表示的函数设定errno变量，所以如果在程序中不参照errno，就可重入。

另外，如果想利用信标提高可重入性，就必须用标准库生成工具指定**reent**选项。除rand、srand函数以外，全部库可重入。但是，不能保证对strtok函数的相同字符串的连续调用。另外，请参照“9.2.2（7）（b）低级接口例程的详细内容”和“9.2.2（7）（d）用于可重入库的低级接口例程的例子”。

可重入栏　○：可重入　×：不可重入　△：设定errno

表10.42　可重入库一览表

标准 include 文件		函数名		可重入
1	stddef.h	1	offsetof	○
2	assert.h	2	assert	×
3	ctype.h	3	isalnum	○
		4	isalpha	○
		5	iscntrl	○
		6	isdigit	○
		7	isgraph	○
		8	islower	○
		9	isprint	○
		10	ispunct	○
		11	isspace	○
		12	isupper	○
		13	isxdigit	○
		14	tolower	○
		15	toupper	○
4	math.h	16	acos	△
		17	asin	△
		18	atan	△
		19	atan2	△
		20	cos	△
		21	sin	△
		22	tan	△
		23	cosh	△
		24	sinh	△
		25	tanh	△
		26	exp	△
		27	frexp	△
		28	ldexp	△
		29	log	△
		30	log10	△
		31	modf	△
		32	pow	△
		33	sqrt	△
		34	ceil	△
		35	fabs	△
		36	floor	△
		37	fmod	△
5	mathf.h	38	acosf	△
		39	asinf	△
		40	atanf	△
		41	atan2f	△
		42	cosf	△
		43	sinf	△
		44	tanf	△
		45	coshf	△
		46	sinhf	△
		47	tanhf	△
		48	expf	△
		49	frexpf	△
		50	ldexpf	△
		51	logf	△
		52	log10f	△
		53	modff	△
		54	powf	△
		55	sqrtf	△
		56	ceilf	△
		57	fabsf	△
		58	floorf	△
		59	fmodf	△
6	setjmp.h	60	setjmp	○
		61	longjmp	○
7	stdarg.h	62	va_start	○
		63	va_arg	○
		64	va_end	○
8	stdio.h	65	fclose	×
		66	fflush	×
		67	fopen	×
		68	freopen	×
		69	setbuf	×
		70	setvbuf	×
		71	fprintf	×
		72	fscanf	×
		73	printf	×
		74	scanf	×

标准 include 文件		函数名		可重入
8	stdio.h	75	sprintf	△
		76	sscanf	△
		77	vfprintf	×
		78	vprintf	×
		79	vsprintf	△
		80	fgetc	×
		81	fgets	×
		82	fputc	×
		83	fputs	×
		84	getc	×
		85	getchar	×
		86	gets	×
		87	putc	×
		88	putchar	×
		89	puts	×
		90	ungetc	×
		91	fread	×
		92	fwrite	×
		93	fseek	×
		94	ftell	×
		95	rewind	×
		96	clearerr	×
		97	feof	×
		98	ferror	×
		99	perror	×
9	stdlib.h	100	atof	△
		101	atoi	△
		102	atol	△
		103	strtod	△
		104	strtol	△
		105	rand	×
9	stdlib.h	106	srand	×
		107	calloc	×
		108	free	×
		109	malloc	×
		110	realloc	×
		111	bsearch	○
		112	qsort	○
		113	abs	○
		114	div	△
		115	labs	○
		116	ldiv	△
10	string.	117	memcpy	○
		118	strcpy	○
		119	strncpy	○
		120	strcat	○
		121	strncat	○
		122	memcmp	○
		123	strcmp	○
		124	strncmp	○
		125	memchr	○
		126	strchr	○
		127	strcspn	○
		128	strpbrk	○
		129	strrchr	○
		130	strspn	○
		131	strstr	○
		132	strtok	×
		133	memset	○
		134	strerror	○
		135	strlen	○
		136	memmove	○

10.3.4　未支持库

在 C 语言规格定义的库中，本编译程序不支持的库如表 10.43 所示。

表 10.43　不支持的库

头文件		库名
1	locale.h*1	setlocale、localeconv
2	signal.h*1	signal、raise
3	stdio.h	remove、rename、tmpfile、tmpnam、fgetpos、fsetpos
4	stdlib.h	strtoul、abort、atexit、exit、getenv、system、 mblen、mbtowc、wctomb、mbstowcs、wcstombs
5	string.h	strcoll、strxfrm
6	time.h*1	clock、difftime、mktime、time、asctime、ctime、gmtime、localtime、 strftime

【注】*1　不支持头文件。

第 11 章　汇编程序的语言规格

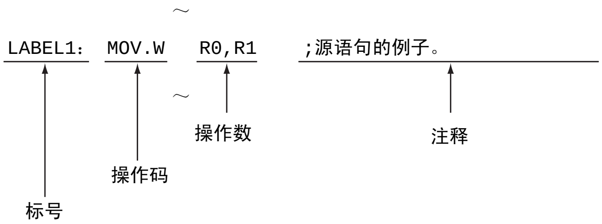
11.1　程序的元素

11.1.1　源语句

（1）源语句的结构

源语句的结构如下所示：
[标号][△操作 [△操作数]][注释]

例：



```
LABEL1： MOV.W  R0,R1   ;源语句的例子。
```

（a）标号

是为了给源语句命名的符号或者局部符号。
符号是程序员定义的名称。

（b）操作码

是执行指令、汇编程序控制指令和预处理器控制语句的助记符（名称）。
执行指令是单片机的指令。
汇编程序控制指令是汇编程序解释后的执行指令。
预处理器控制语句为有关文件包含功能、条件汇编功能、结构化汇编功能和宏功能的语句。

（c）操作数

是操作的执行对象。操作数的个数和种类由操作码决定，有些操作码不需要操作数。

（d）注释

是为了容易理解程序的注释。

（2）源语句的记述方法

用 ASCII 字符记述源语句。但是，在字符串或者注释中可以记述假名/汉字（shift JIS 码和 EUC 码）、LATIN 码。原则上 1 行写 1 条源语句，1 行的最大长度为 8,192 个字符。

（a）标号的记述方法

标号的记述如下：

- 从第 1 列开始记述。

  或者
- 在紧接着标号名之后加冒号（:）。

例：

好的例子：

```
LABEL1                                            ; 从第 1 列开始记述。
  LABEL2:                                         ; 以冒号结束。
```

坏的例子：

```
  LABEL3                                          ;没有从第 1 列开始记述，也没有加冒号，
                                                  ; 所以编汇程序不认为是标号。
```

（b）操作码的记述方法

操作码的记述如下：

- 没有记述标号时：从第 2 列以后开始记述。
- 记述标号时：在与标号之间间隔 1 个以上的空格或者制表符后开始记述。

例：

```
         ADD.W  R0,R1                             ; 没有记述标号时。
LABEL1:  ADD.W  R1,R2                             ; 记述标号时。
```

（c）操作数的记述方法

在与操作之间间隔 1 个以上的空格或者制表符后记述操作数。

例：

```
         ADD.W  R0,R1                             ; ADD 指令的操作数为 2 个。
         SHAL.W R1                                ; SHAL 指令的操作数为 1 个。
```

（d）注释的记述方法

在分号（;）后继续记述。汇编程序将分号到行末的内容认为是注释。

例：

```
         ADD.W R0,R1                              ; R1 加上 R0。
```

（3）跨多行的源语句的记述方法

在以下情况下，为了使程序容易阅读，可将 1 条源语句分为多行进行记述。

- 源语句太长
- 想给每个操作数加注释

必须按照以下（a）～（c）步骤记述跨多行的源语句：
（a）将分开操作数和操作数的逗号（,）作为换行的段落。
（b）在下一行的第 1 列写加号（+）。
（c）在该加号后继续写源语句。
可以在加号后输入空格或者制表符，也能在各行的最后写注释。

例：
（a）

```
          .DATA.W          H'FF00,
  +                        H'FF00,
  +                        H'FFFF
                                              ; 跨 3 行写 1 条源语句的例子。
```

（b）

```
          .DATA.W          H'FF00,            ; 初始值 1
  +                        H'FF00,            ; 初始值 2
  +                        H'FFFF             ; 初始值 3
                                              ; 给每个操作数加注释的例子。
```

11.1.2　保留字

保留字是汇编程序作为具有特别含义的符号而准备的名称。保留字有寄存器名、运算符和定位计数器。寄存器名因各 CPU 种类而不同，请参照各 CPU 的编程手册。保留字不能用作符号。

- 寄存器名
  ER0～ER7、E0～E7、R0～R7、R0H～R7H、R0L～R7L、SP *、CCR、EXR、MACH、MACL、PC、SBR、VBR
- 运算符
  STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD、
- 定位计数器
  $

【注】 * ER7（H8SX、H8S/2600、H8S/2000、H8/300H）或者 R7（H8/300、H8/300L）和 SP 表示相同的寄存器。

11.1.3　符号

（1）符号的作用

符号是程序员定义的名称，有以下作用：

- 地址符号　　：表示数据的保存位置、转移目标等的地址。
- 常数符号　　：表示常数。
- 位数据名　　：表示位操作指令对象的存储器中的 1 位数据。
- 寄存器别名　：表示通用寄存器。
- 段名　　　　：表示段的名称。

符号的使用例如下：

例：

```
（a）  ～
         BRA   SUB1               ; BRA 是转移指令。
       ～                         ; SUB1 表示转移目标的地址符号。
   SUB1:
       ～
```

```
（b）  ～
   MAX:     .EQU     100          ; .EQU 是给符号设定值的汇编程序控制指令。
            MOV.B    #MAX,R0L     ; MAX 表示值 100。
       ～
```

```
（c）  ～
   BSYM:    .BEQU    1,SYM        ; .BEQU 是给位数据设定值的汇编程序控制指令。
            BLD      BSYM         ; BSYM 表示 SYM 的位 1。
   SYM:     .RES.B   1
       ～
```

```
（d）  ～
   MIN:     .REG     R0           ; .REG 是定义寄存器别名的汇编程序控制指令。
            MOV.W    #100,MIN     ; MIN 为 R0 的别名。
       ～
```

```
（e）  ～
         .SECTION   CD,CODE,ALIGN=2
       ～                         ; .SECTION 是声明段的汇编程序控制指令。
                                  ; CD 为此段的名称。
```

（2）符号的命名方法

（a）能用于符号的字符

能使用以下的 ASCII 字符：

- 英文大小写字母（A～Z、a～z）
- 数字（0～9）
- 下划线（_）
- 美元符号（$）

汇编程序区别符号中的英文大小写字母。

（b）头字符

限以下任一字符：

- 英文大小写字母（A～Z、a～z）
- 下划线（_）
- 美元符号（$）

【注】 1 个美元符号（$）的字符是表示定位计数器的保留字。

（c）最大字符数

没有限制。

（d）不能用作符号的名称

以下的名称不能用作符号：

(i) 保留字

- 寄存器助记符（ER0～ER7、E0～E7、R0～R7、R0H～R7H、R0L～R7L、SP、CCR、EXR、MACH、MACL、PC、SBR、VBR）
- 运算符（STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD）
- 定位计数器（$）

(ii) 汇编程序生成符号

- 内部符号*
  _ $$*mmmmm*　　（*m* 是数字（0～F））
- 结构化汇编生成符号
  _$I*nnnnn*
  _$S*nnnnn*
  _$F*nnnnn*
  _$W*nnnnn*
  _$R*nnnnn*　　（*n* 是数字（0～9））

【注】*　内部符号是汇编程序的内部处理所需的符号。内部符号不输出到汇编列表和目标模块。

（e）符号的定义和参照

符号的定义通过标号的记述进行，在参照时记述为操作数。作为例外，将 SECTION 控制指令和.MACRO 控制指令定义的符号记述为操作数。另外，将 MACRO 控制指令定义的符号（宏名）记述为操作后进行参照（宏调用）。

在参照符号时，有时参照先于定义出现，这样的参照称为前方参照。必须注意：通常可进行这样的参照，但是也有不允许参照的情况。

由多个源文件构成程序时，需要参照文件间的符号。能从其它源文件参照定义符号的定义称为外部定义。反之，参照其它源文件的定义符号的参照称为外部参照。

外部定义由.EXPORT、.GLOBAL、.BEXPORT 各控制指令声明。

外部参照由.IMPORT、.GLOBAL、.BIMPORT 各控制指令声明。

必须注意：外部参照和前方参照相同，也有不允许参照的情况。

11.1.4 常数

（1）整数常数

整数常数用基数表现。基数是表示该整数常数为几进制数的记述。

- 2 进制数 …… 用基数 B'*和 2 进制表现的数值记述
- 8 进制数 …… 用基数 Q'*和 8 进制表现的数值记述
- 10 进制数 …… 用基数 D'*和 10 进制表现的数值记述
- 16 进制数 …… 用基数 H'*和 16 进制表现的数值记述

【注】* B'：为 BINARY（2 进制）。
Q'：为 OCTAL（8 进制）。因为 O 容易和数字的零混淆，所以使用 Q。
D'：为 DECIMAL（10 进制）。
H'：为 HEXADECIMAL（16 进制）。

汇编程序不区别基数的英文大小写字母。基数和数值之间不能有空格，也可以省略基数。没有基数的整数常数通常作为 10 进制数处理（能通过.RADIX 汇编程序控制指令指定为几进制数）。

例：

```
    .DATA.B B'10001000                              ;
    .DATA.B Q'210                                   ; 这些源语句都表示
    .DATA.B D'136                                   ; 相同的数值。
    .DATA.B H'88                                    ;
```

（2）字符常数

字符常数是将字符码作为值的常数。必须用双引号（"）括起来记述 4 个字节以内的字符，能记述 ASCII 符号、shift JIS 码、EUC 码的假名/汉字或者 LATIN1 码。ASCII 字符能使用 H'09（制表符）、H'20（空格）～H'7E（~）。将双引号用作数据时，必须连续写 2 个。如果记述假名/汉字，必须在 shift JIS 码时指定 sjis 选项、EUC 码时指定 euc 选项、LATIN1 码时指定 latin1 选项。另外，不能混合使用 shift JIS 码、EUC 码和 LATIN1 码。

例：

（a）

```
    .DATA.L "ABC"           ; 和.DATA.L H'00414243 相同。
    .DATA.W "AB"            ; 和.DATA.W H'4142 相同。
    .DATA.B "A"             ; 和.DATA.B H'41 相同。
                                              ; A 的 ASCII 码 … H'41
                                              ; B 的 ASCII 码 … H'42
                                              ; C 的 ASCII 码 … H'43
```

（b）

```
    .DATA.B """"            ; 为 1 个双引号字符的字符常数。
    .DATA.L "汉字"          ; 汉字
```

11.1.5　定位计数器

定位计数器指向分配目标码（将执行指令和数据转换为计算机能识别的格式代码）的地址（定位）。定位计数器的值（以下称为定位计数器值）根据目标码的输出自动变化。另外，也能用汇编程序控制指令故意改变定位计数器值。

例：

```
             ～
.ORG         H'1000          ; 给定位计数器设定 H'1000。
.DATA.W      H'FF            ; 此汇编程序控制指令的目标码长度为 2 个字节。
                             ; 定位计数器值变为 H'1002。
.DATA.W      H'F0            ; 此汇编程序控制指令的目标码长度为 2 个字节。
                             ; 定位计数器值变为 H'1004。
.DATA.W      H'10            ; 此汇编程序控制指令的目标码长度为 2 个字节。
                             ; 定位计数器值变为 H'1006。
                             ; .ORG 是设定定位计数器值的汇编程序控制指令。
                             ; .DATA 是将数据确保到存储器的汇编程序控制指令。
                             ; .W 指定以字（=2 个字节）为单位进行数据处理。
             ～
```

能用美元符号（$）参照定位计数器。

例：

```
LABEL1:      .EQU $          ; 给 LABEL1 符号设定此时的定位计数器值。
                             ; .EQU 是给符号设定值的汇编程序控制指令。
```

11.1.6　表达式

表达式将常数、符号和运算符进行组合，用于执行指令和汇编程序控制指令的操作数。

（1）表达式的元素

表达式由项、运算符和括号构成。

（a）项

项有以下内容：

- 常数
- 定位计数器（$）
- 符号（寄存器别名除外）
- 根据上述的项和运算符得出的运算结果

单独项也是一种表达式。

（b）运算符

运算符一览表如**表 11.1** 所示。

表 11.1　运算符一览表

运算分类	运算符	运算内容	编程方法
算术运算	+	一元加	+ 项
	-	一元减	- 项
	+	加法	项 1 + 项 2
	-	减法	项 1 – 项 2
	*	乘法	项 1 * 项 2
	/	除法	项 1 / 项 2
逻辑运算	~	一元否定	~ 项
	&	逻辑与	项 1 & 项 2
	|	逻辑或	项 1 | 项 2
	~	逻辑异或	项 1 ~ 项 2
移位运算	<<	算术左移	项 1 << 项 2
	>>	算术右移	项 1 >> 项 2
段集合运算	STARTOF	求段集的起始地址	STARTOF 段名
	SIZEOF	以字节为单位求段集的大小	SIZEOF 段名
抽出运算	HIGH	抽出高位字节	HIGH 项
	LOW	抽出低位字节	LOW 项
	HWORD	抽出高位字	HWORD 项
	LWORD	抽出低位字	LWORD 项

【注】　HWORD 和 LWORD 不能用于 H8/300 或者 H8/300L。

（c）括号

能用圆括号()更改运算的优先顺序。

（d）运算的顺序

在 1 个表达式中有多个运算时，由运算符的优先顺序和指定的括号决定运算处理的顺序。汇编程序遵循以下规则进行运算处理：

- 规则 1
  从用括号括起来的运算开始处理。
  有多重括号时，优先内侧括号的运算。
- 规则 2
  从优先顺序高的运算符开始处理。
- 规则 3
  当运算符的优先顺序相同时，按照运算符的结合律的方向进行处理。

运算符的优先顺序和结合律如**表 11.2** 所示。

表 11.2　运算符的优先顺序和结合律

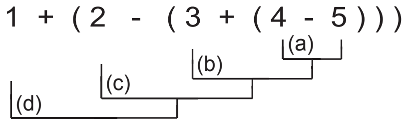
优先顺序		运算符	结合律
1	(高)	+ - ~ STARTOF SIZEOF HIGH LOW HWORD LWORD *1	按从右到左的顺序进行运算处理。
2	↑	* /	按从左到右的顺序进行运算处理。
3	│	+ -	按从左到右的顺序进行运算处理。
4	│	<< >>	按从左到右的顺序进行运算处理。
5	↓	&	按从左到右的顺序进行运算处理。
6	(低)	| ~	按从左到右的顺序进行运算处理。

【注】　相同优先顺序的运算符按照结合律的方向进行运算。
*1　运算符表示一元运算符。

表达式的记述例子如下所示：

例：

(i)

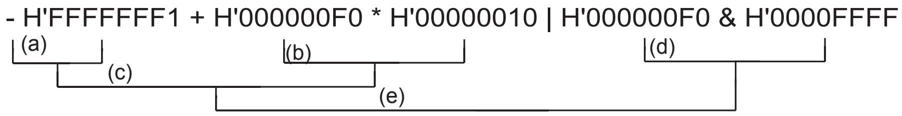


汇编程序按(a)～(d)的顺序计算。

(a)的结果 ……… -1
(b)的结果 ……… 2
(c)的结果 ……… 0
(d)的结果 ……… 1

最后结果为1。

(ii)

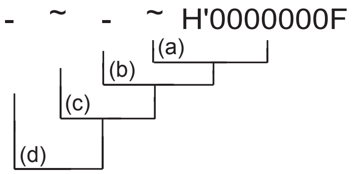


汇编程序按(a)～(e)的顺序计算。

(a)的结果 ……… H'0000000F
(b)的结果 ……… H'00000F00
(c)的结果 ……… H'00000F0F
(d)的结果 ……… H'000000F0
(e)的结果 ……… H'00000FFF

最后结果为H'0000FFF。

(iii)



汇编程序按(a)～(d)的顺序计算。

(a)的结果 ········· H'FFFFFFF0
(b)的结果 ········· H'00000010
(c)的结果 ········· H'FFFFFFEF
(d)的结果 ········· H'00000011

最后结果为H'00000011。

（2）运算的详细内容

（a）STARTOF 运算

求用优化连接编辑程序连接被指定段之后的段起始地址。

（b）SIZEOF 运算

求用优化连接编辑程序连接被指定段之后的段大小。

例：

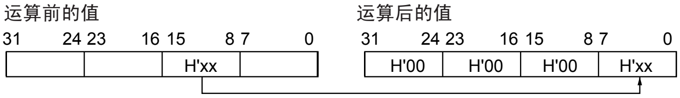
```
           .CPU     2600A
           .SECTION INIT_RAM,DATA,ALIGN=2
           .RES.B   H'100
;
           .SECTION INIT_DATA,DATA,ALIGN=2
INIT_BGN   .DATA.L  STARTOF INIT_RAM                      ; [1]
INIT_END   .DATA.L  STARTOF INIT_RAM + SIZEOF INIT_RAM    ; [2]
;
;
           .SECTION MAIN,CODE,ALIGN=2
INITIAL:
           MOV.L    @INIT_BGN,ER1
           MOV.L    @INIT_END,ER2
           MOV.W    #0,R3
LOOP:
           CMP.L    ER1,ER2
           BEQ      END
           MOV.W    R3,@ER1
           ADDS.L   #1,ER1
           BRA      LOOP
END:
           SLEEP
           .END
```

用零初始化段(INIT_RAM)的数据区。

[1]：求段（INIT_RAM）的起始地址。
[2]：求段（INIT_RAM）的结束地址。

（c）HIGH 运算

抽出 4 个字节值的低位 2 个字节的高位字节。

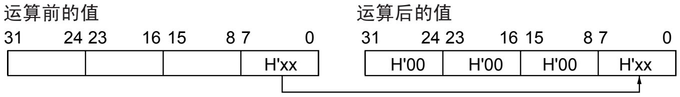


例：

```
LABEL:     .EQU            H'00007FFF
           MOV.W           #HIGH LABEL, R0   ; 将 H'7F 赋值给 R0。
```

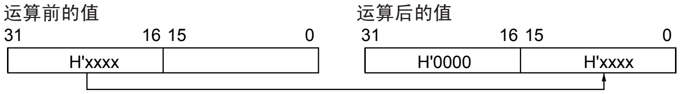
（d）LOW 运算

抽出 4 个字节值的最低位字节。



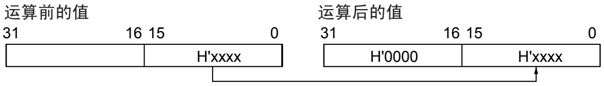
（e）HWORD 运算

抽出 4 个字节值的高位 2 个字节。



（f）LWORD 运算

抽出 4 个字节值的低位 2 个字节。



（3）有关表达式的注意事项

（a）内部处理

汇编程序对表达式的值进行 32 位带符号处理。

不论操作数为多长（8 位/16 位/32 位），运算都以 32 位带符号运算进行。因此，如下例的情况就会出错。

例：

```
           MOV.B   #~H'80:8,R0L
```

汇编程序将 H'80 解释为 H'00000080。因此，~H'80 为 H'FFFFFF7F。

另外，因为 H'FFFFF7F 超出 8 位值的范围，所以出错。

为了避免出错，必须进行如下处理：

```
           MOV.B   #H'7F:8,R0L                 ; 直接记述运算结果的值
           MOV.B   #~H'80&H'FF:8,R0L           ; 使用逻辑与，将低位设定为有效
           MOV.B   #LOW ~H'80:8,R0L            ; 通过低位字节的抽出，将低 8 位
                                               ; 设定为有效。
```

（b）逻辑运算

在逻辑运算中，不能将相对地址或者外部参照符号作为项。

（c）算术运算

对于必须确定值的位置，在乘法/除法运算中不能将相对地址或外部参照符号作为项。

另外，除法运算中不能将 0 作为除数。

例：

```
.IMPORT          SYM
.DATA            SYM/10            ; 正常
.ORG             SYM/10            ; 错误
```

11.1.7　字符串

字符串是一串字符数据。字符串中能使用以下 ASCII 字符：

ASCII码　┌─ H '09　（制表符）
　　　　　└─ H'20　（空格）～H'7E（~）

字符串中的 1 个字符表示具有该字符的 ASCII 码的字节长的数据。另外，能记述 shift JIS 码或者 EUC 码的假名/汉字。如果将假名/汉字作为字符记述，就必须在 shift JIS 码时指定 sjis 选项、EUC 码时就指定 euc 选项。在使用 LATIN1 码时，必须指定 latin1 选项。如果不指定，就取决于主机的日语代码。

必须将字符排列用双引号（"）括起来记述字符串。作为表示数据的字符使用双引号时，必须连续写 2 个双引号。

例：

```
.SDATA           "Hello!"          ; 确保字符串数据 Hello!。
.SDATA           "汇编程序"        ; 确保字符串数据汇编程序。
.SDATA           """Hello!"""      ; 确保字符串数据"Hello!"。
```

;　　　.SDATA 是将字符串数据确保到存储器的汇编程序控制指令。

【注】　字符常数和字符串的不同点：

字符常数是数值，数据的长度为 1 个字节、2 个字节或 4 个字节。字符串不作为数值处理，数据的长度在 1 个字节到 255 个字节之间。

11.1.8　局部标号

（1）局部标号的作用

局部标号是在地址符号之间局部有效的标号。因为局部标号与有效范围外的其它符号不冲突，所以能不必在意其它符号名而进行局部控制。和通常的地址符号一样，能通过标号的记述定义局部标号，能在操作数内参照局部标号。

另外，除了在调试时不能参照局部标号以外，也不能给以下位置指定局部标号。

- 宏名
- 段名
- 目标模块名
- .ASSGINA、.ASSIGNC、.EQU、.BEQU、.ASSIGN、.REG、.DEFINE 标号
- .EXPORT、.IMPORT、.GLOBAL、.BEXPORT、.BIMPORT 的操作数

局部标号的使用例如下所示：
例：

```
LABEL1:                                           ; 局部块 1 的开始
?0001:     CMP.W   R1、R2
           BEQ     ?0002
           BRA     ?0001
?0002:
LABEL2:                                           ; 局部块 2 的开始
?0001:     CMP.W   R1、R2
           BGE     ?0002
           BRA     ?0001
?0002:
LABEL3:
```

（2）局部标号的命名方法

（a）头字符

局部标号是头字符为“?”开始的字符串。

（b）能用于局部标号的字符

局部标号是头字符以外的字符由以下 ASCII 字符构成的字符串。
- 英文大小写字母（A～Z、a～z）
- 数字（0～9）
- 下划线（_）
- 美元符号（$）

汇编程序区别英文大小写字母。

（c）最大字符数

2 个字符以上 16 个字符以内。如果记述 17 个字符以上，就不作为局部符号处理。

（3）局部标号的有效范围

局部标号的有效范围称为局部块。局部块的分离由地址符号或者.SECTION 汇编程序控制指令进行。在该局部块内能参照此局部块内定义的局部标号。属于不同局部块的局部标号即使同名，也被解释为别的标号，不产生错误。

【注】 由.ASSIGNA、ASSIGNC、.EQU、.BEQU、.ASSIGN、.REG 的汇编程序控制指令定义的地址符号不被认为有效范围的分离。

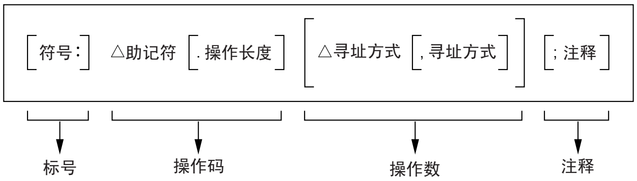
11.2　执行指令

11.2.1　执行指令的概要

执行指令是单片机的指令。

单片机在解读存储器中的执行指令的目标码后执行。

原则上，执行指令的源语句记述如下：



本节说明助记符、操作长度和寻址方式。

（1）助记符

助记符表示执行指令，是表示处理内容的英文省略名。

汇编程序不区别助记符中的英文大小写字母。

（2）操作长度

操作长度为操作数据的单位。

能指定的操作长度因执行指令而不同。

汇编程序不区别操作长度的英文大小写字母。

指定内容	数据长度	
B	字节	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

（3）寻址方式

表示存取对象的数据区和转移目标地址等。
能指定的寻址方式因执行指令而不同。
寻址方式一览表如表 11.3 所示：

表 11.3　寻址方式一览表

寻址方式	名　称	解　　说
ERn、Rn、En RnL、RnH	寄存器直接	寄存器的内容。
@ERn、 @Rn	存储器间接	存储区。 (E)Rn 的值表示区域的起始地址。
@ERn+、 @Rn+、 @ERn-、 @Rn-	后增/递减寄存器间接	存储区。递增*1/递减*2 前的(E)Rn 的值表示区域的起始地址。 单片机先参照(E)Rn，然后递增/递减。
@-ERn、 @-Rn、 @+ERn、 @+Rn	先减/递增寄存器间接	存储区。递减*2/递增后的(E)Rn 的值表示区域的起始地址。 单片机先递减/递增(E)Rn，然后参照。
@(disp,ERn) @(disp,Rn)	带位移量寄存器间接*3	存储区。区域的起始地址为(E)Rn 的值＋位移量（disp）。 (E)Rn 的内容不变。
@(disp,RnL.B) @(disp,Rn.W) @(disp,ERn.L)	带位移量变址寄存器间接	存储区。区域的起始地址为 RnL.B/Rn.W/ERn.L 的值＋位移量（disp）。 (E)Rn 的内容不变。
@abs	绝对地址	存储区。区域的起始地址为指定的绝对地址（abs）。
#imm	立即数	表示常数。
@@abs	存储器间接	存储区。 指定存储器中的操作数，此内容作为转移地址进行转移。
@@vec:7	扩展存储器间接	存储区。 指定存储器中的操作数，作为转移地址转移此内容。
@(disp,PC)	带位移量 PC 相对	存储区。区域的起始地址为 PC 的值＋位移量（disp）。
@(RnL.B,PC) @(Rn.W,PC) @(ERn.L,PC)	带变址寄存器 PC 相对	存储区。区域的起始地址为 PC 的值＋RnL.B/Rn.W/ERn.L 的值。 (E)Rn 的内容不变。
<CCR> <EXR> <MACH> <MACL> <SBR> <VBR>	控制寄存器	<CCR>　：表示 CPU 的内部状态。 <EXR>　：表示跟踪和中断。 <MACH>,<MACL>　：表示乘加运算结果。 <SBR>　：表示短基址值。 <VBR>　：表示向量基值。

【注】 *1　递增
操作长度为字节时加 1、字（2 个字节）时加 2、长字（4 个字节）时加 4。

*2　递减
操作长度为字节时减 1、字（2 个字节）时减 2、长字（4 个字节）时减 4。

*3　位移量
是 2 点之间的距离。本汇编语言以字节为单位表现位移量。

11.2.2　有关执行指令的注意事项

能指定的操作长度根据助记符和寻址方式的组合而不同。

（1）H8SX 系列的执行指令长度和寻址方式

（a）执行指令的长度

H8SX最高模式、高级模式、中级模式和正常模式的执行指令和操作长度的组合如**表11.4**所示。

表 11.4　H8SX 系列的执行指令和操作长度的组合

执行指令	操作长度				
	B	W	L	省略时	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	○	○	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	—	—	—	—	*1
BCLR	○	×	×	B	
BCLR/EQ	○	×	×	B	
BCLR/NE	○	×	×	B	
BFLD	○	×	×	B	
BFST	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BISTZ	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BRA/BC	—	—	—	—	*1
BRA/BS	—	—	—	—	*1
BRA/S	—	—	—	—	*1
BSET	○	×	×	B	
BSET/EQ	○	×	×	B	
BSET/NE	○	×	×	B	
BSR	—	—	—	—	*1
BSR/BC	—	—	—	—	*1
BSR/BS	—	—	—	—	*1
BST	○	×	×	B	
BSTZ	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CLRMAC	—	—	—	—	*1,3
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVS	×	○	○	W	
DIVU	×	○	○	W	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	—	—	—	—	*1
JSR	—	—	—	—	*1
LDC	○	○	○	B/L	*4
LDM	×	×	○	L	
LDMAC	×	×	○	L	*3
MAC	—	—	—	—	*1,3
MOV	○	○	○	B	
MOVA	×	×	○	L	*5
MOVFPE	○	×	×	B	
MOVMD	○	○	○	B	
MOVSD	○	×	×	B	
MOVTPE	○	×	×	B	
MULS	×	○	○	W	
MULS/U	×	×	○	L	*3
MULU	×	○	○	W	
MULU/U	×	×	○	L	*3
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	—	—	—	—	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	—	—	—	—	*1
RTE/L	—	—	—	—	*1
RTS	—	—	—	—	*1
RTS/L	—	—	—	—	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	

执行指令	操作长度 B	 W	 L	 省略时	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	—	—	—	—	*1
STC	○	○	○	B/L	*4
STM	×	×	○	L	
STMAC	×	×	○	L	*3
SUB	○	○	○	B	

执行指令	操作长度 B	 W	 L	 省略时	
SUBS	×	×	○	L	
SUBX	○	○	○	B	
TAS	○	×	×	B	
TRAPA	—	—	—	—	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1 不能指定长度。

*2 最高模式、高级模式和中级模式为 L（长字），正常模式为 W（字）。

*3 在指定有乘法器时指令有效。

*4 在控制寄存器为 CCR 或者 EXR 时，可指定 B（字节）和 W（字），省略时为 B。
在控制寄存器为 SBR 或者 VBR 时，只为 L（长字）。

*5 在输出缩短格式的目标码时，必须指定 C。此时，即使源程序和目的寄存器号不同，汇编程序也不输出错误，而以目的寄存器号输出目标码。

```
MOVA/B.L @(10:16,R1.W),ER1  ；一般格式  目标码 H’78197A99000A
MOVA/B.C @(10:16,R1.W),ER1  ；缩短格式  目标码 H’7A99000A
```

（b）寻址方式

H8SX最高模式、高级模式、中级模式和正常模式的寻址方式如表11.5所示：

表 11.5　H8SX 系列的寻址方式

寻址方式	记述 [*1]
寄存器直接方式	{ ERn | En | Rn | RnH | RnL }
寄存器间接方式	@ERn
后增寄存器间接方式	@ERn+
后减寄存器间接方式	@ERn-
先增寄存器间接方式	@+ERn
先减寄存器间接方式	@-ERn
带位移量寄存器间接方式	@(disp [:{2 | 16 | 32}], ERn)
带位移量变址寄存器间接方式	@(disp [:{16 | 32}], { RnL.B | Rn.W | ERn.L })
绝对寻址方式	@abs[: { 8 | 16 | 24 | 32}]
立即数方式	#imm[: {3 | 4 | 5 | 8 | 16 | 32}]
存储器间接方式	@@abs[:8]
扩展存储器间接方式	@@vec:7
带位移量程序计数器相对方式	d[: { 8 | 16 }]
程序计数器变址相对方式	{ RnL.B | Rn.W | ERn.L }
控制寄存器	CCR、EXR、MACH、MACL、SBR、VBR

【注】*1
n …… 寄存器号（0 ～ 7 [*2]）
disp …… 位移量
abs …… 绝对地址
imm …… 立即数
vec …… 绝对地址

*2 ER7 和 SP（堆栈指针）相同。

（2）H8S/2600 系列的执行指令长度和寻址方式

（a）执行指令的长度

H8S/2600高级模式和正常模式的执行指令和操作长度的组合如表11.6所示：

表 11.6　H8S/2600 系列的执行指令和操作长度的组合

执行指令	操作长度				
	B	W	L	省略时	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	—	—	—	—	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	—	—	—	—	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CLRMAC	—	—	—	—	*1
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	—	—	—	—	*1
JSR	—	—	—	—	*1
LDC	○	○	×	B	
LDM	×	×	○	L	
LDMAC	×	×	○	L	
MAC	—	—	—	—	*1
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	—	—	—	—	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	—	—	—	—	*1
RTS	—	—	—	—	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	—	—	—	—	*1
STC	○	○	×	B	
STM	×	×	○	L	
STMAC	×	×	○	L	
SUB	○	○	○	B	
SUBS	×	×	○	L	
SUBX	○	×	×	B	
TAS	○	×	×	B	
TRAPA	—	—	—	—	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1　不能指定长度。

*2　高级模式为 L（长字），正常模式为 W（字）。

（b）寻址方式

H8S/2600高级模式和正常模式的寻址方式如表11.7所示：

表 11.7　H8S/2600 系列的寻址方式

寻址方式	记述 [*1]
寄存器直接方式	{ ERn | En | Rn | RnH | RnL }
寄存器间接方式	@ERn
后增寄存器间接方式	@ERn+
先减寄存器间接方式	@-ERn
带位移量寄存器间接方式	@(disp[:{16 | 32}], ERn)
绝对寻址方式	@abs[: { 8 | 16 | 24 | 32}]
立即数方式	#imm[: { 8 | 16 | 32 }]
存储器间接方式	@@abs[:8]
带位移量程序计数器相对方式	d[: { 8 | 16 }]
控制寄存器	CCR、EXR、MACH、MACL

【注】 *1　n　……　寄存器号（0 ～ 7 [*2]）

disp　……　位移量

abs　……　绝对地址

imm　……　立即数

*2　ER7 和 SP（堆栈指针）相同。

（3）H8S/2000 系列的执行指令长度和寻址方式

（a）执行指令的长度

H8S/2000高级模式和正常模式的执行指令和操作长度的组合如表11.8所示：

表 11.8　H8S/2000 系列的执行指令和操作长度的组合

执行指令	操作长度				
	B	W	L	省略时	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	—	—	—	—	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	—	—	—	—	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	—	—	—	—	*1
JSR	—	—	—	—	*1
LDC	○	○	×	B	
LDM	×	×	○	L	
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	—	—	—	—	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	—	—	—	—	*1
RTS	—	—	—	—	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	—	—	—	—	*1
STC	○	○	×	B	
STM	×	×	○	L	
SUB	○	○	○	B	
SUBS	×	×	○	L	
SUBX	○	×	×	B	
TAS	○	×	×	B	
TRAPA	—	—	—	—	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1　不能指定长度。

*2　高级模式为 L（长字），正常模式为 W（字）。

（b）寻址方式

H8S/2000高级模式和正常模式的寻址方式如表11.9所示：

表 11.9　H8S/2000 系列的寻址方式

寻址方式	记述 [*1]
寄存器直接方式	{ ERn | En | Rn | RnH | RnL }
寄存器间接方式	@ERn
后增寄存器间接方式	@ERn+
先减寄存器间接方式	@-ERn
带位移量寄存器间接方式	@(disp[:{16 | 32}], ERn)
绝对寻址方式	@abs[: { 8 | 16 | 24 | 32}]
立即数方式	#imm[: { 8 | 16 | 32 }]
存储器间接方式	@@abs[:8]
带位移量程序计数器相对方式	d[: { 8 | 16 }]
控制寄存器	CCR、EXR

【注】 *1 n …… 寄存器号（0 ～ 7 [*2]）

disp …… 位移量

abs …… 绝对地址

imm …… 立即数

*2 ER7 和 SP（堆栈指针）相同。

（4）H8/300H 系列的执行指令长度和寻址方式

（a）执行指令的长度

H8/300H高级模式和正常模式的执行指令和操作长度的组合如表11.10所示：

表 11.10　H8/300H 系列的执行指令和操作长度的组合

执行指令	操作长度				
	B	W	L	省略时	
ADD	○	○	○	B	
ADDS	×	×	○	L	
ADDX	○	×	×	B	
AND	○	○	○	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	—	—	—	—	*1
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	—	—	—	—	*1
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	○	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	○	○	B	
DIVXS	○	○	×	B	
DIVXU	○	○	×	B	
EEPMOV	○	○	×	B	
EXTS	×	○	○	W	
EXTU	×	○	○	W	
INC	○	○	○	B	
JMP	—	—	—	—	*1
JSR	—	—	—	—	*1
LDC	○	○	×	B	
MOV	○	○	○	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXS	○	○	×	B	
MULXU	○	○	×	B	
NEG	○	○	○	B	
NOP	—	—	—	—	*1
NOT	○	○	○	B	
OR	○	○	○	B	
ORC	○	×	×	B	
POP	×	○	○	*2	
PUSH	×	○	○	*2	
ROTL	○	○	○	B	
ROTR	○	○	○	B	
ROTXL	○	○	○	B	
ROTXR	○	○	○	B	
RTE	—	—	—	—	*1
RTS	—	—	—	—	*1
SHAL	○	○	○	B	
SHAR	○	○	○	B	
SHLL	○	○	○	B	
SHLR	○	○	○	B	
SLEEP	—	—	—	—	*1
STC	○	○	×	B	
SUB	○	○	○	B	
SUBS	×	○	○	L	
SUBX	○	×	×	B	
TRAPA	—	—	—	—	*1
XOR	○	○	○	B	
XORC	○	×	×	B	

【注】 *1　不能指定长度。
*2　高级模式为 L（长字），正常模式为 W（字）。

（b）地址形式

H8/300H高级模式和正常模式的寻址方式如表11.11所示：

表 11.11　H8/300H 系列的寻址方式

寻址方式	记述 [*1]
寄存器直接方式	{ ERn | En | Rn | RnH | RnL }
寄存器间接方式	@ERn
后增寄存器间接方式	@ERn+
先减寄存器间接方式	@-ERn
带位移量寄存器间接方式	@(disp[:{16 | 24}], ERn)
绝对寻址方式	@abs[: { 8 | 16 | 24 }]
立即数方式	#imm[: { 8 | 16 | 32 }]
存储器间接方式	@@abs[:8]
带位移量程序计数器相对方式	d[: { 8 | 16 }]
控制寄存器	CCR

【注】 *1 n …… 寄存器号（0 ～ 7 [*2]）
disp …… 位移量
abs …… 绝对地址
imm …… 立即数
*2 ER7 和 SP（堆栈指针）相同。

（5）H8/300、H8/300L 系列的执行指令长度和寻址方式

（a）执行指令的长度

H8/300、H8/300L的执行指令和操作长度的组合如表11.12所示：

表 11.12　H8/300, H8/300L 系列的执行指令和操作长度的组合

执行指令	操作长度				
	B	W	L	省略时	
ADD	○	○	×	B	
ADDS	×	○	×	W	
ADDX	○	×	×	B	
AND	○	×	×	B	
ANDC	○	×	×	B	
BAND	○	×	×	B	
Bcc	—	—	—	—	*
BCLR	○	×	×	B	
BIAND	○	×	×	B	
BILD	○	×	×	B	
BIOR	○	×	×	B	
BIST	○	×	×	B	
BIXOR	○	×	×	B	
BLD	○	×	×	B	
BNOT	○	×	×	B	
BOR	○	×	×	B	
BSET	○	×	×	B	
BSR	—	—	—	—	*
BST	○	×	×	B	
BTST	○	×	×	B	
BXOR	○	×	×	B	
CMP	○	○	×	B	
DAA	○	×	×	B	
DAS	○	×	×	B	
DEC	○	×	×	B	
DIVXU	○	×	×	B	
EEPMOV	—	—	—	—	*
INC	○	×	×	B	
JMP	—	—	—	—	*
JSR	—	—	—	—	*
LDC	○	×	×	B	
MOV	○	○	×	B	
MOVFPE	○	×	×	B	
MOVTPE	○	×	×	B	
MULXU	○	×	×	B	
NEG	○	×	×	B	
NOP	—	—	—	—	*
NOT	○	×	×	B	
OR	○	×	×	B	
ORC	○	×	×	B	
POP	×	○	×	W	
PUSH	×	○	×	W	
ROTL	○	×	×	B	
ROTR	○	×	×	B	
ROTXL	○	×	×	B	
ROTXR	○	×	×	B	
RTE	—	—	—	—	*
RTS	—	—	—	—	*
SHAL	○	×	×	B	
SHAR	○	×	×	B	
SHLL	○	×	×	B	
SHLR	○	×	×	B	
SLEEP	—	—	—	—	*
STC	○	×	×	B	
SUB	○	○	×	B	
SUBS	×	○	×	W	
SUBX	○	×	×	B	
XOR	○	×	×	B	
XORC	○	×	×	B	

【注】 *　不能指定长度。

（b）寻址方式

H8/300和H8/300L的寻址方式如表11.13所示：

表 11.13　H8/300、H8/300L 系列的寻址方式

寻址方式	记述 [*1]
寄存器直接方式	{ Rn | RnH | RnL }
寄存器间接方式	@Rn
后增寄存器间接方式	@Rn+
先减寄存器间接方式	@-Rn
带位移量寄存器间接方式	@(disp[:16], Rn)
绝对寻址方式	@abs[: { 8 | 16 }]
立即数方式	#imm[: { 8 | 16 }]
存储器间接方式	@@abs[:8]
带位移量程序计数器相对方式	d[: 8]
控制寄存器	CCR

【注】 *1 n …… 寄存器号（0 ～ 7 [*2]）
disp …… 位移量
abs …… 绝对地址
imm …… 立即数
*2 R7 和 SP（堆栈指针）相同。

11.3　汇编程序控制指令

汇编程序控制指令是汇编程序解释和执行的指令。
格式的下划线表示省略时的解释。
汇编程序控制指令一览表如表 11.14 所示。

表 11.14　汇编程序控制指令一览表

分类	助记符	功能
有关 CPU 的内容	.CPU	指定 CPU 种类。
有关 8 位短绝对区的内容	.SBR	指定 8 位短绝对区的基点。
有关段或者定位计数器的内容	.SECTION	声明段。
	.ORG	设定定位计数器值。
	.ALIGN	将定位计数器值调整为边界调整数的倍数。
有关符号的内容	.EQU	给符号设定值。
	.ASSIGN	给符号设定或者重新设定值。
	.REG	设定寄存器别名。
	.BEQU	设定位数据名。
确保数据或者数据区的内容	.DATA	确保整数数据。
	.DATAB	确保整数数据块。
	.SDATA	确保字符串数据。
	.SDATAB	确保字符串数据块。
	.SDATAC	确保带计数的字符串数据。
	.SDATAZ	确保以零结束的字符串数据。
	.RES	确保数据区。
	.SRES	确保字符串数据区。
	.SRESC	确保带计数的字符串数据区。
	.SRESZ	确保以零结束的字符串数据区。
有关外部定义或者外部参照的内容	.EXPORT	声明外部定义符号。
	.IMPORT	声明外部参照符号。
	.GLOBAL	声明外部参照符号或者外部定义符号。
	.BEXPORT	声明外部定义 BEQU 符号。
	.BIMPORT	声明外部参照 BEQU 符号。
	.ABS8	指定 8 位短绝对地址符号。
	.NOABS8	将指定的 8 位短绝对地址符号设定为无效。
有关目标模块的内容	.OUTPUT	控制目标模块和调试信息的输出。
	.DEBUG	控制符号调试信息的部分输出。
	.LINE	更改调试信息的文件名和行号。
	.DISPSIZE	设定位移量长度。
有关汇编列表的内容	.PRINT	控制汇编列表的输出。
	.LIST	控制源程序列表的部分输出。
	.FORM	设定汇编列表的行数和列数。
	.HEADING	设定源程序列表页眉。
	.PAGE	将源程序列表换页。
	.SPACE	将空行输出到源程序列表。

分类	助记符	功能
其它	.PROGRAM	设定目标模块名。
	.RADIX	指定无基数指定的整数常数的基数。
	.END	指定源程序的结束和入口点。
	.STACK	对指定的符号定义堆栈值。

CPU 种类的指定

.CPU

格　式

```
△.CPU△<CPU 种类>
      <CPU 种类>＝{ AE5                                              |
                    H8SXX [:<地址空间的位宽>][:{M|D|MD}]              |
                    H8SXA [:<地址空间的位宽>][:{M|D|MD}]              |
                    H8SXM [:<地址空间的位宽>][:{M|D|MD}]              |
                    H8SXN [:{M|D|MD}]                                |
                    2600A [:<地址空间的位宽>]                         |
                    2600N                                            |
                    2000A [:<地址空间的位宽>]                         |
                    2000N                                            |
                    300HA [:<地址空间的位宽>]                         |
                    300HN                                            |
                    300 | 300L                                       }
```

不能记述标号。

说　明

.CPU 指定要建立的目标程序的 CPU 种类、运行模式、地址空间的位宽和有无乘除法器。
只能在 CPU 种类为最高模式、高级模式和中级模式时指定地址空间的位宽。
CPU 种类和地址空间的位宽如下：

子选项名	含　义
AE5	建立用于 AE5 的目标程序。
H8SXX [:<地址空间的位宽>] [:{M|D|MD}]	建立用于 H8SX 最高模式的目标程序。 〈地址空间的位宽〉为 28 或者 32 的数值，分别表示 256M 字节和 4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 32。 能指定乘除法器。
H8SXA [:<地址空间的位宽>] [:{M|D|MD}]	建立用于 H8SX 高级模式的目标程序。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节和 4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。 能指定乘除法器。
H8SXM [:<地址空间的位宽>] [:{M|D|MD}]	建立用于 H8SX 中级模式的目标程序。 〈地址空间的位宽〉为 20 或者 24 的数值，分别表示 1M 字节和 16M 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。 能指定乘除法器。
H8SXN[:{M|D|MD}	建立用于 H8SX 正常模式的目标程序。 能指定乘除法器。
2600A [:<地址空间的位宽>]	建立用于 H8S/2600 高级模式的目标程序。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节和 4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
2600N	建立用于 H8S/2600 的正常模式的目标程序。

2000A [：<地址空间的位宽>]	建立用于 H8S/2000 高级模式的目标程序。 〈地址空间的位宽〉为 20、24、28 或者 32 的数值，分别表示 1M 字节、16M 字节、256M 字节和 4G 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
2000N	建立用于 H8S/2000 正常模式的目标程序。
300HA [：<地址空间的位宽>]	建立用于 H8/300H 高级模式的目标程序。 〈地址空间的位宽〉为 20 或者 24 的数值，分别表示 1M 字节和 16M 字节的地址空间。 省略〈地址空间的位宽〉时解释为 24。
300HN	建立用于 H8/300H 正常模式的目标程序。
300	建立 H8/300 的目标程序。
300L	建立 H8/300L 的目标程序。

有无乘法器/除法器的指定如下:

乘法器/除法器	指定方法
无/无	无指定
有/无	M
无/有	D
有/有	MD

有乘法器时的追加指令为 MAC、LDMAC、STMAC、CLRMAC、MULU/U、MULS/U。
有除法器时，不存在追加指令。

必须在源程序开始处记述本控制指令。除有关汇编列表的控制指令以外，如果在源程序开始处没有本控制指令，就产生错误。
另外，此记述只限 1 次有效。
CPU 种类的优先顺序为 cpu 选项、.CPU 控制指令、H38CPU 环境变量的顺序。
如果省略指定，H38CPU 环境变量设定的 CPU 种类有效。

例

```
        .CPU        2600A:20            ; 汇编成用于 H8S/2600 高级模式的 1Mbyte
        .SECTION    A,CODE,ALIGN=2      ; 模式。
        MOV.L       ER0,ER1
        MOV.L       ER0,ER2
```

8 位短绝对区的基点指定

.SBR

格　式　△.SBR△[<常数>]
不能记述标号。

说　明　.SBR 声明 8 位短绝对区的基点。

如果指定.SBR <常数>，就将指定的常数值作为 8 位短绝对区的基点。必须将基点的最低 8 位置 0。

如果只指定.SBR，8 位短绝对区的基点就因 SBR 选项指定的有无而不同。如果指定 SBR 选项，就为 SBR 选项指定的值。没有指定 SBR 选项时，根据地址空间的位宽，为以下的基点：

	CPU/运行模式	8 位短绝对地址基点
H8SX 最高模式	H8SXX[:32]	H'FFFFFF00
	H8SXX:28	H'0FFFFF00
H8SX 高级模式	H8SXA:32	H'FFFFFF00
	H8SXA:28	H'0FFFFF00
	H8SXA[:24]	H'00FFFF00
	H8SXA:20	H'000FFF00
H8SX 中级模式	H8SXM[:24]	H'00FFFF00
	H8SXM:20	H'000FFF00
H8SX 正常模式	H8SXN	H'0000FF00

在 CPU 为 H8SXN、H8SXM、H8SXA、H8SXX 时，能指定 SBR 控制指令。

另外，实际上，在将地址设定到 SBR（短地址基址寄存器）时，需要记述 LDC 指令。

例

```
.CPU        H8SXA:32
.SECTION    A,CODE,ALIGN=2
.SBR      H'10000               ; 将地址 H'00010000 声明为 SBR。
MOV.L     #H'10000,ER1
LDC.L     ER1,SBR               ; 实际上，将地址设定到 SBR。
～
MOV.B     @H'FFFFFF00,R0L       ; 以 16 位存取。
MOV.B     @H'00010050,R0H       ; 以 8 位存取。
～
.SBR                            ; 清除 SBR 的声明。
MOV.L      #H'FFFFFF00,ER1
LDC.L      ER1,SBR              ; 实际上，将 SBR 恢复到默认值。
～
MOV.B     @H'FFFFFF00,R0L       ; 以 8 位存取。
MOV.B     @H'00010050,R0H       ; 以 32 位存取。
～
```

段的声明

.SECTION

格　式

```
△.SECTION△<段名>[,<段属性>[,<格式种类>]]
   <段属性>          ={ CODE    |
                        DATA    |
                        STACK   |
                        DUMMY   }
   <格式种类>        ={ LOCATE  = <起始地址>   |
                        ALIGN   = <边界调整数>   }
```

不能记述标号。

说　明

.SECTION 是声明段以及指定重新开始的汇编程序控制指令。
段是程序的 1 个分隔、连接的处理单位。
(1) 段的开始
段名的编程方法和符号名的编程方法相同。
另外，区别段的英文大小写字母。

段属性如下：
- CODE　…… 代码段
- DATA　…… 数据段
- STACK …… 堆栈段
- DUMMY …… 虚段

如果指定 locate=<起始地址>，就以绝对寻址方式输出目标程序。
如果指定 align=<边界调整数>，就以相对寻址方式输出目标程序。
优化连接编辑程序将该段的起始地址调整为边界调整数的倍数的绝对地址。
没有指定格式种类时，设定 align=2。

- 绝对寻址方式

  在绝对寻址方式，设定段的起始地址。
  起始地址的最大值如下：

	CPU/运行模式	最大值
H8SX 最高模式	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFF
H8SX 高级模式	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFF
	H8SXA[:24]	H'00FFFFFF
	H8SXA:20	H'000FFFFF
H8SX 中级模式	H8SXM[:24]	H'00FFFFFF
	H8SXM:20	H'000FFFFF
H8SX 正常模式	H8SXN	H'0000FFFF
H8S/2600 高级模式	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFF
	2600A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2600 正常模式	2600N	H'0000FFFF
H8S/2000 高级模式	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFF
	2000A[:24]	H'00FFFFFF
	2000A:20	H'000FFFFF
H8S/2000 正常模式	2000N	H'0000FFFF

	CPU/运行模式	最大值
H8/300H 高级模式	300HA[:24]	H'00FFFFFF
	300HA:20	H'000FFFFF
H8/300H 正常模式	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

- 相对寻址方式

  在相对寻址方式，设定段的边界调整数。

  在连接目标模块时，优化连接编辑程序将相对地址段的起始地址调整为边界调整数的倍数。

  边界调整数能指定 2^n 值。

在不用本控制指令声明段时，就作为默认段，进行如下设定：

```
    .SECTION  P,CODE,ALIGN=2
```

另外，在以下任一情况下，汇编程序准备了默认段。

- 在不声明段时记述了执行指令。
- 在不声明段时记述了确保数据的汇编程序控制指令。
- 在不声明段时记述了.ALIGN 汇编程序控制指令。
- 在不声明段时记述了.ORG 汇编程序控制指令。
- 在不声明段时参照了定位计数器。
- 在不声明段时记述了只有标号的行。

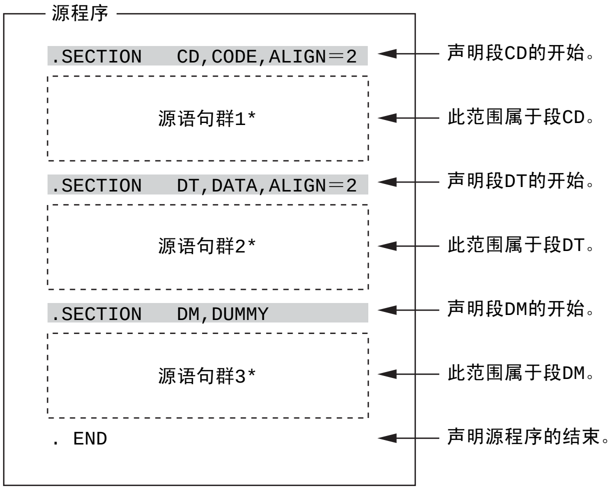
(2)段的重新开始

重新开始源程序中已存在的段。

段的重新开始指定已存在的段名。

段属性和格式种类继续最初声明的内容。

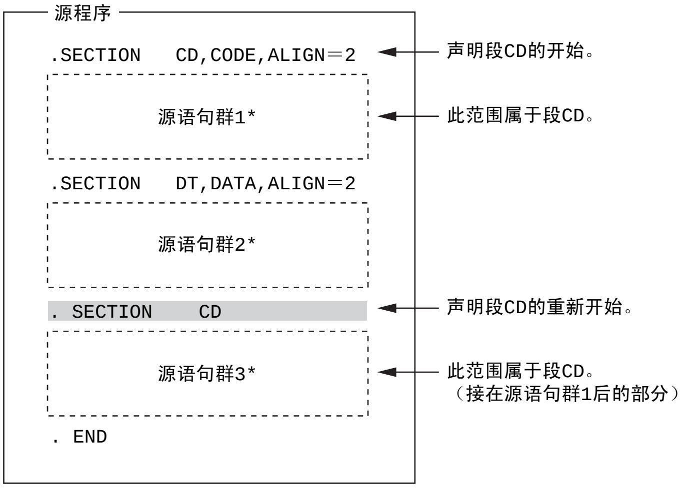
举例说明有关段的声明。



```
.SECTION   CD,CODE,ALIGN=2
    源语句群1*
.SECTION   DT,DATA,ALIGN=2
    源语句群2*
.SECTION   DM,DUMMY
    源语句群3*
. END
```

【注】* 在此例中，假设“源语句群1～3”中不出现.SECTION。

能重新开始源程序中已存在的段。
段的重新开始指定已存在的段名。
举例说明有关段的重新开始。



源程序

```
.SECTION   CD,CODE,ALIGN＝2
        源语句群1*
.SECTION   DT,DATA,ALIGN＝2
        源语句群2*
. SECTION    CD
        源语句群3*
. END
```

声明段CD的开始。

此范围属于段CD。

声明段CD的重新开始。

此范围属于段CD。
（接在源语句群1后的部分）

【注】* 在此例中，假设“源语句群1～3”中不出现.SECTION。

例　　在此例中，假设“～”的部分中不出现.SECTION。

```
            . ALIGN   2               ；  此范围属于默认段P。
            . DATA.W  H'0102,H'0304   ；  默认段P为代码段，并且为
              ～                      ；  边界调整数＝2的相对地址段。

      . SECTION    CD, CODE, ALIGN＝2
            MOV.W     R0, R1          ；  此范围属于段CD。
            MOV.W     R0, R2          ；  段CD为代码段，并且为
              ～                      ；  边界调整数＝2的相对地址段。

      . SECTION    DT, DATA, LOCATE＝H' 00001000
  X1：      . DATA.W  H'2222          ；  此范围属于段DT。
            . DATA.W  H'3333          ；  段DT为数据段，并且为
              ～                      ；  起始地址＝ H'00001000的绝对地址段。
            . END
```

定位计数器值的设定

.ORG

格　式　　△.ORG△<定位计数器值>

不能记述标号。

说　明　　.ORG 设定段内的定位计数器值。

能通过.ORG 将执行指令和数据分配到特定的地址。

定位计数器值的指定如下：

- 指定常数值或者段内的地址。
  并且
- 指定时，不使用前方参照符号。

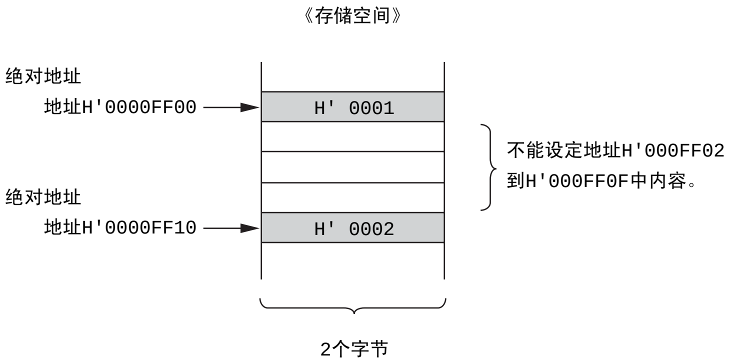
定位计数器的最大值如下：

	CPU/运行模式	最大值
H8SX 最高模式	H8SXX[:32]	H'FFFFFFFF
	H8SXX:28	H'0FFFFFFF
H8SX 高级模式	H8SXA:32	H'FFFFFFFF
	H8SXA:28	H'0FFFFFFF
	H8SXA[:24]	H'00FFFFFF
	H8SXA:20	H'000FFFFF
H8SX 中级模式	H8SXM[:24]	H'00FFFFFF
	H8SXM:20	H'000FFFFF
H8SX 正常模式	H8SXN	H'0000FFFF
H8S/2600 高级模式	2600A:32	H'FFFFFFFF
	2600A:28	H'0FFFFFFF
	2600A[:24]	H'00FFFFFF
	2600A:20	H'000FFFFF
H8S/2600 正常模式	2600N	H'0000FFFF
H8S/2000 高级模式	2000A:32	H'FFFFFFFF
	2000A:28	H'0FFFFFFF
	2000A[:24]	H'00FFFFFF
	2000A:20	H'000FFFFF
H8S/2000 正常模式	2000N	H'0000FFFF
H8/300H 高级模式	300HA[:24]	H'00FFFFFF
	300HA:20	H'000FFFFF
H8/300H 正常模式	300HN	H'0000FFFF
H8/300	300	H'0000FFFF
H8/300L	300L	H'0000FFFF

在用绝对地址段指定时，通过段的起始地址以上的值指定定位计数器值。如果用绝对地址段指定本控制指令，设定的定位计数器值就为绝对地址；如果用相对地址段指定本控制指令，就为相对地址。

例

```
.SECTION  DT,DATA,LOCATE=H'0000FF00
.DATA.W    H'0001
.ORG  H'0000FF10               ; 设定定位计数器值。
.DATA.L    H'0002              ; 将整数数据 H'0002 确保在
～                             ; 绝对地址 H'0000FF10 中。
```



定位计数器值的调整

.ALIGN

格　式　　△.ALIGN△<边界调整数>

不能记述标号。

说　明　　.ALIGN 将段内的定位计数器值调整为边界调整数的倍数。

能通过.ALIGN 将执行指令和数据分配到特定的边界（地址分隔）。

定位计数器值的指定如下：

- 指定常数值。

  并且

- 指定时，不使用前方参照符号。

能给边界调整数指定 2^n 的值。

边界调整数因地址空间的指定而不同。

在相对地址段使用.ALIGN 时，必须设定为

　.SECTION 指定的边界调整数 ≥ .ALIGN 指定的边界调整数。

如果在代码段中记述.ALIGN，汇编程序就将 NOP 指令的目标码*填充到存储器，调整定位计数器值。将 H'00 填充到不完全的字节长的区域。

如果在数据段、虚段和堆栈段中记述.ALIGN，汇编程序就只调整定位计数器值，而不给存储器填充目标码。

【注】* 这样的目标码不表现在汇编列表中。

例

```
        .CPU        2600A
        .SECTION    P,DATA,ALIGN=2     ; …… [1]
        .DATA.B     H'11               ; …… [2]
        .ALIGN      2                  ; …… [3]
        .DATA.W     H'2222
        ～
```

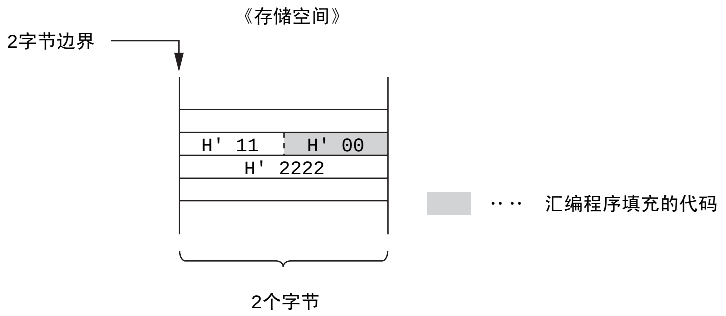
[1] 在连接目标模块时，将相对地址段的起始地址调整为 2 的倍数地址。

[2] 为了确保 1 个字节的数据，下一个数据的定位计数器值变为奇数地址。

[3] 将定位计数器值调整为 2 的倍数地址（偶数地址）。

假设字节长的整数数据 H'11 原来就位于 2 个字节的边界。

汇编程序按下图进行目标码的填充以及边界调整。



给符号设定值

.EQU

格　式	<符号>[:]△.EQU△<符号值>

说　明

.EQU 给符号设定值。
用.EQU 定义的符号不能重新定义。
符号值的指定如下：

- 指定常数值、地址和外部参照符号的值*。
  并且
- 指定时，不使用前方参照符号。

符号的允许值为 H'00000000～H'FFFFFFFF。
【注】* 能记述外部参照符号、外部参照符号＋常数、外部参照符号－常数。

例

```
         ～
X1:     .EQU      10                ;X1 的值为 10。
X2:     .EQU      20                ;X2 的值为 20。
         CMP.W     #X1,R0            ;和 CMP.W #10,R0 相同。
         BNE       LABEL1
         CMP.W     #X2,R0            ;和 CMP.W #20,R0 相同。
         BEQ       LABEL2
         ～
```

给符号设定值

.ASSIGN

格　式　　<符号> [:] △.ASSIGN△<符号值>

说　明　　.ASSIGN 是给符号设定值的汇编程序控制指令。

能通过.ASSIGN 重新定义由.ASSIGN 定义的符号。

符号值的指定如下。

- 指定常数值或者地址。

  并且

- 指定时，不使用前方参照符号。

符号的允许值为 H'00000000～H'FFFFFFFF。

从定义的位置开始，.ASSIGN 定义有效。

.ASSIGN 定义的符号有以下使用限制：

- 不能外部参照或者外部定义。
- 调试程序不能参照。

例

```
             ～
X1:     .ASSIGN     1
X2:     .ASSIGN     2
        CMP.W       #X1,R0             ;和 CMP.W #1,R0 相同。
        BNE         LABEL1
        CMP.W       #X2,R0             ;和 CMP.W #2,R0 相同。
        BEQ         LABEL2
             ～
X1:     .ASSIGN     3
X2:     .ASSIGN     4
        CMP.W       #X1,R0             ;和 CMP.W #3,R0 相同。
        BNE         LABEL3
        CMP.W       #X2,R0             ;和 CMP.W #4,R0 相同。
        BEQ         LABEL4
             ～
```

寄存器别名的定义

.REG

格　式　　<符号>［:］△.REG△(<寄存器名>)

说　明　　.REG 给寄存器取别名。

寄存器名如下:

- 单寄存器 … 给 1 个寄存器取别名。能在任何能使用寄存器的位置指定。能给寄存器名指定通用寄存器。
- 多寄存器 … 给 2 个以上的寄存器取别名。但是，只限于 CPU 种类为 H8SX 系列、H8S/2600 系列和 H8S/2000 系列。能指定为 LDM 指令、STM 指令和.REG 指令的操作数。而且，在 H8SX 系列中，也能指定为 RTS/L 和 RTE/L 指令的操作数。能给寄存器名指定 32 位通用寄存器。

寄存器名的编程方法如下：

指定方法	说明	使用例
单寄存器	指定 R0L～R7L、R0H～R7H、R0～R7、E0～E7 或者 ER0～ER7。	SINGLEREG　.REG　(R0) 给寄存器 R0 指定 SINGLEREG 的别名。
多寄存器	用连字符（-）分开，以范围的格式一次指定多个寄存器。 如果右侧的寄存器号小于左侧的寄存器号，就出错，忽视.REG 指令。	RNG1　.REG　(ER0-ER3) 给 4 个寄存器 ER0、ER1、ER2、ER3 取 RNG1 的别名。 RNG2　.REG　(ER3-ER0) 因为右侧的号小，所以出错。*
寄存器别名的重新设定	预先将已定义的寄存器别名指定为操作数。	ER00　.REG　(ER0-ER3) ER01　.REG　(ER00) 给 4 个寄存器 ER0～ER3 取 ER01 的别名。

【注】　用.REG 定义的寄存器别名不能重新定义。

从已定义的位置开始，.REG 定义有效。

.REG 定义的符号有以下使用限制:

- 不能外部参照或者外部定义。
- 调试程序不能参照。

H8SX 系列中能指定的寄存器组合如下：

(ERn-ERn+1)其中 n 为 0～6、(ERn-ERn+2)其中 n 为 0～5、
(ERn-ERn+3)其中 n 为 0～4

H8S/2600 系列、H8S/2000 系列中能指定的寄存器的组合如下：

(ER0-ER1)、(ER2-ER3)、(ER4-ER5)、(ER6-ER7)、(ER0-ER2)、
(ER4-ER6)、(ER0-ER3)、(ER4-ER7)

例

```
        .CPU      2600A
RLST1:  .REG      (R0)              ; …… [1]
RLST2:  .REG      (ER0-ER2)         ; …… [2]
        MOV.W     RLST1,@ER6
        LDM.L     @SP+,(RLST2)
        STM.L     (RLST2),@-SP
```

[1]　将 R0 寄存器指定为 RLST1。

[2]　将 3 个寄存器 ER0、ER1、ER2 指定为 RLST2。

.BEQU

格　式　<符号>[:]△.BEQU△<位号>, <置换符号名>

说　明　.BEQU 给位操作指令的对象存储器中的 1 位数据命名。
能将位数据名指定为位操作指令的操作数。
将被指定的位数据名置换为#xx,@aa 格式。

位号的指定如下：

- 指定常数值或者地址。
  并且
- 指定时，不使用前方参照符号。

能给位号指定 0～7 的值。

置换符号名的指定如下：

CPU 种类	置换符号名
H8SX 系列	8 位绝对寻址方式 (@aa:8)
H8S/2600 系列	16 位绝对寻址方式(@aa:16)
H8S/2000 系列	32 位绝对寻址方式(@aa:32)
H8/300H 系列	8 位绝对寻址方式 (@aa:8)
H8/300 系列	
H8/300L 系列	

【注】　从已定义的位置开始，.BEQU 定义有效。
能用.BEXPORT、.BIMPORT 外部定义/参照.BEQU 定义的符号。

例

```
         .CPU    2600A:32
AD1      .EQU    H'FFFFFF00
AD2      .EQU    H'FFFF8000
AD1B0    .BEQU   0,AD1
AD1B1    .BEQU   1,AD1
AD2B2    .BEQU   2,AD2
AD2B3    .BEQU   3,AD2

         .SECTION   A,CODE,ALIGN=2
         BSET.B  AD1B0                ; BSET.B    #0,@AD1:8
         BSET.B  AD1B1                ; BSET.B    #1,@AD1:8
         BSET.B  AD2B2                ; BSET.B    #2,@AD2:16
         BSET.B  AD2B3                ; BSET.B    #3,@AD2:16
```

位数据名的内容如下：

AD1B0　……　地址 H'FFFFFF00 的位 0
AD1B1　……　地址 H'FFFFFF00 的位 1
AD2B2　……　地址 H'FFFF8000 的位 2
AD2B3　……　地址 H'FFFF8000 的位 3

备　考　能指定位数据的位操作指令如下：
BSET、BCLR、BNOT、BTST、BAND、BIAND、BOR、BIOR、BXOR、BIXOR、BLD、BILD、BST、BIST

整数数据的确保

.DATA

格　式　　[<符号>[:]]△.DATA[.操作长度]△整数数据[,…]

　　　　　　　　<操作长度> = { B | W | L }

说　明　　.DATA 按照指定的长度将整数数据确保在存储器中。

操作长度和整数数据的范围如下：

大小	数据长度		整数数据的范围（10 进制表现）	
B	字节	（1 个字节）	H'00000000～H'000000FF	(0～255)
			H'FFFFFF80～H'FFFFFFFF	(-128～-1)
W	字	（2 个字节）	H'00000000～H'0000FFFF	(0～65,535)
			H'FFFF8000～H'FFFFFFFF	(-32,768～-1)
L	长字	（4 个字节）	H'00000000～H'FFFFFFFF	(0～4,294,967,295)
			H'80000000～H'FFFFFFFF	(-2,147,483,648～-1)

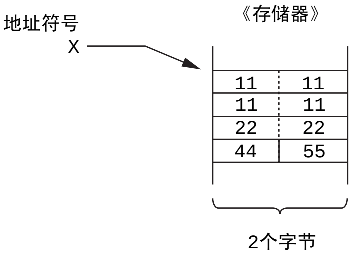
如果省略操作长度，汇编程序就解释为.DATA.B（字节长）。

能给整数数据指定包含相对地址、外部参照符号和前方参照符号的任意值。

能指定的整数数据的范围根据操作长度而不同。

例

```
       .SECTION   A,DATA,ALIGN=2
X:
       .DATA.L    H'11111111          ;
       .DATA.W    H'2222              ; 确保整数数据。
       .DATA.B    H'44,H'55           ;
       ～
```



【注】数据为16进制数。

整数数据块的确保

.DATAB

格　式　[<符号>[:]]△.DATAB[.<操作长度>]△<块数>,<整数数据>
<操作长度> = { B | W | L }

说　明　.DATAB 按照指定的长度将块数的整数数据确保在存储器中。

确保的数据长度取决于操作长度。
如果省略操作长度，就为.DATAB.B（字节长）。
块数的指定如下：
- 指定常数值。
  并且
- 指定时，不使用前方参照符号。

操作长度和块数的范围如下：

长度	数据长度		块数的范围（10 进制表现）	
B	字节	（1 个字节）	H'00000001～H'FFFFFFFF	(1～4,294,967,295)
W	字	（2 个字节）	H'00000001～H'7FFFFFFF	(1～2,147,483,647)
L	长字	（4 个字节）	H'00000001～H'3FFFFFFF	(1～1,073,741,823)

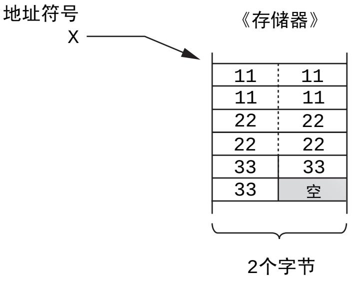
能给整数数据指定包含相对地址、外部参照符号和前方参照符号的任意值。能指定的块数范围和整数数据的范围根据操作长度而不同。

整数数据的范围如下：

长度	整数数据的范围（10 进制表现）	
B	H'00000000~H'000000FF	(0～255)
	H'FFFFFF80～H'FFFFFFFF	(-128～-1)
W	H'00000000～H'0000FFFF	(0～65,535)
	H'FFFF8000～H'FFFFFFFF	(-32,768～-1)
L	H'00000000～H'FFFFFFFF	(0～4,294,967,295）
	H'80000000～H'FFFFFFFF	(-2,147,483,648～-1)

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .DATAB.L    1,H'11111111          ;
        .DATAB.W    2,H'2222              ; 确保整数数据块。
        .DATAB.B    3,H'33                ;
        ～
```



【注】数据为16进制数。

字符串数据的确保

.SDATA

格　式　　[<符号>[:]]△.SDATA△"<字符串>"[,…]

说　明　　.SDATA 将字符串数据确保在存储器中。

将字符用双引号（"）括起来指定字符串。
将双引号（"）指定为字符时，连续记述 2 个。

给字符串指定控制字符时，紧接在双引号（"）括起来的字符串后用尖括号（<>）将控制码括起来记述。

　"字符串"<控制码>

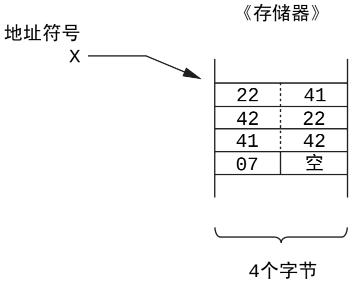
控制码的指定如下：

- 指定常数值。

  并且
- 指定时，不使用前方参照符号。

例

```
        .SECTION    A,DATA,ALIGN=2
        .SDATA      """AB"""              ; 含有双引号的例子。
        .SDATA      "AB"<H'07>            ; 附加控制字符的例子。
```



【注】1　数据为16进制数。

【注】2　字符A的ASCII码‥‥ H'41
字符B的ASCII码‥‥ H'42
字符"的ASCII码‥‥ H'22

字符串数据块的确保

.SDATAB

格　式　　[<符号>[:]]△.SDATAB△<块数>,"<字符串>"

说　明　　.SDATAB 将块数的字符串数据连续确保在存储器中。

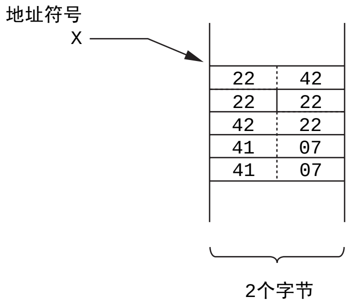
块数的指定如下：
- 指定常数值。
  并且
- 指定时，不使用前方参照符号。

必须给块数指定 1 以上的值。
块数的上限值必须在字符串数据的长度×块数为 H'FFFFFFFF（4,294,967,295 个字节）以内。
将字符用双引号（"）括起来指定字符串。
将双引号（"）指定为字符时，连续记述 2 个。
给字符串指定控制字符时，紧接在双引号（"）括起来的字符串后用尖括号（<>）将控制码括起来记述。。
　"字符串"<控制码>
控制码的指定如下：
- 指定常数值。
  并且
- 指定时，不使用前方参照符号。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SDATAB     2,"""B"""           ; 含有双引号的例子。
        .SDATAB     2,"A"<H'07>         ; 附加控制字符的例子。

        ~
```

《存储器》



22	42
22	22
42	22
41	07
41	07

【注】1　数据为16进制数。

【注】2　字符A的ASCII码 ‥‥ H' 41
　　　　字符B的ASCII码 ‥‥ H' 42
　　　　字符"的ASCII码 ‥‥ H' 22

带计数的字符串数据的确保

.SDATAC

格　式　　`[<符号>[:]]△.SDATAC△"<字符串>"[,…]`

说　明　　.SDATAC 将带计数的字符串数据确保在存储器中。

带计数的字符串是在字符串的前头附加 1 个字节计数的字符串。

确保字符串数据时，在字符串数据的前头附加 1 个字节的计数（表示字符串字节数的数据）。

计数不包含计数本身的 1 个字节。

将字符用双引号（"）括起来指定字符串。

将双引号（"）指定为字符时，连续记述 2 个。

给字符串指定控制字符时，紧接在双引号（"）括起来的字符串后用尖括号（<>）将控制码括起来记述。

　"字符串"<控制码>

控制码的指定如下：

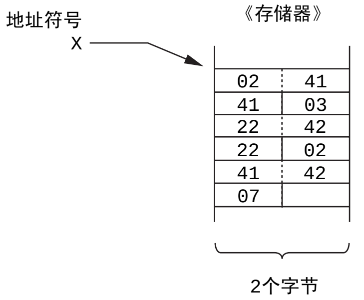
- 指定常数值。

　并且

- 指定时，不使用前方参照符号。

例

```
         .SECTION    A,DATA,ALIGN=2
X:
         .SDATAC     "AA"                 ; 确保带计数的字符串数据。
         .SDATAC     """B"""              ; 含有双引号的例子。
         .SDATAC     "AB"<H'07>           ; 附加控制字符的例子。
```



【注】1　数据为16进制数。

【注】2　字符A的ASCII码 ‥‥ H' 41
　　　　字符B的ASCII码 ‥‥ H' 42
　　　　字符"的ASCII码 ‥‥ H' 22

以零结束的字符串数据的确保

.SDATAZ

格　式　　[<符号>[:]]△.SDATAZ△"<字符串>"[,…]

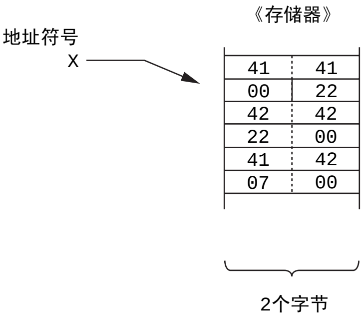
说　明　　.SDATAZ 将以零结束的字符串数据确保在存储器中。
确保数据时，在字符串数据的最后附加 1 个字节的 0。

将字符用双引号（"）括起来指定字符串。
将双引号（"）指定为字符时，连续记述 2 个。
给字符串指定控制字符时，紧接在双引号（"）括起来的字符串后用尖括号（<>）将控制码括起来记述。。
　"字符串"<控制码>
控制码如下指定：

- 指定常数值。
  并且
- 指定时，不使用前方参照符号。

例

```
        .SECTION    A,DATA,ALIGN=2
X:
        .SDATAZ     "AA"                  ; 确保以零结束的字符串数据。
        .SDATAZ     """BB"""              ; 含有双引号的例子。
        .SDATAZ     "AB"<H'07>            ; 附加控制字符的例子。
        ～
```



【注】1　数据为16进制数。

【注】2　字符A的ASCII码 ‥‥ H' 41
字符B的ASCII码 ‥‥ H' 42
字符"的ASCII码 ‥‥ H' 22

数据区的确保

.RES

格　式

```
[<符号>[:]]△.RES[.<操作长度>]△<区域数>
          <操作长度> = { B | W | L }
```

说　明

.RES 确保用于整数数据的数据区。
用指定长度确保<区域数>的区域。
确保的数据区的单位容量取决于操作长度。
数据的长度和区域数的范围如下：

大小	数据长度	区域数的范围（10 进制表现）
B	字节　（1 个字节）	H'00000001～H'FFFFFFFF　(1～4,294,967,295)
W	字　　（2 个字节）	H'00000001～H'7FFFFFFF　(1～2,147,483,647)
L	长字　（4 个字节）	H'00000001～H'3FFFFFFF　(1～1,073,741,823)

如果省略操作长度，就为字节长。

区域数的指定如下：

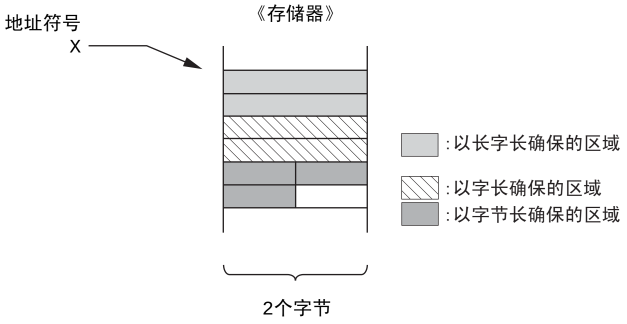
- 指定常数值。

  并且

- 指定时，不使用前方参照符号。

例

```
        .SECTION    A,DATA,ALIGN=2
        .ALIGN   4
X:
        .RES.L      1          ; 确保 1 个长字长的区域。
        .RES.W      2          ; 确保 2 个字长的区域。
        .RES.B      3          ; 确保 3 个字节长的区域。
        ～
```



.SRES

格　式　　[<符号>[:]]△.SRES△<字符串区容量>[,…]

说　明　　.SRES 确保用于字符串的数据区。

确保指定区域容量（字节数）的区域。

字符串区容量的指定如下：

- 指定常数值。

  并且

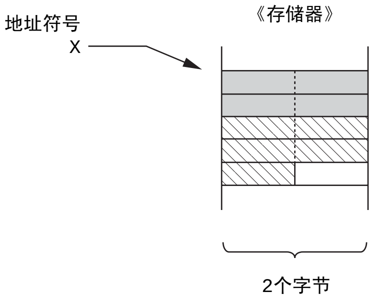
- 指定时，不使用前方参照符号。

字符串区容量的允许值为 H'00000001～H'FFFFFFFF。

（10 进制表现为 1～4,294,967,295）

例

```
        .SECTION   A,DATA,ALIGN=2
X:
        .SRES      4                     ; 确保 4 个字节的区域。
        .SRES      5                     ; 确保 5 个字节的区域。
        ～
```



带计数的字符串数据区的确保

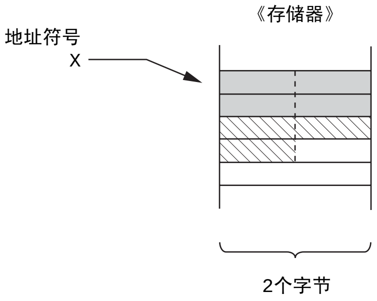
.SRESC

格　式　　[<符号>[:]]△.SRESC△<字符串区容量>[,…]

说　明　　.SRESC 将用于带计数的字符串数据区确保在存储器中。
确保给指定区域容量（字节数）附加 1 个字节的计数后的区域。
字符串区容量的指定如下：
・指定常数值。
　并且
・指定时，不使用前方参照符号。
字符串区容量的允许值为 H'00000000～H'000000FF（10 进制表现为 0～255）。
被确保在存储器中的区域容量为字符串区容量＋用于计数的 1 个字节。

例

```
       .SECTION    A,DATA,ALIGN=2
X:
        .SRESC     3                ; 确保 3 个字节+用于计数的 1 个字节。
        .SRESC     2                ; 确保 2 个字节+用于计数的 1 个字节。
        ～
```



以零结束的字符串数据区的确保

.SRESZ

格　式　　[<符号>[:]]△.SRESZ△<字符串区容量>[,…]

说　明　　.SRESZ 将用于以零结束的字符串数据区确保在存储器中。

确保给指定区域容量（字节数）附加零结束 1 个字节后的区域。

字符串区容量的指定如下：

- 指定常数值。

  并且

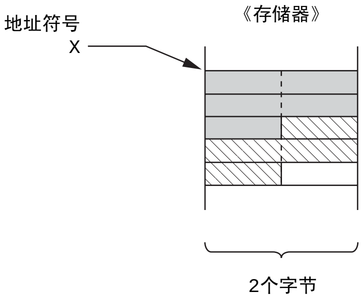
- 指定时，不使用前方参照符号。

字符串区容量的允许值为 H'00000000～H'000000FF（10 进数表现为 0～255）。

被确保在存储器上的区域容量为字符串区容量＋用于零结束的 1 个字节。

例

```
        .SECTION   A,DATA,ALIGN=2
X:
        .SRESZ     4             ; 确保 4 个字节+用于零结束的 1 个字节。
        .SRESZ     3             ; 确保 3 个字节+用于零结束的 1 个字节。
        ～
```



外部定义符号的声明

.EXPORT

格　式　　△.EXPORT△<符号>[:{ 8 | 16}][,…]

不能记述标号。

说　明　　.EXPORT 声明外部定义符号。

在从其它源程序开始参照源程序内定义的符号时声明。

以下的符号能被指定为外部定义符号:

- 具有绝对值的符号
- 具有地址值的符号

但是，不能指定用.ASSIGN 控制指令定义的符号和具有虚段地址值的符号。

另外，通过给符号名附加存取长度（:8 或者:16），以 8 位或者 16 位绝对寻址方式对该符号进行存取。但是，对于前方参照符号，忽视指定的存取长度。

本控制指令的声明与存取长度的有无无关，最初设定的符号有效，第 2 次以后的声明无效。另外，如果在本控制指令前指定 ABS8/NOABS8 控制指令，就遵循 ABS8/NOABS8 控制指令的指示。

从其它源程序外部参照符号时，需要对应外部定义符号声明，在参照该符号的其它源程序中进行外部参照符号的声明（.IMPORT）。

例　　在文件 B 中参照文件 A 定义的符号的例子。

- 文件 A

```
        .EXPORT    X                   ; 将 X 声明为外部定义符号。
        ～
X:      .EQU  H'10000000               ; 定义 X。
        ～
```

- 文件 B

```
        .IMPORT    X                   ; 将 X 声明为外部参照符号。
        ～
        .SECTION   A,DATA,ALIGN=2
        .DATA.L    X                   ; 参照 X。
        ～
```

外部参照符号的声明

.IMPORT

格　式　　△.IMPORT△<符号>[:{ 8 | 16}][,…]

不能记述标号。

说　明　　.IMPORT 声明外部参照符号。

在参照其它源程序内定义的符号时声明。

不能对源程序内定义的符号进行外部参照符号的声明。

另外，通过给符号名附加存取长度（:8 或者:16），以 8 位或者 16 位绝对寻址方式对该符号进行存取。但是，对于前方参照符号，忽视指定的存取长度。

本控制指令的声明与存取长度的有无无关，最初设定的符号有效，第 2 次以后的声明无效。另外，如果在本控制指令前指定 ABS8/NOABS8 控制指令，就遵循 ABS8/NOABS8 控制指令的指示。

外部参照符号时，需要对应外部参照符号声明，在声明该符号的源程序中进行外部定义符号声明（.EXPORT）。

例　　在文件 B 中参照文件 A 定义的符号的例子。

・文件 A

```
        .CPU       2600A
        .EXPORT    X                  ; 将 X 声明为外部定义符号。
        ～
        .SECTION   A,CODE,ALIGN=2
X:      .EQU  H'10000000              ; 定义 X。
        ～
```

・文件 B

```
        .IMPORT    X                  ; 将 X 声明为外部参照符号。
        ～
        .SECTION   A,DATA,ALIGN=2
        .DATA.L    X                  ; 参照 X 。
        ～
```

外部定义符号或者外部参照符号的声明

.GLOBAL

格　式	`△.GLOBAL△<符号>[:{ 8 | 16}][,…]` 不能记述标号。

说　明

.GLOBAL 声明外部定义符号或者外部参照符号。

在参照其它源程序内定义的符号时或者在从其它源程序参照源程序内定义的符号时声明。

本控制指令将源程序内没有定义的符号作为外部参照符号，并且将源程序内定义的符号作为外部定义符号。

以下的符号能被指定为外部定义符号:

- 具有绝对值的符号
- 具有地址值的符号

但是，不能指定用.ASSIGN 控制指令定义的符号和具有虚段地址值的符号。

另外，通过给符号名附加存取长度（:8 或者:16），以 8 位或者 16 位绝对寻址方式对该符号进行存取。但是，对于前方参照符号，忽视指定的存取长度。

本控制指令的声明与存取长度的有无无关，最初设定的符号有效，第 2 次以后的声明无效。另外，如果在本控制指令前指定 ABS8/NOABS8 控制指令，就遵循 ABS8/NOABS8 控制指令的指示。

例

```
        .CPU        2600A
        .GLOBAL     PROG1          ; 将 PROG1 声明为外部定义符号。
        .GLOBAL     PROG2          ; 将 PROG2 声明为外部参照符号。
;
        .SECTION    A,CODE,ALIGN=2
PROG1:
        MOV.L       ER0,ER1
        JSR         @PROG2:24
        MOV.L       ER1,ER2
        RTS
;
```

位数据名的外部定义符号的声明

.BEXPORT

格　式　△.BEXPORT△<符号>[,…]

不能记述标号。

说　明　.BEXPORT 声明.BEQU 指定的位数据名的外部定义符号。

在从其它源程序参照源程序内定义的.BEQU 符号时声明。

例　在文件 B 中参照文件 A 定义的符号的例子。

- 文件 A

```
          .CPU        2600A:32
          .BEXPORT    AD1B0           ; 将 AD1B0 声明为外部定义符号。

AD1       .EQU        H'FFFFFF00
AD1B0     .BEQU       0,AD1
        ～
```

- 文件 B

```
          .BIMPORT    AD1B0           ; 将 AD1B0 声明为外部参照符号。
        ～
          .SECTION    A,CODE,ALIGN=2
          BSET.B      AD1B0           ; 参照 AD1B0。
        ～
```

位数据名的外部参照符号的声明

.BIMPORT

格　式　　△.BIMPORT△<符号>[,…]

不能记述标号。

说　明　　.BIMPORT 声明.BEQU 指定的位数据名的外部参照符号。

在参照其它源程序内定义的.BEQU 符号时声明。

如果在定义.BIMPORT 后用.BEQU 以外的指令定义符号，就输出警告。同样地，如果在定义.BEQU 后用.BIMPORT 定义符号，也输出警告。

外部参照.BEQU 符号时，需要在定义该符号的源程序中进行外部定义符号的声明（.BEXPORT）。

例　　在文件 B 中参照文件 A 定义的符号的例子。

- 文件 A

```
          .BIMPORT   AD1B0          ; 将 AD1B0 声明为外部参照符号。
        ～
          .SECTION   A,CODE,ALIGN=2
          BSET.B     AD1B0          ; 参照 AD1B0。
        ～
```

- 文件 B

```
          .CPU       2600A:32
          .BEXPORT   AD1B0          ; 将 AD1B0 声明为外部定义符号。

AD1       .EQU       H'FFFFFF00
AD1B0     .BEQU      0,AD1
        ～
```

8 位短绝对地址符号的指定

.ABS8
.NOABS8

格　式　　△.ABS8△[<符号>[,…]]

不能记述标号。

△.NOABS8

不能记述标号。

说　明　　.ABS8 指定以 8 位绝对寻址方式存取的符号。如果只指定.ABS8，就以本控制指令后的全部外部参照/定义符号为对象。

如果指定.NOABS8，就从对象排除本控指令后的被指定为 8 位绝对寻址方式存取的全部外部参照/定义符号。

存取长度的优先顺序

优先顺序		存取长度的格式
高	1	绝对寻址方式的确保长度
↕	2	.IMPORT/.EXPORT/.GLOBAL 控制指令的存取长度 .ABS8/.NOABS8 控制指令
低	3	abs8/abs16 选项

例

```
.CPU     H8SXX:32
.IMPORT  sym1,sym3,sym5
.IMPORT  sym2:16
.IMPORT  sym4:8
MOV.B    @sym1  ,R1H        ;32 位        (无指定)
MOV.B    @sym2  ,R1H        ;16 位        (指定.IMPORT 的存取长度)
MOV.B    @sym3:8,R1H        ; 8 位        (指定确保长度)
MOV.B    @sym4  ,R1H        ; 8 位        (指定.IMPORT 的存取长度)
MOV.B    @sym5  ,R1H        ;32 位        (无指定)
MOV.B    @(sym1+sym2),R1H   ;16 位*       (无指定，混合指定 16 位)
.ABS8    sym1
MOV.B    @sym1  ,R1H        ; 8 位        (指定.abs8)
MOV.B    @sym2  ,R1H        ;16 位        (指定.IMPORT 的存取长度)
MOV.B    @sym3:8,R1H        ; 8 位        (指定确保长度)
MOV.B    @sym4  ,R1H        ; 8 位        (指定.IMPORT 的存取长度)
MOV.B    @sym5  ,R1H        ;32 位        (无指定)
MOV.B    @(sym1+sym2),R1H   ; 8 位*       (混合指定 8 位和 16 位)
.NOABS8
MOV.B    @sym1  ,R1H        ;32 位        (指定.noabs8)
MOV.B    @sym2  ,R1H        ;16 位        (指定.IMPORT 的存取长度)
MOV.B    @sym3:8,R1H        ; 8 位        (指定确保长度)
MOV.B    @sym4  ,R1H        ;32 位        (指定.noabs8)
MOV.B    @sym5  ,R1H        ;32 位        (无指定)
MOV.B    @(sym1+sym2),R1H   ;16 位*       (混合指定 32 位和 16 位)
```

备　考　　如果以绝对寻址方式记述多个外部符号，存取长度就使用最小的存取长度。

目标模块/调试信息的输出控制

.OUTPUT

格　式

```
△.OUTPUT△<输出指定> [,…]
          <输出指定> = { obj | noobj |
                         dbg | nodbg }
```

不能记述标号。

说　明

.OUTPUT 控制目标模块或者调试信息的输出。

(1) 目标模块的输出

控制目标模块的输出。

输出种类如下：

输出种类	输出控制
obj	输出
noobj	抑制输出

(2) 调试信息的输出

控制调试信息的输出。

输出种类如下：

输出种类	输出控制
dbg	输出
nodbg	抑制输出

本指定在输出目标模块时有效。

如果 2 次以上使用.OUTPUT，并且指定的内容发生矛盾，就出错。

有关目标模块和调试信息的输出，汇编程序优先选项的指定。

在省略本控制指令时解释为 obj 和 nodbg。

例

有关目标模块和调试信息的输出，是假设没有指定选项的结果说明。

例 1：

```
        .OUTPUT     OBJ                   ; 输出目标模块。
        ～                                ; 不输出调试信息。
```

例 2：

```
        .OUTPUT     OBJ,DBG               ; 输出目标模块和
        ～                                ; 调试信息。
```

例 3：

```
        .OUTPUT     OBJ,NODBG             ; 输出目标模块。
        ～                                ; 不输出调试信息。
```

备　考

调试信息是在用调试程序进行程序调试时所必需的信息，为目标模块的一部分。

调试信息包含有关源语句行的信息和有关符号的信息等。

符号调试信息的部分输出控制

.DEBUG

格　式　　△.DEBUG△<输出指定>

　　　　　<输出指定> = { ON | OFF }

不能记述标号。

说　明　　.DEBUG 控制符号调试信息的部分输出。

在只要输出源程序符号中的调试所需的符号时使用。如果限定调试所需的符号，输出符号调试信息，就能缩短汇编时间等。

.DEBUG 的指定只在输出目标模块和调试信息时有效。

输出种类如下：

输出种类	输出控制
on	输出
off	抑制输出

能多次指定本控制指令。另外，对于本控制指令的源语句，指定内容有效。

本控制指令只在输出调试信息时有效。

在省略本控制指令时解释为 on。

例

```
.SECTION    A,CODE,ALIGN=2
.DEBUG      OFF                ; 汇编程序从下一个源语句开始
                               ; 不输出符号调试信息。
～
.DEBUG      ON                 ; 汇编程序从下一个源语句开始
                               ; 输出符号调试信息。
～
```

补　充　　符号调试信息表示有关调试信息中的符号的信息。

行号的变更

.LINE

格　式　△.LINE△["<文件名>",]<行号>

不能记述标号。

说　明　.LINE 更改调试信息的文件名和行号。
本控制指令支持 C/C++源级调试。
C/C++编译程序在输出汇编源程序时插入本控制指令。
因此，即使对于编译程序输出的汇编源程序，也能输出 C/C++源文件的行信息。

从指定本控制指令的行的下一行开始，汇编程序管理的文件名和行号变为由本控制指令指定的内容。
本控制指令指定的文件名和行号只在记述本控制指令的文件内有效。

例

```
ch38 -code=asmcode -debug test.c
```

C源程序（test.c）

```
int      func()       /*1*/
{                     /*2*/
  int  i, j;          /*3*/
                      /*4*/
  j=0;                /*5*/
  for (i=1;i<=10;i++) /*6*/
    j+=i;             /*7*/
  return(j);          /*8*/
}                     /*9*/
```

汇编源程序（test.src）

```
          .CPU       2600A:24
          .EXPORT    _func
          .SECTION   P,CODE,ALIGN=2
          .LINE      "/asm/test.c",1
_func:                   ; function: func
          .LINE      2
          .LINE      5
          .LINE      6
          SUB.L      ER0,ER0
          MOV.B      #1,R0L
          .LINE      6
L5:
          .LINE      7
          ADD.W      R0,E0
          .LINE      6
          INC.W      #1,R0
          .LINE      6
          CMP.W      #10,R0
          BLE        L5:8
          .LINE      8
          MOV.W      E0,R0
          .LINE      9
          RTS
          .END
```

位移量长度的设定

.DISPSIZE

格　式　　△.DISPSIZE△<对象项目>=<位数> [,…]

<对象项目> = { FBR | XBR | FRG | XRG | FWD | XTN | ALL }

不能记述标号。

说　明　　.DISPSIZE 设定转移指令的位移量或者在带位移量的寄存器间接方式的位移量为前方参照值或外部参照值时的位移量的默认长度。

本控制指令的对象为没有指定位移量长度（:8、:16、:24、:32）的位移量。

对象项目如下：

指定项目	内容
FBR	前方参照的转移指令
XBR	外部参照的转移指令
FRG	带前方参照位移量的寄存器间接方式
XRG	带外部参照位移量的寄存器间接方式
FWD	同时指定 FBR、FRG
XTN	同时指定 XBR、XRG
ALL	同时指定 FBR、XBR、FRG、XRG

位数如下：

CPU	输出方法 [*1]
H8SX 最高模式	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX 高级模式	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX 中级模式	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8SX 正常模式	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8S/2600 高级模式	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8S/2600 正常模式	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8S/2000 高级模式	FBR=8, 16、XBR=8, 16、FRG=16, 32、XRG=16, 32、FWD=16、XTN=16、ALL=16
H8S/2000 正常模式	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8/300H 高级模式	FBR=8, 16、XBR=8, 16、FRG=16, 24、XRG=16, 24、FWD=16、XTN=16、ALL=16
H8/300H 正常模式	FBR=8, 16、XBR=8, 16、FRG=16、XRG=16
H8/300, H8/300L	FBR=8、XBR=8、FRG=16、XRG=16

【注】下划线为省略指定时的设定。

[*1] 在 H8/300 和 H8/300L 中，固定为 FBR=8、XBR=8、FRG=16、XRG=16，所以没有意义。

能多次指定本控制指令。
指定内容对于本控制指令以后的源语句有效。
FBR 在没有 optimize 选项和 br_relative 选项时有效。

例

```
        .CPU       2600A

        .SECTION   A,CODE,ALIGN=2
        .DISPSIZE  FBR=16            ; [1]
        BRA        sym               ; 和 BRA sym:16 相同。

        .DISPSIZE  FBR=8             ; [2]
        BRA        sym               ; 和 BRA sym:8 相同。
sym:
        MOV.W      R0,R1
```

[1] 将前方参照的转移指令的位移量长度设定为 16 位。
[2] 将前方参照的转移指令的位移量长度设定为 8 位。

汇编列表的输出控制

.PRINT

格 式 △.PRINT△<输出指定> [,…]

<输出指定> = { LIST | NOLIST | SRC | NOSRC |
CREF | NOCREF | SCT | NOSCT }

不能记述标号。

说 明 .PRINT 通过输出指定控制下列各列表的输出/抑制输出：

(1) 汇编列表
(2) 源程序列表
(3) 对照表
(4) 段信息列表

根据各输出指定的控制内容如下：

项目	输出指定 [*1]		含义	控制内容
	输出	输出抑制		
(1)	list	nolist	汇编列表的输出控制 [*2]	输出/抑制输出汇编列表
(2)	src	nosrc	源程序列表的输出控制 [*3 *4]	输出/抑制输出源程序列表
(3)	cref	nocref	对照表的输出控制 [*3 *5]	输出/抑制输出对照表
(4)	sct	nosct	段信息列表的输出控制 [*3 *6]	输出/抑制输出段信息列表

【注】 *1 本指定只限 1 次有效。
*2 在没有指定 list/nolist 选项时有效。
*3 只在输出汇编列表时有效。
*4 在没有指定 source/nosource 选项时有效。
*5 在没有指定 cross_reference/nocross_reference 选项时有效。
*6 在没有指定 section/nosection 选项时有效。

如果 2 次以上使用.PRINT，并且指定的内容发生矛盾，就出错。

例

```
        .PRINT      LIST, SRC, NOCREF, NOSCT
;
        .SECTION    A,CODE,ALIGN=2
START
        MOV.W       R0,R1
        MOV.W       R0,R2
```

只将源程序输出到汇编列表。

源程序列表的部分输出控制

.LIST

格　式

```
△.LIST△<输出指定> [,…]
        <输出指定> = { ON   | OFF    |
                       COND | NOCOND | DEF | NODEF | CALL | NOCALL |
                       EXP  | NOEXP  | STR | NOSTR | CODE | NOCODE }
```

不能记述标号。

说　明

.LIST 根据输出指定进行如下输出：
(1) 源语句的部分输出
(2) 预处理器功能的源语句的部分输出
(3) 目标码显示行的部分输出

根据各输出指定的控制内容如下：

项目	输出指定		含义	控制内容
	输出	抑制输出		
(1)	on	off	源语句的输出控制	本指令以后的源语句
(2)	cond	nocond	条件不成立的输出控制 *1	.AIF、.AIFDEF 的不成立部分
	def	nodef	定义的输出控制 *1	宏定义部分 .AREPEAT、.AWHILE 定义部分 .INCLUDE 控制语句 .ASSIGNA、.ASSIGNC 控制语句
	call	nocall	调用的输出控制 *1	宏调用语句 .AIF、.AIFDEF、.AENDI 控制语句
	exp	noexp	展开的输出控制 *1	宏展开部分 .AREPEAT、.AWHILE 展开部分
	str	nostr	结构化的输出控制 *1	结构化汇编展开部分
(3)	code	nocode	目标码显示行的输出控制 *1	控制指令的目标码显示超过源语句的行数的部分

【注】*1　本指定在没有指定 show/noshow 选项时有效。

本控制指令不显示在源程序列表。
能多次指定本控制指令，指定内容对于本控制指令以后的源语句有效。

例

```
       .PRINT     list
;
       .list      off                 ; ········· [1]
       .include   "bbb.h"             ;
       .list      on                  ; ········· [2]
       .section   A,CODE,ALIGN=2
START
      MOV.W       R0,R1
      MOV.W       R0,R2
```

抑制部分的源语句输出。 [1]～[2]之间的源语句不输出到源程序列表。

汇编列表行数/列数的设定

.FORM

格　式　　△.FORM△<长度指定>[,…]

　　　　<长度指定> = { LIN = <行数> | COL = <列数> }

不能记述标号。

说　明　　.FORM 设定汇编列表的每 1 页的行数和每 1 行的列数。

行数和列数的指定如下：

- 指定常数值。

  并且

- 指定时，不使用前方参照符号。

<行数>、<列数>的允许值的范围和既不指定.FORM 也不指定选项时的解释如下：

指定内容	含义	允许值 *3	没有指定时
LIN＝<行数>	每 1 页的行数 *1	20 ～ 255	60
COL＝<列数>	每 1 行的列数 *2	79 ～ 255	132

【注】 *1　在没有指定 lines 选项时有效。

　　　*2　在没有指定 columns 选项时有效。

　　　*3　在指定范围以外的值时，小于 20 时指定 20，大于 255 时指定 255。此时。不显示错误。

有关汇编列表的行数和列数，汇编程序优先选项的指定。

在 1 个源程序中，能多次使用.FORM。

设定的长度指定对本控制指令以后的页有效。

例　　有关汇编列表的行数和列数的设定，是假设没有指定选项的结果说明。

```
～
.FORM  LIN=60,COL=200           ; 从此页开始将汇编列表的每 1 页设定为
                                ; 60 行。
                                ; 另外，从此行开始将汇编列表的每 1 行
                                ; 设定为 200 列。
～
.FORM  LIN=55,COL=150           ; 从此页开始将汇编列表的每 1 页设定为
                                ; 55 行。
                                ; 另外，从此行开始将汇编列表的每 1 行
                                ; 设定为 150 列。
～
```

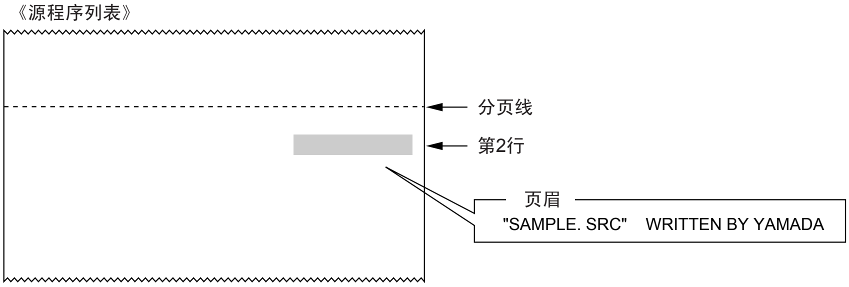
源程序列表页眉的设定

.HEADING

格　式	△.HEADING△"<字符串>" 不能记述标号。
说　明	.HEADING 设定显示在源程序列表页眉的标题。 页眉能设定 60 个字符以内的字符串。但是，即使超过 60 个字符，也不输出错误信息。 将字符用双引号(")括起来指定字符串。 将双引号指定为字符时，连续记述 2 个。 在 1 个源程序中，能多次使用.HEADING。 如果本控制指令在列表的第 1 行，就从此页显示页眉的标题；如果设定在页的第 2 行以后，就从下一页开始显示页眉的标题。

例

```
∼
.HEADING    """SAMPLE.SRC""  WRITTEN BY YAMADA"
∼
```



源程序列表的换页

.PAGE

格　式　　△.PAGE
不能记述标号。

说　明　　.PAGE 换页源程序列表。
本控制指令在列表的第 1 行时，此换页指定无效。
另外，在源程序列表上不显示本控制指令。
本控制指令在输出源程序列表时有效。

例

```
        ～
        .PRINT      LIST
        .SECTION    A,CODE,ALIGN=2
START
        MOV.W       R0,R1
        MOV.W       R0,R2

;
        .PAGE
        .SECTION    B,DATA,ALIGN=2
DAT
        .DATA.W     H'0001
        .DATA.W     H'0002
        ～
```

《源程序列表》

```
    4  00000000  0D01               4             MOV.W      R0,R1
    5  00000002  0D02               5             MOV.W      R0,R2









--------------------------------------------------------------------------------  ←换页

  *** H8S,H8/300 ASSEMBLER Ver. 4.0 ***   07/18/00 21:28:14
  PROGRAM NAME =

    9  00000000                     9             .SECTION   B,DATA,ALIGN=2
   10  00000000                    10      DAT
   11  00000000  0001              11             .DATA.W    H'0001
   12  00000002  0002              12             .DATA.W    H'0002
```

源程序列表的空行输出

.SPACE

格　式　　△.SPACE[△<行数>]

不能记述标号。

说　明　　.SPACE 将指定行数的空行输出到源程序列表。

如果省略操作数，就输出 1 行空行。

行数的指定如下：

- 指定常数值。

  并且

- 指定时，不使用前方参照符号。

行数的允许值为 1～50。

小于 1 时为 1；大于 50 时为 50。此时，不显示错误。

通过本控制指令输出的空行没有行号等的显示。

另外，在输出空行后发生换页时，汇编程序不输出换页后的空行。

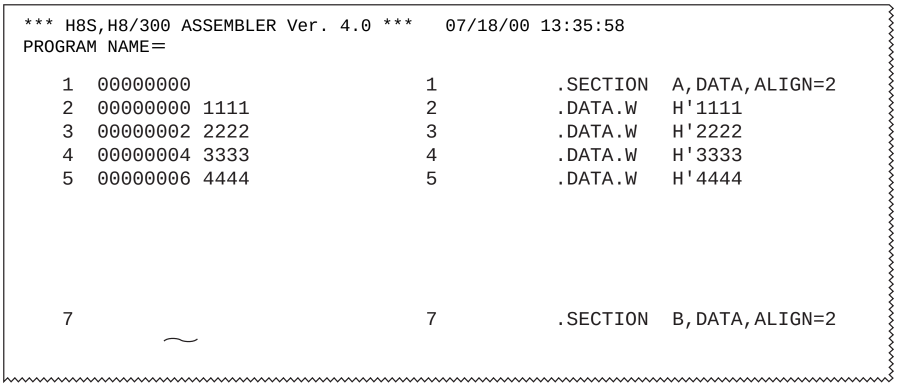
在源程序列表上不显示本控制指令。

本控制指令在输出源程序列表时有效。

例

```
        .SECTION    A,DATA,ALIGN=2
        .DATA.W     H'1111
        .DATA.W     H'2222
        .DATA.W     H'3333
        .DATA.W     H'4444              ; 在替换段的位置
        .SPACE  5                       ; 插入 5 行空行。
        .SECTION    B,DATA,ALIGN=2
        ～
```

《源程序列表》



```
*** H8S,H8/300 ASSEMBLER Ver. 4.0 ***   07/18/00 13:35:58
PROGRAM NAME＝

   1  00000000                        1          .SECTION   A,DATA,ALIGN=2
   2  00000000 1111                   2          .DATA.W    H'1111
   3  00000002 2222                   3          .DATA.W    H'2222
   4  00000004 3333                   4          .DATA.W    H'3333
   5  00000006 4444                   5          .DATA.W    H'4444






   7                                  7          .SECTION   B,DATA,ALIGN=2
             ～
```

目标模块名的设定

.PROGRAM

格　式　　△.PROGRAM△<目标模块名>

不能记述标号。

说　明　　.PROGRAM 设定目标模块名。

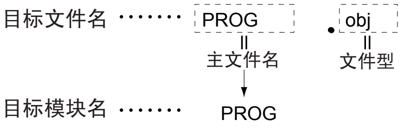
目标模块名是优化连接编辑程序为了识别目标模块所需要的名称。

目标模块名的命名方法和符号的命名方法相同。

汇编程序区别目标模块名的英文大小写字符。

本控制指令的最初设定有效，汇编程序忽视第 2 次以后的设定。

没有本控制指令的设定时，设定默认（隐含）的目标模块名。默认值为目标文件（目标模块的输出文件）的主文件名。



即使目标模块名与程序中使用的符号名重复也没关系。

例

```
      .PROGRAM     PROG1                ; 将 PROG1 设定为
                                        ; 目标模块名。
      ～
```

基数的指定

.RADIX

格　式　　△.RADIX△<基数指定>

　　　　<基数指定>={B ｜ Q ｜ D ｜ H}

不能记述标号。

说　明

.RADIX 对没有指定基数的整数常数进行基数设定。

根据基数指定的内容决定没有基数的整数常数为几进制数。

在指定没有基数的整数常数为 16 进制数（指定基数 H）时，如果整数常数的最高位为 A～F，就必须在此之前加 0。

（汇编程序将以 A～F 开始的记述当作符号）

本控制指令的指定从指定的位置开始有效。

指定内容	没有基数的整数常数
B	2 进制数
Q	8 进制数
D	10 进制数
H	16 进制数

如果省略本控制指令，没有基数的整数常数就为 10 进制数。

例

• 例 1

```
        ～
        .RADIX  D
X:      .EQU        100                 ; 100 是 10 进制数。
        ～
        .RADIX  H
Y:      .EQU        64                  ; 64 是 16 进制数。
        ～
```

• 例 2

```
        ～
        .RADIX  H
Z:      .EQU        0F                  ; 如果只写 F，就被当作符号，
                                        ; 所以在前头加 0。
        ～
```

源程序结束/入口点的指定

.END

格　式　　△.END［△<执行起始地址>］
不能记述标号。

说　明　　.END 表示源程序的结束。
在本控制指令出现处，汇编程序结束汇编处理。
将操作数指定的符号作为入口点。
给符号指定外部定义符号。

例

```
        .EXPORT    START
        .SECTION   P,CODE,ALIGN=4
START:
        ～
        .END       START         ; 声明源程序的结束。
                                 ; 符号 START 为入口点。
```

对指定的符号定义堆栈值

.STACK

格　式　　△.STACK△<符号>=<堆栈值>
不能记述标号。

说　明　　对符号定义由栈解析工具参照的堆栈使用量。
对于 1 个符号能定义的堆栈值只有 1 次有效。如果指定 2 次以上，该定义就无效。
另外，能指定的堆栈值只为 H’00000000～H’FFFFFFFE 范围内的 2 的倍数，在指定其它值时，该定义无效。
堆栈值的指定如下：
- 指定常数值。
  并且
- 指定时，不使用前方参照符号、外部参照符号和相对地址符号。

例

```
                 ～
        .STACK  SYMBOL=H’100
                 ～
```

11.4　文件包含功能

文件包含是将其他源文件取入要汇编的源文件的功能（以下将被取入的源文件称为 include 文件）。
作为有关文件包含功能的控制语句，有.INCLUDE 控制语句。
将指定的 include 文件取到记述了.INCLUDE 控制语句的位置。

例：

源程序

```
        .INCLUDE    "FILE. H"

        .SECTION    CD1,CODE,ALIGN＝2
        MOV.W       #ON, R0

        ～
```

include文件　FILE. H

```
ON:     . EQU    1
OFF:    . EQU    0
```

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

文件包含结果（源列表）

```
        .INCLUDE    "FILE. H"
ON:          . EQU      1
OFF:         . EQU      0

        .SECTION    CD1, CODE, ALIGN＝2
        MOV.W       #ON, R0

        ～
```

指定 include 文件的包含

.INCLUDE

格　式　　△.INCLUDE△"<文件名>"

不能记述标号。

说　明　　.INCLUDE 取入指定的 include 文件。

作为文件名，在只指定主文件名时，没有文件扩展名的文件名有效（汇编程序不假设文件扩展名）。

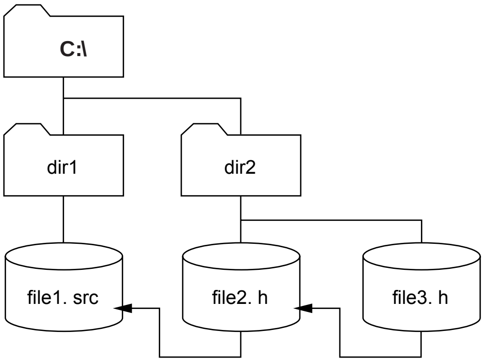
能以包含目录路径名的形式指定文件名。

以绝对路径（根目录开始的路径）或者相对路径（当前目录开始的路径）指定目录。

也能在 include 文件中取入其它文件。包含能嵌套 30 层。

能通过 include 选项更改由.INCLUDE 指定的目录名。

例　　在目录为如下图结构时，进行以下操作：



从根目录(c:\)启动汇编程序

输入源文件为 c:\dir1\file1.src

file1.src 包含 file2.h

file2.h 包含 file3.h

启动命令如下：

```
  >asm38 c:\dir1\file1.src (RET)
```

file1.src 需要以下 include 控制语句。

```
  .INCLUDE "dir2\file2.h"  ; \为当前目录（指定相对路径）。
  或者
  .INCLUDE "\dir2\file2.h" ; 指定绝对路径
```

file2.h 需要以下 include 控制语句。

```
  .INCLUDE "file3.h"   ; \dir2 为当前目录（指定相对路径）。
  或者
  .INCLUDE "\dir2\file3.h" ; 指定绝对路径
```

【注】 为 UNIX 时，必须将反斜线(\)改为斜线(/)。

11.5　条件汇编功能

11.5.1　条件汇编功能的概要

条件汇编功能简单实现如下汇编：

- 将源程序的字符串置换为其它字符串
- 根据条件决定是否对一部分源程序进行汇编
- 对一部分源程序重复展开进行汇编

（1）预处理器变量

用于记述汇编条件的变量称为预处理器变量。
预处理器变量的型有整型和字符型。

（a）整型的预处理器变量

通过.ASSIGNA 控制语句或者 assigna 选项定义整数值（能用.ASSIGNA 控制指令重新定义）。
参照时，在预处理器变量的前头加“\”和“&”<\&>。
例：

```
FLAG: .ASSIGNA  1                              ; 给 FLAG 设定整数值 1。
       ～
      .AIF \&FLAG EQ 1                         ; 和.AIF 1 EQ 1 相同。
      MOV.W    R0,R1                           ; 汇编 MOV.W R0,R1。
      .AENDI
       ～
```

（b）文字型的预处理器变量

通过.ASSIGNC 控制语句或者 assignc 选项定义字符串（能用.ASSIGNC 控制指令重新定义）。
参照时，给预处理器变量的前头加“\”和“&”<\&>。
例：

```
FLAG: .ASSIGNC  "ON"                           ; 给 FLAG 设定字符串 ON。
       ～
      .AIF "\&FLAG" EQ "ON"                    ; 和.AIF "ON" EQ "ON"相同
      MOV.W         R0,R1                      ; 汇编 MOV.W R0,R1。
      .AENDI
       ～
```

（2）置换符号

用.DEFINE 控制语句定义。
能通过符号的指定置换一部分源程序。
编程如下：
例：

```
SYM1: .DEFINE  "R1"
       ～
      MOV.W   SYM1,R0                          ; 被置换为 MOV.W R1,R0。
       ～
```

（3）条件汇编

能根据条件决定是否对一部分源程序进行汇编。条件汇编的条件有用关系运算符判断的比较型条件汇编和用置换符号判断的定义型条件汇编。

（a）比较型条件汇编

比较型条件汇编根据条件的成立与否决定汇编范围。编程如下：

```
                  ～
 .AIF        比较型条件
      条件成立时汇编的部分
 .AELIF      比较型条件
      条件成立时汇编的部分                            可省略此部分
 .AELSE
      全部条件都不成立时汇编的部分
 .AENDI
                  ～
```

例：

```
    ～
 .AIF  "\&FLAG" EQ "ON"
 MOV.W    R0,R2                       ; FLAG为"ON"时汇编。
 MOV.W    R1,R3                       ;
 .AELSE
 MOV.W    R2,R0                       ; FLAG不为"ON"时汇编。
 MOV.W    R3,R1                       ;
 .AENDI
    ～
```

（b）定义型条件汇编

定义型条件汇编根据是否定义了置换符号决定汇编范围。编程如下：

```
               ～
.AIFDEF   定义型条件
    定义了条件置换符号时汇编的部分
.AELSE
    不定义条件置换符号时汇编的部分                     可省略此部分
.AENDI
               ～
```

例：

```
    ～
 .AIFDEF   FLAG
 MOV.W    R0,R3                                   ; 在由.AIFDEF 控制语句参照前，
 MOV.W    R1,R4                                   ; 用.DEFINE 控制语句定义 FLAG
 MOV.W    R2,R5                                   ; 时汇编。
 .AELSE
 MOV.W    R3,R0                                   ; 在由.AIFDEF 控制语句参照前，
 MOV.W    R4,R1                                   ; 没有用.DEFINE 控制语句定义 FLAG
 MOV.W    R5,R2                                   ; 时汇编。
 .AENDI
    ～
```

（4）重复展开

能对一部分源程序进行指定次数的重复展开，汇编被展开的部分。编程如下：

```
                ～
.AREPEAT    重复次数
    重复对象部分
.AENDR
                ～
```

例：

```
 MOV.B   R1L,R1H
.AREPEAT 2                                   ; 指定重复次数。
ADD.B   R0L,R1L
ADD.B   R2L,R3L
.AENDR
ADD.B   R3L,R1H
```

[展开后]

```
MOV.B   R1L,R1H
ADD.B   R0L,R1L  ⎫
ADD.B   R2L,R3L  ⎬ 被展开的部分
ADD.B   R0L,R1L  ⎪
ADD.B   R2L,R3L  ⎭
ADD.B   R3L,R1H
```

重复 2 次展开.AREPEAT～.AENDR 之间的源语句，汇编被展开的部分。

（5）条件重复展开

条件成立时，能对一部分源程序重复展开进行汇编。编程如下：

```
                     ～
    .AWHILE          重复条件
        重复对象部分
    .AENDW
                     ～
```

例：

```
         MOV.B  R0H,R0L
 COUNT   .ASSIGNA   2                ; 给 COUNT 设定 2。
         .AWHILE    \&COUNT NE 0     ; COUNT≠0 时，进行展开。
         ADD.B  R0L,R1L
         ADD.B  R0L,R2L
         INC.B  R0L
 COUNT   .ASSIGNA   \&COUNT-1        ; COUNT 减去 1。
         .AENDW
         MOV.B  R0L,@SP
```

[展开后]

```
         MOV.B      R0H,R0L
         ADD.B      R0L,R1L
         ADD.B      R0L,R2L
         INC.B      R0L
 COUNT   .ASSIGNA   \&COUNT-1
         ADD.B      R0L,R1L
         ADD.B      R0L,R2L
         INC.B      R0L
 COUNT   .ASSIGNA   \&COUNT-1
         MOV.B      R0L,@SP
```

被展开的部分

只在 COUNT≠0 时重复展开.AWHILE～.AENDW 之间的源语句，汇编被展开的部分。

11.5.2　有关条件汇编功能的控制语句

条件汇编功能的控制语句一览表如表 11.15 所示：

表 11.15　条件汇编功能一览表

分类	助记符	功能
有关变量定义	.ASSIGNA	定义整型预处理器变量。 能重新定义。
	.ASSIGNC	定义字符型预处理器变量。 能重新定义。
	.DEFINE	定义预处理器置换字符串。 不能重新定义。
有关条件转移	.AIF .AELIF .AELSE .AENDI	根据条件决定是否对一部分源程序进行汇编。 条件成立时，汇编.AIF 后的源程序；不成立时，汇编 AELIF 或者.AELSE 后的源程序。
	.AIFDEF .AELSE .AENDI	根据置换符号的定义决定是否对一部分源程序进行汇编。 定义了置换符号时，汇编.AIFDEF 后的源程序；没有定义时，汇编.AELSE 后的源程序。
有关重复展开	.AREPEAT .AENDR	对一部分源程序（在.AREPEAT 和.AENDR 之间）进行指定次数的重复展开，汇编被展开的部分。
	.AWHILE .AENDW	条件成立时，对一部分源程序（在.AWHILE 和.AENDW 之间）重复展开进行汇编。
	.EXITM	中断.AREPEAT、.AWHILE 的重复展开。
其它	.AERROR	进行预处理器展开时的错误处理。
	.ALIMIT	设定预处理器的.AWHILE 展开的上限值。

整型预处理器变量的定义

.ASSIGNA

格　式　　<预处理器变量名>[:]△.ASSIGNA△<值>

说　明　　.ASSIGNA 控制语句定义预处理器变量。

预处理器变量名的命名方法与符号的命名方法相同。

另外，预处理器变量名的最大字符数为 32 个字符，区别英文大小写字母。

能通过.ASSIGNA 重新定义由.ASSIGNA 定义的预处理器变量。

用以下格式指定预处理器变量的值：

- 常数（整数常数、字符常数）
- 已定义的预处理器变量
- 有关上述项的表达式

定义的预处理器变量对本控制语句以后的源语句有效。

能在以下的位置参照预处理器变量。

- .ASSIGNA、.ASSIGNC 控制语句
- .AIF、.AELIF、.AREPEAT、.AWHILE 控制语句
- 宏体（.MACRO～.ENDM 之间的源语句）

参照预处理器变量时，必须在前面加“\”和“&”<\&>记述。

　\&预处理器变量名[']

在要明确区别预处理器变量名和源语句时，记述单引号（'）。

用选项定义了预处理器字符串时，对同名的预处理器变量的.ASSIGNA 无效。

例

```
FLAG    .ASSIGNA   1                 ; 给 FLAG1 设定 1。
;
        .SECTION   A,CODE,ALIGN=2
START
        .AIF       \&FLAG EQ 1       ; 和.AIF 1 EQ 1 相同。
        MOV.W      R0,R2
        .AENDI
        .AIF       \&FLAG EQ 2       ; 和.AIF 1 EQ 2 相同。
        MOV.W      R1,R2
        .AENDI
;
FLAG    .ASSIGNA   2                 ; 将 FLAG 的值更改为 2。
;
        .AIF       \&FLAG EQ 1       ; 和.AIF 2 EQ 1 相同。
        MOV.W      R0,R2
        .AENDI
        .AIF       \&FLAG EQ 2       ; 和.AIF 2 EQ 2 相同。
        MOV.W      R1,R2
        .AENDI
```

通过.AIF 参照整型预处理器变量 FLAG。

字符型预处理器变量的定义

.ASSIGNC

格　式　　`<预处理器变量名>[:]△.ASSIGNC△"<字符串>"`

说　明　　.ASSIGNC 控制语句定义字符型预处理器变量。

预处理器变量名的命名方法和符号的命名方法相同。

另外，预处理器变量名的最大字符数为 32 个字符，区别英文大小写字母。

能通过.ASSIGNC 重新定义由.ASSIGNC 定义的预处理器变量。

将字符和已定义的预处理器变量用双引号（"）括起来指定字符串。

定义的预处理器变量对本控制语句以后的源语句有效。

能在以下的位置参照预处理器变量。

- .ASSIGNA、.ASSIGNC 控制语句
- .AIF、.AELIF、.AREPEAT、.AWHILE 控制语句
- 宏体（.MACRO～.ENDM 之间的源语句）

在参照预处理器变量时，必须在前面加“\”和“&”<\&>后记述。

　\&预处理器变量名[']

在要明确区别预处理器变量名和源语句时，记述单引号（'）。

用命令行/选项定义了预处理器字符串时，对同名的预处理器变量的.ASSIGNC 无效。

例

```
FLAG1 .ASSIGNC   "ON"                    ; 给 FLAG1 设定"ON"。
;
      .SECTION   A,CODE,ALIGN=2
START
      .AIF       "\&FLAG1" EQ "ON"       ; 和.AIF "ON" EQ "ON"相同。
      MOV.W      R0,R2
      .AENDI
      .AIF       "\&FLAG1" EQ "OFF"      ; 和.AIF "ON" EQ "OFF"相同。
;
FLAG2 .ASSIGNC   "OFF"                   ; 将 FLAG 更改为字符串 OFF。
;
      .AIF       "\&FLAG2" EQ "ON"       ; 和.AIF "OFF" EQ "ON"相同。
      MOV.W      R3,R5
      .AENDI

      .AIF       "\&FLAG2" EQ "OFF"      ; 和.AIF "OFF" EQ "OFF"相同。
      MOV.W      R4,R5
      .AENDI
;
FLAG  .ASSIGNC   "\&FLAG1' AND \&FLAG2"  ; 为了明确区别 FLAG 和 AND,
                                         ; 使用"'"。
                                         ; FLAG 结果为"ON AND OFF"。
```

通过.AIF 参照字符型预处理器变量 FLAG。

预处理器置换字符串的定义

.DEFINE

格　式	<符号>[:]△.DEFINE△"<置换字符串>"

说　明

.DEFINE 控制语句指定将符号置换成对应的字符串。

.DEFINE 和.ASSIGNC 的不同点如下：

(a) 用.ASSIGNC 定义的符号只能用于预处理器语句，而用.DEFINE 定义的符号能用于任何的语句。

(b) 以“\&符号”格式参照.ASSIGNA、.ASSIGNC 定义的符号，而以“符号”格式参照.DEFINE 定义的符号。

(c) 不能重新定义.DEFINE 定义的符号。

(d) 用选项定义了置换符号时，对同名的符号的.DEFINE 无效。

例

```
SYM1: .DEFINE    "R1"
      MOV.W      SYM1,R0                 ; 置换为 MOV.W R1,R0。
```

用.DEFINE 定义了同名的符号时，以 a～f 或者 A～F 开头的 16 进制数为置换对象。必须在非置换对象的前头加 0。

```
A0:   .DEFINE    "0"
      MOV.W      #H’A0,R0                ; 置换为 MOV.W #H’0,R0。
      MOV.W      #H’0A0,R0               ; 没被置换。
```

用.DEFINE 定义了同名符号时，基数（B’、Q’、D’、H’）为置换对象。在定义 B、Q、D、H、b、q、d、h 单个字符的符号时必须注意。

```
B:    .DEFINE    "H"
      MOV.W      #B’10,R0                ; 置换为 MOV.H #H’10,R0。
```

备　考

不能用.DEFINE 控制指令置换以下的控制指令：

.AENDI、.AENDR、.AENDW、.AIFDEF、.END、.ENDM、.ENDF、.ENDI、.ENDS、.ENDW

.AIF, .AELIF, .AELSE, .AENDI

格　式

```
 △.AIF△<项 1>△<关系运算符>△<项 2>
    <.AIF 条件成立时汇编的源语句>
[△.AELIF△<项 1>△<关系运算符>△<项 2>
    <.AELIF 条件成立时汇编的源语句>]
[△.AELSE
    <全部条件不成立时汇编的源语句>]
 △.AENDI
```

不能记述标号。

说　明

在.AIF～.AELIF～.AELSE～.AENDI 之间记述的源语句中，汇编条件成立的部分。
能省略.AELIF 和.AELSE。
另外，在.AIF 和.AELSE 之间能重复指定.AELIF。
能通过各操作指定的操作数如下：

操作	操作数
.AIF	比较型条件
.AELIF	比较型条件
.AELSE .AENDI	不可记述

给<项 1>、<项 2>记述值或者字符串。但是，如果将值和字符串进行比较，条件总是不成立。
用常数或者预处理器变量指定值。
将字符或者预处理器变量用双引号（"）括起来指定字符串。在将双引号（"）指定为字符时，连续记述 2 个双引号（""）。

关系运算符的条件如下：

EQ	项 1 ＝ 项 2
NE	项 1 ≠ 项 2
GT	项 1 ＞ 项 2
LT	项 1 ＜ 项 2
GE	项 1 ≥ 项 2
LE	项 1 ≤ 项 2

【注】字符串比较只为 EQ、NE 时有效。

例

```
.AIF      \&TYPE EQ 1
MOV.W     R0,R3                   ; TYPE 为 1 时汇编。
MOV.W     R1,R4
.AELIF    \&TYPE EQ 2
MOV.W     R0,R2                   ; TYPE 为 2 时汇编。
MOV.W     R1,R3
.AELSE
MOV.W     R0,R4                   ; TYPE 既不为 1 也不为 2 时
MOV.W     R1,R5                   ; 汇编。
.AENDI
```

定义型条件汇编

.AIFDEF, .AELSE, .AENDI

格　式

```
 △.AIFDEF△<置换符号>
    <定义了置换符号时汇编的源语句>
[△.AELSE
    <没有定义置换符号时汇编的源语句>]
 △.AENDI
```

不能记述标号。

说　明

.AIFDIF、.AELSE、.AENDI 根据置换符号的定义决定是否汇编。能省略.AELSE。
能通过各操作指定的操作数如下：。

操作	操作数
.AIFDEF	定义型条件
.AELSE	不可记述
.AENDI	

通过.DEFINE 控制语句或者 define 选项定义置换符号。
用选项定义了记述的置换符号时或者在参照前用本控制语句定义时，条件成立。
另外，如果在参照后用控制语句定义记述的置换符号时或者没有定义时，条件不成立。

例

```
 .AIFDEF   FLAG
MOV.W      R0,R3                  ;在用.DEFINE 控制语句定义 FLAG 时
MOV.W      R1,R4                  ;汇编。
.AELSE
MOV.W      R0,R2                  ; 在没有用.DEFINE 控制语句定义
MOV.W      R1,R3                  ; FLAG 时汇编。
.AENDI
```

重复展开

.AREPEAT, .AENDR

格　式

```
△.AREPEAT△<次数>
   <重复展开后汇编的源语句>
△.AENDR
```

不能记述标号。

说　明

.AREPEAT、.AENDR 为只进行指定次数的重复展开后汇编的控制语句。
能通过各操作指定的操作数如下：

操作	操作数
.AREPEAT	重复展开的次数
.AENDR	不可记述

对记述在.AREPEAT～.AENDR 之间的源语句进行.AREPEAT 指定次数的重复展开后汇编（和重复复制源语句相同，并不在执行时循环）。
用常数或者预处理器变量指定次数。
如果对次数指定 0 以下的值，就不展开。

例

```
      MOV.B      @SP,R0L
      .AREPEAT   3
      SHAL.B     R0L
      .AENDR
      MOV.B      R0L,@SP
```

[展开后]

```
      MOV.B      @SP,R0L
      SHAL.B     R0L
      SHAL.B     R0L
      SHAL.B     R0L
      MOV.B      R0L,@SP
```

带条件的重复展开

.AWHILE, .AENDW

格　式

```
△.AWHILE△<项 1>△<关系运算符>△<项 2>
   <重复展开后汇编的源语句>
△.AENDW
```

不能记述标号。

说　明

.AWHILE、.AENDW 只在条件成立时重复展开。
在.AWHILE 指定的条件成立时，重复展开记述在.AWHILE～.AENDW 之间的源语句后汇编（和重复复制源语句相同，并不在执行时循环）。

对<项 1>、<项 2>记述值或者字符串。但是，如果将值和字符串进行比较，条件总是不成立。
用常数或者预处理器变量指定值。
将字符或者预处理器变量用双引号（"）括起来指定字符串。将双引号（"）指定为字符时，连续记述 2 个双引号（""）。
带条件的重复展开在条件不成立时结束。
如果条件总是成立，就重复 65,535 次或者.ALIMIT 控制语句指定的展开次数，所以必须注意条件的指定。
关系运算符的条件如下：

关系运算符	条件
EQ	项 1 ＝ 项 2
NE	项 1 ≠ 项 2
GT	项 1 ＞ 项 2
LT	项 1 ＜ 项 2
GE	项 1 ≥ 项 2
LE	项 1 ≤ 项 2

【注】字符串的比较只在 EQ、NE 时有效。

例

```
; 只在 COUNT≠0 时重复展开。
COUNT   .ASSIGNA   2                    ; 给 COUNT 设定 2。
        .AWHILE    \&COUNT NE 0         ; 条件为 COUNT≠0。
        ADD.B      R0L,R1L
        ADD.B      R0L,R2L
        INC.B      R0L
COUNT   .ASSIGNA   \&COUNT-1            ; COUNT 减去 1。
        .AENDW

; 只在 STOP≤10 时重复展开。
STOP    .ASSIGNA   0                    ; 给 STOP 置 0。
        .AWHILE    \&STOP LE 10         ; 条件为 STOP≤10。
        ADD.B      R0L,R1L
        ADD.B      R0L,R2L
        INC.B      R0L
STOP    .ASSIGNA   \&STOP+3             ; STOP 加上 3。
        .AENDW
```

展开的中断

.EXITM

格　式

△.EXITM

不能记述标号。

说　明

.EXITM 中断重复展开（.AREPEAT～.AENDR）和带条件的重复展开（.AWHILE～.AENDW）。在各展开中出现本控制语句时，中断展开。

本控制语句也用于宏展开的中断。在组合使用宏指令和重复展开时，必须注意本控制语句的位置。

例

```
COUNT .ASSIGNA    0               ; 给 COUNT 置 0。
      .AWHILE     1 EQ 1          ; 指定无限展开（条件总是成立）。
      ADD.W       R0,R1
      ADD.W       R2,R3

COUNT .ASSIGNA    \&COUNT+1       ; COUNT 加上 1。
      .AIF        \&COUNT EQ 2    ; 条件为 COUNT=2。
      .EXITM                      ; 条件成立，中断.AWHILE。
      .AENDI
      .AENDW
```

如果 COUNT 被更新，.AIF 条件成立，就汇编.EXITM。

在汇编.EXITM 后，立即中断.AWHILE 的展开。

展开结果如下：

```
       ADD.W      R0,R1    … COUNT 为 0 时
       ADD.W      R2,R3
       ADD.W      R0,R1    … COUNT 为 1 时
       ADD.W      R2,R3
```

此后，COUNT 为 2，中断展开。

预处理器展开时的错误处理

.AERROR

格　式　　△.AERROR
不能记述标号。

说　明　　如果汇编.AERROR，就发生错误 670，结束汇编程序。
.AERROR 能用于检查预处理器变量值等。

例

```
      ～
      .AIF  \&FLAG  EQ 1
      ADD.W       R0,R1
      INC.W       R0
      .AELSE
      .AERROR                    ; \&FLAG 不为 1 时，发生错误。
      .AENDI
      ～
```

展开的上限值设定

.ALIMIT

格　式　　△.ALIMIT△<次数>
不能记述标号。

说　明　　通过带条件的重复展开（.AWHILE～.AENDW）设定源语句展开次数的上限值。
用以下格式指定<次数>的值。

- 常数（整数常数、字符常数）
- 已经定义的预处理器变量
- 有关上述项的表达式

如果超过.ALIMIT 指定的上限值，就发生警告 854，中断展开。
不指定.ALIMIT 时，展开次数的上限值为 65,535。
通过本控制指令的重新指定，能更改重复展开次数的上限值。
上限值的重新指定对本控制指令以后的源语句有效。

例

```
COUNT .ASSIGNA    3               ; 给 COUNT 设定 3。
      .ALIMIT     10              ; 给重复的上限值指定 10。
      .AWHILE     \&COUNT NE 4
      ADD.W       R0,R1           ; [1]
      ADD.W       R0,R1           ; [1]
      INC.W       R0              ; [1]
COUNT .ASSIGNA    \&COUNT-1       ; [1]
      .AENDW
```

只在 COUNT≠4 时展开[1]。
展开 10 次后，输出警告 854，中断重复展开。

11.6　宏功能

11.6.1　宏功能的概要

本汇编言語能给程序中常用的一连串处理命名，定义为 1 个指令（宏指令）。此种定义称为宏定义。宏定义的方法如下：

```
                ～
.MACRO          宏名
    宏体
.ENDM
                ～
```

宏名是给宏指令起的名字，宏体为宏指令的内容。

调用使用被定义的宏指令称为宏调用。宏调用的方法如下：

```
                ～
被定义的宏名
                ～
```

宏定义和宏调用的例子如下所示：

例：

```
～
.MACRO   SUM        ; 将进行 R1、R2、R3 合计计算的处理
ADD.W    R2,R1      ; 定义为宏指令 SUM。
ADD.W    R3,R1
.ENDM
～

SUM                 ; 调用宏指令 SUM。
                    ; 展开宏体
                    ;   ADD.W   R2,R1
                    ;   ADD.W   R3,R1
```

可通过参数的使用，更改一部分宏体后展开宏指令。

步骤如下：
（1）宏定义
　　用.MACRO语句定义形式参数（记述在宏名后）。
　　使用形式参数记述宏体（在形式参数的前头加反斜线“\”）。
（2）宏调用
　　指定宏参数调用宏指令。

在展开宏指令时，将形式参数置换为对应的宏参数。

例：

```
～
.MACRO    SUM ARG1      ; 定义形式参数 ARG1。
MOV.W     R1,\ARG1      ; 使用 ARG1 记述宏体。
ADD.W     R2,\ARG1
ADD.W     R3,\ARG1
.ENDM
～
SUM       R0            ;  指定宏参数 R0 调用宏 SUM。
                        ;  用宏参数置换宏体中的形式参数
                        ;  展开  ADD.W    R1,R0
                        ;        ADD.W    R2,R0
                        ;        ADD.W    R3,R0
```

11.6.2　有关宏功能的控制语句

宏功能的控制语句一览表如表 11.16 所示。

表 11.16　宏功能的控制语句一览表

助记符	功能
.MACRO .ENDM	定义宏指令。
.EXITM	中断宏指令的展开。 请参照 11.5.2　.EXITM。

.MACRO, .ENDM

格　式　　△.MACRO△<宏名>[△<形式参数>[,…]]

△.ENDM

<形式参数>：<形式参数名>[=<形式参数的默认值>]

不能记述标号。

说　明　　.MACRO、.ENDM 定义宏。

定义宏指令（宏定义）是指将.MACRO～.ENDM 之间的源语句（宏体）命名为宏指令。

能用各操作指定的操作数如下：

操作	操作数
.MACRO	・宏指令 ・形式参数 记述默认值（可省略）
.ENDM	不能记述

(1) 宏名

宏名是给宏指令起的名字。

要在展开宏指令时置换一部分宏体进行展开时指定。

在进行宏展开（宏调用）时，将形式参数置换为指定的字符串（宏参数）。

在宏体中，给要置换的部分记述形式参数名。宏体中的形式参数的参照方法如下：

\形式参数名[']

在明确区别形式参数名和源语句时记述单引号(')。

(2) 形式参数

能给形式参数设定默认的形式参数。给默认的形式参数指定在宏调用时省略宏参数时的置换字符串。

形式参数的写法和符号名的写法相同。

形式参数名的最大字符数为 32 个字符，区别英文大小写字母。

(3) 默认的形式参数

默认的形式参数包含以下字符时，必须将字符串用双引号（"）或者尖括号括起来（<>）。

- 空格
- 制表符
- 逗号（,）
- 分号（;）
- 双引号（"）
- 尖括号（<>）

在宏展开中，去掉将字符串括起来的双引号（"）和尖括号（<>）后进行置换。

(4) 限制

在以下情况下不能定义宏指令：

- 宏体（.MACRO～.ENDM）
- .AREPEAT～.AENDR 之间
- .AWHILE～.AENDW 之间

不能给宏体记述.END。

不能给.ENDM 的标号记述符号。

如果给.ENDM 标号记述符号，就忽视.ENDM，不显示错误。

例

```
; 将进行 R3、R4、R5 合计计算的处理定义为宏指令 SUM。
        ～
        .MACRO    SUM
        MOV.W     R3,R1
        ADD.W     R4,R1
        ADD.W     R5,R1
        .ENDM
        ～
        SUM                             ; 调用宏指令 SUM。
                                        ; 展开宏体
                                        ;   MOV.W   R3,R1
                                        ;   ADD.W   R4,R1
                                        ;   ADD.W   R5,R1
```

```
; 将形式参数 P1、P2、P3 的加法结果输出到 R0 的处理定义为宏指令 TOTAL。
        ～
        .MACRO    TOTAL P1,P2,P3
        MOV.W     \P1,R0
        ADD.W     \P2,R0
        ADD.W     \P3,R0
        .ENDM
        ～
        TOTAL     R1,R2,R3              ; 调用宏指令 TOTAL。
                                        ; 展开宏体
                                        ;   MOV.W   R1,R0
                                        ;   ADD.W   R2,R0
                                        ;   ADD.W   R3,R0
```

11.6.3　宏体

在.MACRO 和.ENDM 之间记述的一连串源语句称为宏体。通过宏调用（调用宏指令），展开和汇编宏体。本节说明宏体具有的功能和记述方法。

（1）形式参数的参照

在宏展开中，给要置换为宏参数的部分记述形式参数。形式参数的参照方法如下：

\形式参数［'］

在要明确区别形式参数名和源语句时，记述单引号（'）。

例：

```
          .MACRO    PLUS1 P,P1              ; P、P1 为形式参数。
          ADD.W     #1,\P1                  ; 参照形式参数 P1。
          .SDATA    "\P'1"                  ; 参照形式参数 P。
          .ENDM
          PLUS1     R,R1                    ; 展开 PLUS1。
[展开结果]
          ADD.W     #1,R1                   ; 参照形式参数 P1。
          .SDATA    "R1"                    ; 参照形式参数 P。
```

（2）预处理器变量（.ASSIGNA、.ASSIGNC）的参照

在宏体中能参照预处理器变量。预处理器变量的参照方法如下：

\&预处理器变量名［'］

在要明确区别预处理器变量和源语句时，记述单引号（'）。

例：

```
          .MACRO    PLUS1
          ADD.W     #1,R\&V1                ; 参照预处理器变量 V1。
          .SDATA    "\&V'1"                 ; 参照预处理器变量 V。
          .ENDM
   V:     .ASSIGNC  "R"                     ; 定义预处理器变量 V。
   V1:    .ASSIGNA  1                       ; 定义预处理器变量 V1。
          PLUS1                             ; 展开 PLUS1。
[展开结果]
          ADD.W     #1,R1                   ; 参照预处理器变量 V1。
          .SDATA    "R1"                    ; 参照预处理器变量 V。
```

（3）宏生成号

宏体中有标号时，如果多次进行宏调用，符号名就会重复。为了避免这种状况，必须使用宏生成号。宏生成号是宏展开所固有的 5 位 10 进制数（00000～99999）。必须如下记述宏生成号。

\@

作为符号名的一部分，如果记述宏生成号，就在每次调用宏时为固有的符号名，并且能避免重复。能对 1 个宏体记述 2 个以上的宏生成号，但是在 1 次宏调用中展开相同的宏生成号。另外，将宏生成号展开为数字，所以不能记述在符号名的前头。

例：

```
         .MACRO MCO Rn
         MOV.W  \Rn, \Rn
         BEQ    LAB\@:8
         MOV.W  #H'0,\Rn
LAB\@:
         INC.W  \Rn
         .ENDM

         MCO    R1                          ; 每次展开 MCO 时
         MCO    R2                          ; 生成不同符号。
```

[展开结果]

```
         MOV.W  R1,R1
         BEQ    LAB00000:8
         MOV.W  #H'0,R1
LAB00000:
         INC.W  R1
;
         MOV.W  R2,R2
         BEQ    LAB00001:8
         MOV.W  #H'0,R2
LAB00001:
         INC.W  R2
```

（4）宏处理除外

如果在宏体中有反斜线“\”，就为宏置换处理的对象。因此，要将反斜线作为 ASCII 字符记述时，需要进行宏置换处理除外的处理。宏处理除外的写法如下：

\（宏处理除外的字符串）

在宏展开中去掉反斜线“\”和括号。

例：

```
.MACRO BACK_SLASH_SET
\(MOV.W   #"\",R0)；将“\”展开为 ASCII 字符。
.ENDM

BACK_SLASH_SET
```

[展开结果]

```
MOV.W    #"\",R0                         ；将“\”展开为 ASCII 字符。
```

（5）宏内注释

在宏展开中不希望展开宏体的注释时，记述宏内注释。宏内注释的写法如下：

\;注释

例：

```
.MACRO COMMENT_IGNORE Rn
MOV.W  \Rn,@-SP                          \;保存\Rn 的数据。
.ENDM

COMMENT_IGNORE R1
```

[展开结果]

```
MOV.W  R1,@-SP
```

（6）字符列操作函数

能在宏体中记述字符串操作函数。有如下的字符串操作函数：

.LEN 函数　　：　字符串的长度
.INSTR 函数　：　字符串的检索
.SUBSTR 函数 ：　字符串的抽出

11.6.4　宏调用

展开由宏定义进行定义的宏指令称为宏调用。

用以下的格式记述宏调用：

[<符号>[:]]△<宏名> [△<宏参数>[,…]]
<宏参数>[=<形式参数名>]=<字符串>

在调用宏以前宏定义（.MACRO）宏名。

给宏参数指定宏展开中置换的字符串。

此时，需要用与宏名对应的宏定义（.MACRO）声明形式参数。

（1）宏参数的指定方法

宏参数的指定方法有位置指定和关键字指定。

（2）位置指定

位置指定是将宏定义（.MACRO）声明的形式参数和宏参数的排列顺序一致的指定方法。

（3）关键字指定

关键字指定是对由宏定义（.MACRO）声明的形式参数的形式参数名，用等号（=）分开指定的方法。

（4）宏参数的记述

宏参数中包含以下字符时，必须将字符串用双引号（"）或者尖括号（<>）括起来。

- 空格
- 制表符
- 逗号（,）
- 分号（;）
- 双引号（"）
- 尖括号（<>）

在宏展开中，去掉将字符串括起来的双引号和尖括号后进行替换。

例：

```
       .MACRO    SUM    FROM=0,TO=6          ; 定义宏指令 SUM、形式参数 FROM、TO。

       MOV.W     R\FROM,R0
COUNT  .ASSIGNA  \FROM+1
       .AWHILE   \&COUNT LE \TO
       AND.W     R\&COUNT,R0                 ; 用形式参数记述宏体。
COUNT  .ASSIGNA  \&COUNT+1
       .AENDW
       .ENDW

       SUM       0,3                         ; 任何一个的结果都相同。
       SUM       TO=3
```

用宏参数置换宏体中的形式参数，展开结果如下：

[展开结果]

```
       MOV.W    R0,R0
       AND.W    R1,R0
       AND.W    R2,R0
       AND.W    R3,R0
```

11.6.5　字符串操作函数

能在宏体中使用的字符串操作函数一览表如**表 11.17** 所示：

表 11.17　字符串操作函数一览表

.LEN	返回字符串的长度。
.INSTR	检索字符串。
.SUBSTR	抽出字符串。

字符串的长度

.LEN

格　式　　`.LEN[△]("<字符串>")`

说　明　　.LEN 计算字符串长度，将结果置为省略基数的 10 进制数。

将字符用双引号（"）括起来指定字符串。

将双引号指定为字符时，连续记述 2 个。

能给字符串指定宏的形式参数和预处理器变量。

```
  .LEN("\形式参数名")
  .LEN("\&预处理器变量名")
```

只有宏体（.MACRO～.ENDM）能记述本函数。

例

```
        .MACRO    RESERVE_LENGTH P1
        .SRES     .LEN("\P1")
        .ENDM

        RESERVE_LENGTH  ABCDEF
        RESERVE_LENGTH  ABC

[展开结果]
        .SRES     6                     ; "ABCDEF"的字符数是 6。
        .SRES     3                     ; "ABC"的字符数是 3。
```

字符列的检索

.INSTR

格　式　　`.INSTR[△]("<字符串 1>","<字符串 2>"[,<检索开始位置>])`

说　明　　.INSTR 检索字符串 1 中是否含有字符串 2，将字符串起始位置为 0 的检索位置置换为省略基数的 10 进制数。
字符串 1 中不含有字符串 2 时，置换为-1。

将字符用双引号（"）括起来指定字符串。
将双引号（"）指定为字符时，连续记述 2 个。

用以字符串 1 起始位置为 0 的数值指定检索开始位置。
在省略时，设定 0。

能在字符串、检索开始位置中指定宏的形式参数和预处理器变量。

```
 .INSTR("\形式参数名", .... )
 .INSTR("\&预处理器变量名", .... )
```

只有宏体（.MACRO～.ENDM）能记述本函数。

例

```
        .MACRO    FIND_STR   P1
        .DATA.W   .INSTR("ABCDEFG","\P1",0)
        .ENDM

        FIND_STR  CDE
        FIND_STR  H
```

[展开结果]

```
        .DATA.W   2                     ; 从"ABCDEFG"的第 2 个字符（起始位置
                                        ; 为 0）起有"CDE"。
        .DATA.W   -1                    ; "ABCDEFG"中没有"H"。
```

字符列的抽出

.SUBSTR

格　式　　.SUBSTR[△]("<字符串>",<抽出开始位置>,<抽出长度>)

说　明　　.SUBSTR 从字符串起始位置为 0 的抽出开始位置，将抽出长度的字符串抽出，置换为用双引号(")括起来的字符串。
将字符用双引号括起来指定字符串。
将双引号指定为字符时，连续记述 2 个。

能指定 0 以上的抽出开始位置。
在抽出开始位置和抽出长度不适当时，置换为空字符("")。
另外，能指定 1 以上的抽出长度。

能给抽出开始位置和抽出长度指定宏的形式参数和预处理器变量。

```
  .SUBSTR("\形式参数名", .... )
  .SUBSTR("\&预处理器变量名", .... )
```

只有宏体（.MACRO～.ENDM）能记述本函数。

例

```
        .MACRO    RESERVE_STR  P1=0,P2
        .SDATA    .SUBSTR("ABCDEFG",\P1,\P2)
        .ENDM

        RESERVE_STR     2,2
        RESERVE_STR     ,3            ; 省略宏参数 P1。
```

[展开结果]

```
        .SDATA    "CD"
        .SDATA    "ABC"
```

11.7　结构化汇编功能

结构化汇编功能是对选择处理和重复处理的指令进行展开的功能。

在后述的结构化汇编控制语句中使用的条件码的条件如**表 11.18** 所示。请参照各结构化汇编控制语句的说明。

表 11.18　条件码一览表

条件码	比较执行型时的条件	指定条件码型时的条件
< EQ >	项 1 ＝ 项 2	Z = 1
< NE >	项 1 ≠ 项 2	Z = 0
< GT >	项 1 ＞ 项 2 （带符号的比较）	Z v (N ⊕ V) = 0
< LT >	项 1 ＜ 项 2 （带符号的比较）	N ⊕ V = 1
< GE >	项 1 ≥ 项 2 （带符号的比较）	N ⊕ V = 0
< LE >	项 1 ≤ 项 2 （带符号的比较）	Z v (N ⊕ V) = 1
< HI >	项 1 ＞ 项 2 （无符号的比较）	C v Z = 0
< LO > < CS >	项 1 ＜ 项 2 （无符号的比较）	C = 1
< HS > < CC >	项 1 ≥ 项 2 （无符号的比较）	C = 0
< LS >	项 1 ≤ 项 2 （无符号的比较）	C v Z = 1
< VC >		V = 0
< VS >		V = 1
< PL >		N = 0
< MI >		N = 1
< T >		条件总是成立
< F >		条件总是不成立

【注】 N …… CCR（条件码寄存器）的 N（负）标志
Z …… CCR 的 Z（零）标志
V …… CCR 的 V（溢出）标志
C …… CCR 的 C（进位）标志
v …… 逻辑或
⊕ …… 逻辑异或

11.7.1　有关结构化汇编功能的注意事项

结构化汇编功能是对各控制语句展开一定格式的指令和符号的功能，不进行优化。
因此，各控制语句能指定的内容根据展开的指令规格被限定。
另外，有可能对低效率的指令和不需要的符号进行展开。

（1）指令的展开

在用条件码的条件进行判断时，记述方法根据展开的指令规格被限定。
例：

```
.IF.B  (R0L<LT>#10)                ; 展开的指令出现错误。
       MOV.W    R1,R2
.ENDI
```

将.IF 控制指令展开为 CMP 指令。
但是，在此记述中被展开为 CMP R0L,#10，所以出现错误。
为了避免这种情况，必须进行如下的记述：

```
.IF.B  (#10<LT>R0L)                ; 展开为 CMP #10,R0L。
       MOV.W    R1,R2
.ENDI
```

（2）符号的展开

对各控制语句，展开以下所示的格式符号。

```
.IF      ············  _$I00000 ～ _$I99999
.SWITCH  ······  _$S00000 ～ _$S99999
.FOR[U]  ······  _$F00000 ～ _$F99999
.WHILE   ······  _$W00000 ～ _$W99999
.REPEAT  ······  _$R00000 ～ _$R99999
```

因此，不能使用相同名称的符号。

11.7.2　有关结构化汇编功能的控制语句

结构化汇编功能的控制语句的一览表如表 11.19 所示：

表 11.19　结构化汇编功能的控制语句一览表

.IF	选择处理
.SWITCH	根据条件，选择执行指令
.FOR	重复处理
.WHILE	条件成立时，重复执行
.REPEAT	
.BREAK	中断重复处理，结束处理
.CONTINUE	中断重复处理，继续处理

.IF

格　式

```
 △.IF[.<大小>][:<转移长度>]△<条件>
[△.ELSE [:<转移长度>]]
 △.ENDI
          <大小>       = { B | W | L }
          <转移长度>= { 8 | 16 }
          <条件>       = { 项1<CC>项2 | <CC> }
          <CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                             LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

不记述标号。

说　明

根据.IF 条件的判断结果，选择执行源语句。
在条件成立时，执行.IF～.ELSE 之间的源语句；否则执行.ELSE～.ENDI 之间的源语句。
能省略.ELSE 的语句。此时，如果条件成立，就执行.IF～.ENDI 之间的源语句。

长度、转移长度如下：

(1) 长度

给长度指定以比较运行型比较的项的长度。
如果省略操作长度，就为.IF.B（字节长度）。
本指定在条件码指定型中没有含义。
长度如下：

长度		长度
B	字节	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

(2) 转移长度

转移长度有.IF 和.ELSE 的转移长度。
给.IF 的转移长度指定从.IF 到.ELSE 或者从.IF 到.ENDI（省略.ELSE 时）的转移长度。
给.ELSE 的转移长度指定从.ELSE 到.ENDI 的转移长度。
转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

有关条件码的条件，请参照“**表 11.18　条件码一览表**”。

另外，条件判断有以下 2 种：

(1) 比较运行型

比较运行型用条件码的条件判断 2 个项。
给项指定能被 CMP 指令指定的寻址方式。

(2) 条件码指定型

条件码指定型以 CCR（条件码寄存器）的状态进行条件判断。

限制事项

(1) 在 H8/300、H8/300L 的情况下，不能给长度指定 L。

(2) 在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。

(3) .IF～.ELSE 之间、.ELSE～.ENDI 之间或者.IF～.ENDI 之间（省略.ELSE 时）的源语句的长度不能超过指定的转移长度。

对转移长度的源语句的最大长度如下：

8 …… 约 100 字节

16 …… 约 32,700 字节

(4) 如果使用本控制指令，就展开_$I00000～_$I99999 的符号。因此，不能使用同名的符号。

例

(1)

```
        .IF.W (R0<EQ>R1)
                ADD.W     #1,R0           ; [1]
                MOV.W     R0,R2           ; [1]
        .ELSE
                ADD.W     #1,R1           ; [2]
                MOV.W     R1,R2           ; [2]
        .ENDI
```

是比较运行型的例子。

R0＝R1 时执行[1]，R0≠R1 时执行[2]。

(2)

```
        .IF.B (#H'10<LT>R0L)
                SUB.W     R1,R1           ; [3]
                MOV.W     R1,R2           ; [3]
        .ENDI
```

是比较运行型的例子。

H'10＜R0L（带符号的比较）时执行[3]。

(3)

```
        .IF (<NE>)
                ADD.B     #5:8,R0L        ; [4]
        .ELSE
                MOV.B     R0L,R1L         ; [5]
        .ENDI
```

是条件码指定型的例。

CCR（条件码寄存器）的 Z（零）标志为 0 时执行[4]、为 1 时执行[5]。

(4)

```
        .IF.B (#0<LE>R0L)
            .IF.B (#50<GE>R0L)
                MOV.W     R2,R1           ; [6]
                ADD.W     R3,R1           ; [6]
            .ENDI
        .ENDI
```

是组合.IF 的例子。

0≤R0L≤50（带符号的比较）时执行[6]。

.SWITCH

格　式

```
 △.SWITCH[.<长度>]△<条件 1>
(△.CASE [:<转移长度>]△<条件 2>
[△.BREAK [:<转移长度>]])[,…]
[△.OTHERS]
 △.ENDS
          <长度>       = { B | W | L }
          <转移长度>   = { 8 | 16 }
          <条件 1>     = { <寄存器> | <CC> }
          <条件 2>     = { <项> | <CC> }
          <CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                           LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

不能记述标号。

说　明

根据.SWITCH 和.CASE 条件的判断结果，选择执行源语句。

如果.SWITCH 和.CASE 的条件成立，就执行相应的.CASE～.BREAK 之间的源语句。

.SWITCH 和.CASE 的条件判断按从上到下的顺序进行。

如果省略.BREAK，就执行紧接着的.CASE～.BREAK 之间的源语句或者.OTHERS～.ENDS 之间的源语句。

长度、转移长度如下：

(1) 长度

长度指定以比较运行型比较的寄存器和项的长度。

如果省略操作长度，就为.SWITCH.B（字节长度）。

本指定在条件码指定型中没有含义。

长度如下：

长度	长度	
B	字节	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

(2) 转移长度

转移长度有.CASE 和.BREAK 的转移长度。

给.CASE 的转移长度指定从.CASE 到下一个出现.CASE、.OTHERS 或者.ENDS 的转移长度。

给.BREAK 的转移长度指定从.BREAK 到.ENDS 的转移长度。

转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

条件码的条件请参照“表 11.18　条件码一览表”。

另外，条件判断有以下 2 种：

(1) 比较运行型

比较运行型判断寄存器和项是否相等。

给.SWITCH 指定寄存器。

给.CASE 指定能被 CMP 指令的源操作数指定的寻址方式。

(2) 条件码指定型

条件码指定型以 CCR（条件码寄存器）的状态进行条件判断。

给.SWITCH 指定 CCR。

给.CASE 指定条件码。

限制事项

(1) 在 H8/300、H8/300L 的情况下，不能给长度指定 L。

(2) 在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。

(3) 与各.CASE 对应的源语句或者各.BREAK～.ENDS 之间的源语句的长度不能超出指定的转移长度。

对转移长度的源语句的最大长度如下：

8 …… 约 100 字节

16 …… 约 32,700 字节

(4) 如果使用本控制指令，就展开_$S00000～_$S99999 的符号。因此，不能使用同名的符号。

例

(1)

```
.SWITCH.B  (R0L)
.CASE   #0
        MOV.W     R1,R4            ; [1]
.BREAK
.CASE   #1
        MOV.W     R2,R4            ; [2]
.BREAK
.OTHERS
        MOV.W     R3,R4            ; [3]
.ENDS
```

是比较运行型的例子。

R0L=0 时执行[1]，R0L=1 时执行[2]，否则执行[3]。

(2)

```
.SWITCH.B  (CCR)
.CASE   <CS>
        MOV.W     R0,R3            ; [4]
.BREAK
.CASE   <MI>
        MOV.W     R1,R3            ; [5]
.ENDS
```

是条件码指定型的例子。

CCR（条件码寄存器）的 C（进位）标志为 1 时执行[4]、N（负）标志为 1 时执行[5]

```
(3)
        .SWITCH.B  (R0L)
        .CASE   #0
        .CASE   #1
        .CASE   #2
                MOV.W     R1,R3           ; [6]
        .BREAK
        .CASE   #3
                MOV.W     R2,R3           ; [7]
        .ENDS
```

是省略与.CASE 对应的.BREAK 的例子。

R0L=0、R0L=1、R0L=2 时执行[6]，R0L=3 时执行[7]。

循环指令

.FOR[U]

格　式

```
△.FOR[U][.<长度>][:<转移长度>]△<条件>
△.ENDF
      <长度>        = { B | W | L }
      <转移长度>    = { 8 | 16 }
      <条件>        = { <循环计数器>=<初始值>, <终值> [,[{+ | -}]<增量值>]}
```

不能记述标号。

说　明

用.FOR[U]的循环计数器和终值进行条件判断，只在条件成立时，重复执行.FOR[U]～.ENDF 之间的源语句。

本控制指令有用带符号的范围判断重复条件的.FOR 和用无符号范围判断重复条件的.FORU。

长度、转移长度如下：

(1) 长度

给长度指定用比较运行型比较的寄存器和项的长度。

如果省略各操作长度，就为.FOR[U].B（字节长度）。

长度如下：

长度	长度	
B	长度	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

(2) 转移长度

给转移长度指定从.FOR[U]到.ENDF 的转移长度。

转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

条件如下：

(1) 循环计数器=初始值

指定循环计数器的初始值。

给循环计数器指定寄存器。

给初始值指定能被 MOV 指令的源操作数指定的寻址方式。

(2) 终值

指定与循环计数器比较的终值。

重复处理的条件如下：

增量方向为增加方向 …… 循环计数器≤终值

增量方向为减少方向 …… 循环计数器≥终值

给终值指定能被 CMP 指令的源操作数指定的寻址方式。

(3) 增量值

指定每次的循环时循环计数器进行加减的增量值。
在增量方向为增加方向时指定正号(+)，为减少方向时指定负号(-)。
如果省略指定，就指定正号(+)。
给增量值指定能被 ADD、SUB 指令的源操作数指定的寻址方式。如果省略指定，就指定+#1。

循环计数器的数值范围如下所示。有可能进入无限循环，所以必须充分注意数值的范围。

控制语句	增量方向	长度	循环计数器的数值范围（初始值～终值）		
.FOR	+	B	-128	～	126
		W	-32,768	～	32,766
		L	-2,147,483,647	～	2,147,483,646
	-	B	127	～	-127
		W	32,767	～	-32,767
		L	2,147,483,647	～	-2,147,483,647
.FORU	+	B	0	～	254
		W	0	～	65,534
		L	0	～	4,294,967,294
	-	B	255	～	1
		W	65,535	～	1
		L	4,294,967,295	～	1

限制事项

(1) 在 H8/300、H8/300L 的情况下，不能给长度指定 L。
(2) 在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。
(3) .FOR[U]～.ENDF 之间的源语句的长度不能超出指定的转移长度。
对转移长度的源语句的最大长度如下：
8 …… 约 100 字节
16 …… 约 32,700 字节
(4) 如果使用本控制指令，就展开_$F00000～_$F99999 的符号。因此，不能使用同名的符号。

例

(1)

```
        .FOR.B (R0L=#1,#10,+#1)         ; [1]
               ADD.B     R0L,R1L
        .ENDF
```

为.FOR 的例子。
循环计数器为 R0L，初始值为#1，终值为#10，增量值为+#1。
只在 R0L≤10（带符号的比较）时，重复执行[1]。

(2)

```
        .FOR.W (R0=R1,R2,-R3)
               ADD.B     #1:8,R5L       ; [2]
        .ENDF
```

为.FOR 的例子。
循环计数器为 R0，初始值为 R1，终值为 R2，增量值为-R3。
只在 R0≥R2（带符号的比较）时，重复执行[2]。

(3)

```
       .FORU.B(R0L=#1,#200,+#1)
              ADD.W     R1,R2          ; [3]
              ADD.W     R3,R4          ; [3]
       .ENDF
```

为.FORU 的例子。
循环计数器为 R0L，初始值为#1，终值为#200，增量值为+#1。
只在 R0L≤200（无符号的比较）时，重复执行[3]。

(4)

```
       .FORU.L(ER0=#H'00000100,#H'000001FC,+#4)
              MOV.L     @ER0,ER2                      ; [4]
              MOV.L     ER2,@(H'00001100:32,ER1)      ; [4]
              ADDS.L    #4,ER1                        ; [4]
       .ENDF
```

为.FORU 的例子。
循环计数器为 ER0，初始值为#H'00000100，终值为＃H'000001FC，增量值为+#4。
只在 ER0≤H'000001FC（无符号的比较）时，重复执行[4]。

循环指令

.WHILE

格　式

```
△.WHILE[.<长度>][:<转移长度>]△<条件>
△.ENDW
         <长度>       = { B | W | L }
         <转移长度> = { 8 | 16 }
         <条件>       = { 项 1<CC>项 2 | <CC> }
         <CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                            LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

不能记述标号。

说　明

用.WHILE 进行条件判断，只在条件成立时，重复执行.WHILE～.ENDW 之间的源语句。

长度、转移长度如下：

(1) 长度

给长度指定用比较运行型比较的寄存器和项的长度。

如果省略操作长度，就为.WHILE.B（字节长度）。

本指定对条件码指定型无意义。

长度如下：

长度		长度
B	字节	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

(2) 转移长度

给转移长度指定从.WHILE 到.ENDW 的转移长度。

转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

条件码的条件请参照“**表 11.18　条件码一览表**”。

另外，条件判断有以下 2 种：

(1) 比较运行型

比较运行型用条件码的条件判断 2 个项。

给项指定能被 CMP 指令指定的寻址方式。

(2) 条件码指定型

条件码指定型以 CCR（条件码寄存器）的状态进行条件判断。

限制事项

(1) 在 H8/300、H8/300L 的情况下，不能给长度指定 L。
(2) 在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。
(3) .WHILE～.ENDW 之间的源语句的长度不能超过指定的转移长度。
与转移长度对应的源语句的最大长度如下：
8 …… 约 100 字节
16 …… 约 32,700 字节
(4) 如果使用本控制指令，就展开_$W00000～_$W99999 的符号。因此，不能使用同名的符号。

例

(1)

```
       .WHILE.B  (#50<GT>R0L)
              ADD.W     R1,R2            ; [1]
              ADD.B     #1:8,R0L         ; [1]
       .ENDW
```

是比较运行型的例子。
只在 50>R0L（带符号的比较）时，重复[1]。

(2)

```
       .WHILE.W  (R0<LS>R1)
              MOV.B     R2L,R3L          ; [2]
              SUB.W     R5,R1            ; [2]
       .ENDF
```

是比较运行型的例子。
只在 R0≤R1（无符号的比较）时，重复[2]。

(3)

```
       .WHILE  (<NE>)
              MOV.L     @ER2,ER4         ; [3]
              MOV.L     ER4,@ER3         ; [3]
              ADDS.L    #4,ER2           ; [3]
              ADDS.L    #4,ER3           ; [3]
              SUB.B     R1L,R0L          ; [3]
       .ENDW
```

是条件码指定型的例子。
只在 CCR（条件码寄存器）的 Z（零）标志为 0 时，重复[3]。

(4)

```
       .WHILE  (<PL>)
              MOV.L     ER2,@ER1         ; [4]
              ADDS.L    #4,ER1           ; [4]
              MOV.L     ER3,@ER1         ; [4]
              ADDS.L    #4,ER1           ; [4]
              ADD.W     #-1,R0           ; [4]
       .ENDW
```

是条件码指定型的例子。
只在 CCR（条件码寄存器）的 N（负）标志为 0 时，重复执行[4]。

循环指令

.REPEAT

格　式

```
△.REPEAT
△.UNTIL [.<长度>]△<条件>
        <长度>       = { B | W | L }
        <条件>       = { 项 1<CC>项 2 | <CC> }
        <CC>         = { EQ | NE | GT | LT | GE | LE | HI | LO | HS |
                         LS | CC | CS | VC | VS | PL | MI | T  | F  }
```

不能记述标号。

说　明

用.UNTIL 进行条件判断，在条件成立之前，重复执行.REPEAT～.UNTIL 之间的源语句。因为用.UNTIL 进行条件判断，所以必须执行一次.REPEAT～.UNTIL 之间的源语句。

长度如下：

给长度指定用比较运行型比较的寄存器和项的长度。
如果省略操作长度，就为.UNTIL.B（字节长度）。
本指定对条件码指定型无意义。
长度如下：

长度		长度
B	字节	（1 个字节）
W	字	（2 个字节）
L	长字	（4 个字节）

条件码的条件请参照“**表 11.18 条件码一览表**”。

另外，条件判断有以下 2 种：

(1) 比较运行型
比较运行型用条件码的条件判断 2 个项。
给项指定能被 CMP 指令指定的寻址方式。

(2) 条件码指定型
条件码指定型以 CCR（条件码寄存器）的状态进行条件判断。

限制事项

(1) 在 H8/300、H8/300L 的情况下，不能给长度指定 L。
(2) .REPEAT～.UNTIL 之间的源语句的长度在 CPU 种类是 300、300L 时约为 100 个字节，否则约为 32,700 个字节。
(3) 如果使用本控制指令，就展开_$R00000～_$R99999 的符号。因此，不能使用同名的符号。

例

(1)

```
       .REPEAT
              MOV.L      @ER0,ER2         ; [1]
              MOV.L      ER2,@ER1         ; [1]
              ADDS.L     #4,ER0           ; [1]
              ADDS.L     #4,ER1           ; [1]
       .UNTIL (#H'001000<LS>ER0)
```

是比较运行型的例子。

在 H'001000≤ER0（无符号的比较）成立之前，重复执行[1]。

(2)

```
       .REPEAT
              ADD.W      R2,R3            ; [2]
              ADD.W      R2,R4            ; [2]
              SUB.B      R1L,R0L          ; [2]
       .UNTIL (<EQ>)
```

是条件码指定型的例子。

在 CCR（条件码寄存器）的 Z（零）标志为 1 之前，执行[2]。

处理的中断（结束）

.BREAK

格　式

```
△.BREAK[:<转移长度>]
       <转移长度> = { 8 | 16 }
```

不能记述标号。

说　明

在.FOR[U]、.WHILE、.REPEAT 的重复处理中，不执行.BREAK 以下的源语句而结束重复处理。其具体的处理是在.FOR[U]、.WHILE、.REPEAT 的重复处理中分别转移到.ENDF、.ENDW、.UNTIL 后结束重复处理。

给转移长度指定从.BREAK 到.ENDF、.ENDW 或者.UNTIL 的转移长度。
转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

.SWITCH 也使用本控制语句。
有关.SWITCH 的使用方法，请参照“11.7 .SWITCH”。

限制事项

在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。

例

```
        .WHILE  (<T>)
            .IF.B   (#10<LE>R0L)
                .BREAK
            .ENDI
            ADD.W    R1,R2
            INC.B    R0L
        .ENDW
```

在 10≤R0L 时，结束重复处理。

.CONTINUE

格　式

```
△.CONTINUE[:<转移长度>]
      <转移长度> = { 8 | 16 }
```

不能记述标号。

说　明

在.FOR[U]、.WHILE、.REPEAT 的重复处理中不执行.CONTINUE 以下的源语句而继续重复处理。

其具体的处理是在.FOR[U]、.WHILE、.REPEAT 的重复处理中分别转移到重复处理的条件判断。

给转移长度指定从.CONTINUE 到.ENDF、.ENDW 或者.UNTIL 的转移长度。

转移长度如下：

操作	转移长度
8	8 位
16	16 位

有关省略指定时的设定，请参照“11.3　.DISPSIZE 的 FBR”、“3.2.2　br_relative”和“3.2.2　[no]optimize”。

限制事项

在 H8/300、H8/300L 的情况下，不能给转移长度指定 16。

例

```
.WHILE.B  (#10<GT>R0L)
   INC.B    R0L
   INC.B    R1L
   .IF.B    (#10<LT>R1L)
       .CONTINUE
   .ENDI
   ADD.W    R2,R3              ; [1]
.ENDW
```

在 10＜R1L 时不执行[1]。

第 12 章　编译程序的错误信息

12.1　错误格式和错误级

本章说明用以下格式输出的错误信息和错误内容：

错误号　**（错误级）**　**错误信息**
错误内容

根据错误的重要程度，错误级分为 5 种。

错误级		运行
（I）	信息	继续处理，输出目标程序。
（W）	警告	继续处理，输出目标程序。
（E）	错误	继续处理，不输出目标程序。
（F）	致命错误	输出错误信息的同时，中断处理。
（-）	内部错误	输出错误信息的同时，中断处理。

12.2　信息一览

`C0002 (I) No declarator`

存在没有声明符的声明。

`C0003 (I) Unreachable statement`

存在没有被执行的语句。

`C0004 (I) Constant as condition`

将常数表达式指定为表示 `if` 语句或者 `switch` 语句的条件表达式。

`C0005 (I) Precision lost`

在赋值表达式中，将右边表达式的值转换为左边的型时，有可能失去精度。

`C0006 (I) Conversion in argument`

将函数的参数表达式转换为原型声明指定的参数型。

`C0008 (I) Conversion in return`

将返回语句的表达式转换为函数返回值的型。

`C0010 (I) Elimination of needless expression`

存在不需要的表达式。

`C0011 (I) Used before set symbol "变量名"`

参照没有设定值的局部变量。

C0015 (I) No return value
在返回 void 型以外的型的函数中，返回语句没有返回值，或者没有返回语句。

C0016 (I) Padding in structure
在结构体的成员之间生成了边界调整的空区域。

C0100 (I) Function "函数名" not optimized
函数太大，不能优化。

C0101 (I) Optimizing range divided in function "函数名"
"函数名"的优化范围被分为多个。

C0102 (I) Register is not allocated to "变量名" in "函数名"
不能将寄存器分配给具有 register 存储类的变量。

C0200 (I) No prototype function
没有函数的原型声明。

C0300 (I) #pragma interrupt has no effect
没有#pragma interrupt 指定的函数。

C0301 (I) #pragma abs8 has no effect
没有#pragma abs8 指定的变量。

C0302 (I) #pragma abs16 has no effect
没有#pragma abs16 指定的变量。

C0303 (I) #pragma indirect has no effect
没有#pragma indirect 指定的函数。

C0304 (I) #pragma regsave/noregsave has no effect
没有#pragma regsave/noregsave 指定的函数。

C0305 (I) #pragma inline/inline_asm has no effect
没有#pragma inline/inline_asm 指定的函数。

C0306 (I) #pragma global_register has no effect
没有#pragma global_register 指定的变量。

C0307 (I) #pragma entry has no effect
没有#pragma entry 指定的声明。

C0308 (I) #pragma address has no effect
没有#pragma address 指定的变量。

C1000 (W) Illegal pointer assignment
在指针型之间的赋值中，各指针型所指的型不同。

C1001 (W) Illegal comparison in "运算符"
二元运算符 == 或者 != 的操作数一边是指针型，另一边是指向 0 以外的整型。

C1002 (W) Illegal pointer for "运算符"
二元运算符 ==、!=、>、<、>= 或者 <= 的操作数是指向不同型的指针型。

C1005 (W) Undefined escape sequence
在字符常数或者字符串中，使用了语法没有定义的转义序列（反斜线及其后续的字符）。

C1007 (W) Long character constant
字符常数的长度为 2 个字符。

C1008 (W) Identifier too long
标识符的长度超过 8189 个字符。第 8190 个以后的字符无效。

C1010 (W) Character constant too long
字符常数的长度超过 2 个字符。第 3 个以后的字符无效。

C1012 (W) Floating point constant overflow
浮点常数的值超出了范围。根据符号，假设与＋∞ 或者－∞ 对应的内部表现值。

C1013 (W) Integer constant overflow
整型常数的值超出了 unsigned long 型的范围。假设溢出的高位被忽视的值。

C1014 (W) Escape sequence overflow
在字符常数或者字符串中，表示位模式的转义序列的值超过了 255。低位 1 个字节的值有效。

C1015 (W) Floating point constant underflow
浮点常数的绝对值小于能表现的最小值。假设常数值为 0.0。

C1016 (W) Argument mismatch
原型声明中的参数和函数调用的对应参数的型为指针型，分别指向不同的型。照样设定在函数调用时的参数指针的内部表现。

`C1017 (W) Return type mismatch`
函数的返回型和返回语句表达式的型为指针型，分别指向不同的型。照样设定返回语句的表达式指针的内部表现。

`C1019 (W) Illegal constant expression`
在常数表达式中，关系运算符 <、>、<= 或者 >=的操作数是没有指向相同型的指针型。假设结果值为 0。

`C1020 (W) Illegal constant expression of "-"`
在常数表达式中，二元运算符 – 的操作数是没有指向相同型的指针型。假设结果值为 0。

`C1021 (W) Convert to sjis-space`
日语代码不能转换为指定的输出代码。转换为 Shift JIS 的空格。

`C1022 (W) Convert to euc-space`
日语代码不能转换为指定的输出代码。转换为 EUC 的空格。

`C1023 (W) Can not convert japanese code "字符" to output type`
日语代码不能转换为指定的输出代码。转换为空格。

`C1024 (W) First operand of "运算符" is not lvalue`
给运算符的第一操作数指定了非左边值。

`C1025 (W) Out of float`
浮点常数的有效位数超过了 17 位。第 18 位以后无效。

`C1026 (W) Address of packed member`
取得了有 pack=1 指定的结构体成员的地址。

`C1200 (W) Division by floating point zero`
在常数表达式中，进行了以浮点数 0.0 为除数的除法运算。根据符号，假设与＋∞或者－∞对应的内部表现值。

`C1201 (W) Ineffective floating point operation`
在常数表达式中，进行了 ∞－∞、0.0/0.0 等无效运算。假设与表示无效运算结果的非数值对应的内部表现值。

`C1300 (W) Command parameter specified twice`
指定了 2 次以上相同的编译程序选项。在相同的编译程序选项中，最后指定的选项有效。

`C1302 (W) 'frame' or 'noframe' option ignored`
在指定优化时指定了 frame 选项，或者在没有指定优化时指定了 noframe 选项。忽视选项的指定。

`C1305 (W) 'show=object' option ignored`

在指定汇编源程序的输出时，指定了 `show=object` 选项。忽视选项的指定。

`C1306 (W) 'speed=inline' option ignored`

在没有指定优化时，指定了 `speed=inline` 选项。忽视选项的指定。

`C1307 (W) Section name too long`

段名的长度超过了 8192 个字符。前 8192 个字符有效，忽视以后的字符。

`C1308 (W) 'speed=loop' option ignored`

在没有指定优化时，指定了 `speed=loop` 选项。忽视选项的指定。

`C1310 (W) 'goptimize' option ignored`

在指定汇编源程序的输出时，指定了 `goptimize` 选项。忽视选项的指定。

`C1311 (W) 'cmncode' option ignored`

在没有指定优化时，指定了 `cmncode` 选项。忽视选项的指定。

`C1313 (W) Invalid SBR value`

给 `sbr` 选项的低 `8bit` 指定了 `0` 以外的值。忽视低 `8bit` 的指定。

`C1314 (W) 'ecpp' option ignored`

在指定 C++异常处理功能有效时，指定了 `ecpp` 选项。忽视选项的指定。

`C1315 (W) 'noregexpansion' option ignored`

在 CPU 种类为 H8SX、H8S（没有指定 `legacy=v4` 选项）时，指定了 `noregexpansion` 选项。忽视选项的指定。

`C1316 (W) 'cmncode' option ignored`

在 CPU 种类为 H8SX、H8S（没有指定 `legacy=v4` 选项）时，指定了 `cmncode` 选项。忽视选项的指定。

`C1318 (W) 'align=4' option ignored`

在 CPU 种类不为 H8SX 时，指定了 `align=4` 选项。忽视选项的指定。

`C1319 (W) 'speed=intrinsic' option ignored`

在 CPU 种类不为 H8SX 时，指定了 `speed=intrinsic` 选项。忽视选项的指定。

C1321 (W) 'sbr' option ignored

在 CPU 种类不为 H8SX 时，指定了 sbr 选项。忽视选项的指定。

C1322 (W) 'volatile_loop' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 volatile_loop 选项。忽视选项的指定。

C1323 (W) 'infinite_loop' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 infinite_loop 选项。忽视选项的指定。

C1324 (W) 'ptr16' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 ptr16 选项。忽视选项的指定。

C1325 (W) 'del_vacant_loop' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 del_vacant_loop 选项。忽视选项的指定。

C1326 (W) 'global_alloc' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 global_alloc 选项。忽视选项的指定。

C1327 (W) 'struct_alloc' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 struct_alloc 选项。忽视选项的指定。

C1328 (W) 'const_var_propagate' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 const_var_propagate 选项。忽视选项的指定。

C1329 (W) 'opt_range' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 opt_range 选项。忽视选项的指定。

C1330 (W) 'max_unroll' option ignored

在 CPU 种类不为 H8SX、H8S（没有指定 legacy=v4 选项）时，指定了 max_unroll 选项。忽视选项的指定。

C1331 (W) Section name "S" specified

给段名指定了 S。

有可能与编译程序生成的堆栈区的段名重复。

C1332 (W) 'indirect=extended' option ignored

在 CPU 种类不为 H8SX 时，指定了 indirect=extended 选项。忽视选项的指定。

C1333 (W) 'enable_register' option ignored

给 CPU 种类指定了 300HN、300HA、300、300L、300Reg 或者指定了 legacy=V4 选项。忽视选项的指定。

C1334 (W) 'legacy=v4' option ignored

给 CPU 种类指定了 H8S 以外的种类。忽视选项的指定。

C1335 (W) 'strict_ansi' option ignored

给 CPU 种类指定了 300HN、300HA、300、300L、300Reg 或者指定了 legacy=V4 选项。忽视选项的指定。

C1336 (W) 'cpuexpand=v6' option ignored

在 CPU 种类为 H8SX,H8S（没有指定 legacy=v4 选项）时，忽视相应选项的指定。

C1337 (W) 'noscope' option ignored

给 CPU 种类指定了 300HN、300HA、300、300L、300Reg 或者指定了 legacy=V4 选项。忽视选项的指定。

C1338 (W) Invalid SBR value in H8SXM

给 CPU 种类指定了 H8SXM。SBR 被设定在数据区外。

C1339 (W) 'file_inline' option ignored

给 CPU 种类指定了 300HN、300HA、300、300L、300Reg 或者指定了 legacy=V4 选项。忽视选项的指定。

C1341 (W) 'file_inline_path' option ignored

给 CPU 种类指定了 300HN、300HA、300、300L、300Reg 或者指定了 legacy=V4 选项。忽视选项的指定。

C1342 (W) "选项字符串 1" is interpreted as "选项字符串 2"

作为选项，指定了选项字符串 1，但是没有相应的选项。

将"选项字符串 1"选项解释为"选项字符串 2"选项，进行编译。

C1400 (W) Function "函数名" in #pragma inline is not expanded

#pragma inline 指定的函数没有被直接插入展开。忽视#pragma inline 的指定。

C1401 (W) #pragma abs16 ignored

CPU/运行模式为 H8SXN、H8SXM、2600n、2000n、300hn 或者 300 时，指定了#pragma abs16。忽视#pragma abs16 的指定。

C1403 (W) #pragma asm ignored

目标格式为可再定位目标程序时，指定了#pragma asm。忽视#pragma asm 的指定。

C1404 (W) 'case=table' option ignored by switch
不能将 switch 语句展开为表格式。用 if-then 格式展开 switch 语句。

C1405 (W) Illegal #pragma syntax
指定了无法识别的#pragma 语句。忽视#pragma 的指定。

C1406 (W) Abs8 attribute ignored
忽视了 abs8 的指定。

C1407 (W) #pragma address ignored
对于带初始值的变量的#pragma address 指定无效。

C1510 (W) Illegal bit width
用 CPU 选项指定的位长不正确。

C1511 (W) Illegal value in operand
给操作数指定了范围以外的值。

C2000 (E) Illegal preprocessor keyword
在预处理器语句中，使用了错误的关键字。

C2001 (E) Illegal preprocessor syntax
预处理器语句或者宏调用的指定方法有错。

C2007 (E) Invalid include file name "文件名"
include 文件名的指定不正确。

C2016 (E) Preprocessor constant expression too complex
#if 语句、#elif 语句指定的常数表达式的运算符和操作数的合计超过了 512 个。

C2019 (E) File name too long
文件名的长度超过了 4096 个字符。

C2020 (E) System identifier "名称" redefined
定义了和内部函数同名的符号。

C2021 (E) System identifier "名称" mismatch
使用了在被指定的 CPU/运行模式中没有的内部函数。

C2100 (E) Multiple storage classes
在声明中指定了 2 个以上的存储类说明符。

C2101 (E) Address of register
对于具有寄存器存储类的变量，使用了一元运算符&。

C2102 (E) Illegal type combination
型说明符的组合有错。

C2103 (E) Bad self reference structure

用与主结构体或者联合体相同的型声明结构体或者联合体成员的型。

C2104 (E) Illegal bit field width

表示位字段宽度的常数表达式不为整型，或者将负整数指定为位字段宽度。

C2105 (E) Incomplete tag used in declaration

在 typedef 声明、指针型或者函数的返回型以外的声明中使用了未声明的标记符名、或者由结构体或联合体临时声明的标记符名。

C2106 (E) Extern variable initialized

对在复合语句内指定 extern 存储类的声明，指定了初始值。

C2107 (E) Array of function

元素的型指定了函数型的数组型。

C2108 (E) Function returning array

返回值的型指定了数组型的函数型。

C2109 (E) Illegal function declaration

在复合语句内的函数型变量声明中，指定了 extern 以外的存储类。

C2110 (E) Illegal storage class

在外部定义中，作为存储类，指定了 auto 或者 register。

C2111 (E) Function as a member

给结构体或者联合体的成员的型指定了函数型。

C2112 (E) Illegal bit field

给位字段指定了错误型。位字段允许的型说明符为 char, unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、enum、bool 中的任一型，或者将这些型和 const 或者 volatile 组合的型。

C2113 (E) Bit field too wide

位字段宽度超过了型说明符指定的长度（8、16、32 位）。

C2114 (E) Multiple variable declarations

在相同的有效范围内重复声明了变量名。

C2115 (E) Multiple tag declarations

在相同的有效范围内重复声明了结构体、联合体或者枚举型的标记符名。

C2117 (E) Empty source program

在源程序中没有包含外部定义。

C2118 (E) Prototype mismatch "函数名"
函数的型与以前声明中指定的型不一致。

C2119 (E) Not a parameter name "参数名"
对函数的参数声明列中没有的标识符进行参数声明。

C2120 (E) Illegal parameter storage class
函数的参数声明中指定了 register 以外的存储类。

C2121 (E) Illegal tag name
结构体、联合体或者枚举型和标记符名的组合与以前声明的型和标记符名的组合不同。

C2122 (E) Bit field width 0
指定成员名的位字段宽度为 0。

C2123 (E) Undefined tag name
枚举型的声明中使用了未定义的标记符名。

C2124 (E) Illegal enum value
给枚举型的成员指定了非整数的常数表达式。

C2125 (E) Function returning function
返回值的型为函数型。

C2126 (E) Illegal array size
指定的数组元素的个数超过了限制值。

C2127 (E) Missing array size
没有指定数组元素的个数。

C2129 (E) Illegal initializer type
在变量初始值的指定中，初始值的型不是能给变量赋值的型。

C2130 (E) Initializer should be constant
给结构体型、联合体型、数组型变量的初始值或者静态分配的变量初始值指定了非常数表达式。

C2131 (E) No type nor storage class
在外部数据定义中，没有指定存储类或者型。

C2132 (E) No parameter name
即使函数的参数声明列为空，也进行了参数声明。

`C2133 (E) Multiple parameter declarations`

在（宏）函数定义的参数声明列中重复声明了同名的参数，或者在函数说明符的内外 2 处进行了参数声明。

`C2134 (E) Initializer for parameter`

对参数声明指定了初始值。

`C2135 (E) Multiple initialization`

对相同的变量重复进行了初始化。

`C2136 (E) Type mismatch`

2 次以上声明了具有 `extern` 或者 `static` 存储类的变量或者函数，并且型不一致。

`C2137 (E) Null declaration for parameter`

在函数的参数声明中没有指定标识符。

`C2138 (E) Too many initializers`

在指定结构体、联合体或者数组的初始值时，指定了多于结构体成员个数或者数组元素个数的初始值个数，或者在联合体的最初成员为标量型时，指定了 2 个以上的初始值。

`C2139 (E) No parameter type`

函数声明的参数声明中没有指定型。

`C2140 (E) Illegal bit field`

给联合体指定了位字段。

`C2141 (E) Struct has no member name`

给结构体的最初成员指定了无名的位字段。

`C2142 (E) Illegal void type`

`void` 型的指定方法有错。在以下三种情况时能指定 `void` 型：

（1）指定为指针的指向型；

（2）指定为函数的返回型；

（3）明确指定原型声明的函数没有参数。

`C2143 (E) Illegal static function`

在源文件中存在未定义的 `static` 存储类的函数声明。

`C2150 (E) Multiple function qualifiers`

指定了多个函数限定词。

`C2151 (E) "名称" must be qualified for function type`

不能限定函数型以外的型。

C2152 (E) Illegal attribute combination

不允许关键字的组合方法。

以下"○"表示的关键字为允许的组合。

	_ _near8	_ _near16	_ _abs8	_ _abs16	_ _ptr16	_ _interrupt	_ _inline	_ _indirect	_ _indirect_ex	_ _regsave	_ _noregsave
_ _near8	×	×	○	○	×	×	×	×	×	×	×
_ _near16	×	×	○	○	×	×	×	×	×	×	×
_ _abs8	○	○	×	×	×	×	×	×	×	×	×
_ _abs16	○	○	×	×	○	×	×	×	×	×	×
_ _ptr16	×	×	×	○	×	×	×	×	×	×	×
_ _interrupt	×	×	×	×	×	×	×	×	×	○	○
_ _inline	×	×	×	×	×	×	×	○	○	×	×
_ _indirect	×	×	×	×	×	×	○	×	×	○	○
_ _indirect_ex	×	×	×	×	×	×	○	×	×	○	○
_ _regsave	×	×	×	×	×	○	×	○	○	×	×
_ _noregsave	×	×	×	×	×	○	×	○	○	×	×

C2153 (E) Illegal "名称" specifier

属性说明符的记述方法有错。

C2154 (E) "名称" must be specified for variables

此属性说明符只能指定变量。

C2155 (E) "名称" must be specified for functions

此属性说明符只能指定函数。

C2157 (E) Attribute keyword and pragma cannot be specified for one symbol

不能同时指定属性关键字和#pragma 声明。

C2158 (E) Attribute mismatch

声明之间的属性不一致。

C2159 (E) Multiple entry functions

指定了多个入口函数。

C2160 (E) Illegal '_ _near8/_ _near16' variable size

指定_ _near8/_ _near16 的变量长度超出了范围。

C2161 (E) Illegal '_ _abs8' variable type

_ _abs8 变量的型不正确。

C2162 (E) Illegal '_ _global_register' variable type

_ _global_register 变量的型不正确。

C2163 (E) Illegal '_ _interrupt' function type

中断函数的型不正确。

C2164 (E) Cannot specify '名称' to local storage class

使用了在此不能指定的属性。

C2165 (E) Multiple pointer qualifiers

指定了多个 ptr16。

C2166 (E) '_ _ptr16' must be qualified for data pointer type

ptr16 被指定为非数据指针型。

C2190 (E) Multiple functions on vector "向量号"

给相同的向量号指定了多个函数。

C2200 (E) Index not integer

数组的下标表达式不为整型。

C2201 (E) Cannot convert parameter "n"

不能将函数调用中的第 n 个参数转换为对应的原型声明的参数型。

C2202 (E) Number of parameters mismatch

函数调用中参数的个数和原型声明参数的个数不一致。

C2203 (E) Illegal member reference for "."

运算符 . 左边的表达式的型不是结构体型或者联合体型。

C2204 (E) Illegal member reference for "->"

运算符 -> 左边的表达式的型不是指向结构体型或者联合体型的指针。

C2205 (E) Undefined member name

在参照结构体或者联合体中使用了没有声明的成员名。

C2206 (E) Modifiable lvalue required for "运算符"

前缀或者后缀运算符 ++、-- 用于可赋值的左边值（除数组型、const 型以外的左边值）以外的表达式。

C2207 (E) Scalar required for "!"

一元运算符 ! 用于非标量型的表达式。

C2208 (E) Pointer required for "*"

一元运算符 * 用于非指针型的表达式或者指向 void 型的指针型的表达式。

C2209 (E) Arithmetic type required for "运算符"

一元运算符 + 或者 – 用于非算术型的表达式。

C2210 (E) Integer required for "~"

一元运算符 ~ 用于非整型的表达式。

C2211 (E) Illegal sizeof

sizeof 运算符用于指定位字段的成员、函数型、void 型或者没有指定大小的数组。

C2212 (E) Illegal cast

数据类型转换运算符指定的型为数组型、结构体型或者联合体型，或者数据类型转换运算符的操作数为 void 型、结构体型或者联合体型，不能进行型转换。

C2213 (E) Arithmetic type required for "运算符"

二元运算符 *、/、*= 或者 /= 用于非算术型的表达式。

C2214 (E) Integer required for "运算符"

二元运算符 <<、>>、&、|、^、%、<<=、>>=、&=、|=、^= 或者 %= 用于非整型的表达式。

C2215 (E) Illegal type for "+"

不允许二元运算符 + 的操作数型的组合。只在两边都是算术型或者一边是指针型而另一边是整型时，允许二元运算符 + 的型的组合。

C2216 (E) Illegal type for parameter

给函数调用的参数型指定了 void 型。

C2217 (E) Illegal type for "-"

不允许二元运算符 – 的操作数型的组合。在以下三种情况时允许二元运算符 – 的型的组合：

（1）两边的操作数是算术型。
（2）两边的操作数是同型的指针型。
（3）第 1 个操作数是指针型，第 2 个操作数是整型。

C2218 (E) Scalar required in "?:"

条件运算符 ?: 的第 1 个操作数的型不是标量型。

C2219 (E) Type not compatible in "?:"

条件运算符 ? : 的第 2 个操作数和第 3 个操作数的型不一致。在以下六种情况时，允许条件运算符 ? : 的第 2 个操作数和第 3 个操作数的组合：

（1）两边都是算术型。
（2）两边都是 void 型。
（3）两边都是相同的指针型。
（4）一边是指针型而另一边是 0 的整型常数或者将 0 的整型常数转换为指向 void 型的指针型。
（5）一边是指针型而另一边是指向 void 型的指针型。
（6）两边都是同型的结构体或者联合体。

C2220 (E) Modifiable lvalue required for "运算符"

给赋值运算符 =、*=、/=、%=、+=、-=、<<=、>>=、&=、^= 或者 |= 左边的表达式指定了可赋值的左边值（除数组型、const 型以外的左边值）以外的表达式。

C2221 (E) Illegal type for "运算符"

给后缀运算符 ++ 或者 -- 的操作数指定了指向非标量型、函数型或者 void 型的指针型。

C2222 (E) Type not compatible for "="

赋值运算符 = 两边的表达式不一致。在以下五种情况时，允许赋值运算符 = 两边的表达式的组合：

（1）两边都是算术型。
（2）两边都是相同的指针型。
（3）左边是指针型，右边是 0 的整型常数或者将 0 的整型常数转换为指向 void 型的指针型。
（4）一边是指针型，另一边是指向 void 型的指针型。
（5）两边都是同型的结构体或者联合体。

C2223 (E) Incomplete tag used in expression

在表达式中使用了结构体或者联合体临时声明的标记符名。

C2224 (E) Illegal type for assign

赋值运算符 += 或者 -= 两边的型不正确。

C2225 (E) Undeclared name "名称"

在表达式中使用了没有声明的名称。

C2226 (E) Scalar required for "运算符"

二元运算符 && 或者 || 用于非标量型的表达式。

C2227 (E) Illegal type for equality

不允许相等运算符 == 或者 != 的操作数型的组合。在以下三种情况时，允许等值运算的操作数的组合：

（1）两边都是算术型。
（2）两边都是指向相同型的指针型。
（3）一边是指针型，另一边是 0 的整型常数或者指向 void 型的指针型。

C2228 (E) Illegal type for comparison

不允许关系运算符 >、<、>= 或者 <= 的操作数型的组合。在以下二种情况时，允许关系运算符的操作数的组合：

（1）两边都是算术型。
（2）两边都是指向同型的指针型。

C2230 (E) Illegal function call

在函数调用中使用了非函数型或者非函数型的指针型的表达式。

C2231 (E) Address of bit field

一元运算符 & 用于位字段。

C2232 (E) Illegal type for "运算符"

给前缀运算符 ++ 或者 -- 的操作数指定了指向非标量型、函数型或者 void 型的指针型。

C2233 (E) Illegal array reference

除数组型、函数型或者 void 型以外的非指针型的表达式用作数组。

C2234 (E) Illegal typedef name reference

在表达式中将 typedef 声明的名称用作了变量。

C2235 (E) Illegal cast

进行了从指针到浮点型的数据类型转换。

C2237 (E) Illegal constant expression

在常数表达式中，对指针型的常数→整型的数据类型转换结果进行了运算。

C2238 (E) Lvalue or function type required for "&"

一元运算符 & 用于左边值或者函数型以外的表达式。

C2239 (E) Illegal section name

段名中有不能使用的字符。

C2240 (E) Illegal section naming

段的命名有错。给不同用途的段起了相同的名称。

C2300 (E) Case not in switch

在非 switch 语句中指定了 Case 标号。

C2301 (E) Default not in switch

在非 switch 语句中指定了 default 标号。

C2302 (E) Multiple labels

在一个函数中重复定义了标号名。

C2303 (E) Illegal continue

在 while 语句、for 语句或者 do 语句以外的语句中指定了 continue 语句。

C2304 (E) Illegal break

在 while 语句、for 语句、do 语句或者 switch 语句以外的语句中指定了 break 语句。

C2305 (E) Void function returns value

对返回 void 型的函数中的 return 语句，指定了返回值。

C2306 (E) Case label not constant
case 标号的表达式不是整型的常数表达式。

C2307 (E) Multiple case labels
在一个 switch 语句中重复指定了同值的 case 标号。

C2308 (E) Multiple default labels
在一个 switch 语句中重复指定了 Default 标号。

C2309 (E) No label for goto
没有 goto 语句指定的转移目标的标号。

C2310 (E) Scalar required “while, for, do”
while 语句、for 语句或者 do 语句的控制表达式（判断语句执行的表达式）不是标量型。

C2311 (E) Integer required
switch 语句的控制表达式（判断语句执行的表达式）不是整型。

C2312 (E) Missing (
没有 if 语句、while 语句、for 语句、do 语句或者 switch 语句的控制表达式（判断语句执行的表达式）的左括弧"("。

C2313 (E) Missing ;
在 do 语句的最后没有分号";"。

C2314 (E) Scalar required "if"
if 语句的控制表达式（判断语句执行的表达式）不是标量型。

C2316 (E) Illegal type for return value
不能将 return 语句表达式的型转换为函数的返回型。

C2320 (E) Illegal asm position
#pragma asm 的指定位置不适当。

C2330 (E) Illegal #pragma interrupt declaration
中断函数声明有错。

C2331 (E) Illegal interrupt function call
在程序中，调用或者参照了中断函数声明的函数。

C2332 (E) Function "函数名" in #pragma interrupt already declared
用中断函数声明#pragma interrupt 指定的函数已被声明为通常的函数。

C2333 (E) Multiple interrupt for one function
对相同函数重复声明了中断函数声明#pragma interrupt。

C2334 (E) Illegal parameter in #pragma interrupt function

在中断函数中使用参数的型不一致。只有 void 型能指定为参数的型。

C2335 (E) Missing parameter declaration in #pragma interrupt function

对中断函数声明#pragma interrupt 的堆栈替换指定（sp）或者中断结束指定（sy），使用了没有声明的变量或者函数。

C2336 (E) Parameter out of range in #pragma interrupt function

中断函数声明#pragma interrupt 的参数 tn 值超过了 3。

C2337 (E) Illegal #pragma interrupt function type

中断函数声明有错。

C2340 (E) Illegal #pragma abs8 declaration

短绝对地址变量声明有错。

C2341 (E) Variable "变量名" in #pragma abs8 already declared

短绝对地址变量声明#pragma abs8 指定的变量已被声明为变量。

C2342 (E) Illegal #pragma abs8 symbol type

用变量名以外的型声明了短绝对地址变量声明#pragma abs8 指定的变量。

C2345 (E) Illegal #pragma abs16 declaration

短绝对地址变量声明有错。

C2346 (E) Variable "变量名" in #pragma abs16 already declared

短绝对地址变量声明#pragma abs16 指定的变量已被声明为变量。

C2347 (E) Illegal #pragma abs16 symbol type

用变量名以外的型声明了短绝对地址变量声明#pragma abs16 指定的变量。

C2350 (E) Illegal section name declaration

#pragma section 的指定有错。

C2352 (E) Section name table overflow

段的总数超过了 65280 个。

C2353 (E) Section size overflow regarding "段名"

段的大小超过了 32KB。

C2360 (E) Illegal #pragma indirect declaration

存储器间接函数声明有错。

C2361 (E) Function "函数名" in #pragma indirect already declared
存储器间接函数声明#pragma indirect 指定的函数已被声明为函数。

C2362 (E) Illegal #pragma indirect function type
用非函数型声明或者定义了存储器间接函数声明#pragma indirect 指定的函数。

C2363 (E) Too many indirect identifiers
1 个文件能指定的存储器间接函数的个数超过了限制。1 个文件能指定的存储器间接函数的个数是 256 个。

C2370 (E) Illegal #pragma regsave/noregsave declaration
#pragma regsave 或者#pragma noregsave 声明有错。

C2371 (E) Function "函数名" in #pragma regsave/noregsave already declared
#pragma regsave 或者#pragma noregsave 指定的函数已被声明为函数。

C2372 (E) Illegal #pragma regsave/noregsave function type
用非函数型声明或者定义了#pragma regsave 或者#pragma noregsave 指定的函数。

C2380 (E) Illegal #pragma inline/inline_asm declaration
#pragma inline 或者#pragma inline_asm 声明有错。

C2381 (E) Function "函数名" in #pragma inline/inline_asm already declared
#pragma inline 或者#pragma inline_asm 指定的函数已被声明为函数。

C2382 (E) Illegal #pragma inline/inline_asm function type
用非函数型声明或者定义了#pragma inline 或者#pragma inline_asm 指定的函数。

C2383 (E) #pragma inline_asm ignored
在目标格式为可再定位目标程序时指定了#pragma inline_asm。

C2390 (E) Illegal #pragma global_register declaration
#pragma global_register 声明有错。

C2391 (E) Variable "变量名" in #pragma global_register already declared
#pragma global_register 指定的变量已被声明为变量。

C2392 (E) Illegal #pragma global_register symbol type
用非变量的型声明了#pragma global_register 指定的变量。

C2393 (E) Illegal register
#pragma global_register 指定的寄存器名有错，或者被重复指定。

C2400 (E) Illegal character "字符"
有不正确的字符。

C2401 (E) Incomplete character constant
字符常数中有换行字符。

C2402 (E) Incomplete string
字符串中有换行字符。

C2403 (E) EOF in comment
在注释中有文件结束字符。

C2404 (E) Illegal character code "字符码"
有不正确的字符码。

C2405 (E) Null character constant
在字符常数中没有指定字符，即指定了' '格式的字符常数。

C2407 (E) Incomplete logical line
在非空的源文件的最后字符中指定了反斜线"＼"，或者在反斜线之后指定了换行字符"＼(RET)"。

C2408 (E) Comment nest too deep
注释的嵌套超过 255 层。

C2410 (E) Illegal #pragma entry declaration
#pragma entry 声明有语法错误。

C2411 (E) Function "函数名" in #pragma entry already declared
在#pragma entry 声明前有同名的符号，或者已指定了#pragma。

C2412 (E) Illegal #pragma entry function type
指定的符号不是函数。

C2413 (E) Multiple #pragma entry declaration
有多个#pragma entry 声明。

C2420 (E) Illegal #pragma pack/unpack declaration
#pragma pack、#pragma unpack 声明有语法错误。

C2440 (E) Illegal #pragma stacksize declaration
#pragma stacksize 声明有语法错误。

C2441 (E) Multiple #pragma stacksize declaration
有多个#pragma stacksize 声明。

C2442 (E) Stack size overflow
#pragma stacksize 指定的堆栈容量太大。

C2450 (E) Illegal #pragma option declaration
#pragma option 声明有错。

C2460 (E) Pragma kind mismatch
在声明之间#pragma 种类不一致。

C2470 (E) Illegal #pragma bit_order declaration
#pragma bit_order 声明有错。

C2480 (E) Illegal #pragma address declaration
#pragma address 指定行有错。

C2481 (E) Variable "变量名" in #pragma address already declared
#pragma address 指定的变量已经被不能同时指定的#pragma 定义了。

C2482 (E) Illegal #pragma address symbol type
用#pragma address 指定了变量名以外的符号。

C2483 (E) Illegal address in #pragma address
（1）对边界调整数 2 的变量或者结构体，指定了奇数地址。
（2）对不同的变量指定了相同地址，或者变量的地址有重复。
（3）2 个 address 的指定有重叠。

C2500 (E) Illegal token "语句"
语句的排列不符合语法。

C2501 (E) Division by zero
在常数表达式中整型数据被零除。

C2510 (E) Missing {
没有_ _asm 块开始的"{"。

C2511 (E) Illegal mnemonic
助记符不正确。

C2512 (E) Member reference required for "offset"
offset 运算符用于非成员参照表达式。

C2513 (E) Number of operands mismatch
操作数个数不正确。

C2514 (E) Illegal addressing mode
操作数的寻址方式不正确。

C2515 (E) Illegal register list
寄存器列表的指定不正确。

C2516 (E) Constant required
指定了非常数表达式。

C2517 (E) Illegal value in operand
给操作数指定了范围以外的值。

C2518 (E) Invalid delay slot instruction
将不正确的指令分配到延迟槽。

C2600 (E) #error : "字符串"
如果没有指定 nolist 选项，就将#error 的字符串指定的错误信息显示在列表文件。

C2801 (E) Illegal parameter type in intrinsic function
在内部函数中参数的型不一致。

C2802 (E) Parameter out of range in intrinsic function
在内部函数中参数大小超过了可指定的范围。

C2803 (E) Usage for intrinsic function is wrong
内部函数的使用方法有错。

C3000 (F) Statement nest too deep
if 语句、while 语句、for 语句、do 语句和 switch 语句的嵌套超过了 256 层。

C3006 (F) Too many parameters
在函数声明或者调用中，参数的个数超过了 63 个。

C3007 (F) Too many macro parameters
在宏定义或调用中，参数的个数超过了 63 个。

C3008 (F) Line too long
宏展开后的 1 行的长度超过了 16384 个字符。

C3009 (F) String literal too long

字符串的长度超过了 32767 个字符。字符串的长度是将连续指定的字符串连接后的字节数。在此，字符串的长度不是源程序中的长度，而是包含在字符串数据中的字节数，转义序列也计为 1 个字符。

C3013 (F) Too many switches

switch 语句的个数超过了 2048 个。

C3014 (F) For nest too deep

for 语句的嵌套超过了 128 层。

C3017 (F) Too many case labels

在一个 switch 语句中 case 标号的个数超过了 511 个。

C3018 (F) Too many goto labels

在一个函数中定义的 goto 标号的个数超过了 511 个。

C3019 (F) Cannot open source file "文件名"

不能打开源文件。

C3020 (F) Source file input error

不能读源文件或者 include 文件。

C3021 (F) Memory overflow

不能分配编译程序内部使用的存储区。

C3022 (F) Switch nest too deep

switch 语句的嵌套超过了 128 层。

C3023 (F) Type nest too deep

限定基本型的型（指针型、数组型和函数型）的个数超过了 16 个。

C3024 (F) Array dimension too deep

数组的维数超过了 6 维。

C3025 (F) Source file not found

在命令行中没有指定源文件名。

C3026 (F) Expression too complex

表达式太复杂。

C3027 (F) Source file too complex

程序语句的嵌套太深或者表达式太复杂。

C3030 (F) Too many compound statements

在 1 个函数中复合语句的个数超过了 2048 个。

C3031 (F) Data size overflow

数组或者结构体的大小超过了限制值。数组或者结构体的限制值在 CPU/运行模式是
H8SXN、2600n、2000n、300hn、300 时为 65535；
H8SXA 和 H8SXX 并且指定了 ptr16 选项时、或者是 H8SXM 时为 32767；
H8SXA:20、2600a:20、2000a:20、300ha:20 时为 1048575；
H8SXA:24、2600a:24、2000a:24、300ha:24 时为 16777215；
H8SXA:28、H8SXX:28、2600a:28、2000a:28 时为 268435455；
H8SXX:32、H8SXX:32、2600a:32、2000a:32 时为 4294967295。

C3034 (F) Invalid file name "文件名"

文件名的指定不正确。

C3200 (F) Object size overflow

目标大小超过了存储器的限制。
目标大小的限制值在 CPU/运行模式是
H8SXN、2600n、2000n、300hn、300 时为 65535；
H8SXM:20、H8SXA:20、2600a:20、2000a:20、300ha:20 时为 1048575；
H8SXM:24、H8SXA:24、2600a:24、2000a:24,300ha:24 时为 16777215；
H8SXA:28、H8SXX:28、2600a:28、2000a:28 时为 268435455；
H8SXA:32、H8SXX:32、2600a:32、2000a:32 时为 4294967295。

C3201 (F) Object data size overflow

数据长度超过了限制值。数据长度的限制值在 CPU/运行模式是
H8SXN、H8SXM、2600n、2000n、300hn、300 时为 65535；
H8SXA 和 H8SXX 并且指定了 ptr16 选项时为 65535；
H8SXA:20、2600a:20、2000a:20、300ha:20 时为 1048575；
H8SXA:24、2600a:24、2000a:24、300ha:24 时为 16777215；
H8SXA:28、H8SXX:28、2600a:28、2000a:28 时为 268435455；
H8SXX:32、H8SXX:32、2600a:32、2000a:32 时为 4294967295。

C3202 (F) Illegal stack access

局部变量/临时区和寄存器保存区超出了对于堆栈指针(SP)或者帧指针(FP)的偏移制限值，或者参数区超出了对于 SP 或者 FP 的偏移制限值。
对于 SP、FP 的偏移限制值在 CPU/运行模式是
H8SXN、H8SXM、2600n、2000n、300hn、300 时为 32767；
H8SXA 和 H8SXX 并且指定了 ptr16 选项时为 32767；
H8SXA:20、2600a:20、2000a:20、300ha:20 时为 524287；
H8SXA:24、2600a:24、2000a:24、300ha:24 时为 8388607；
H8SXA:28、H8SXX:28、2600a:28、2000a:28 时为 134217727；
H8SXA:32、H8SXX:32、2600a:32、2000a:32 时为 2147483647。

C3300 (F) Cannot open internal file

不能打开编译程序内部使用的中间文件。

`C3301 (F) Cannot close internal file`
不能关闭编译程序内部生成的中间文件。请确认是否正在存取编译程序生成的中间文件。

`C3302 (F) Cannot input internal file`
不能读编译程序内部生成的中间文件。请确认是否正在存取编译程序生成的中间文件。

`C3303 (F) Cannot output internal file`
不能写编译程序内部生成的中间文件。请增大磁盘容量。

`C3304 (F) Cannot delete internal file`
不能删除编译程序内部生成的中间文件。请确认是否正在存取编译程序生成的中间文件。

`C3305 (F) Invalid command parameter "选项"`
编译程序选项的指定方法有错。

`C3306 (F) Interrupt in compilation`
在编译处理中从标准输入终端检测到“(cntl)+C”键的中断。

`C3307 (F) Compiler version mismatch in "文件名"`
构成编译程序的文件之间的版本不一致。请参照安装指南的组装方法，重新安装编译程序。

`C3320 (F) Command parameter buffer overflow`
命令行的指定超过了 4096 个字符。

`C3322 (F) Lacking cpu specification`
没有指定 CPU/运行模式。请通过 cpu 选项或者环境变量 H38CPU 指定 CPU/运行模式。

`C3323 (F) Illegal environment specified "环境变量"`
编译程序使用的环境变量（CH38TMP、H38CPU）的指定有错。

`C3324 (F) Cannot open subcommand file "文件名"`
不能打开子命令文件。

`C3325 (F) Cannot close subcommand file`
不能关闭子命令文件。请确认是否正在使用子命令文件。

`C3326 (F) Cannot input subcommand file`
不能读子命令文件。

C3327 (F) Cannot find "文件名"
找不到编译程序的执行文件。请再次确认文件名或者路径名。

C4xxx (-) Internal error
在编译程序的内部处理中发生了某些障碍。请将错误的发生情况与销售部门联系。

C5003 (F) #include file "文件名" includes itself
包含了"文件名"的自身文件。

C5004 (F) Out of memory
编译所需的存储器容量不足。请增大系统的存储器容量或者结束其它应用程序。

C5005 (F) Could not open source file "名称"
不能打开"名称"的文件。请确认文件名是否正确。

C5006 (E) Comment unclosed at end of file
没有注释的结束指定“*/”。

C5007 (E) (I) Unrecognized token
有无法认识的字句（宏时为(I)）。

C5008 (E) (I) Missing closing quote
没有字符串的结束指定“"”（宏时为(I)）。

C5009 (I) Nested comment is not allowed
/* */注释嵌套。

C5010 (E) "#" not expected here
在行的前头或者预处理器以外的位置指定了“#”。

C5011 (E) Unrecognized preprocessing directive
有无法认识的预处理器的关键字。

C5012 (E) Parsing restarts here after previous syntax error
重新开始字句的解析。

C5013 (E)(F) Expected a file name
需要文件名。
#include 语句为(F)、#line 语句为(E)。

C5014 (E) Extra text after expected end of preprocessing directive
在预处理器语句后记述了文本。

C5016 (F) "名称" is not a valid source file name
"名称"的文件无效。

C5017 (E) Expected a "]"
没有"]"。

C5018 (E) Expected a ")"
没有")"。

C5019 (E) Extra text after expected end of number
在数值后记述了文本。

C5020 (E) Identifier "名称" is undefined
没有定义"名称"的符号。

C5021 (W) Type qualifiers are meaningless in this declaration
指定了没有含义的型限定符。型限定符无效。

C5022 (E) Invalid hexadecimal number
16 进制数的记述有错。

C5024 (E) Invalid octal digit
8 进制数的记述有错。

C5025 (E) Quoted string should contain at least one character
字符常数为空。

C5026 (E) Too many characters in character constant
字符常数中的字符个数太多。

C5027 (W) Character value is out of range
字符的值超出了范围。超出的值被舍去。

C5028 (E) Expression must have a constant value
表达式的值不是常数。

C5029 (E) Expected an expression
需要表达式。

C5030 (E) Floating constant is out of range
浮点数的值超出了范围。

C5031 (E) Expression must have integral type
表达式的型必须是整型。

C5032 (E) Expression must have arithmetic type
表达式的型必须是算术型。

C5033 (E) Expected a line number
#line 语句需要行号。

C5034 (E) Invalid line number
#line 语句的行号无效。

C5035 (F) #error directive: "行号"
使用了#error 语句。

C5036 (E) The #if for this directive is missing
#if 语句的指定方法有错。

C5037 (E) The #endif for this directive is missing
#endif 语句的指定方法有错。

C5038 (W) Directive is not allowed -- an #else has already appeared
#else 语句已出现。忽视本指定。

C5039 (E) Division by zero
发生了被零除。

C5040 (E) Expected an identifier
需要标识符。

C5041 (E) Expression must have arithmetic or pointer type
表达式的型必须是算术型或者指针型。

C5042 (E) Operand types are incompatible ("型 1" and "型 2")
"型 1"和"型 2"的操作数的型不适合。

C5044 (E) Expression must have pointer type
表达式的型必须是指针型。

C5045 (W) #undef may not be used on this predefined name
不能取消系统定义的宏名。#undef 指定无效。

C5046 (W) This predefined name may not be redefined
不能重新定义系统定义的宏名。#define 指定无效。

C5047 (W) Incompatible redefinition of macro "名称" (declared at line "行号")
"名称"宏的重新定义与以前的定义不同。重新定义的宏有效。

C5049 (E) Duplicate macro parameter name
重复定义了宏的参数名。

C5050 (E) "##" may not be first in a macro definition
在#define 宏的最初位置指定了##。

C5051 (E) "##" may not be last in a macro definition
在#define 宏的最后位置指定了##。

C5052 (E) Expected a macro parameter name
没有紧接在#后的宏参数。

C5053 (E) Expected a ":"
需要":"。

C5054 (W) Too few arguments in macro invocation
宏展开时的实际参数不足。

C5055 (W) Too many arguments in macro invocation
宏展开时的实际参数太多。

C5056 (E) Operand of sizeof may not be a function
不能给 sizeof 运算的操作数指定函数。

C5057 (E) This operator is not allowed in a constant expression
不能在常数表达式中指定此运算符。

C5058 (E) This operator is not allowed in a preprocessing expression
不能在预处理器的表达式中指定此运算符。

C5059 (E) Function call is not allowed in a constant expression
不能在常数表达式中调用函数。

C5060 (E) This operator is not allowed in an integral constant expression
不能在整型常数表达式中指定此运算符。

C5061 (W) Integer operation result is out of range
整数的运算结果超出了值的范围。假设为溢出的高位被忽视的值。

C5062 (W) Shift count is negative
移位计数是负值。按指定内容进行运算。

C5063 (W) Shift count is too large
移位计数超过了有效位数。按指定的内容进行运算。

C5064 (W) Declaration does not declare anything
没有指定声明的符号。忽视声明。

C5065 (E) Expected a ";"
需要";"。

C5066 (E) Enumeration value is out of "int" range
enum 型成员的值超出了 int 型的范围。

C5067 (E) Expected a "}"
需要"}"。

C5068 (W) Integer conversion resulted in a change of sign
进行了伴有符号转换的整型转换。照样设定位串。

C5069 (W) Integer conversion resulted in truncation
进行了舍去高位字节的整型转换。设定舍去后的值。

C5070 (E) Incomplete type is not allowed
指定了不完全型。

C5071 (E) Operand of sizeof may not be a bit field
给 sizeof 运算符的操作数指定了位字段。

C5075 (E) Operand of "*" must be a pointer
*运算符的操作数的型不是指针型。

C5077 (E) This declaration has no storage class or type specifier
没有存储类或者型的指定。

C5079 (E) Expected a type specifier
需要型说明符。

C5080 (E) A storage class may not be specified here
在此不能指定存储类。

C5081 (E) More than one storage class may not be specified
不能指定多个存储类。

C5083 (W) Type qualifier specified more than once
指定了多个 const/volatile 限定符。忽视多余的指定。

C5084 (E) Invalid combination of type specifiers
型的组合不正确。

C5085 (E) Invalid storage class for a parameter
给形式参数指定了不适当的存储类。

C5086 (E) Invalid storage class for a function
给函数指定了不适当的存储类。

C5087 (E) A type specifier may not be used here
不能指定型。

C5088 (E) Array of functions is not allowed
不能指定将函数作为元素的数组。

C5089 (E) Array of void is not allowed
不能指定将 void 型作为元素的数组。

C5090 (E) Function returning function is not allowed
不能指定将函数型作为返回型的函数。

C5091 (E) Function returning array is not allowed
不能指定将数组作为返回型的函数。

C5093 (E) Function type may not come from a typedef
不能使用 typedef 声明的函数型。

C5094 (E) The size of an array must be greater than zero
数组的大小必须大于 0。

C5095 (E) Array is too large
数组太大。

C5097 (E) A function may not return a value of this type
函数不能返回此型的值。

C5098 (E) An array may not have elements of this type
数组不能将此型作为元素。

C5100 (E) Duplicate parameter name
形式参数的名称有重复。

C5101 (E) "名称" has already been declared in the current scope
在相同作用域内已存在"名称"的声明。

C5103 (E) Class is too large
类太大。

C5105 (E) Invalid size for bit field
位字段的大小不正确。

C5106 (E) Invalid type for a bit field
位字段的型不正确。

C5107 (E) Zero-length bit field must be unnamed
不能给长度为 0 的位字段命名。

C5108 (W) Signed bit field of length 1
指定带符号整型的长度 1 的位字段。用指定的型处理。

C5109 (E) Expression must have (pointer-to-) function type
表达式必须是指向函数型的指针型。

C5110 (E) Expected either a definition or a tag name
需要声明的定义或者标记符名。

C5111 (I) Statement is unreachable
是不被执行的语句。有可能通过优化被删除。

C5112 (E) Expected "while"
需要 while 关键字。

C5114 (E) Entity-kind "名称" was referenced but not defined
没有被参照的"名称"的定义。

C5115 (E) A continue statement may only be used within a loop
continue 语句在循环中有效。

C5116 (E) A break statement may only be used within a loop or switch
break 语句在循环或者 switch 语句中有效。

C5117 (W) Non-void entity-kind "名称" should return a value
非 void 型的函数没有返回值。返回值不定。

C5118 (E) A void function may not return a value
返回 void 型的函数不能有返回值。

C5119 (E) Cast to type "型" is not allowed
不能指定指向"型"的数据类型转换。

C5120 (E) Return value type does not match the function type
返回值和函数的型不一致。

C5121 (E) A case label may only be used within a switch
在 switch 语句以外的位置使用了 case 标号。

C5122 (E) A default label may only be used within a switch
在 switch 语句以外的位置使用了 default 标号。

C5123 (E) Case label value has already appeared in this switch
在 switch 语句中已存在 case 标号的值。

C5124 (E) Default label has already appeared in this switch
在 switch 语句中已存在 Default 标号的值。

C5125 (E) Expected a "("
需要"("。

C5126 (E) Expression must be an lvalue
表达式必须是左边值。

C5127 (E) Expected a statement
需要语句。

C5128 (I) Loop is not reachable from preceding code
不被执行的循环语句。

C5129 (E) A block-scope function may only have extern storage class
在块内声明的函数必须是 extern 存储类。

C5130 (E) Expected a "{"
需要"{"。

C5131 (E) Expression must have pointer-to-class type
表达式必须是指向类的指针型。

C5132 (E) Expression must have pointer-to-struct-or-union type
表达式必须是结构体或者联合体的指针型。

C5133 (E) Expected a member name
需要成员名。

C5134 (E) Expected a field name
需要字段名。

C5135 (E) Entity-kind "名称" has no member "成员名"
"名称"没有"成员名"。

C5136 (E) Entity-kind "名称" has no field "字段名"
"名称"没有"字段名"。

C5137 (E) Expression must be a modifiable lvalue
表达式必须是可修正的左边值。

C5139 (E) Taking the address of a bit field is not allowed
不能参照位字段的地址。

C5140 (E) Too many arguments in function call
函数调用的实际参数的个数太多。

C5142 (E) Expression must have pointer-to-object type
表达式必须是指向目标的指针型。

C5143 (F) Program too large or complicated to compile
程序太大或者太复杂。

C5144 (E) A value of type "型 1" cannot be used to initialize an entity of type "型 2"
初始值的"型 1"和变量的"型 2"不同。

C5145 (E) Entity-kind "名称" may not be initialized
不能初始化"名称"。

C5146 (E) Too many initializer values
初始值的个数太多。

C5147 (E) Declaration is incompatible with "名称"(declared at line "行号")
以前声明的"名称"的型不一致。

C5148 (E) Entity-kind "名称" has already been initialized
已设定"名称"的初始值。

C5149 (E) A global-scope declaration may not have this storage class
不能对全局作用域内的声明指定此存储类。

C5150 (E) A type name may not be redeclared as a parameter
在形式参数中不能重复声明型名。

C5151 (E) A typedef name may not be redeclared as a parameter
在形式参数中不能重复声明型名。

C5153 (E) Expression must have class type
表达式必须是类的型。

C5154 (E) Expression must have struct or union type
表达式必须是结构体或者联合体型。

C5157 (E) Expression must be an integral constant expression
表达式必须是整型的常数表达式。

C5158 (E) Expression must be an lvalue or a function designator
表达式必须为左边值或者函数名。

C5159 (E) Declaration is incompatible with previous "名称" (declared at line "行号")
与以前使用的"名称"的型不一致。

C5160 (E) Name conflicts with previously used external name "名称"
以前使用的外部名"名称"和名称有重复。

C5161 (I) Unrecognized #pragma
有无法认识的#pragma 指定。忽视#pragma 指定。

C5163 (F) Could not open temporary file "名称"
不能打开"名称"的临时文件。请确认编译程序的环境设定和主机环境的文件系统是否异常。

C5164 (F) Name of directory for temporary files is too long ("名称")
临时文件的"名称"太长。

C5165 (E) Too few arguments in function call
函数调用的实际参数的个数不足。

C5166 (E) Invalid floating constant
浮点常数的指定不正确。

C5167 (E) Argument of type "型 1" is incompatible with parameter of type "型 2"
实际参数的型"型 1"和形式参数的型"型 2"不一致。

C5168 (E) A function type is not allowed here
不允许函数型。

C5169 (E) Expected a declaration
需要声明。

C5170 (W) Pointer points outside of underlying object
指针指向的区域超过了目标的范围。

C5171 (E) Invalid type conversion
数据类型转换的型不正确。

C5172 (I) External/internal linkage conflict with previous declaration
以前的声明和外部/内部连接不同。假设内部连接。

C5173 (E) Floating-point value does not fit in required integral type
将浮点数型的值转换为整型时，超出了值的范围。

C5174 (I) Expression has no effect
无效果的表达式。有可能通过优化被删除。

C5175 (W) Subscript out of range
数组的下标超出了范围。用指定的索引继续处理。

C5177 (W) Entity-kind "entity" was declared but never referenced
有不被参照的声明。

C5179 (W) Right operand of "%" is zero
%运算符的右边值是 0。用指定的表达式来评价。

C5182 (F) Could not open source file "名称" (no directories in search list)
不能打开"名称"的文件。请确认目录是否存在。

C5183 (E) Type of cast must be integral
数据类型转换的型必须是整型。

C5184 (E) Type of cast must be arithmetic or pointer
数据类型转换的型必须是算术型或者指针型。

C5185 (I) Dynamic initialization in unreachable code
不执行初始化表达式。执行时不设定初始值。

C5186 (W) Pointless comparison of unsigned integer with zero
0 和无符号整数的无意义比较。按指定的内容评价表达式。

C5187 (I) Use of "=" where "==" may have been intended
"=="是有意图的表达式，使用了"="。按照指定的内容评价表达式。

C5189 (F) Error while writing "文件名" file
写文件失败。

C5191 (W) Type qualifier is meaningless on cast type
给数据类型转换的型指定了无含义的型限定符。忽视指定的型。

C5192 (W) Unrecognized character escape sequence
指定了无法认识的转义序列字符。照样使用值。

C5193 (I) Zero used for undefined preprocessing identifier
0 用于预处理器语句的判断式。按指定的内容评价表达式。

C5219 (F) Error while deleting file "文件名"
不能删除"文件名"的文件。

C5221 (W) Floating-point value does not fit in required floating-point type
不能将浮点型转换为被要求的浮点型。认为是无穷大的值。

C5224 (W) The format string requires additional arguments
实际参数的个数不足于格式字符串中要求的参数。

C5225 (W) The format string ends before this argument
实际参数的个数多于格式字符串中要求的参数。

C5226 (W) Invalid format string conversion
格式转换的格式和实际参数的型不同。

C5229 (W) Bit field cannot contain all values of the enumerated type
位字段不能保持全部 enum 型的值。舍去结果。

C5235 (E) Variable "名称" was declared with a never-completed type
声明了不完全型的"名称"变量。

C5236 (W)(I) Controlling expression is constant
控制表达式是常数(I)。控制表达式是地址常数(W)。按指定的内容评价表达式。

C5237 (I) Selector expression is constant
switch 语句的控制表达式是常数。

C5238 (E) Invalid specifier on a parameter
在参数声明中使用了不正确的说明符。

C5239 (E) Invalid specifier outside a class declaration
在类声明外使用了不正确的说明符。

C5240 (E) Duplicate specifier in declaration
在一个声明中重复使用了说明符。

C5241 (E) A union is not allowed to have a base class
union 型不能持有基类。

C5242 (E) Multiple access control specifiers are not allowed
重复使用了存取说明符。

C5243 (E) Class or struct definition is missing
不能对应 class 定义的括弧。

C5244 (E) Qualified name is not a member of class "型" or its base classes
限定名不是类或者基类的成员的"型"。

C5245 (E) A nonstatic member reference must be relative to a specific object
非静态成员的参照没有对应目标。

C5246 (E) A nonstatic data member may not be defined outside its class
不能在类以外的位置定义非静态数据成员。

C5247 (E) Entity-kind "名称" has already been defined
已定义了"名称"。

C5248 (E) Pointer to reference is not allowed
不允许指向参照型的指针型。

C5249 (E) Reference to reference is not allowed
不允许指向参照型的参照型。

C5250 (E) Reference to void is not allowed
不允许 void 型的参照型。

C5251 (E) Array of reference is not allowed
不允许参照型的数组。

C5252 (E) Reference entity-kind "名称" requires an initializer
参照型的定义"名称"需要初始值。

C5253 (E) Expected a ","
需要逗号","。

C5254 (E) Type name is not allowed
不允许型名。

C5255 (E) Type definition is not allowed
不允许型的定义。

C5256 (E) Invalid redeclaration of type name "名称" (declared at line "行号")
不能重新定义"名称"的型名。

C5257 (E) Const entity-kind "名称" requires an initializer
const 型的定义"名称"需要初始值。

C5258 (E) "this" may only be used inside a nonstatic member function
在非静态成员函数外使用了"this"。

C5259 (E) Constant value is not known
const 型的值不明确。

C5261 (I) Access control not specified ("名称" by default)
没有基类的存取控制指定。假设存取控制指定为"名称"。

C5262 (E) Not a class or struct name
没有基类指定的类或者结构体。

C5263 (E) Duplicate base class name
　　重复指定了基类。

C5264 (E) Invalid base class
　　基类不正确。

C5265 (E) Entity-kind "名称" is inaccessible
　　不能存取"名称"。

C5266 (E) "名称" is ambiguous
　　指定的"名称"不明确。

C5269 (E) Implicit conversion to inaccessible base class "型" is not allowed
　　不允许对不可存取的类进行隐含的型转换。

C5274 (E) Improperly terminated macro invocation
　　在宏调用中文件结束。

C5276 (E) Name followed by "::" must be a class or namespace name
　　紧接在::后的名称必须是类名或者 namespace 名。

C5277 (E) Invalid friend declaration
　　友元声明的指定不正确。

C5278 (E) A constructor or destructor may not return a value
　　构造函数和析构函数没有返回值。

C5279 (E) Invalid destructor declaration
　　析构函数的声明不正确。

C5280 (E)(W) Declaration of a member with the same name as its class
　　声明了和类名相同名称的成员名。
　　　(W) 非 static 变量名
　　　(E) static 变量名、typedef 名、enum 成员等

C5281 (E) Global-scope qualifier (leading "::") is not allowed
　　不允许全局作用域决定的运算符。

C5282 (E) The global scope has no "名称"
　　没有给全局作用域声明"名称"。

C5283 (E) Qualified name is not allowed
　　不允许限定名。

C5284 (W) NULL reference is not allowed
　　不允许参照 NULL。按指定的内容评价表达式。

C5285 (E) Initialization with "{...}" is not allowed for object of type "型"
在"型"的目标中不允许{ }格式的初始化。

C5286 (E) Base class "type" is ambiguous
基类的型不明确。

C5287 (E) Derived class "type" contains more than one instance of class "型"
派生型包含多个相同"型"的类。

C5288 (E) Cannot convert pointer to base class "型 1" to pointer to derived class "型 2" -- base class is virtual
不能将虚拟基类"型 1"的指针型转换为派生类"型 2"的指针型。

C5289 (E) No instance of constructor "名称" matches the argument list
构造函数"名称"的参数不一致。

C5290 (E) Copy constructor for class "型" is ambiguous
类"型"的复制构造函数不明确。

C5291 (E) No default constructor exists for class "型"
没有类"型"的默认构造函数。

C5292 (E) "名称" is not a nonstatic data member or base class of class "型"
"名称"不是非静态数据成员或者基类"型"。

C5293 (E) Indirect nonvirtual base class is not allowed
不允许非虚拟的间接基类。

C5294 (E) Invalid union member -- class "型" has a disallowed member function
存在不能指定为 union 成员的类"型"的成员函数。

C5297 (E) Expected an operator
需要运算符。

C5298 (E) Inherited member is not allowed
不能使用继承的成员。

C5299 (E) Cannot determine which instance of entity-kind "名称" is intended
不能决定重载函数的"名称"。

C5300 (E) A pointer to a bound function may only be used to call the function
指向成员函数的指针用于非函数调用。

C5302 (E) Entity-kind "名称" has already been defined
已定义了函数"名称"。

C5304 (E) No instance of entity-kind "名称" matches the argument list
函数"名称"的参数不一致。

C5305 (E) Type definition is not allowed in function return type declaration
不能用函数的返回型的声明定义型。

C5306 (E) Default argument not at end of parameter list
默认参数的声明不在参数列表的最后。

C5307 (E) Redefinition of default argument
重新定义默认参数。

C5308 (E) More than one instance of entity-kind "名称" matches the argument list:
参数列表一致，重载函数"名称"不明确。

C5309 (E) More than one instance of constructor "名称" matches the argument list:
参数列表一致，构造函数不明确。

C5310 (E) Default argument of type "型 1" is incompatible with parameter of type "型 2"
默认值的"型 1"和参数的"型 2"不一致。

C5311 (E) Cannot overload functions distinguished by return type alone
不能重载返回型不同的函数。

C5312 (E) No suitable user-defined conversion from "型 1" to "型 2" exists
没有从"型 1"到"型 2"的适当的用户定义转换。

C5313 (E) Type qualifier is not allowed on this function
不能给函数指定型限定符(const、volatile)。

C5314 (E) Only nonstatic member functions may be virtual
给静态成员函数指定了 virtual。

C5315 (E) The object has type qualifiers that are not compatible with the member function
目标的型限定符(const、volatile)和成员函数的型限定符不一致。

C5316 (E) Program too large to compile (too many virtual functions)
虚拟函数的个数太多。

C5317 (E) Return type is not identical to nor covariant with return type "型" of overridden virtual function entity-kind "名称"
虚拟函数"名称"的返回型"型"不同。

C5318 (E) Override of virtual entity-kind "名称" is ambiguous
虚拟函数"名称"的置换不明确。

C5319 (E) Pure specifier ("= 0") allowed only on virtual functions
在非虚拟函数中指定了纯说明符"=0"。

C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)
纯说明符的格式不正确。只允许"=0"。

C5321 (E) Data member initializer is not allowed
数据成员的初始化指定不正确。

C5322 (E) Object of abstract class type "型" is not allowed:
不能定义抽象类"型"的目标。

C5323 (E) Function returning abstract class "型" is not allowed:
不能定义返回抽象类"型"的函数。

C5324 (I) Duplicate friend declaration
重复指定了友元声明。

C5325 (E) Inline specifier allowed on function declarations only
inline 说明符只在函数声明中有效。

C5326 (E) "inline" is not allowed
不允许 inline 指定。

C5327 (E) Invalid storage class for an inline function
inline 函数的存储类不正确。

C5328 (E) Invalid storage class for a class member
类成员的存储类不正确。

C5329 (E) Local class member entity-kind "名称" requires a definition
没有局部类成员"名称"的定义。

C5330 (E) Entity-kind "名称" is inaccessible
不能存取"名称"。

C5332 (E) Class "type" has no copy constructor to copy a const object
在类"型"中没有复制 const 型目标的复制构造函数。

C5333 (E) Defining an implicitly declared member function is not allowed
　　不能定义隐含声明的成员函数。

C5334 (E) Class "型" has no suitable copy constructor
　　类"型"没有适当的复制构造函数。

C5335 (E) Linkage specification is not allowed
　　不能指定连接说明符。

C5336 (E) Unknown external linkage specification
　　指定了无法认识的连接指定。

C5337 (E) Linkage specification is incompatible with previous "名称"
　　(declared at line "行号")
　　与以前指定的连接说明符"名称"不一致。

C5338 (E) More than one instance of overloaded function "名称" has "C" linkage
　　存在具有 C 连接的多个重载函数"名称"。

C5339 (E) Class "型" has more than one default constructor
　　类"型"有多个默认构造函数。

C5341 (E) "operator 运算符" must be a member function
　　"运算符"的运算符函数必须是成员函数。

C5342 (E) Operator may not be a static member function
　　不允许静态成员函数的运算符函数。

C5343 (E) No arguments allowed on user-defined conversion
　　对于用户定义转换，不允许参数。

C5344 (E) Too many parameters for this operator function
　　运算符函数的参数的个数太多。

C5345 (E) Too few parameters for this operator function
　　运算符函数的参数的个数不足。

C5346 (E) Nonmember operator requires a parameter with class type
　　非成员函数的运算符函数需要类的型的参数。

C5347 (E) Default argument is not allowed
　　不允许默认参数。

C5348 (E) More than one user-defined conversion from "型 1" to "型 2" applies:
　　从"型 1"到"型 2"的用户定义型转换不明确。

C5349 (E) No operator "运算符" matches these operands
运算符函数"运算符"的操作数不一致。

C5350 (E) More than one operator "运算符" matches these operands:
运算符函数"运算符"的操作数不明确。

C5351 (E) First parameter of allocation function must be of type "size_t"
operator new 的第一参数必须是 size_t 型。

C5352 (E) Allocation function requires "void *" return type
operator new 的返回型必须是 void *型。

C5353 (E) Deallocation function requires "void" return type
operator delete 的返回型必须是 void 型。

C5354 (E) First parameter of deallocation function must be of type "void *"
operator delete 的第一参数必须是 void *型。

C5356 (E) Type must be an object type
型必须是目标型。

C5357 (E) Base class "type" has already been initialized
基类已被初始化。

C5359 (E) Entity-kind "名称" has already been initialized
"名称"已被初始化。

C5360 (E) Name of member or base class is missing
成员名或者基类有错。

C5363 (E) Invalid anonymous union -- nonpublic member is not allowed
无名 union 的成员不是公开成员。

C5364 (E) Invalid anonymous union -- member function is not allowed
在无名 union 中不允许成员函数。

C5365 (E) Anonymous union at global or namespace scope must be declared static
需要 static 声明全局或者 namespace 作用域的无名 union。

C5366 (E) Entity-kind "名称" provides no initializer for:
不能给"名称"指定初始化。

C5367 (E) Implicitly generated constructor for class "型" cannot initialize:
不能初始化被隐含生成的类"型"构造函数。

C5368 (E) Entity-kind "名称" defines no constructor to initialize the following:
"名称"没有定义初始化的构造函数。

C5369 (E) Entity-kind "名称" has an uninitialized const or reference member
"名称"的 const 或者参照成员没有被初始化。

C5370 (W) Entity-kind "名称" has an uninitialized const field
"名称"的 const 字段没有被初始化。

C5371 (E) Class "型" has no assignment operator to copy a const object
没有定义复制 const 目标的类"型"的赋值运算符函数。

C5372 (E) Class "型" has no suitable assignment operator
类"型"没有定义适当的赋值运算。

C5373 (E) Ambiguous assignment operator for class "型"
类"型"的赋值运算符函数不明确。

C5375 (E) Declaration requires a typedef name
需要 typedef 名的声明。

C5377 (E) "virtual" is not allowed
不能指定 virtual。

C5378 (E) "static" is not allowed
不能指定 static。

C5380 (E) Expression must have pointer-to-member type
表达式必须是指向成员的指针型。

C5381 (I) Extra ";" ignored
忽视了多余的";"。

C5382 (W) Nonstandard member constant declaration (standard form is a static const integral member)
const 成员的声明不是标准格式。初始化无效。

C5384 (E) No instance of overloaded "名称" matches the argument list
重载函数"名称"的参数列表不一致。

C5386 (E) No instance of entity-kind "名称" matches the required type
不存在被要求的型的重载函数"名称"。

C5388 (E) "operator->" for class "型 1" returns invalid type "型 2"
类"型 1"的 operator->运算函数的返回型"型 2"不正确。

C5389 (E) A cast to abstract class "型" is not allowed:
不允许抽象类"型"的数据类型转换。

C5391 (E) A new-initializer may not be specified for an array
不能根据 new 初始化数组。

C5392 (E) Member function "名称" may not be redeclared outside its class
在类的外侧重新声明了成员函数"名称"。

C5393 (E) Pointer to incomplete class type is not allowed
不允许指向不完全类的指针型。

C5394 (E) Reference to local variable of enclosing function is not allowed
不允许参照包含局部类的函数的局部变量。

C5397 (E) Implicitly generated assignment operator cannot copy:
隐含生成的赋值运算符函数不能正确地复制目标。

C5399 (I) Entity-kind "名称" has an operator newxxxx() but no default operator deletexxxx()
"名称"具有 operator new，但是不具有默认的 operator delete。

C5400 (I) Entity-kind "名称" has a default operator deletexxxx() but no operator newxxxx()
"名称"具有默认的 operator delete，但是不具有 operator new。

C5401 (E) Destructor for base class "型" is not virtual
基类"型"的析构函数不是 virtual。

C5403 (E) Entity-kind "名称" has already been declared
重新声明了成员函数"名称"。

C5404 (E) Function "main" may not be declared inline
不能 inline 声明 main 函数。

C5405 (E) Member function with the same name as its class must be a constructor
和类名相同名称的成员函数必须是构造函数。

C5407 (E) A destructor may not have parameters
析构函数不能有参数。

C5408 (E) Copy constructor for class "型 1" may not have a parameter of type "型 2"
类"型 1"的复制构造函数不能有"型 2"的参数。

C5409 (E) Entity-kind "名称" returns incomplete type "型"
函数"名称"的返回型是不完全型"型"。

C5410 (E) Protected entity-kind "名称" is not accessible through a "型" pointer or object
不能经过指向"型"的指针或目标后存取限定公开名"名称"。

C5411 (E) A parameter is not allowed
不允许形式参数。

C5412 (E) An "asm" declaration is not allowed here
不允许 asm 声明。

C5413 (E) No suitable conversion function from "型 1" to "型 2" exists
没有从"型 1"到"型 2"的适当的转换函数。

C5414 (W) Delete of pointer to incomplete class
删除了指向不完全型类的指针。

C5415 (E) No suitable constructor exists to convert from "型 1" to "型 2"
没有从"型 1"转换到"型 2"的适当的构造函数。

C5416 (E) More than one constructor applies to convert from "型 1" to "型 2":
从"型 1"转换到"型 2"的构造函数不明确。

C5417 (E) More than one conversion function from "型 1" to "型 2" applies:
从"型 1"转换到"型 2"的函数不明确。

C5418 (E) More than one conversion function from "型" to a built-in type applies:
从"型"转换到 built-in 型的函数不明确。

C5424 (E) A constructor or destructor may not have its address taken
不能参照构造函数或者析构函数的地址。

C5427 (E) Qualified name is not allowed in member declaration
在成员声明中不能使用限定名。

C5429 (E) The size of an array in "new" must be non-negative
对于 new 指定的数组大小，不允许负值。

C5430 (W) Returning reference to local temporary
在函数中已将局部区域的参照设定到返回值。

C5432 (E) "enum" declaration is not allowed
不允许 enum 型声明。

C5433 (E) Qualifiers dropped in binding reference of type "型 1" to initializer of type "型 2"
给参照型"型 1"的初始值指定了 const/volatile 限定的型"型 2"。

C5434 (E) A reference of type "型 1" (not const-qualified) cannot be initialized with a value of type "型 2"
不能用"型 2"的值初始化非 const 型限定的型"型 1"的参照。

C5435 (E) A pointer to function may not be deleted
不能删除指向函数的指针。

C5436 (E) Conversion function must be a nonstatic member function
转换函数必须是非静态成员函数。

C5437 (E) Template declaration is not allowed here
在此作用域中不允许模板声明。

C5438 (E) Expected a "<"
需要"<"。

C5439 (E) Expected a ">"
需要">"。

C5440 (E) Template parameter declaration is missing
模板的参数声明不正确。

C5441 (E) Argument list for entity-kind "名称" is missing
模板"名称"的实际参数列表不正确。

C5442 (E) Too few arguments for entity-kind "名称"
模板"名称"的实际参数不足。

C5443 (E) Too many arguments for entity-kind "名称"
模板的实际参数太多。

C5445 (E) Entity-kind "名称 1" is not used in declaring the parameter types of entity-kind "名称 2"
不使用模板"名称 1"的参数"名称 2"。

C5449 (E) More than one instance of entity-kind "名称" matches the required type
重载函数"名称"不明确。

C5452 (E) Return type may not be specified on a conversion function
没有指定转换函数的返回型。

C5456 (E) Excessive recursion at instantiation of entity-kind "名称"
模板"名称"的示例被递归生成。

C5457 (E) "名称" is not a function or static data member
"名称"不是函数或者静态数据成员。

C5458 (E) Argument of type "型 1" is incompatible with template parameter of type "型 2"
实际参数的型"型 1"与模板的参数"型 2"不一致。

C5459 (E) Initialization requiring a temporary or conversion is not allowed
不允许初始化要求临时和转换。

C5461 (E) Initial value of reference to non-const must be an lvalue
非 const 型的参照的初始值必须是左边值。

C5463 (E) "template" is not allowed
不允许"template"指定。

C5464 (E) "型" is not a class template
"型"不是类模板。

C5466 (E) "main" is not a valid name for a function template
"main"不能用于函数模板的名称。

C5467 (E) Invalid reference to entity-kind "名称" (union/nonunion mismatch)
"名称"的参照不正确。

C5468 (E) A template argument may not reference a local type
模板的实际参数不能参照局部型。

C5469 (E) Tag kind of "名称 1" is incompatible with declaration of entity-kind "名称 2" (declared at line "行号")
标记符名"名称 1"的种类和"名称 2"的声明不一致。

C5470 (E) The global scope has no tag named "名称"
在全局作用域中没有标记符名"名称"。

C5471 (E) Entity-kind "名称 1" has no tag member named "名称 2"
"名称 1"没有标记符成员"名称 2"。

C5473 (E) Entity-kind "名称" may be used only in pointer-to-member declaration
在指向成员的指针型的声明中必须使用 typedef 名"名称"。

C5475 (E) A template argument may not reference a non-external entity
模板的实际参数不能参照非外部名。

C5476 (E) Name followed by "::~" must be a class name or a type name
紧接在::~后的名称必须是类名或者型名。

C5477 (E) Destructor name does not match name of class "型"
类名"型"和析构函数名不一致。

C5478 (E) Type used as destructor name does not match type "型"
用于析构函数名的型和"型"不一致。

C5479 (I) Entity-kind "名称" redeclared "inline" after being called
在调用函数后声明了 inline"名称"。此后，inline 指定有效。

C5481 (E) Invalid storage class for a template declaration
模板声明的存储类指定不正确。

C5484 (E) Invalid explicit instantiation declaration
模板的实际参数不正确。

C5485 (E) Entity-kind "名称" is not an entity that can be instantiated
不能进行模板"名称"的实体化。

C5486 (E) Compiler generated entity-kind "entity" cannot be explicitly instantiated
不能对编译程序生成的函数进行实体化。

C5487 (E) Inline entity-kind "名称" cannot be explicitly instantiated
不能进行直接插入函数"名称"的实体化。

C5488 (E) Pure virtual entity-kind "名称" cannot be explicitly instantiated
不能进行纯虚拟函数"名称"的实体化。

C5489 (E) Entity-kind "名称" cannot be instantiated -- no template definition was supplied
由于没有定义模板，所以不能进行"名称"的实体化。

C5490 (E) Entity-kind "名称" cannot be instantiated -- it has been explicitly specialized
不能进行"名称"的实体化。

C5493 (E) No instance of entity-kind "名称" matches the specified type
重载函数"名称"和被指定的型不一致。

C5496 (E) Template parameter "名称" may not be redeclared in this scope
在作用域中重复声明了模板参数"名称"。

C5497 (W) Declaration of "名称" hides template parameter
"名称"的声明隐藏模板参数。

C5498 (E) Template argument list must match the parameter list
模板的实际参数和形式参数不一致。

C5499 (E) Conversion function to convert from "型 1" to "型 2" is not allowed
不允许从"型 1"到"型 2"的函数转换。

C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"
后缀运算函数的第 2 参数的型必须是 int 型。

C5501 (E) An operator name must be declared as a function
必须将运算符名声明为函数。

C5502 (E) Operator name is not allowed
不允许运算符名。

C5503 (E) Entity-kind "名称" cannot be specialized in the current scope
在作用域中"名称"不明确。

C5505 (E) Too few template parameters -- does not match previous declaration
模板的参数不足。

C5506 (E) Too many template parameters -- does not match previous declaration
模板的参数太多。

C5507 (E) Function template for operator delete(void *) is not allowed
不允许 operator delete(void *)的函数模板。

C5508 (E) Class template and template parameter may not have the same name
类模板和模板的参数同名。

C5510 (E) A template argument may not reference an unnamed type
参照了没有命名的模板实际参数的型。

C5511 (E) Enumerated type is not allowed
不允许 enum 型。

C5512 (W) Type qualifier on a reference type is not allowed
不能给参照型指定 const/volatile 限定。

C5513 (E) A value of type "型 1" cannot be assigned to an entity of type "型 2"
由于型不一致，所以"型 1"的值不能赋值到"型 2"的实体。

C5514 (W) Pointless comparison of unsigned integer with a negative constant
对负常数和无符号整数进行了比较。

C5515 (E) Cannot convert to incomplete class "型"
不能型转换到不完全型"型"。

C5516 (E) Const object requires an initializer
const 型的目标需要初始值。

C5517 (E) Object has an uninitialized const or reference member
目标持有未初始化的 const 型成员或者参照型成员。

C5519 (E) Entity-kind "名称" may not have a template argument list
"名称"不能持有模板实际参数。

C5520 (E) Initialization with "{...}" expected for aggregate object
必须用{...}的格式初始化集合型对象。

C5521 (E) Pointer-to-member selection class types are incompatible ("型 1" and "型 2")
成员指针型的类的型"型 1"和"型 2"不一致。

C5522 (W) Pointless friend declaration
对自身进行了友元声明。

C5526 (E) A parameter may not have void type
不能指定 void 型的参数。

C5529 (E) This operator is not allowed in a template argument expression
不允许模板的实际参数表达式指定的运算。

C5530 (E) Try block requires at least one handler
没有对应 try 语句的 catch 语句。

C5531 (E) Handler requires an exception declaration
catch 语句的(...)需要异常声明。

C5532 (E) Handler is masked by default handler
根据默认处理程序进行了处理程序的屏蔽。

C5533 (E) Handler is potentially masked by previous handler for type "型"
有可能根据具有"型"的以前的处理程序进行处理程序的屏蔽。

C5534 (I) Use of a local type to specify an exception
指定了使用局部型的异常处理。

C5535 (I) Redundant type in exception specification
在异常处理中有冗余型的指定。

C5536 (E) Exception specification is incompatible with that of previous entity-kind "名称" (declared at line "行号"):
异常处理的指定和以前指定的"名称"不一致。

C5540 (E) Support for exception handling is disabled
没有指定进行异常处理的选项(exception)。

C5541 (W) Omission of exception specification is incompatible with previous entity-kind "名称" (declared at line "行号")
异常处理的省略形式和以前的"名称"不一致。

C5542 (F) Could not create instantiation request file "名称"
在进行模板的实体化时，不能建立使用的文件"名称"。

C5543 (E) Non-arithmetic operation not allowed in nontype template argument
对应的模板实际参数不允许非算术型转换。

C5544 (E) Use of a local type to declare a nonlocal variable
给非局部变量指定了局部型。

C5545 (E) Use of a local type to declare a function
给函数声明指定了局部型。

C5546 (E) Transfer of control bypasses initialization of:
不进行初始化处理。

C5548 (E) Transfer of control into an exception handler
进入异常处理程序。

C5549 (I) Entity-kind "名称" is used before its value is set
在给"名称"设定值之前，使用了"名称"。

C5550 (W) Entity-kind "名称" was set but never used
没有使用"名称"。

C5551 (E) Entity-kind "名称" cannot be defined in the current scope
在此作用域中不能定义"名称"。

C5552 (W) Exception specification is not allowed
不允许异常处理的指定。异常处理无效。

C5553 (W) External/internal linkage conflict for entity-kind "名称"
(declared at line "行号")
"名称"的外部/内部连接指定发生冲突。设定外部连接。

C5554 (W) Entity-kind "名称" will not be called for implicit or explicit
conversions
不能既隐含又明确地调用转换函数"名称"。

C5555 (E) Tag kind of "名称" is incompatible with template parameter of type
"型"标记符"名称"的种类和模板的参数的"型"不一致。

C5556 (E) Function template for operator new(size_t) is not allowed
不允许 operator new(size_t)的函数模板。

C5558 (E) Pointer to member of type "型" is not allowed
指向成员的指针型"型"有错。

C5559 (E) Ellipsis is not allowed in operator function parameter list
不能给运算符函数的参数列表省略指定(...)。

C5598 (E) A template parameter may not have void type
不能给模板的参数指定 void 型。

C5601 (E) A throw expression may not have void type
不能给 throw 表达式指定 void 型。

C5603 (E) Parameter of abstract class type "型" is not allowed:
不允许抽象类"型"的参数。

C5604 (E) Array of abstract class "型" is not allowed:
不允许抽象类"型"的数组。

C5610 (W) Entity-kind "名称 1" does not match "名称 2" -- virtual function
override intended?
"名称 1"和"名称 2"不一致。按指定的内容继续处理。

C5611 (W) Overloaded virtual function "名称 1" is only partially overridden
in entity-kind "名称 2"
"名称 1"的重载虚拟函数在"名称 2"中只有部分的虚拟函数为置换对象。按指定的内容继续处理。

C5612 (E) Specific definition of inline template function must precede its
first use
必须在调用前定义直接插入指定的模板函数。

C5624 (E) "名称" is not a type name
"名称"不是型的名称。

C5641 (F) "名称" is not a valid directory
"名称"不是正确的目录。

C5642 (F) Cannot build temporary file name
不能建立编译程序使用的临时文件。

C5656 (E) Transfer of control into a try block
控制从外侧的块转移到 try 块。

C5657 (W) Inline specification is incompatible with previous "名称" (declared at line "行号")
直接插入指定和以前的声明"名称"不一致。

C5658 (E) Closing brace of template definition not found
没有模板定义的右括弧。

C5660 (E) Invalid packing alignment value
Pack 的值不正确。忽视指定。

C5662 (W) Call of pure virtual function
纯虚拟函数调用了函数。

C5663 (E) Invalid source file identifier string
#pragma 指定的语法有错。

C5664 (E) A class template cannot be defined in a friend declaration
在友元声明中不能定义类模板。

C5673 (E) A reference of type "型 1" cannot be initialized with a value of type "型 2"
不能用"型 2"的值初始化 const/volatile 型"型 1"的参照。

C5674 (E) Initial value of reference to const volatile must be an lvalue
const/volatile 型的参照的初始值必须是左边值。

C5678 (I) Call of entity-kind "名称" (declared at line "行号") cannot be inlined
不能直接插入展开函数调用"名称"。

C5679 (I) Entity-kind "名称" cannot be inlined
不直接插入展开函数"名称"。

C5693 (E) <typeinfo> must be included before typeid is used
为了使用 typeid，必须包含<typeinfo>。

C5694 (E) "名称" cannot cast away const or other type qualifiers
没有"名称"的数据类型转换的结果 const 等的属性。

C5695 (E) The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *
dynamic_cast 的型必须是指向完全类的型的指针型、参照型或者 void *型。

C5696 (E) The operand of a pointer dynamic_cast must be a pointer to a complete class type
dynamic_cast 指针的操作数必须是指向完全类的型的指针型。

C5697 (E) The operand of a reference dynamic_cast must be an lvalue of a complete class type
dynamic_cast 的参照的操作数必须是完全类的型的左边值。

C5698 (E) The operand of a runtime dynamic_cast must have a polymorphic class type
执行时 dynamic_cast 的操作数必须是多态类的型。

C5701 (E) An array type is not allowed here
不允许数组型。

C5702 (E) Expected an "="
需要赋值。

C5703 (E) Expected a declarator in condition declaration
需要说明符。

C5704 (E) "名称", declared in condition, may not be redeclared in this scope
在此作用域中不能重新声明"名称"。

C5705 (E) Default template arguments are not allowed for function templates
不能给函数模板指定默认的实际参数。

C5706 (E) Expected a "," or ">"
需要","或者 ">"。

C5707 (E) Expected a template parameter list
需要模板的参数列表。

C5708 (W) Incrementing a bool value is deprecated
bool 型的值被递增。值递增后继续处理。

C5709 (E) bool type is not allowed
不能递减 bool 型的值。

C5710 (E) Offset of base class "名称 1" within class "名称 2" is too large
类"名称 2"内的基类"名称 1"太大。

C5711 (E) Expression must have bool type (or be convertible to bool)
表达式的型必须是 bool 型或者能转换到 bool 型的型。

C5717 (E) The type in a const_cast must be a pointer, reference, or pointer to member to an object type
const_cast 的型必须是指针型、参照型或者指向成员的指针型。

C5718 (E) A const_cast can only adjust type qualifiers; it cannot change the underlying type
const_cast 不能调整 const/volatile 以外的型。

C5719 (E) mutable is not allowed
不允许 mutable 的指定。

C5720 (W) Redeclaration of entity-kind "名称" is not allowed to alter its access
不能通过"名称"的重新声明更改存取的指定。以前声明的存取指定有效。

C5722 (W) Use of alternative token "<:" appears to be unintended
使用了 2 个字符的记述 "<:" 。解释为"[" 。

C5723 (W) Use of alternative token "%:" appears to be unintended
使用了 2 个字符的记述 "%:" 。解释为"#" 。

C5724 (E) namespace definition is not allowed
在文件作用域或者 namespace 作用域中允许 namespace 的定义。

C5725 (E) Name must be a namespace name
namespace 的名称不正确。

C5726 (E) Namespace alias definition is not allowed
在此不允许 namespace 的别名定义。

C5727 (E) namespace-qualified name is required
需要 namespace 的限定名。

C5728 (E) A namespace name is not allowed
不允许 namespace 名。

C5730 (E) Entity-kind "名称" is not a class template
"名称"不是类模板的成员。

C5732 (E) Allocation operator may not be declared in a namespace
在 namespace 中声明了 operator new 函数。

C5733 (E) Deallocation operator may not be declared in a namespace
在 namespace 中声明了 operator delete 函数。

C5734 (E) Entity-kind "名称 1" conflicts with using-declaration of entity-kind "名称 2"
名称"名称 1"与 using 声明名"名称 2"冲突。

C5735 (E) Using-declaration of entity-kind "名称 1" conflicts with entity-kind "名称 2" (declared at line "行号")
using 声明的名称发生冲突。

C5737 (W) Using-declaration ignored -- it refers to the current namespace
对当前的 namespace 作用域的名称进行了 using 声明。忽视 using 声明。

C5738 (E) A class-qualified name is required
需要类的限定名。

C5741 (W) Using-declaration of entity-kind "名称" ignored
using 声明"名称"无效。

C5742 (E) Entity-kind "名称 1" has no actual member "名称 2"
"名称 1"没有"名称 2"的成员。

C5750 (E) Entity-kind "名称" (declared at line "行号") was used before its template was declared
在声明模板前使用了"名称"。

C5751 (E) Static and nonstatic member functions with same parameter types cannot be overloaded
不能重载具有相同参数的型的静态成员函数和非静态成员函数。

C5752 (E) No prior declaration of entity-kind "名称"
没有 namespace 模板函数"名称"的声明。

C5753 (E) A template-id is not allowed
在此，不允许模板(template 名<template 实际参数>)。

C5754 (E) A class-qualified name is not allowed
在此，不允许类限定名。

C5755 (E) Entity-kind "名称" may not be redeclared in the current scope
在此作用域中不能重新声明"名称"。

C5756 (E) Qualified name is not allowed in namespace member declaration
在 namespace 成员的声明中不允许指定的限定名。

C5757 (E) Entity-kind "名称" is not a type name
"名称"不是型名。

C5761 (E) Typename may only be used within a template
只能在模板中使用 typename 关键字。

C5766 (W) Exception specification for virtual entity-kind "名称 1" is incompatible with that of overridden entity-kind "名称 2"
虚拟函数的异常指定"名称 1"和"名称 2"不一致。

C5767 (W) Conversion from pointer to smaller integer
将指针转换为小于指针大小的型。

C5768 (W) Exception specification for implicitly declared virtual entity-kind "名称 1" is incompatible with that of overridden entity-kind "名称 2"
编译程序生成的隐含虚拟函数"名称 1"的异常指定和"名称 2"不一致。

C5771 (E) "explicit" is not allowed
只能对类声明内的构造函数指定 explicit。

C5772 (E) Declaration conflicts with "名称" (reserved class name)
与保留的类名 type_info 冲突。

C5773 (E) Only "()" is allowed as initializer for array entity-kind "名称"
数组"名称"的初始化指定不正确。

C5774 (E) "virtual" is not allowed in a function template declaration
在函数模板中不能指定 virtual。

C5775 (E) Invalid anonymous union -- class member template is not allowed
无名 union 的指定不正确。

C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "名称"
模板参数的嵌套和以前的声明"名称"不一致。

C5777 (E) This declaration cannot have multiple "template <...>" clauses
此声明不能声明多个模板。

C5779 (E) "名称", declared in for-loop initialization, may not be redeclared in this scope
在此作用域中不能重新声明由 for 语句的初始化表达式声明的"名称"。

C5782 (E) Definition of virtual entity-kind "名称" is required here
在此需要虚拟函数的定义"名称"。

C5784 (E) A storage class is not allowed in a friend declaration
不能给友元声明指定存储类。

C5785 (E) Template parameter list for "名称" is not allowed in this declaration
在此声明中不允许"名称"的模板的参数排列。

C5786 (E) entity-kind "名称" is not a valid member class or function template
"名称"不是有效的成员或者函数模板。

C5787 (E) Not a valid member class or function template declaration
不是有效的成员或者函数模板声明。

C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration
定义模板函数后不能指定包含模板参数排列的模板声明。

C5789 (E) Explicit specialization of entity-kind "名称 1" must precede the first use of entity-kind "名称 2"
显式模板实体的定义"名称 1"必须在使用最初的模板"名称 2"前。

C5790 (E) Explicit specialization is not allowed in the current scope
在此作用域中不允许显式模板实体的定义。

C5791 (E) Partial specialization of entity-kind "名称" is not allowed
不允许模板"名称"的部分定义。

C5792 (E) Entity-kind "名称" is not an entity that can be explicitly specialized
"名称"不是模板的示例。

C5793 (E) Explicit specialization of entity-kind "名称" must precede its first use
显式模板实体"名称"的定义必须在最初开始使用前。

C5794 (W) Template parameter "模板参数" may not be used in an elaborated type specifier
对 class 指定不能使用模板参数。class 指定无效，模板有效。

C5795 (E) Specializing entity-kind "名称" requires "template<>" syntax
对于"名称"的模板的实体定义，需要 tempalte<>格式。

C5800 (E) This declaration may not have extern "C" linkage
此声明不能有 extern "C" 的连接。

C5801 (E) "名称" is not a class or function template name in the current scope
在此作用域中"名称"不是类模板或者函数模板。

C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard
在重新声明未参照的函数模板时，指定了默认参数。
忽略默认参数。

C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed
在重新声明已被参照的函数模板时，指定了默认参数。

C5804 (E) Cannot convert pointer to member of base class "型1" to pointer to member of derived class "型2" -- base class is virtual
不能将虚拟基类"型1"的成员指针转换为派生类"型2"的成员指针。

C5805 (E) Exception specification is incompatible with that of entity-kind "名称" (declared at line "行号"):
throw 异常指定和"名称"的异常指定不一致。

C5806 (W) Omission of exception specification is incompatible with entity-kind "名称" (declared at line "行号")
throw 异常指定的省略和"名称"的异常指定不一致。"名称"有效。

C5807 (E) The parse of this expression has changed between the point at which it appeared in the program and the point at which the expression was evaluated -- "typename" may be required to resolve the ambiguity
中途结束默认实际参数的表达式的评价。

C5808 (E) Default-initialization of reference is not allowed
不允许参照型的默认初始化。

C5809 (E) Uninitialized entity-kind "名称" has a const member
未初始化的"名称"有 const 型成员。

C5810 (E) Uninitialized base class "型" has a const member
未初始化的基类"型"有 const 型成员。

C5811 (E) Const entity-kind "名称" requires an initializer -- class "型" has no explicitly declared default constructor
const 型的"名称"需要初始化指定。没有明确声明类"型"的默认构造函数。

C5812 (W) Const object requires an initializer -- class "型" has no explicitly declared default constructor
const 型目标需要初始化指定。不能存在没有明确声明类"型"的默认构造函数。

C5815 (I) Type qualifier on return type is meaningless
给模板实体化的返回型指定了无意义的限定型。限定型有效。

C5817 (E) Static data member declaration is not allowed in this class
局部类不能有静态数据成员。

C5818 (E) Template instantiation resulted in an invalid function declaration
模板实体化的函数声明不正确。

C5822 (E) Invalid destructor name for type "型"
"型"的析构函数名不正确。

C5824 (E) Destructor reference is ambiguous -- both entity-kind "名称 1" and entity-kind "名称 2" could be used
使用了"名称 1"和"名称 2"。析构函数的参照不明确。

C5825 (W) Virtual inline entity-kind "名称" was never defined
没有虚拟直接插入成员函数"名称"的定义。

C5826 (W) Entity-kind "名称" was never referenced
不参照函数的参数"名称"。

C5827 (E) Only one member of a union may be specified in a constructor initializer list
在析构函数的初始化中只能指定联合体的一个成员。

C5831 (I) Support for placement delete is disabled
operator delete 函数的型不正确。继续处理。

C5832 (E) No appropriate operator delete is visible
找不到适当的 operator delete 函数。

C5833 (E) Pointer or reference to incomplete type is not allowed
不允许指向不完全型的指针或者参照型。

C5834 (E) Invalid partial specialization -- entity-kind "名称" is already fully specialized

对已被特殊化的"名称"进行了部分特殊化。

C5835 (E) Incompatible exception specifications

异常指定的型不一致。

C5836 (W) Returning reference to local variable

将局部变量的参照指定为返回值。继续指定的处理。

C5837 (W) Omission of explicit type is nonstandard ("int" assumed)

没有型指定。假设 int 型。

C5838 (E) More than one partial specialization matches the template argument list of entity-kind "名称"

部分特殊化模板"名称"的模板实际参数不明确。

C5840 (E) A template argument list is not allowed in a declaration of a primary template

不能给主模板声明指定模板实际参数。

C5841 (E) Partial specializations may not have default template arguments

部分特殊化模板不能有默认模板参数。

C5842 (E) Entity-kind "名称 1" is not used in template argument list of entity-kind "名称 2"

部分特殊化模板"名称 1"不能用于"名称 2"的模板实际参数。

C5843 (E) The type of partial specialization template parameter entity-kind "名称" depends on another template parameter

部分特殊化模板"名称"的模板形式参数取决于其它模板的形式参数。

C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter

部分特殊化模板的模板实际参数包含了取决于模板形式参数的非型实际参数。

C5845 (E) This partial specialization would have been used to instantiate entity-kind "名称"

此部分特殊化模板正要进行主模板"名称"的实体化。

C5846 (E) This partial specialization would have been made the instantiation of entity-kind "名称" ambiguous

此部分特殊化模板为不明确的"名称"的实体化。

C5847 (E) Expression must have integral or enum type
表达式的型必须是整型或者 enum 型。

C5848 (E) Expression must have arithmetic or enum type
表达式的型必须是算术型或者 enum 型。

C5849 (E) Expression must have arithmetic, enum, or pointer type
表达式的型必须是算术型、enum 型或者指针型。

C5850 (E) Type of cast must be integral or enum
数据类型转换的型必须是整型或者 enum 型。

C5851 (E) Type of cast must be arithmetic, enum, or pointer
数据类型转换的型必须是算术型、enum 型或者指针型。

C5852 (E) Expression must be a pointer to a complete object type
表达式的型必须是指向完全目标型的指针型。

C5853 (E) A partial specialization of a member class template must be declared in the class of which it is a member
在包含该成员的类中必须声明成员类的部分特殊化模板。

C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant
部分特殊化模板的非型模板实际参数必须是非型的形式参数名或者常数。

C5855 (E) Return type is not identical to return type "型" of overridden virtual function entity-kind "名称"
函数的返回型和被覆盖（override）的虚拟函数"名称"的返回型"型"不一致。

C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member
必须在包含该成员的 namespace 中声明部分特殊化模板。

C5858 (E) Entity-kind "名称" is a pure virtual function
"名称"是纯虚拟函数。

C5859 (E) Pure virtual entity-kind "名称" has no overrider
不能覆盖（override）纯虚拟函数"名称"。

C5861 (E) Invalid character in input line
行中出现不正确的字符。

C5862 (E) Function returns incomplete type "型"
函数的返回型"型"是不完全型。

C5864 (E) "名称" is not a template
"名称"不是模板。

C5865 (E) A friend declaration may not declare a partial specialization
在友元声明中不能指定部分特殊化模板。

C5867 (W) Declaration of "size_t" does not match the expected type "型"
size_t 型与期待的"型"不同。

C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)
在 2 个模板实际参数列表的最后指定的">>"之间需要空白。

C5870 (W) Invalid multibyte character sequence
有不正确的 2 个字节的字符。

C5871 (E) Template instantiation resulted in unexpected function type of "型 1" (the meaning of a name may have changed since the template declaration -- the type of the template is "型 2")
具有"型 2"的模板实体化的结果，建立了没有期待的型"型 1"的函数。

C5873 (E) Non-integral operation not allowed in nontype template argument
在非型的模板实际参数中不允许非整型的运算。

C5875 (W) Embedded C++ does not support templates
Embedded C++规格不支持模板功能。

C5876 (W) Embedded C++ does not support exception handling
Embedded C++规格不支持异常处理功能。

C5877 (W) Embedded C++ does not support namespaces
Embedded C++规格不支持 namespace 功能。

C5878 (W) Embedded C++ does not support run-time type information
Embedded C++规格不支持运行时型信息功能。

C5879 (W) Embedded C++ does not support the new cast syntax
Embedded C++规格不支持 new 的数据类型转换功能。

C5880 (W) Embedded C++ does not support using-declarations
Embedded C++规格不支持 using 声明功能。

C5881 (W) Embedded C++ does not support "mutable"
Embedded C++规格不支持 mutable 功能。

C5882 (W) Embedded C++ does not support multiple or virtual inheritance
Embedded C++规格不支持多重继承/虚拟继承功能。

C5885 (E) "型 1" cannot be used to designate constructor for "型 2"
"型 1"不能用于构造函数的"型 2"。

C5891 (E) An explicit template argument list is not allowed on this declaration
在此声明中不允许显式模板实际参数。

C5894 (E) Entity-kind "名称" is not a template
"名称"不是模板。

C5896 (E) Expected a template argument
期待模板的实际参数。

C5898 (E) Nonmember operator requires a parameter with class or enum type
非成员运算符函数需要类或者 enum 型的形式参数。

C5900 (E) Using-declaration of entity-kind "名称" is not allowed
不允许"名称"的 using 声明。

C5901 (E) Qualifier of destructor name "型 1" does not match type "型 2"
"型 1"的析构函数的限定名和"型 2"不一致。

C5902 (W) Type qualifier ignored
型限定名不正确。型限定名无效。

C5916 (E) Cannot convert pointer to member of derived class "型 1" to pointer to member of base class "型 2" -- base class is virtual
不能将指向派生类"型 1"成员的指针型转换为指向虚拟基类"型 2"成员的指针型。

C5919 (F) Invalid output file: "名称"
模板信息文件的"名称"不正确。
请确认编译程序的环境设定和主机环境的文件系统是否异常。

C5920 (F) Cannot open output file: "名称"
不能打开模板信息文件"名称"。
请确认编译程序的环境设定和主机环境的文件系统是否异常。

C5926 (F) Cannot open definition list file: "名称"
不能打开文件"名称"。
请确认编译程序的环境设定和主机环境的文件系统是否异常。

C5928 (E) Incorrect use of va_start
va_start 的使用方法有错。

C5929 (E) Incorrect use of va_arg
va_arg 的使用方法有错。

C5930 (E) Incorrect use of va_end
va_end 的使用方法有错。

C5935 (E) "typedef" may not be specified here
不能指定 typedef。

C5936 (W) Redeclaration of entity-kind "名称" alters its access
通过"名称"的重新声明更改存取指定。
被重新定义的存取指定有效。

C5937 (E) A class or namespace qualified name is required
需要类或者 namespace 的限定名。

C5940 (W) Missing return statement at end of non-void entity-kind "名称"
非 void 型返回的函数"名称"没有 return 语句。return 值不定。

C5941 (W) Duplicate using-declaration of "名称" ignored
重复指定 using 声明"名称"。重复 using 声明无效。

C5946 (E) Name following "template" must be a member template
紧接着"template"后的名称必须是成员模板。

C5947 (E) Name following "template" must have a template argument list
紧接着"template"后的名称必须是模板实际参数。

C5952 (E) A template parameter may not have class type
不能给模板形式参数指定类的型名。

C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template
不能给类模板的成员声明指定默认模板实际参数。

C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor
在构造函数的 try 块的处理程序中不允许返回语句。

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "长度" bits
指定的位数超过了位字段的型的"长度"。
将位数按照位字段的型的长度，继续处理。

C5960 (E) Type used as constructor name does not match type "型"
用于构造函数名的型和"型"不一致。

C5961 (W) Use of a type with no linkage to declare a variable with linkage
用未连接的型声明有连接的变量。作为有连接的变量。

C5962 (W) Use of a type with no linkage to declare a function
用未连接的型声明有连接的函数。作为有连接的函数。

C5963 (E) Return type may not be specified on a constructor
不能给构造函数指定返回型。

C5964 (E) Return type may not be specified on a destructor
不能给析构函数指定返回型。

C5965 (E) Incorrectly formed universal character name
universal character 的格式不正确。

C5966 (E) Universal character name specifies an invalid character
universal character 指定的字符不正确。

C5967 (E) A universal character name cannot designate a character in the basic character set
在基本字符集中不能将 universal character 指定为字符。

C5968 (E) This universal character is not allowed in an identifier
在标识符中不允许 universal character。

C5978 (E) A template friend declaration cannot be declared in a local class
在局部类中不能声明模板的友元函数。

C5979 (E) Ambiguous "?" operation: second operand of type "型 1" can be converted to third operand type "型 2", and vice versa
三元运算符"?:"的第 2 个表达式的"型 1"和第 3 个表达式的"型 2"是可互相转换的型，不明确。

C5980 (E) Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type
调用目标，但是没有定义 operator()函数或者指向函数的指针型转换函数。

C5982 (E) There is more than one way an object of type "型" can be called for the argument list
有 2 个以上能从实际参数列表调用的"型"的目标。

C5984 (W) Operator new and operator delete cannot be given internal linkage
用 static 定义了 delete。

C5985 (E) Storage class "mutable" is not allowed for anonymous unions
不能将 mutable 指定为无名联合体。

C5987 (E) Abstract class type "型" is not allowed as catch type:
不能用 catch 接受抽象类。

C5988 (E) A qualified function type cannot be used to declare a nonmember function or a static member function
带限定函数型不能用于非成员函数和 static 成员函数的声明。

C5989 (E) A qualified function type cannot be used to declare a parameter
带限定函数型不能用于函数参数的指定。

C5990 (E) Cannot create a pointer or reference to qualified function type
不能建立指向带限定函数型的指针型和参照型。

C5991 (W) Extra braces are nonstandard
集合型的初始化子列表中有多余的'{'。

C5994 (E) An empty template parameter list is not allowed in a template template parameter declaration
不能将具有空模板参数的模板指定为模板参数。

C5995 (E) Expected "class"
指定为模板参数的类模板需要类。

C5996 (E) The "class" keyword must be used when declaring a template template parameter
指定为模板参数的类模板不是结构体。

C5998 (E) A qualified name is not allowed for a friend declaration that is a function definition
在带 friend 指定的函数定义中不能指定带名称空间的函数名。

C5999 (E) "型" is not compatible with "型"
指定的类模板与模板参数的形式不一致。

C6000 (W) A storage class may not be specified here
在此不能指定存储类说明符。

C6006 (E) A template template parameter cannot have the same name as one of its template parameters
指定为模板参数的类模板名与其自身的模板参数名相同。

C6007 (W) "函数名 1" is hidden by "函数名 2" -- virtual function override intended?
"函数名 1"隐藏了虚拟函数"函数名 2"。

C6008 (E) A parameter of a template template parameter cannot depend on the type of another template parameter
指定为模板参数的模板所具有的非型参数的型不能依存于其它型参数。

C6009 (E) "示例名" is not an entity that can be defined
要生成没有实体的示例。

C6010 (E)Destructor name must be qualified
析构函数名需要限定。

C6013 (E) A qualified friend template declaration must refer to a specific previously declared template
必须在参照前定义限定友元模板。

C6018 (E) "类名" has no member class "成员名"
使用了非类的成员。

C6019 (E) The global scope has no class named "类名"
类中的名称使用了文件作用域运算符。

C6020 (E) Recursive instantiation of template default argument
生成了模板的默认参数的递归示例。

C6021 (E) Access declarations and using-declarations cannot appear in unions
在 union 中不能使用 using 指定。

C6022 (E) "名称" is not a class member
不是类的成员。

C6028 (W) Invalid redeclaration of nested class
在类中二重定义了类。

C6035 (E) "模板名" cannot be declared in this scope
在此作用域中不能声明 template。

C6057 (E) _ _evenaccess qualifier is applied to only integer type
_ _evenaccess 限定符不能限定非整型。

C6058 (E) Expected a section name string
　　_ _sectop/_ _secend 没有段名。

C6059 (E) Expected a section name
　　#pragma section 的段名不正确。

C6060 (E) Invalid pragma declaration
　　#pragma 的语法不正确。

C6061 (E) "名称" has already been specified by other pragma
　　此符号已被其它的#pragma 指定。

C6062 (E) Pragma may not be specified after definition
　　不能对定义符号后的声明指定#pragma。

C6063 (E) Invalid kind of pragma is specified to this symbol
　　指定了不正确的#pragma。

C6064 (I) This pragma has no effect
　　没有此#pragma 指定的符号。

C6065 (E) _ _regparam? must be qualified for function type
　　_ _regparam2/_ _regparam3 不能限定非函数型。

C6066 (E) Illegal 属性说明符 specifier
　　属性说明符的记述方法有错。

C6067 (E) Multiple pointer qualifiers
　　指定了多个_ _ptr16。

C6068 (E) _ _ptr16 must be qualified for data pointer type
　　给非数据指针型指定了_ _ptr16。

12.3　C 库函数的错误信息

在库函数中，如果在执行库函数中发生错误，就有可能将错误号设定到标准库的头文件 `<stddef.h>` 定义的宏 `errno`。在错误号中定义了对应的错误信息，并能输出错误信息。输出错误信息的程序例如下所示：

例

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

void main(void)
{
          FILE *fp;

          fp=fopen(“file”, “w”);
          fp=NULL;

          fclose(fp);                        /* error occurred  */

          printf(“%s\n”, strerror(errno));   /* print error message */
}
```

说明

由于将值为`NULL`的文件指针作为实际参数传递给`fclose`函数，所以出错。此时给`errno`设定对应的错误号。

如果将错误号作为实际参数传递给`strerror`函数，`strerror`函数就返回对应错误信息的字符串指针。根据`printf`函数的字符串输出指定，输出错误信息。

表 12.1　C 库函数的错误信息一览表

错误号	错误信息/说明	设定错误号的函数
1100 （ERANGE）	DATA OUT OF RANGE 发生溢出。	frexp, ldexp, modf, ceil, floor, fmod, strtol, atoi, atol, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceilf, cos, cosf, cosh, coshf, exp, expf, floorf, fmodf, ldexpf, log, log10, log10f, logf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexpf
1101 （EDOM）	DATA OUT OF DOMAIN 没有定义对于数学函数的参数的结果值。	acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceil, ceilf, cos, cosf, cosh, coshf, exp, expf, floor, floorf, fmod, fmodf, ldexp, ldexpf, log, log10, log10f, logf, modf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexp, frexpf
1102 （EDIV）	DIVISION BY ZERO 被零除。	div, ldiv
1104 （ESTRN）	TOO LONG STRING 字符串的长度超过了 32767 个字符。	strtol, strtod, atoi, atol, atof
1106 （PTRERR）	INVALID FILE POINTER 给文件指针的值指定了 NULL 指针常数。	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 （ECBASE）	INVALID RADIX 基数的指定有错。	strtol, atoi, atol
1202 （ETLN）	NUMBER TOO LONG 表现数值的字符串的长度超过了 17 位。	strtod, fscanf, scanf, sscanf, atof
1204 （EEXP）	EXPONENT TOO LARGE 指数的位数超过了 3 位。	strtod, fscanf, scanf, sscanf, atof
1206 （EEXPN）	NORMALIZED EXPONENT TOO LARGE 在进行一次字符串的 IEEE 规格 10 进制格式的正规化时，指数的位数超过了 3 位。	strtod, fscanf, scanf, sscanf, atof
1210 （EFLOATO）	OVERFLOW OUT OF FLOAT float 型的 10 进制数值超出了 float 型的范围（溢出）。	strtod, fscanf, scanf, sscanf, atof
1220 （EFLOATU）	UNDERFLOW OUT OF FLOAT float 型的 10 进制数值超出了 float 型的范围（下溢）。	strtod, fscanf, scanf, sscanf, atof
1230 （EOVER）	FLOATING POINT OVERFLOW 数值常数超出了 double 型的范围（溢出）。	strtod, fscanf, scanf, sscanf, atof
1240 （EUNDER）	FLOATING POINT UNDERFLOW 数值常数超出了 double 型的范围（下溢）。	strtod, fscanf, scanf, sscanf, atof

错误号	错误信息/说明	设定错误号的函数
1300 （NOTOPN）	FILE NOT OPEN 不能打开文件。	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen
1302 （EBADF）	BAD FILE NUMBER 对于输入专用文件，调用了输出函数；或者对于输出专用文件，调用了输入函数。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 （ECSPEC）	ERROR IN FORMAT 用带格式的输入/输出函数指定的格式有错。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror

第 13 章　汇编程序的错误信息

13.1　错误格式和错误级

本章说明用以下格式输出的错误信息和错误内容：

错误号　（错误级）　错误信息
错误内容

根据错误的重要程度，错误级分为 3 种。

错误级		运行
（W）	警告	继续处理。
（E）	错误	继续处理。
（F）	致命错误	中断处理。

13.2　信息一览

```
10  (E)  NO INPUT FILE SPECIFIED
```

没有指定输入源文件。
请指定输入源文件。

```
20  (E)  CANNOT OPEN FILE 文件名
```

不能打开指定的文件。
请检查文件名、目录名等。

```
30  (E)  INVALID COMMAND PARAMETER
```

选项有错。
请检查选项。

```
40  (E)  CANNOT ALLOCATE MEMORY
```

在处理中存储器不足。
发生在用户能够使用的存储量极少时。
如果有其它执行中的处理，请在结束该处理后重新启动汇编程序。
即使如此还发生本错误时，请检查主机系统的存储管理方法。

```
50  (E)  INVALID FILE NAME 文件名
```

包含目录的文件名太长或者文件名有错。
请检查文件名。
调试程序可能无法处理此时汇编程序输出的目标模块。

60　(W)　INVALID VALUE 文件名
给 SBR 选项的常数值的低 8 位指定了 0 以外的值。
请检查常数值。
汇编程序将常数值的低 8 位更改为 0。

101　(E)　SYNTAX ERROR IN SOURCE STATEMENT
源语句有语法错误。
请检查全部源语句。

102　(E)　SYNTAX ERROR IN DIRECTIVE
汇编程序控制命令的源语句有语法错误。
请检查全部源语句。

103　(E) .END NOT FOUND
程序中没有.END。
请记述.END。

104　(E)　LOCATION COUNTER OVERFLOW
定位计数器的值超过了最大值。
请缩小程序。

105　(E)　ILLEGAL INSTRUCTION IN STACK SECTION
记述了在堆栈段内确保执行指令、数据的汇编程序控制指令。
请删除确保执行指令、数据的汇编程序控制命令。

106　(E)　TOO MANY ERRORS
由于错误太多，中止了显示。
请检查全部源语句。

108　(E)　ILLEGAL CONTINUATION LINE
跨多行记述的源语句有错。
请检查记述方法。

150　(E)　INVALID DELAY SLOT INSTRUCTION
延迟槽指令（紧接在延迟转移指令后的执行指令）不适当。
请更改执行指令的记述顺序等，改正延迟槽指令。

200　(E)　UNDEFINED SYMBOL REFERENCE
没有定义参照的符号。
请定义符号。

201　(E)　ILLEGAL SYMBOL OR SECTION NAME
作为符号（包含段名），指定了保留字（寄存器名、运算符、定位计数器）。
请修正符号（包含段名）。

202　(E)　ILLEGAL SYMBOL OR SECTION NAME
符号（包含段名）有错。
请修正符号（包含段名）。

203　(E)　ILLEGAL LOCAL LABEL
局部标号的指定有错。
请修正局部标号的指定。

300　(E)　ILLEGAL MNEMONIC
操作码有错。
请修正操作码。

301　(E)　TOO MANY OPERANDS OR ILLEGAL COMMENT
汇编程序控制指令的操作数过多，或者注解有错。
请修正操作数或者注解。

304　(E)　LACKING OPERANDS
操作数不足。
请修正操作数。

306　(E)　SYNTAX ERROR IN REGISTER LIST
多个寄存器的指定方法有错。
请修正多个寄存器的指定。

307　(E)　ILLEGAL ADDRESSING MODE OR OBJECT CODE SIZE
指定了操作数不允许的寻址方式，或者确保的长度（:8、:16、:24、:32）有错。
请修正操作数或者确保的长度。

308　(E)　SYNTAX ERROR IN OPERAND
操作数有语法错误。
请修正操作数。

400　(E)　CHARACTER CONSTANT TOO LONG
字符常数超过了 4 个字符。
请修正字符常数。

402　(E)　ILLEGAL VALUE IN OPERAND
作为操作数，为范围以外的值。
请更改值。

403　(E)　ILLEGAL OPERATION FOR RELATIVE VALUE
对相对地址指定了乘除运算或逻辑运算。
请修正运算内容。

404　(E)　ILLEGAL IMMEDIATE DATA

给#1,2,4 、#0～3、#0～7 的操作数指定了相对值。
不能使用相对值，请修正值。

407　(E)　MEMORY OVERFLOW

在算式的计算中，用于计算的存储器不足。请简化运算内容。

408　(E)　DIVISION BY ZERO

指定了被 0 除的运算。
请更改运算内容。

409　(E)　REGISTER IN EXPRESSION

在表达式中出现了寄存器名。
请修正运算内容。

411　(E)　INVALID STARTOF/SIZEOF OPERAND

在 STARTOF 运算或者 SIZEOF 运算中指定了段名以外的内容。
请修正运算内容。

412　(E)　ILLEGAL SYMBOL IN EXPRESSION

给移位数指定了相对值或者相对符号。
请修正运算内容。

413　(E)　ILLEGAL DISPLACEMENT VALUE

位移量的值不适当。
请将位移量设定为偶数。

500　(E)　SYMBOL NOT FOUND

在需要标号的汇编程序控制指令的源语句中没有标号。
请记述标号。

501　(E)　ILLEGAL ADDRESS VALUE IN OPERAND

段的起始地址的指定有错或者定位计数器值的指定有错。
请修正起始地址或者定位计数器的值。

502　(E)　ILLEGAL SYMBOL IN OPERAND

给操作数指定了不适当的值（前方参照符号、外部参照符号、相对地址符号、未定义符号）。
请修正操作数。

503　(E)　UNDEFINED EXPORT SYMBOL

将文件中没有定义的符号声明为外部定义符号。
请定义符号或者取消作为外部定义符号的声明。

504　(E)　INVALID RELATIVE SYMBOL IN OPERAND
给操作数指定了不适当的值（前方参照符号、外部参照符号）。
请修正操作数。

505　(E)　ILLEGAL OPERAND
操作数的名称有错。
请修正操作数。

506　(E)　ILLEGAL OPERAND
指定了操作数不允许的元素。
请修正操作数。

508　(E)　ILLEGAL VALUE IN OPERAND
给操作数指定了范围以外的值。
请修正操作数。

510　(E)　ILLEGAL BOUNDARY VALUE
边界调整数的指定有错。
请修正边界调整数。

511　(E)　ILLEGAL DISPLACEMENT SIZE
.DISPSIZE 的位数有错。
请修正位数。

512　(E)　ILLEGAL EXECUTION START ADDRESS
执行起始地址有错。
请修正执行起始地址。

513　(E)　ILLEGAL REGISTER NAME
寄存器名有错。
请修正寄存器名。

514　(E)　INVALID EXPORT SYMBOL
将不能外部定义的符号声明为外部定义符号。
请取消外部定义符号的声明。

516　(E)　EXCLUSIVE DIRECTIVES
控制指令指定的内容有矛盾。
请检查相关的控制指令。

517　(E)　INVALID VALUE IN OPERAND
给操作数指定了不适当的值（前方参照符号、外部参照符号、其它段的相对地址符号）。
请修正操作数。

518 (E) INVALID IMPORT SYMBOL
将文件中定义的符号声明为外部参照符号。
请取消外部参照符号的声明。

520 (E) ILLEGAL .CPU DIRECTIVE POSITION
.CPU 控制指令不在程序的起始位置，或者被多次指定。
请在程序的起始位置只指定一次. CPU 控制指令。

521 (E) ILLEGAL SYMBOL IN OPERAND
在指定 optimize 选项时，给指定常数值的操作数指定了持有地址值的符号或者定位计数器的值。
在指定持有地址值的符号或者定位计数器值的情况下，不能指定 optimize选项。

523 (E) ILLEGAL OPERAND
.LINE 控制指令的操作数有错。
请修正 LINE 控制指令的操作数。

524 (E) ILLEGAL ADDRESSING SPACE SIZE
给.CPU 控制指令的操作数指定了不允许的地址空间的位宽。
请修正地址空间的位宽。

525 (E) ILLEGAL .LINE DIRECTIVE POSITION
在宏展开或者带条件重复展开中，指定了.LINE 控制指令。
请改变.LINE 控制指令的指定位置。

526 (E) STRING TOO LONG
操作数的字符串超过了 255 个字符。
给.SDATA、.SDATAB、.SDATAC、.SDATAZ 控制指令的操作数指定的字符串必须在 255 个字符以内。

527 (E) CANNOT SUPPORT COMMON SECTION SINCE VERSION 4
给段属性指定了 COMMON。
不能使用公共段。
在优化连接编辑程序的 start 选项中，能使用冒号（:）将多个段分配在同一地址。

528 (E) SPECIFICATION OF THE ADDRESS OVERLAPS
重复分配了段内的地址。
请检查.SECTION 控制指令和.ORG 控制指令的指定内容。

529 (E) THE ADDRESS BETWEEN SECTIONS OVERLAPS
重复分配了段之间的地址。
请检查.SECTION 控制指令和.ORG 控制指令的指定内容。

532　(E)　ILLEGAL OPERAND
.STACK 的操作数有错。
请将堆栈值修正为 2 的倍数。

533　(E)　ILLEGAL .STACK DIRECTIVE POSITION
在宏展开或者带条件的重复展开中，指定了.STACK。
请改变.STACK 的指定位置。

600　(E)　INVALID CHARACTER
源程序中有不适当的字符。
请修正不适当的字符。

601　(E)　INVALID DELIMITER
分隔符不适当。
请修正分隔符。

602　(E)　INVALID CHARACTER STRING FORMAT
字符串有错。
请修正字符串。

603　(E)　SYNTAX ERROR IN SOURCE STATEMENT
源语句有语法错误。
请检查全部源语句。

604　(E)　ILLEGAL SYMBOL IN OPERAND
给指定绝对值的操作数指定了相对值、前方参照值以及未定义符号。
请修正值。

610　(E)　MULTIPLE MACRO NAMES
要用.MACRO 定义的宏名已经被定义。
请修正宏名。

611　(E)　MACRO NAME NOT FOUND
.MACRO 的操作数没有宏名。
请记述宏名。

612　(E)　ILLEGAL MACRO NAME
.MACRO 的宏名有错。
请修正宏名。
不能给宏名指定执行指令、控制指令（句点（.）除外）、控制语句（句点（.）除外）的助记符。

613　(E)　ILLEGAL .MACRO DIRECTVE POSITION
宏体（.MACRO～.ENDM 之间）、.AREPEAT～.AENDR 之间、.AWHILE～.AENDW 之间有.MACRO。
请删除.MACRO。

614　(E)　MULTIPLE MACRO PARAMETERS
在声明宏定义（.MACRO）的形式参数中，形式参数名有重复。
请修正形式参数名。

615　(E)　ILLEGAL .END DIRECTIVE POSITION
宏体（.MACRO～.ENDM 之间）有.END。
请删除.END。

616　(E)　MACRO DIRECTIVES MISMATCH
.ENDM 与.MACRO 不对应，或者.EXITM 不在宏体（.MACRO～.ENDM 之间）、.AREPEAT～.AENDR 之间、.AWHILE～.AENDW 之间。
请删除.ENDM 或者.EXITM。

618　(E)　MACRO EXPANSION TOO LONG
在宏展开中，1 行的字符数超过了 8,192 个字符。
请修正为 8,192 个字符以内。

619　(E)　ILLEGAL MACRO PARAMETER
在宏调用中宏参数的形式参数名有错，或者宏体（.MACRO～.ENDM 之间）的形式参数名有错。
请修正形式参数名。
如果宏体的形式参数名有错，就在展开宏时出错。

620　(E)　UNDEFINED PREPROCESSOR VARIABLE
没有定义正在参照的预处理器变量。
请定义预处理器变量。

621　(E)　ILLEGAL .END DIRECTIVE POSITION
宏展开中有.END。
请删除.END。

622　(E)　')' NOT FOUND
没有右括号（宏处理除外）。
请记述右括号。

623　(E)　SYNTAX ERROR IN STRING FUNCTION
字符串操作函数有语法错误。
请检查字符串操作函数。

624　(E)　MACRO PARAMETERS MISMATCH
在宏调用中位置指定的宏参数的个数太多。
请修正宏参数的个数。

630　(E)　SYNTAX ERROR IN OPERAND
结构化汇编控制语句的操作数有语法错误。
请检查全部源语句。

631　(E)　END DIRECTIVE MISMATCH
在配对的控制语句中，结束控制语句不一致。
请检查控制语句。

632　(E)　SYNTAX ERROR IN OPERAND
结构化汇编控制语句的操作数的条件码有错。
请修正条件码。

633　(E)　ILLEGAL .BREAK OR .CONTINUE DIRECTIVE POSITION
.BREAK、.CONTINUE 不在.FOR[U] ～ .ENDF 之间、.WHILE～.ENDW 之间或者.REPEAT～.UNTIL 之间。
请删除.BREAK 和.CONTINUE。

634　(E)　EXPANSION TOO LONG
在结构化汇编展开中，1 行的字符数超过了 8,192 个字符。
请修正为 8,192 个字符以内。

640　(E)　SYNTAX ERROR IN OPERAND
条件汇编程序控制语句的操作数有语法错误。
请检查全部源语句。

641　(E)　INVALID RELATIONAL OPERATOR
条件汇编程序控制语句的操作数的关系运算符有错。
请修正关系运算符。

642　(E)　ILLEGAL .END DIRECTIVE POSITION
在.AREPEAT～.AENDR 之间、.AWHILE～.AENDW 之间有.END。
请删除.END。

643　(E)　DIRECTIVE MISMATCH
没有与.AREPEAT 或.AWHILE 配对的.AENDR 或.AENDW。
请检查控制语句。

644　(E)　ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION
在.AIF～.AENDI 之间有.AENDR 或.AENDW。
请删除.AENDR 或.AENDW。

645　(E)　EXPANSION TOO LONG
在.AREPEAT 或.AWHILE 展开中，1 行的字符数超过了 8,192 个字符。
请修正为 8,192 个字符以内。

650　(E)　INVALID INCLUDE FILE
.INCLUDE 的文件名有错。
请修正文件名。

651 (E) CANNOT OPEN INCLUDE FILE
不能打开.INCLUDE 的文件。
请修正文件名。

652 (E) INCLUDE NEST TOO DEEP
文件包含的嵌套超过了 30 层。
请将嵌套设定在 30 层以下。

653 (E) SYNTAX ERROR IN OPERAND
.INCLUDE 的操作数有语法错误。
请修正操作数。

660 (E) .ENDM NOT FOUND
没有与.MACRO 配对的.ENDM。
请记述.ENDM。

661 (E) END DIRECTIVE NOT FOUND
在结构化汇编程序控制语句中没有结束控制语句。
请记述结束控制语句。

662 (E) ILLEGAL .END DIRECTIVE POSITION
在.AIF～.AENDI 之间有.END。
请删除.END。

663 (E) ILLEGAL .END DIRECTIVE POSITION
include 文件中有.END。
请删除.END。

664 (E) ILLEGAL .END DIRECTIVE POSITION
.AIF～.AENDI 之间有.END。
请删除.END。

665 (E) ILLEGAL SYMBOL IN OPERAND
在指定 optimize 选项时，给预处理器控制指令指定了非预处理器变量的符号。
请修正符号。
另外，在指定非预处理器变量的符号时，不能指定 optimize 选项。

667 (E) EXPANSION TOO LONG
在.DEFINE 控制语句中，1 行的字符数超过了 8,192 个字符。
请修正为 8,192 个字符以内。

668 (E) ILLEGAL VALUE IN OPERAND
.AIFDEF 控制指令的操作数有错。
请通过.DEFINE 控制语句的符号指定此控制指令的操作数。

669　(E)　STRING TOO LONG

操作数的字符串超过了 255 个字符。
给.ASSIGNC 控制指令、.DEFINE 控制指令、字符串操作函数（.LEN、.INSTR、.SUBSTR）的操作数指定的字符串必须在 255 个字符以内。

670　(E)　SUCCESSFUL CONDITION .AERROR

根据.AIF 的条件处理了含.AERROR 控制语句的语句。
为了不处理.AERROR，请检查条件表达式。

800　(W)　SYMBOL NAME TOO LONG

预处理器变量名或者 define 置换符号名超过了 32 个字符。
请修正符号名。汇编程序忽视第 33 个以后的字符。

801　(W)　MULTIPLE SYMBOLS

再次定义了已被定义的符号。
请取消符号的重新定义。
汇编程序忽视第 2 次以后的定义。

805　(W)　ILLEGAL OPERATION SIZE

结构化汇编程序控制语句的转移长度（:8、:16）有错。
请修正转移长度。

807　(W)　ILLEGAL OPERATION SIZE

操作长度有错。
请修正操作长度。
汇编程序忽视操作长度的指定。

808　(W)　ILLEGAL CONSTANT SIZE

一部分整数常数的记述有错。
请修正记述。
解释长度有字节（.B）和字（.W），分别解释为 1 个字节和 2 个字节的带符号的值。

810　(W)　TOO MANY OPERANDS

执行指令的操作数太多，或者注释有错。
请修正操作数或者注释。
汇编程序忽视多余的操作数的指定。

811　(W)　ILLEGAL SYMBOL DEFINITION

对不能记述标号的汇编程序控制指令的源语句记述了标号。
请删除标号。
汇编程序忽视标号。

813 (W) SECTION ATTRIBUTE MISMATCH
通过段的重新开始指定了不同种类的段，或者通过绝对地址段的重新开始再次指定了段的起始地址。
重新开始段时，不能指定段的种类和起始地址。
已经开始的段的指定照样有效。

814 (W) ILLEGAL OBJECT CODE SIZE
确保长度（:8、:16、:24、:32）有错。
请修正确保长度。
#xx:2、#xx:3 是手册中记述的记号，不能记述在汇编程序中。

815 (W) MULTIPLE MODULE NAMES
再次设定了目标模块名。
请只设定 1 次目标模块。
汇编程序忽视第 2 次以后的设定。

816 (W) START ODD ADDRESS
从奇数地址开始确保了偶数字节的数据或者数据区。
请修正为偶数地址。

817 (W) OPERATION SIZE MISMATCH
对于字节长度（.B），指定了@-SP、@SP+。
虽然照样能输出目标代码，但是 SP（堆栈指针）为奇数值，所以请不要使用。

818 (W) ILLEGAL ACCESS SIZE
存取长度（:8、:16）有错。
请修正存取长度。

819 (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED
在 H8/300H、H8S、H8SX CPU 中，请将@Rn+、 @-Rn、 @+Rn、@Rn-、@(d,Rn)、@Rn 指定的 Rn 修正为 ERn。

825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION
记述了在虚设段中确保执行指令和数据的汇编程序控制指令。
请删除确保执行指令和数据的汇编程序控制指令。
汇编程序忽视确保执行指令和数据的汇编程序控制指令的记述。

830 (W) OPERATION SIZE MISMATCH
对字节长度（.B）指定了 ERn、Rn，或者对字长度（.W）指定了 ERn。
请修正寄存器。
以字节长为 RnL、字长为 Rn，生成目标代码。

832　(W)　MULTIPLE 'P' DEFINITIONS
作为默认段名的 P 与其它符号 P 重复。
请不要重复 P。
汇编程序将 P 认为默认段名，其它符号 P 的定义无效。

835　(W)　ILLEGAL VALUE IN OPERAND
给执行指令的操作数指定了范围以外的值。
请修正值。
汇编程序将值校正到范围以内后生成目标码。

836　(W)　CONSTANT SIZE OVERFLOW
整数常数的值超过了整数常数的解释长度（.B、.W）的范围。
请修正整数常数的值。
解释长度有字节（.B）和字（.W），分别解释为 1 个字节和 2 个字节的带符号的值。

837　(W)　SOURCE STATEMENT TOO LONG
源语句 1 行的长度超过了 8,192 个字节。
请缩短注释语句等，将 1 行控制在 8,192 个字节以内，或者请分成多行记述源语句。

838　(W)　ILLEGAL CHARACTER CODE
在注释、字符串以外的语句中指定了 shift JIS、EUC 或者 LATIN1 代码，或者没有指定 sjis、euc 或者 latin1 选项。
请在注释、字符串中指定 shift JIS、EUC 或者 LATIN1 代码，或者请指定 sjis、euc、latin1 选项。

850　(W)　ILLEGAL SYMBOL DEFINITION
给标号字段指定符号。
请删除符号。

851　(W)　MACRO SERIAL NUMBER OVERFLOW
宏生成序号超过了 99,999。
请减少宏调用的次数。

852　(W)　UNNECESSARY CHARACTER
在操作数后有字符。
请修正操作数。

853　(W)　NEGATIVE IMMEDIATE VALUE
对.FOR[U]的增量值记述了#-xx。
请修正为-#xx。
照样展开.FOR[U]。

854　(W)　.AWHILE ABORTED BY .ALIMIT
展开次数达到了.ALIMIT 控制语句设定的上限值而中断展开。
请检查重复展开的条件。

855　(W)　ILLEGAL VALUE IN OPERAND
给 SBR 控制指令的常数值的最低 8 位指定了 0 以外的值。
请检查常数值。
汇编程序将常数值的最低 8 位改为 0。

856　(W)　MULTIPLE SYMBOLS
对同一符号再次定义了堆栈值。
请不要重新定义堆栈值。
汇编程序忽视第 2 次以后的定义。

870　(W)　ILLEGAL DISPLACEMENT VALUE
位移量的值不适当。
请将位移量设定为偶数。
汇编程序按指定的内容生成目标码。

871　(W)　MISSING DELAY SLOT INSTRUCTION
没有延迟槽指令（紧接在延迟转移指令后的执行指令）。
请改变执行指令的记述顺序，追加延迟槽指令。
汇编程序照样输出目标码。

901　(F)　SOURCE FILE INPUT ERROR
在输入源文件时发生错误。
请确认磁盘的空容量。
请删除磁盘上不需要的文件，以确保需要的空容量。

902　(F)　MEMORY OVERFLOW
存储器不足（不能处理有关临时信息）。
请分割程序。

903　(F)　LISTING FILE OUTPUT ERROR
在输出列表文件时发生错误。
请确认磁盘的空容量。
请删除磁盘上不需要的文件，以确保需要的空容量。

904　(F)　OBJECT FILE OUTPUT ERROR
在输出目标文件时发生错误。
请确认磁盘的空容量。
请删除磁盘上不需要的文件，以确保需要的空容量。

905　(F)　MEMORY OVERFLOW
存储器不足（不能处理有关源程序行的信息）。
请分割程序。

906　(F)　MEMORY OVERFLOW
存储器不足（不能处理有关符号的信息）。
请分割程序。

907　(F)　MEMORY OVERFLOW
存储器不足（不能处理有关段的信息）。
请分割程序。

908　(F)　SECTION OVERFLOW
段的个数太多。
请分割程序。
指定 goptimize 选项时，段的上限在输出调试信息时为 62,265 个；在不输出调试信息时为 65,274 个。

933　(F)　LACKING CPU SPECIFICATION
没有设定 CPU 种类。
请用 CPU 选项、CPU 控制指令或者 H38CPU 环境变量指定 CPU 种类。

935　(F)　SUBCOMMAND FILE INPUT ERROR
在输入子命令文件时发生错误。
请确认磁盘的空容量。
请删除磁盘上不需要的文件，以确保需要的空容量。

954　(F)　MEMORY OVERFLOW
存储器不足。
请分割源程序。

955　(F)　LOCAL BLOCK NUMBER OVERFLOW
在局部标号的有效范围内的局部块的个数超过了 100,000 个。
请分割源程序。

956　(F)　EXPAND FILE INPUT/OUTPUT ERROR
在预处理器展开的文件输出时发生错误。
请确认磁盘的空容量。
请删除磁盘上不需要的文件，以确保需要的空容量。

957　(F)　MEMORY OVERFLOW
存储器不足。
请分割源程序。

964　(F)　MEMORY OVERFLOW
存储器不足（不能处理有关符号的信息）。
请分割源程序。

970　(F)　MEMORY OVERFLOW

存储器不足（段太大）。

可能通过.ORG 控制指令给定位计数器设定了较大的偏移量，或者通过.DATAB 控制指令等确保了较大的数据区。

请分割段或者缩小数据区。

第 14 章　优化连接编辑程序的错误信息

14.1　错误格式和错误级

本章说明用以下格式输出的错误信息和错误内容：

错误号　（错误级）　错误信息
错误内容

根据错误的重要程度，错误级分为 5 种。

	错误级		运行
L0000 - L0999 P0000 - P0999	（I）	信息	继续处理。
L1000 - L1999 P1000 - P1999	（W）	警告	继续处理。
L2000 - L2999 P2000 - P2999	（E）	错误	继续选项解析处理，中断处理。
L3000 - L3999 P3000 - P3999	（F）	致命错误	中断处理。
L4000 - P4000 -	（-）	内部错误	中断处理。

以 L 开始的错误号是优化连接编辑程序的输出信息。

以 P 开始的错误号是预连接程序的输出信息。不能用 nomessage 选项或者 change_message 选项指定以 P 开始的错误号。

14.2　信息一览

L0001 (I) Section "段" created by optimization "优化"
根据"优化"的优化，建立了"段"。

L0002 (I) Symbol"符号" created by optimization "优化"
根据"优化"的优化，建立了"符号"。

L0003 (I) "文件"-"符号" moved to "段" by optimization
根据 variable_access 的优化，移动了"文件"中的"符号"。

L0004 (I) "文件"-"符号" deleted by optimization
根据 symbol_delete 的优化，删除了"文件"中的"符号"。

L0005 (I) The offset value from the symbol location has been changed by optimization : "文件"-"段"-"符号±offset"

在"符号±offset"范围内因优化发生了长度的变更，所以更改了 offset 值。请确认是否没有问题。在想抑制 offset 值变更的情况下，请在"文件"汇编时解除 goptimize 选项的指定。

L0100 (I) No inter-module optimization information in "文件"

在"文件"中没有模块间的优化信息。"文件"不作为模块间的优化对象。要作为模块间优化对象时，请在编译和汇编时指定 goptimize 选项。但是，asmsh 没有 goptimize 选项。

L0101 (I) No stack information in "文件"

在"文件"中没有堆栈信息。"文件"可能是汇编程序的输出文件或者 SYSROF->ELF 转换文件。在优化连接编辑程序输出的堆栈信息文件中不包含该文件的内容。

L0102 (I) Stack size "容量" specified to the undefined symbol "符号" in "文件"

给"文件"中未定义符号的"符号"指定了堆栈容量的"容量"。

L0103 (I) Multiple stack sizes specified to the symbol "符号"

对符号的"符号"指定了多个堆栈容量。

P0200 (I) "示例" no longer needed in "文件"

"文件"中有不使用的"示例"。

P0201 (I) "示例" assigned to file "文件"

将"示例"分配到"文件"。

P0202 (I) Executing: "命令"

为了生成示例而正在执行"命令"。

P0203 (I) "示例" adopted by file "文件"

"示例"被分配到"文件"。

L0300 (I) Mode type "模式种类1" in "文件" differ from "模式种类2"

输入了不同模式种类的文件。

L0400 (I) Unused symbol "文件"-"符号"

没有使用"文件"中的"符号"。

L1000 (W) Option "选项" ignored

"选项"无效。忽视"选项"。

L1001 (W) Option "选项 1" is ineffective without option "选项 2"
"选项 1"需要"选项 2"。忽视"选项 1"。

L1002 (W) Option "选项 1" cannot be combined with option "选项 2"
不能同时指定"选项 1"和"选项 2"。忽视"选项 1"。

L1003 (W) Divided output file cannot be combined with option "选项"
指定"选项"时，不能指定分割输出文件。忽视选项的指定。将第 1 个输入的文件名用作输出文件名。

L1004 (W) Fatal level message cannot be changed to other level : "序号"
不能更改 Fatal 级信息的级。忽视"序号"的指定。
能用 change_message 选项更改的错误是 Information/Warning/Error 级。

L1005 (W) Subcommand file terminated with end option instead of exit option
在 end 选项之后没有处理指定。假设 exit 选项进行处理。

L1006 (W) Options following exit option ignored
忽视 exit 选项后的选项。

L1007 (W) Duplicate option : "选项"
"选项"有重复。最后指定的选项有效。

L1008 (W) Option "选项" is effective only in cpu type "CPU 种类"
"选项"只能在"CPU 种类"时有效。忽视"选项"。

L1011 (W) Duplicate module specified in option "选项" : "模块"
用"选项"2 次指定了相同的模块。忽视第 2 次的指定。

L1012 (W) Duplicate symbol/section specified in option "选项" : "名称"
用"选项"2 次指定相同的符号名或者段名。忽视第 2 次的指定。

L1013 (W) Duplicate number specified in option "选项" : "序号"
用"选项"指定了相同的错误号。最后指定的序号有效。

L1100 (W) Cannot find "名称" specified in option "选项"
找不到"选项"指定的符号名或者段名。忽视"名称"的指定。

L1101 (W) "名称" in rename option conflicts between symbol and section
用 rename 选项指定的"名称"存在于段名和符号名。符号名为更改的对象。

L1102 (W) Symbol "符号" redefined in option "选项"
"选项"指定的符号已被定义。照样继续处理。

L1103 (W) Invalid address value specified in option "选项" : "地址"
"选项"指定的"地址"是无效值。忽视"地址"的指定。

L1104 (W) Invalid section specified in option "选项" : "段"
在"选项"中不能指定没有初始值的段。
忽视"段"的指定。

L1110 (W) Entry symbol "符号" in entry option conflicts
在编译或者汇编时，将 entry 选项指定的"符号"以外的符号指定为入口符号。优先指定的选项。

L1120 (W) Section address is not assigned to "段"
没有"段"的地址指定。将"段"分配到最后。

L1121 (W) Address cannot be assigned to absolute section "段" in start option
"段"是绝对地址段。忽视对绝对地址段的地址指定。

L1122 (W) Section address in start option is incompatible with alignment : "段"
start 选项指定的"段"地址与边界调整数发生了矛盾。
根据边界调整数校正段地址。

L1130 (W) Section attribute mismatch in rom option : "段 1、段 2"
rom 选项指定的"段 1"和"段 2"的属性、边界调整数不同。作为"段 2"的边界调整数，其中较大的一方有效。

L1140 (W) Load address overflowed out of record-type in option "选项"
指定了小于地址值的 record 格式。用其它的 record 格式输出超出指定的 record 格式范围。

L1141 (W) Cannot fill unused area from "地址" with the specified value
因为空区的容量不为 space 选项的指定值的倍数，所以指定的数据不能输出到"地址"后的地址。

L1150 (W) Sections in fsymbol option have no symbol
fsymbol 选项指定的段没有外部定义符号。忽视 fsymbol 选项。

L1160 (W) Undefined external symbol "符号"
参照了未定义的"符号"。

L1170 (W) Specified SBR addresses conflict

指定了多个不同的SBR地址。作为SBR=USER处理。

L1171 (W)Least significant byte in SBR="常数" ignored

SBR 选项指定的地址"常数"的低 8 位无效。

L1200 (W) Backed up file "文件 1" into "文件 2"

将"文件 1"备份到"文件 2"。

L1300 (W) No debug information in input files

输入文件中没有调试信息。忽视 debug、sdebug、compress 选项的指定。请确认在编译或者汇编时是否指定了相应的选项。

L1301 (W) No inter-module optimization information in input files

输入文件中没有模块间的优化信息。忽视 optimize 选项。请在编译或者汇编时指定 goptimize 选项。

L1302 (W) No stack information in input files

输入文件中没有堆栈信息。忽视 stack 选项。在输入文件为汇编程序的输出文件或者 SYSROF->ELF 转换文件时，stack 选项无效。

L1310 (W) "段" in "文件" is not supported in this tool

"文件"中有非支持段。忽视"段"。

L1311 (W) Invalid debug information format in "文件"

"文件"中的调试信息不是 dwarf2。删除 debug 信息。

L1320 (W) Duplicate symbol "符号" in "文件"

"符号"有重复。优先先输入的文件中的符号。

L1321 (W) Entry symbol "符号" in "文件" conflicts

输入了多个有入口符号定义的目标文件。先输入的文件中的入口符号有效。

L1322 (W) Section alignment mismatch : "段"

输入了不同边界调整数的同名段。指定的最大边界调整数有效。

L1323 (W) Section attribute mismatch : "段"

输入了不同属性的同名段。在绝对段和相对段的情况下，作为绝对段处理。在 read/write 属性不同的情况下，两者都允许。

L1324 (W) Symbol size mismatch : "符号" in "文件"

输入了大小不同的公共符号或者定义符号。优先定义符号。在都为公共符号的情况下，优先先输入的文件中的符号。

L1330 (W) Cpu type "CPU 种类 1" in "文件" differ from "CPU 种类 2"
输入了不同 CPU 种类的文件。将 H8SX 作为 CPU 种类继续处理。

L1400 (W) Stack size overflow in register optimization
在寄存器优化中，堆栈存取码超出了编译程序的堆栈量限制值。忽视寄存器优化的指定。

L1401 (W) Function call nest too deep
函数的调用嵌套太深，不能进行寄存器优化。

L1410 (W) Cannot optimize "文件"-"段" due to
multi label relocation operation
不能优化持有多个标号的再定位运算的段。"文件"中的"段"不作为优化对象。

L1420 (W) "文件" is newer than "配置"
"文件"更新在"配置"之后。忽视配置信息。

L1500 (W) Cannot check stack size
没有堆栈段，所以不能检查编译时 stack 选项指定的堆栈容量的匹配性。
为了检查编译时 stack 选项的匹配性，在编译或者汇编时需要指定 goptimize 选项。

L1501 (W) Stack size overflow : "堆栈容量"
在编译时，堆栈段大小超出了 stack 选项指定的"堆栈容量"。请更改编译时的选项，或者更改程序以便能减少堆栈容量。

L1502 (W) Stack size in "文件" conflicts with that in another file
用多个文件指定了不同的堆栈容量。请确认编译时的选项。

P1600 (W) An error occurred during name decoding of "示例"
"示例"不能解码。用编码名输出信息。

L2000 (E) Invalid option : "选项"
不支持"选项"。

L2001 (E) Option "选项" cannot be specified on command line
不能在命令行指定"选项"。请在子命令文件中指定。

L2002 (E) Input option cannot be specified on command line
在命令行指定了 input 选项。请在命令行指定无 input 选项的输入文件。

L2003 (E) Subcommand option cannot be specified in subcommand file
在子命令文件中指定了 subcommand 选项。不能嵌套 subcommand 选项。

L2004 (E) Option "选项 1" cannot be combined with option "选项 2"
不能同时指定"选项 1"和"选项 2"。

L2005 (E) Option "选项" cannot be specified while processing "进程"
对于"进程"的处理不能指定"选项"。

L2006 (E) Option "选项 1" is ineffective without option "选项 2"
"选项 1"需要"选项 2"。

L2010 (E) Option "选项" requires parameter
"选项"需要指定参数。

L2011 (E) Invalid parameter specified in option "选项" : "参数"
在"选项"中指定了无效的参数。

L2012 (E) Invalid number specified in option "选项" : "值"
在"选项"指定中指定了无效值。请确认值的范围。

L2013 (E) Invalid address value specified in option "选项" : "地址"
"选项"指定的"地址"是无效值。请用 0～FFFFFFFF 的 16 进制数指定。

L2014 (E) Illegal symbol/section name specified in "选项" : "名称"
"选项"指定的段或者符号名使用了不正确的字符。在段/符号名中能使用的是数字、英文字、_、$（不以数字开头）。

L2016 (E) Invalid alignment value specified in option "选项" :
"边界调整数"
"选项"指定的"边界调整数"是无效值。
请用 10 进制数指定 2 的乘方的值（1、2、4、8、16、32）。

L2020 (E) Duplicate file specified in option "选项" : "文件"
在"选项"指定中 2 次指定了相同的文件。

L2021 (E) Duplicate symbol/section specified in option "选项" : "名称"
在"选项"指定中 2 次指定了相同的符号名或者段名。

L2022 (E) Address ranges overlap in option "选项" : "地址范围"
"选项"指定的"地址范围"有重复。

L2100 (E) Invalid address specified in cpu option : "地址"
在 cpu 选项中指定了 cpu 不能指定的地址。

L2101 (E) Invalid address specified in option "选项" : "地址"
"选项"指定的"地址"超出了 cpu 能指定的地址范围或者 cpu 选项指定的范围。

L2110 (E) Section size of second parameter in rom option is not 0 : "段"
给 rom 选项的第 2 个参数指定了长度不为 0 的"段"。

L2111 (E) Absolute section cannot be specified in rom option : "段"
用 rom 选项指定了绝对地址段。

L2120 (E) Library "文件" without module name specified as input file
作为输入文件，指定了没有模块名的库文件。

L2121 (E) Input file is not library file : "文件(模块)"
输入文件指定的"文件(模块)"不是库文件。

L2130 (E) Cannot find file specified in option "选项" : "文件"
找不到"选项"指定的文件。

L2131 (E) Cannot find module specified in option "选项" : "模块"
没有"选项"指定的模块。

L2132 (E) Cannot find "名称" specified in option "选项"
不存在"选项"指定的符号或者段。

L2133 (E) Cannot find defined symbol "名称" in option "选项"
不存在"选项"指定的外部定义符号。

L2140 (E) Symbol/section "名称" redefined in option "选项"
"选项"指定的符号或者段已被定义。

L2141 (E)Module "模块" redefined in option "选项"
"选项"指定的模块已被注册。

L2200 (E) Illegal object file : "文件"
P2200
输入了 ELF 以外的格式。

L2201 (E) Illegal library file : "文件"
"文件"不是库文件。

L2202 (E) Illegal cpu information file : "文件"
"文件"不是 cpu 信息文件。

L2203 (E) Illegal profile information file : "文件"
"文件"不是配置信息文件。

L2210 (E) Invalid input file type specified for option "选项" :
"文件(种类)"
在指定"选项"时输入了不能处理的文件（种类）。

L2211 (E) Invalid input file type specified while processing "进程" :
"文件(种类)"
对于"进程"的处理输入了不能处理的文件（种类）。

L2220 (E) Illegal mode type "模式种类" in "文件"
输入了不同模式种类的文件。

L2221 (E) Section type mismatch : "段"
输入了不同属性（有无初始值）的同名段。

L2300 (E) Duplicate symbol "符号" in "文件"
"符号"有重复。

L2301 (E) Duplicate module "模块" in "文件"
"模块"有重复。

L2310 (E) Undefined external symbol "符号" referenced in "文件"
在"文件"中参照了未定义的"符号"。

L2311 (E) Section "段1" cannot refer to overlaid section :
"段2"-"符号"
在指定相同地址的覆盖段之间有符号参照。
请不要将"段1"和"段2"分配到相同地址。

L2320 (E) Section address overflowed out of range : "段"
"段"地址超出了能使用的地址范围。

L2321 (E) Section "段1" overlaps section "段2"
"段1"和"段2"的地址有重复。请更改 start 选项的地址指定。

L2322 (E) Section size too large: "段"
段"段"太大。
$TBR 段的大小必须在 1024 个字节以内。

L2330 (E) Relocation size overflow : "文件"-"段"-"偏移"
再定位运算结果超出了再定位长度。可能是不能到达转移目标或者参照了必须分配到特定地址的符号等。请确认在编译或者汇编列表中"段"的"偏移"位置的参照符号是否被分配到正确的位置。

L2331 (E) Division by zero in relocation value calculation :
"文件"-"段"-"偏移"

在再定位运算中发生了被 0 除的运算。请确认在编译或者汇编列表中"段"的"偏移"位置的运算是否有问题。

L2332 (E) Relocation value is odd number :
"文件"-"段"-"偏移"

再定位运算的结果是奇数。请确认在编译或者汇编列表中"段"的"偏移"位置的运算是否有问题。

L2340 (E) Symbol name in section "段" is too long

fsymbol 指定的"段"中的符号的字符数超过了 8174 个字符。

L2400 (E) Global register in "文件" conflicts : "符号","寄存器"

在"文件"中指定的全局寄存器已被分配了别的符号。

L2401 (E) near8,near16 symbol "符号" is outside near memory area

"符号"没有被分配到 near8、near16 范围内。为了能正确计算地址，请更改 start 指定或者解除编译时的 near 指定。

L2402 (E) Number of register parameter conflicts with that in another file : "函数"

在多个文件中，对"函数"指定了不同的寄存器参数的个数。

L2410 (E) Address value specified by map file differs from one after linkage as to "符号"

作为"符号"的地址值，编译时使用的外部符号分配信息文件中的地址和连接后的地址不同。请确认在编译时指定 map 选项的前后是否更改了程序。另外根据 optlnk 的优化，编译时指定 map 选项前后的排列顺序也可能发生了变化。请将编译时的 map 选项或者 optlnk 优化选项置为无效。

L2411 (E) Map file in "文件" conflicts with that in another file

在输入文件之间使用了编译时不同的外部符号分配信息文件。

L2412 (E) Cannot open file : "文件"

不能打开"文件"（外部符号分配信息文件）。请确认文件名和存取权是否正确。

L2413 (E) Cannot close file : "文件"

不能关闭"文件"（外部符号分配信息文件）。磁盘容量可能没有空间。

L2414 (E) Cannot read file : "文件"

不能读"文件"（外部符号分配信息文件）。磁盘容量可能没有空间。

L2415 (E) Illegal map file : "文件"

"文件"（外部符号分配信息文件）格式不正确。请确认文件名是否正确。

L2416 (E) Order of functions specified by map file differs from one after linkage as to "函数名"

在编译时使用的外部符号分配信息文件内的信息和连接后的分配信息中，函数"函数名"和其它函数的排列顺序不同。函数中 static 变量的地址有可能与外部符号分配信息文件和连接后的结果不同。

P2500 (E) Cannot find library file : "文件"

没有指定的库"文件"。

P2501 (E) "示例" has been referenced as both an explicit specialization and a generated instantiation

对于已定义的示例，需要生成示例。

对于正在使用"示例"的文件，请确认在建立可再定位目标文件时是否没有使用 form=relocate。

P2502 (E) "示例" assigned to "文件 1" and "文件 2"

在"文件 1"和"文件 2"中重复定义了"示例"。

对于正在使用"示例"的文件，请确认在建立可再定位目标文件时是否没有使用 form=relocate。

L3000 (F) No input file

没有输入文件。

L3001 (F) No module in library

库中的模块个数为 0。

L3002 (F) Option "选项 1" is ineffective without option "选项 2"

"选项 1"需要"选项 2"。

L3100 (F) Section address overflow out of range : "段"

"段"的地址超出了可使用的上限区域。

请更改 start 选项的地址指定。

关于地址空间的详细内容，请参照各 CPU 的硬件手册。

L3101 (F) Section "段 1" overlaps section "段 2"

"段 1"和"段 2"的地址有重复。请更改 start 选项的地址指定。

L3102 (F) Section contents overlap in absolute section "段"
绝对地址段的段内数据地址有重复。请修正源程序。

L3110 (F) Illegal cpu type "cpu 种类" in "文件"
输入了不同 cpu 种类的文件。

L3111 (F) Illegal encode type "Endian 种类" in "文件"
输入了不同Endian种类的文件。

L3112 (F) Invalid relocation type in "文件"
"文件"中有不支持的再定位类型。请确认编译和汇编的版本是否正确。

L3200 (F) Too many sections
段的个数超过了极限值。如果指定多个文件的输出，就有可能解决此问题。

L3201 (F) Too many symbols
符号个数超过了极限值。如果指定多个文件的输出，就有可能解决此问题。

L3202 (F) Too many modules
模块的个数超过了极限值。请分库建立。

L3300 (F) Cannot open file : "文件"
P3300
不能打开"文件"。请确认文件名和存取权是否正确。

L3301 (F) Cannot close file : "文件"
不能关闭"文件"。磁盘容量可能没有空间。

L3302 (F) Cannot write file : "文件"
不能写"文件"。磁盘容量可能没有空间。

L3303 (F) Cannot read file : "文件"
P3303
不能读"文件"。输入了空文件或者磁盘容量可能没有空间。

L3310 (F) Cannot open temporary file
P3310
不能打开临时文件。HLNK_TMP 的指定不正确或者磁盘容量可能没有空间。

L3311 (F) Cannot close temporary file
不能关闭临时文件。磁盘容量可能没有空间。

L3312 (F) Cannot write temporary file
不能写临时文件。磁盘容量可能没有空间。

L3313 (F) Cannot read temporary file
不能读临时文件。磁盘容量可能没有空间。

L3314 (F) Cannot delete temporary file
不能删除临时文件。磁盘容量可能没有空间。

L3320 (F) Memory overflow
P3320
优化连接编辑程序内部使用的存储器不足。请增加存储器。

L3400 (F) Cannot execute "装入模块"
不能启动"装入模块"。请确认是否设定了"装入模块"的路径。

L3410 (F) Interrupt by user
检测到标准输入终端的“(cntl)+C”键中断。

L3420 (F) Error occurred in "装入模块"
在执行"装入模块"中发生错误。

P3500 (F) Bad instantiation request file -- instantiation assigned to more than one file
示例生成指定文件有错。
请重新编译连接对象的文件。

P3501 (F) Instantiation loop
示例生成处理进入了循环处理。
输入文件名有可能与其它文件的示例生成要求文件一致。为了使文件名（除了扩展名）不一致，请更改文件名。

P3502 (F) Cannot create instantiation request file "文件"
不能建立示例生成指定文件。
请确认目标建立目录的存取权是否正确。

P3503 (F) Cannot change to directory "目录"
不能移动到"目录"。
请确认是否存在"目录"。

P3504 (F) File "文件" is read-only
是只读"文件"。
请更改存取权。

L4000 (-) Internal error : ("内部错误号") "文件 行号" / "注释"
P4000
在优化连接编辑程序的处理中发生了内部错误。
请将信息中的内部错误号、文件、行号、注释与销售部门联系。

第 15 章　标准库生成工具/格式转换程序的错误信息

15.1　错误格式和错误级

本章说明用以下格式输出的错误信息和错误内容：

错误号　（错误级）　错误信息
错误内容

根据错误的重要程度，错误级分为 3 种。

		错误级	运行
G1000 - G1999	（W）	警告	继续处理。
G2000 - G2999	（E）	错误	继续选项解析处理，中断处理。
G3000 - G3999	（F）	致命错误	中断处理。

15.2　信息一览

`G1001 (W) Debug information ignored`

在转换对象文件中，存在用`#pragma option`进行了有优化和无优化指定的函数。在删除调试信息后进行转换。

`G1002 (W) Command parameter specified twice`

二次以上指定了相同的选项。在相同的选项中，最后指定的选项有效。请确认在选项的指定中是否有错。

`G2001 (E) Cannot open file "文件"`

不能打开文件。请确认文件名和存取权是否正确。

`G2002 (E) Illegal file type "文件"`

在从 SYSROF 向 ELF 转换时指定了目标文件或者库文件以外的文件，或者在从 ELF 向 SYSROF 转换时指定了装入模块文件以外的文件。请确认文件种类后重新执行。

`G2003 (E) Illegal file format "文件"`

文件格式不正确。请确认文件内容后重新执行。

`G3001 (F) Invalid command parameter "参数"`

指定了不正确的命令参数。请确认命令参数的内容后重新执行。

`G3002 (F) No input file`

没有输入文件。

`G3003 (F) Command parameter buffer overflow`

命令行的指定超过了 32767 个字符。

`G3101 (F) Cannot open file "文件"`

不能打开文件。请确认文件名和存取权是否正确。

`G3102 (F) Cannot input file "文件"`

不能从指定文件输入。请确认是否在存取转换对象文件。

`G3103 (F) Cannot create file "文件"`

不能生成文件。请确认磁盘容量是否有空间。

`G3104 (F) Cannot output file "文件"`

不能写文件。请解除写禁止。

`G3105 (F) Cannot open internal file`

不能打开内部生成的临时文件。请确认是否在存取临时文件。

`G3106 (F) Cannot output internal file`

不能输出到内部生成的临时文件。有可能磁盘容量没有空间或者磁盘有物理错误。

`G3107 (F) Memory overflow`

不能分配内部使用的存储区。请在确保必要的存储器后重新执行。

`G3108 (F) Illegal format in archive "文件"`

指定的文件不是档案文件的格式。

`G3109 (F) Cannot find "文件名"`

不能找到文件。请确认环境变量 PATH 的设定。

`G3201 (F) Cannot execute compiler`

不能启动编译程序。请确认编译程序的路径名和环境变量。

`G3202 (F) Cannot execute optlinker`

不能启动优化连接编辑程序。请确认优化连接编辑程序的路径名。

`G3203 (F) Interrupt by user`

在执行中检测到中断。

`G3204 (F) Cannot execute assembler`

不能启动汇编程序。请确认汇编程序的路径名。

`G3300 (F) Already existent file "文件"`

文件已经存在。

第 16 章　极限值

16.1　编译程序的极限值

编译程序的极限值如表 16.1 所示。
请在此极限值的范围内建立源程序。

表 16.1　编译程序的极限值

	分类	项目			极限值
1	启动	能用 define 选项指定的宏名总数			无限制
2		文件名的长度			无限制（取决于 OS）
3	源程序	1 行的字符数			32,768 个字符 （H8SX/H8S） 16,384 个字符 （300H/300）
4		1 个文件的源程序行数			无限制
5		能编译的源程序的总行数			无限制
6	预处理器	#include 语句的嵌套深度			无限制
7		#define 语句的宏名总数			无限制
8		宏定义、宏调用可指定的参数			无限制
9		宏名的再置换个数			无限制
10		#if、#ifdef、#ifndef、#else、#elif 语句的嵌套深度			无限制
11		在#if、#elif 语句中能指定的运算符、非运算符的合计数			无限制
12	声明	函数定义数			无限制
13		为外部连接的标识符（外部名）的个数			无限制
14		在 1 个函数中有效标识符（内部名）的个数			无限制
15		限定基本型的指针型、数组型、函数型的合计数			16 个
16		数组的维数			6 维
17		数组/ 结构体 的大小 *1	H8SX H8/2600 H8/2000 H8/300H H8/300	正常模式 正常模式 正常模式 正常模式	65,535 个字节
			H8SX H8SX H8SX	中级模式 高级模式（有 ptr16） 最高模式（有 ptr16）	32,767 个字节
			H8/300H	高级模式	16,777,215 个字节
			H8SX H8SX H8/2600 H8/2000	高级模式（无 ptr16） 最高模式（无 ptr16） 高级模式 高级模式	2,147,483,647 个字节 4,294,967,295 个字 （指定 legacy=v4 选项时）
18	语句	复合语句的嵌套深度			无限制
19		根据迭代语句（while 语句、do 语句、for 语句）、选择语句（if 语句、switch 语句）的组合产生的嵌套深度			4,096 层 （H8SX/H8S） 256 层（300H/300）
20		在 1 个函数中能指定的 goto 标号的个数			2,147,483,646 个 （H8SX/H8S） 511 个（300H/300）

	分类	项目	极限值
21	语句	switch 语句的个数	2,048 个 *4
22		switch 语句的嵌套深度	2,048 层（H8SX/H8S） 128 层（300H/300）
23		在 1 个 switch 语句中能指定的 case 标号的个数	2,147,483,646 个 （H8SX/H8S） 511 个（300H/300）
24		for 语句的嵌套深度	2,048 层 （H8SX/H8S） 128 层（300H/300）
25	表达式	字符串的长度	32,766 字符
26		用函数定义、函数调用能指定的参数	2,147,483,646 个 （H8SX/H8S） 63 个（300H/300） *2
27		在 1 个表达式中能指定的运算符和非运算符的合计数	约 500 个
28	标准库	用 open 函数能同时打开的文件个数	可变 *3

【注】 在指定 legacy=v4 选项时，极限值和 300H/300 相同。

*1　在高级、中级、最高模式中，如果指定地址空间的位宽，就优先对应地址空间位宽的地址空间大小。在 H8SX 的高级模式和最高模式中，根据 ptr16 选项的有无，值不同。

*2　在非静态函数成员时为 62 个。

*3　详细内容请参照“9.2.2(5)　C/C++库函数的初始设定（_INITLIB）”。

*4　本版本的变更内容。

16.2　汇编程序的极限值

汇编程序的极限值如表 16.2 所示。

表 16.2　汇编程序的极限值

	项目	极限值		
1	1 行的字符数	8192 个字符		
2	字符常数	最多 4 个字符		
3	符号长度	无限制 *1		
4	符号个数	无限制		
5	外部参照符号的个数	无限制		
6	外部定义符号的个数	无限制		
7	段的最大容量 *2	H8SX	最高模式………………………	最多 H'FFFFFFFF 字节
		H8SX	高级模式………………………	最多 H'FFFFFFFF 字节
		H8SX	中级模式………………………	最多 H'00FFFFFF 字节
		H8SX	正常模式………………………	最多 H'0000FFFF 字节
		H8S/2600	高级模式………………………	最多 H'FFFFFFFF 字节
		H8S/2600	正常模式………………………	最多 H'0000FFFF 字节
		H8S/2000	高级模式………………………	最多 H'FFFFFFFF 字节
		H8S/2000	正常模式………………………	最多 H'0000FFFF 字节
		H8/300H	高级模式………………………	最多 H'00FFFFFF 字节
		H8/300H	正常模式………………………	最多 H'0000FFFF 字节
		H8/300	………………………………	最多 H'0000FFFF 字节
		H8/300L	………………………………	最多 H'0000FFFF 字节
8	段的个数	指定 goptimize 选项时	有调试 ………………………	H'FEF1 个
			无调试 ………………………	H'FEFA 个
		未指定 goptimize 选项时	有调试 ………………………	H'FEF2 个
			无调试 ………………………	H'FEFB 个
9	文件 include	最多嵌套 30 层		
10	字符串长度	最多 255 个字符		
11	文件名长度	无限制（取决于 OS）		

【注】 *1　预处理器变量名、宏名和宏形式参数名最多 32 个字符。
能用-DEFINE/.DEFINE 指定的置换符号的字符个数没有限制。但是，置换字符串最多 255 个字符，并且 1 行中能记述的字符个数最多 8192 个字符。

*2　段的最大容量因地址空间的指定而不同。

第 17 章　面向 AE5 的功能支持

17.1　编译程序的功能

17.1.1　概要

说明面向 AE5 的追加功能的使用方法。

在使用编译程序支持的 EEPROM 存取内部函数时，由于此内部函数使用 EEPMOV.B 和 EEPMOV/P.W 指令，所以必须包含头文件<machine.h>或者<eeprom.h>并指定 EEPROM 选项。

另外，在__asm{ }块中记述 EEPMOV/P.W 指令时，也必须指定 EEPROM 选项（除了指定-CPU=AE5 以外）。

在用 HEW 指定 EEPROM 选项时，必须在选项窗口的编译程序选项卡的<其它>种类的[用户指定选项 :]的编辑框中记述“-eeprom”（不区别英文大小写字母）。

17.1.2　编译程序选项

V.6.01 追加的选项如下所示。
除此以外，能使用在 CPU 种类为 H8SXA 时可用的选项/扩展功能（但是，set_vbr 内部函数除外）。

表 17.1　特殊选项一览表

项目		命令行格式	对话菜单	指定内容
1	CPU 种类	cpu = ae5	编译程序 <CPU>	指定 CPU。
2	EEPMOV/P.W 指令的使用	eeprom	编译程序 <其它> [用户指定选项 :]	允许使用 EEPMOV/P.W 指令

CPU 种类

CPu

编译程序 <CPU>

格　式　　`CPu = AE5`

说　明　　指定建立的目标程序的 CPU 种类。
如果给予选项指定 AE5，就不能指定乘除运算器。
将地址空间大小固定为 24bit。
如果指定了本选项，eeprom 选项总是有效。

EEPROM 存取

EEPROM

编译程序 <其它>[用户指定选项 :]

格　式　　`EEPROM`

说　明　　允许使用通过 EEPMOV.B 指令或者 EEPMOV/P.W 指令存取 EEPROM 的内部函数。
在指定本选项并包含了头文件<eeprom.h>时，将如下所示的内部函数展开为有 ECR 或 EPR 设定的 EEPMOV.B 或者 EEPMOV/P.W 指令。

- eepromb
- eepromw
- eepromb_epr
- eepromw_epr

另外，允许在_ _asm{ }块内记述 EEPMOV/P.W 指令。

备　考　　关于内部函数，请参照各内部函数的解说。
关于 ECR 和 EPR，请参照硬件手册。

17.1.3　内部函数

表 17.2　内部函数一览表

项目		规格	功能
1	特殊指令	unsigned char eepromb(void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval)	在设定 ECR 后，用 EEPMOV.B 指令从 src 指向的地址开始，将 size 指定的字节数的数据块传送到 dst 指向的地址。
		unsigned int eepromw(void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval)	在设定 ECR 后，用 EPMOV/P.W 指令从 src 指向的地址开始，将 size 指定的字节数的数据块传送到 dst 指向的地址。
		unsigned char eepromb_epr(void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)	在设定 EPR 和 ECR 后，用 EEPMOV.B 指令从 src 指向的地址开始，将 size 指定的字节数的数据块传送到 dst 指向的地址。
		unsigned int eepromw_epr(void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)	在设定 EPR 和 ECR 后，用 EEPMOV/P.W 指令从 src 指定的地址开始，将 size 指定的字节数的数据块传送到 dst 指向的地址。

块传送指令（有 ECR 设定）

*unsigned char eepromb(void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval)*

*unsigned int eepromw(void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval)*

说　明	从 `src` 指向的地址开始，将 `size` 指定的字节数的数据块传送到 `dst` 指向的地址。`eepromb` 函数用 `EEPMOV.B` 指令进行块传送，`eepromw` 函数用 `EEPMOV/P.W` 指令进行块传送。 本内部函数在将 `dst`、`src`、`size` 设定到寄存器以及对 `ecr` 指向的地址设定 `ecrval` 后执行块传送指令。 在传送成功时返回 `0`，失败时返回未被传送的数据长度。对于 `eepromb` 函数，能给 `size` 指定 `0` 到 `255` 的数据；对 `eepromw` 関数，能给 `size` 指定 `0` 到 `65535` 的数据。在给 `size` 指定 `0` 时，不进行块传送。
头文件	`machine.h / eeprom.h`
返回值	未被传送的数据长度(`0`～`size`)

参　数

`dst`	指向传送目标的指针
`src`	指向传送源的指针
`size`	传送长度
`ecr`	硬件寄存器 ECR 的地址
`ecrval`	给硬件寄存器 ECR 的设定值

例

```
#include <eeprom.h>
#define ecr_ptr ((volatile unsigned char *)(0xZZZZZZ))

char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromb(b, a, 10, ecr_ptr, 1);
}
```

备　考

本内部函数只在 CPU 种类为 AE5 或 H8SX 并且指定了`-eeprom` 选项时有效。
如果在给 CPU 种类指定了 AE5 以外的种类时使用本内部函数，就需要在编译时指定 `eeprom` 选项。
有关 ECR、EPR 以及其它的相关事项，请参照硬件手册。

块传送指令（有 EPR、ECR 设定）

*unsigned char eepromb_epr(void *dst, const void *src, unsigned char size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)*

*unsigned int eepromw_epr(void *dst, const void *src, unsigned int size, volatile unsigned char *ecr, unsigned char ecrval, volatile unsigned char *epr, unsigned char eprval)*

说　明	从 src 指向的地址开始，将 size 指定的字节数的数据块传送到 dst 指向的地址。eepromb_epr 函数用 EEPMOV.B 指令进行块传送，eepromw_epr 函数用 EEPMOV/P.W 指令进行块传送。 本内部函数在将 dst、src、size 设定到寄存器、对 epr 指向的地址设定 eprval 以及对 ecr 指向的地址设定 ecrval 后执行块传送指令。 在传送成功时返回 0，失败时返回未被传送的数据长度。对于 eepromb_epr 函数，能给 size 指定 0 到 255 的数据；对 eepromw_epr 函数，能给 size 指定 0 到 65535 的数据。在给 size 指定 0 时，不进行块传送。
头文件	machine.h / eeprom.h
返回值	未被传送的数据长度(0～size)

参　数

dst	指向传送目标的指针
src	指向传送源的指针
size	传送长度
ecr	硬件寄存器 ECR 的地址
ecrval	给硬件寄存器 ECR 的设定值
epr	硬件寄存器 EPR 的地址
eprval	给硬件寄存器 EPR 的设定值

例

```
#include <eeprom.h>
#define ecr_ptr ((volatile unsigned char *)(0xZZZZZZ))
#define epr_ptr ((volatile unsigned char *)(0xWWWWWW))

char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromb_epr(b, a, 10, ecr_ptr, 1, epr_ptr, 1);
}
```

备　考

本内部函数只在 CPU 种类为 AE5 或 H8SX 并且指定了-eeprom 选项时有效。
如果在给 CPU 种类指定了 AE5 以外的种类时使用本内部函数，就需要在编译时指定 eeprom 选项。
有关 ECR、EPR 以及其它的相关事项，请参照硬件手册。

17.2　汇编程序的功能

V.6.01 追加的选项如下所示。

除此以外，能使用在 CPU 种类为 H8SXA 时可用的选项/扩展功能。但是，用 SBR 和 VBR 的指令不能使用。

表 17.3　特殊选项一览表

项目		命令行格式	对话菜单	指定内容
1	CPU 种类	-CPu = AE5	汇编程序 <CPU>	指定 CPU。
2	EEPMOV/P.W 指令的使用	-EEPROM	汇编程序 <其它> [用户指定选项 :]	允许使用 EEPMOV/P.W 指令

CPU 种类

CPu

汇编程序 <CPU>

格　式　　CPu = AE5

说　明　　指定建立的目标程序的 CPU 种类。
如果给予选项指定 AE5，就不能指定乘除运算器。
将地址空间的大小固定为 24bit。
如果指定了本选项，eeprom 选项总是有效。

EEPROM 存取

EEPROM

汇编程序<其它>[用户指定选项 :]

格　式　　EEPROM

说　明　　允许使用 EEPMOV/P.W 指令。

备　考　　本选项只在 CPU 种类为 H8SX 时有效。

第 18 章　版本升级的注意事项

18.1　版本升级时的注意事项

说明从旧版本（H8S,H8/300 系列 C/C++编译程序包 Ver.3.x 及以前的版本）升级后使用的注意事项。

18.1.1　程序的运行保证

在升级编译程序版本后进行程序开发时，程序的运行有可能变化。创建程序时，必须注意以下各点，充分测试用户程序。

（1）取决于执行时间和时序的程序

语言规格不规定程序的执行时间。因此，根据编译程序版本的不同，有可能产生程序的执行时间和 I/O 等外围设备的时序偏差、或者如中断处理等异步处理的时间差异等，导致程序的运行发生变化。

（2）一个表达式含有 2 个以上副作用的程序

在一个表达式中含有 2 个以上的副作用时，根据编译程序的版本，运行有可能发生变化。

例：

```
a[i++]=b[i++];            /* i 的增量顺序不定。*/
f(i++, i++);              /* 根据增量顺序，参数值发生变化。*/
                          /* i 的值为 3 时，为 f(3, 4)或者 f(4, 3)。*/
```

（3）取决于结果溢出或者不适当运算的程序

如果产生溢出或者进行了不适当运算，就不能保证结果值。因此，根据版本的变化，运行有可能发生变化。

例：

```
int a, b;
x=(a*b)/10;                 /* 根据 a 和 b 的值的范围，有可能发生溢出 */
```

（4）变量没被初始化和型的不一致

在变量没被初始化时，或者参数和返回值的型在调用侧和被调用侧没有对应时，会存取不正确的值。因此，根据编译程序的版本，运行有可能发生变化。

例：

file 1:

```
int f(double d)
{
          :
}
```

file 2:

```
int g(void)
{
          f(1);
}
```

函数调用侧的参数为int型，
但是函数定义侧的参数为
double型，所以不能正确
参照值。

上述记载的信息并不表示全部可能出现的状况。因此，客户必须在正确使用本编译程序的基础上，充分测试用户程序。

18.1.2　与旧版本的兼容性

以下说明在连接旧版本（Ver.4.x 及以前的版本）的编译程序及其附属的汇编程序和连接编辑程序输出的目标文件或库文件时、或者在使用旧版本使用的调试程序时的注意点。

（1）strict_ansi（从 Ver.6.01 开始的版本）

如果指定 strict_ansi 选项，浮点的运算结果就有可能与旧版本（Ver.4.x 及以前的版本）不同。为了得到相同的运算结果，必须在不指定 strict_ansi 选项的情况下进行编译，或者在对全部文件指定 strict_ansi 选项后重新进行编译。

（2）cpuexpand（从 Ver.6.01 开始的版本）

如果指定 cpuexpand 选项，运算结果就有可能与旧版本（Ver.4.x 及以前的版本）不同。为了得到和旧版本相同的运算结果，在 CPU 为 2600A/2600N/2000A/2000N 时，必需在指定 legacy=v4 选项后进行编译；在 CPU 为 H8SX 中任何一个 CPU 时，必须请在对全部文件指定 cpuexpand 选项后重新进行编译。

对于运算结果不同的表达式，请参照“2.2.2　目标选项”中的 cpuexpand=v6 选项说明。

（3）code=asmcode（从 Ver.6.01 开始的版本）

如果从 Ver.6.01 指定 code=asmcode 选项，就在汇编源程序中输出.STACK 控制指令。

因此，需使用.STACK 对应的汇编程序（Ver.6.01）。

（4）带初期值变量的分配段的变更（从 Ver.6.01 开始的版本）

在 Ver6.00 中，如果定义了带初期值的变量，并且给 CPU 种类指定了 H8/300、H8/300H、H8S/2000、H8/2600，就将该变量输出到 D 段。

在 Ver.6.01 中，如果给 CPU 种类指定了 H8S/2600 或 H8S/2000，就将该变量输出到 C 段。在 CPU 种类为 H8/300、H8/300H 时，该变量仍然被输出到 D 段。

如果给 CPU 种类指定了 H8SX，就与版本无关，总是被输出到 C 段。

（5）目标格式（从 Ver.4.0 开始的版本）

格式从原来的 SYSROF 更改为标准格式的 ELF 格式。另外，调试信息格式也更改为标准格式的 DWARF2 格式。

将旧版本（Ver.3.x 及以前的版本）的编译程序和汇编程序输出的目标文件（SYSROF）输入到优化连接编辑程序时，必须用文件转换程序将目标文件转换为 ELF 格式。但是，不能转换连接编辑程序输出的可再定位文件（扩展名 rel）和含有可再定位文件的库文件。

另外，在使用支持 SYSROF 格式和 ELF/DWARF1 格式装入模块的调试程序时，必须用文件转换程序将 ELF/DWARF2 格式装入模块转换为 SYSROF 或者 ELF/DWARF1 格式。但是，如果在同一文件中存在用#pragma option（Ver.4.0 的追加功能）指定有或者无优化的函数，就不能转换调试信息的部分，而只转换目标部分。

（6）函数接口更改选项的追加（从 Ver.4.0 开始的版本）

作为更改函数接口规则的选项，追加了 structreg 和 longreg。在指定此选项时，必须全部重新编译。另外，还必须修正与汇编例程的接口。

（7）堆栈区（从 Ver.4.0 开始的版本）

能用 stack 选项指定堆栈的计算大小。

在省略该选项时，stack=medium（用 2 个字节进行堆栈的计算）有效，所以需要更改时，必须用 stack 选项指定。

（8）const 数据输出段（从 Ver.4.0 开始的版本）

在指定 volatile 选项时，Ver.3.x 将有 const 声明的变量输出到 D 段，但是在 Ver4. 0 中，更改为输出到 C 段。

（9）数据分配（从 Ver.4.0 开始的版本）

通过 align/noalign 选项，能按照各边界调整重新排列数据。

在省略此该选项时，align（按照边界调整进行重新排列）有效，所以需要抑制重新排列时，必须指定 noalign 选项。

（10）$ABS8C、$ABS8D、$ABS8B 段的边界调整（从 Ver.4.0 开始的版本）

在指定#pragma abs8 或者_ _abs8、abs8 选项时，将被输出的$ABS8C、$ABS8D、$ABS8B 段的边界调整数从原来的 2 更改为 1。

因此，指定#pragma abs8 或者_ _abs8、abs8 选项时有效的变量条件是：从“成员中具有 char、unsigned char 型变量/数组、或者 char、unsigned char 型变量/数组的结构体/类”更改为“边界调整为 1 的变量/数组/结构体/类”。

（11）包含文件的原点（从 Ver.4.0 开始的版本）

废除了 chgincpath 选项。在以目录相对格式检索被指定的 include 文件时，总是从有源文件的目录开始检索。

（12）C++程序（从 Ver.4.0 开始的版本）

因为更改了编码规则和执行方式，所以不能连接旧版本编译程序建立的 C++目标文件。必须在重新编译后使用。

另外，也更改了在设定执行环境时使用的全局类目标初始处理/后处理的库函数名。请参照“**9.2.2 执行环境的设定**”，进行修正。

（13）公共段的废除（汇编程序，从 Ver.4.0 开始的版本）

随着目标格式的变更，废除了公共段的支持。

（14）.END 控制指令的入口指定（汇编程序，从 Ver.4.0 开始的版本）

能用.END 控制指令进行入口指定的符号只为外部定义符号。

（15）模块间的优化（从 Ver.4.0 开始的版本）

旧版本的编译程序和汇编程序输出的目标文件不能成为模块间的优化对象。对于需要模块间优化对象的文件，必须重新进行编译和汇编。

（16）有关优化连接编辑程序支持的目标

根据版本的不同，优化连接编辑程序支持的编译程序/汇编程序不同。
各连接编辑程序支持的工具版本如下所示。
不能保证没有记载的版本目标文件的连接处理。

优化连接编辑程序 Ver.7.00：Ver.4.0 以前的编译程序、Ver.4.0 以前的汇编程序
优化连接编辑程序 Ver.8.00：Ver.6.0 以前的编译程序、Ver.6.0 以前的汇编程序
优化连接编辑程序 Ver.9.00：Ver.6.01 以前的编译程序、Ver.6.01 以前的汇编程序

（17）选项的统一

对于旧版本和 Ver.6.01，以下所示的编译程序选项必须相同。

cpu、regparam、pack、structreg/nostructreg、longreg/nolongreg、stack、double=float、rtti、exception

但是，在将旧版本的目标文件和 Ver.6.01 的目标文件连接时，不能使用如下所示的 Ver.6.01 的选项：

byteenum、bit_order=right、indirect=extended、ptr16、sbr

在将 Ver.4.0 以前版本的目标文件和 Ver.6.01 目标文件连接时，不能使用从 Ver.4.0 开始支持的以下选项：

structreg、longreg、stack=small/medium、double=float、rtti、exception

在将 Ver.3.0 以前版本的目标文件和 Ver.6.01 目标文件连接时，不能使用从 Ver.3.0 开始支持的以下选项：

regparam=3、 pack=1

18.1.3　命令行接口

（1）汇编程序（Ver.4.0）、优化连接编辑程序（Ver.7.0）命令行的指定方法

在文件名和选项之间需要空格。
另外，文件名和选项的指定顺序没有限制。

（2）优化连接编辑程序选项 （从 Ver.7.0 开始的版本）

废除了会话形式的选项指定的支持。另外，将旧版本的模块间优化工具（optlnk38）、连接编辑程序（lnk）、库生成程序（lbr）、目标转换程序（cnvs）合并成优化连接编辑程序（optlnk）。因此，命令行规格有了很大的更改。更改的命令一览表如**表 18.1** 和**表 18.2** 所示。

表 18.1　连接命令的变更一览表

No.	命令名	Ver.6.0	Ver.7.0	备考
1	start	start=段(地址) 缩短型 st	start=段/地址 缩短型 star	—
2	rom	rom=(rom 段, ram 段)	rom=rom 段/ram 段	—
3	define	define=外部名(定义值)	define=外部名=定义值	—
4	rename	rename= ed=更改前(更改后), er=更改前(更改后), un=更改前(更改后) 缩短型 re	rename= (更改前=更改后), (更改前=更改后), — 缩短型 ren	根据目标格式的变更，废除了单元的概念
5	delete	delete= ed=单元.符号 un=单元	delete=(符号) —	根据目标格式的变更，废除了单元的概念
6	print / noprint	print noprint	list —	可省略文件名
7	mlist	mlist	list	—
8	information	information	message	—
9	directory	directory	HLNK_DIR(环境变量)	—
10	form	缩短型 f	缩短型 fo	—
11	output / nooutput	缩短型 o 可指定 nooutput	缩短型 ou 不可指定 nooutput	只能指定 output
12	cpu	缩短型 c	缩短型 cp	可直接指定范围
13	elf / sysrof / sysrofplus	elf / sysrof / sysrofplus	废除	总是为 ELF
14	exclude / noexclude	exclude / noexclude	废除	总是为 exclude
15	align_section	align_section	废除	总是有效*
16	check_section	check_section	废除	总是有效*
17	cpucheck	cpucheck	废除	总是有效*
18	udf / noudf	udf / noudf	废除	总是输出*
19	udfcheck	udfcheck	废除	总是有效*
20	echo / noecho	echo / noecho	废除	总是抑制
21	exchange	exchange	废除	根据目标格式的变更，废除了单元的概念
22	autopage	autopage	废除	无对象 cpu
23	abort	abort	废除	废除了会话格式
24	list	list	废除	不同于 Ver.7.0 的 list 选项
25	library / nolibrary	可指定 nolibrary	不可指定 nolibrary	只能指定 library
26	exit	不可省略	可省略	—
27	debug / nodebug	省略时：nodebug	省略时： 取决于输入文件 debug 信息的有无	—

【注】* 能用 change_message 选项置为无效。

表 18.2　库生成程序命令的变更一览表

No.	命令名	Ver.2.0	Ver.7.0	备考
1	add	add	input	—
2	directory	directory	HLNK_DIR(环境变量)	—
3	slist	slist	list show	—
4	list	list (s)	list show	—
5	delete	缩短型 d	缩短型 del	—
6	create	create (s | u)	library form=library(s | u)	—
7	output	output (s | u)	output form=library(s | u)	—
		缩短型 o	缩短型 ou	
8	replace	缩短型 r	缩短型 rep	—
9	abort	abort	废除	废除了会话格式
10	exit	不可省略	可省略	—

18.1.4　提供的内容

在“H8S,H8/300 系列 C/C++编译程序包”提供的内容中，从 Ver.4 的程序包开始更改了以下文件：

（1）CPU 信息文件生成工具

能用 optlnk 的 cpu 选项直接指定地址范围。

能使用旧版本 CPU 信息文件生成工具建立的 cpu 信息文件。在更改/建立 CPU 信息时，必须用 cpu 选项直接指定地址范围。

（2）标准库文件

为了能任意指定函数接口和优化选项，由原来的标准库文件提供更改为由标准库生成工具提供。

（3）头文件

根据 bool 型的支持，删除了 defbool.h。

18.1.5　列表文件规格

（1）编译列表（从 Ver.4.0 开始的版本）

为了易看，更改了列数。另外，能选择制表符的列数。

（2）优化连接编辑程序（从 Ver.7.0 开始的版本）

更新了原来的连接映像列表和库列表的格式。

18.2　追加/改善的内容

18.2.1　共同的追加/改善

（1）编译程序 Ver.4.0、汇编程序 Ver.4.0 和优化连接编辑程序 Ver.7.0 主要追加/改善的功能

（a）限制值的放宽

大幅度放宽了源程序和命令行的限制。

- 文件名长度：251 个字节 → 无限制
- 符号长度：251 个字节 → 无限制
- 符号数：65,535 个 → 无限制
- 源程序行数：C/C++：32,767 行、ASM：65,535 行 → 无限制
- C 程序行长度：8,192 个字符 → 16,384 个字符
- C 程序字符串长度：512 个字符 → 16,384 个字符
- 子命令文件行长度：ASM：300 个字节、optlnk：512 个字节 →无限制
- 优化连接编辑程序 ROM 选项的参数个数：64 个 → 无限制

（b）目录名和文件名的连字符（-）

能给目录名和文件名指定连字符（-）。

（c）版权显示抑制

能通过 logo/nologo 选项，指定版权显示的有无。

（d）错误信息的前缀

根据综合开发环境的错误帮助功能支持，在编译程序、优化连接编辑程序的错误信息的前头附加了前缀。

18.2.2　编译程序的追加/改善

（1）Ver.4.0 主要追加/改善的功能

（a）关键字的支持

能使用关键字（__interrupt、__indirect、__entry、__abs8、__abs16、__regsave、__noregsave、__inline、_ _global_register），对函数或者变量的声明及定义指定属性。

（b）向量表的生成功能

能使用#pragma interrupt、#pragma indirect、#pragma entry 和__interrupt、__indirect、__entry 的 vect 指定，自动生成函数的向量表。

（c）__evenaccess 的支持

保证用__evenaccess 指定的常数和变量的偶数字节的存取。

（d）寄存器参数的指定扩展

能使用__regparam2 和__regparam3，给各函数指定寄存器参数的个数。

（e）函数单位选项的指定

能使用#pragma option，以函数为单位指定选项。

（f）数据的 near 分配支持

能使用__near8 和__near16，优化数组和结构体的地址计算码。但是，指针长度不变。

（g）堆栈的 near 分配支持

能使用 stack 选项优化堆栈区的堆栈地址计算码。

（h）内部函数的追加

追加了以下的内部函数：

- 无符号溢出运算

（i）double=float 的支持

通过 double=float 选项，将 double 型声明数据和浮点常数作为 float 型处理。

（j）noregsave 函数的强化支持

调用#pragma noregsave、__noregsave 声明函数时，为了在调用函数侧保证寄存器而进行了更改。

（k）多环境变量的指定

能通过 include 目录的环境变量（CH38）指定多个目录。

（l）结构体参数/返回值的寄存器传递

能使用 structreg 选项，用寄存器传递长度较小的结构体参数/返回值。

（m）4 字节参数/返回值的寄存器传递（cpu=300）

能使用 longreg 选项，用寄存器传递 4 字节参数/返回值。

（n）非 volatile 变量的循环外移动条件

循环判断式中的非 volatile 的外部变量即使在循环内没有副作用（函数调用、赋值等），也总是抑制循环外移动优化。

（o）speed=loop=1 | 2 的支持

能通过 speed=loop=1|2 选项控制循环展开优化的执行。

（p）通过边界调整的数据分配更改

通过 align 选项，能按各边界调整进行数据的重新分配，使边界调整的空间达到最小限。

（q）隐式声明的追加

用#define 隐式声明_ _HITACHI_ _、_ _HITACHI_VERSION_ _等。

（r）static 标号名

为了能在#pragma asm～#pragma endasm 和#pragma inline_asm 函数中参照文件作用域的 static 标号，将标号名更改为_ _$(名称)。

但是，连接列表中显示_(名称)。

（s）语言规格的扩展/变更

- 抑制 union 初始化时的错误。

例：

```
union{
    char c[4];
}uu={ {'a','b','c'} };
```

- 能在位字段中记述 enum。

例：

```
struct{
    enum E1{a,b,c} m1:2;
    enum E1         m2:2;
};
```

- 对枚举符的最后的","，抑制错误输出。

例：

```
enum E1{a,b,c,}m1;
```

- 能同时赋值和声明联合体。

例：

```
union U{
    int a,b;
}u1;
void test(){
    union U u2 = u1;
}
```

- 放宽了在 C 编译时对地址的数据类型转换的错误检查。
  在记述对地址的数据类型转换时，必须指定 C 编译（lang=c 选项）。

例：

```
          int x;
          short addr1=(short)&x;
```

- 放宽了 C 程序的函数/变量声明和#pragma 声明的出现顺序的限制。

例：

```
          void f(void);
          #pragma interrupt f
          void f(void){}        // 函数声明后的#pragma 声明有效。
                                // （在 Ver.3.x 时出错）
```

- 更改了 C++程序的函数/变量声明和#pragma 声明的出现顺序的规约。

例：

```
          void f(void){}
          #pragma interrupt f   // 函数定义后的#pragma 声明无效。
          void f(void);         // 与函数定义后的#pragma 声明有关的函数声明为错误。
```

- 作为 C++语言规格，也支持异常处理和摸板功能。

（2）Ver.4.0 → Ver.6.0 主要追加/改善的功能

（※ 不存在 Ver.5.0，缺号）

（a）新 CPU 的支持

支持了 CPU 种类为 H8SX 的目标文件的生成。

（b）2 个字节长度指针的支持（仅 H8SX）

通过指定_ _ptr16 关键字或者 ptr16 选项，能使用 2 个字节长度的指针。
在 H8SX 的高级模式和最高模式中有效。

（c）位字段排列顺序的指定

通过指定#pragma bit_order 或者 bit_order 选项，能指定对存储器的位字段成员的填写方法。

（d）扩展存储器间接方式的函数调用（仅 H8SX）

通过指定_ _indirect_ex 关键字或者 indirect=extended 选项，能声明作为扩展存储器间接调用的函数。另外，#pragma indirect section 不仅能转换用于存储器间接（@@aa:8）调用的函数地址区的＄INDIRECT 段，而且还能转换用于扩展存储器间接（@@aa:7）调用的函数地址区的＄EXINDIRECT 段的段名。

（e）汇编功能（仅 H8SX）

通过指定_ _asm 关键字，能在 C/C++源程序中记述汇编语言。

（f）#line 输出的抑制指定

通过指定 noline 选项，能抑制预处理器展开时的#line 输出。

（g）memcpy 函数、strcpy 函数的直接插入展开的指定（仅 H8SX）

通过指定 library 选项，能直接插入展开 memcpy 和 strcpy 二个库函数。

（h）错误级的变更

通过指定 change_message 选项，能对信息级和警告级信息进行个别错误级的更改。

（i）8 位绝对区的地址指定（仅 H8SX）

通过指定 sbr 选项，能指定 8 位绝对地址区的分配地址。

（j）优化功能的强化（仅 H8SX）

通过追加以下选项，能更加详细地指定优化的内容。

opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、volatile_loop

（k）内部函数的追加

追加了以下内部函数：

- H8SX 的 64 位乘法（mulsu、muluu）
- H8SX 的块传送指令（movmdb、movmdw、movmdl、movsd）
- 块传送指令的内部函数的强化（eepmovb、eepmovw、eepmovi）
- 修改了 MOVFPE 指令的内部函数（_movfpe）

（l）通配符的支持

能用通配符指定输入文件。

（m）编译程序极限值的变更

将 1 个文件中能记述的 switch 语句从 256 更改为 2048。

（n）信息显示的规格变更

在 Ver.4.0 中，如果在命令行指定了多个 message 或 nomessage 选项，最后指定的选项有效。在本版本中，抑制命令行内的各 nomessage 选项指定序号的并集的序号信息。

（o）枚举型数据的型

指定 byteenum 选项时，如果 enum 值的范围为 0～255，就将该 enum 声明的枚举型数据作为 unsigned char 型处理。

（p）直接插入展开

CPU 种类为 H8SX 时，speed=inline=<数值>选项的<数值>含义和其它 CPU 种类不同。为 H8SX 时，<数值>表示程序大小的增加率；其它 CPU 时，表示能直接插入展开的函数的结点最大数。

（q）1 个字节边界调整数据段和 4 个字节边界调整数据段（仅 H8SX）

如果指定 align=4 选项，就将长度为奇数的数据输出到 1 个字节边界调整的段或者将长度为 4 的倍数的数据输出到 4 个字节边界调整的段。

（r）段名

如果用 section 选项将 P、C、B、D 段更改为段名 S，就发生警告错误。S 为堆栈用的保留段名。

（s）隐式声明的追加

用#define 隐式声明__H8SXN__、__H8SXM__、__H8SXA__、__H8SXX__、__HAS_MULTIPLIER__、__HAS_DIVIDER__、__INTRINSIC_LIB__、__DATA_ADDRESS_SIZE__、__H8__、__RENESAS_VERSION__和__RENESAS__。

（t）可重入库的支持

如果对库生成工具指定 reent 选项，就生成可重入库。

（u）小端法空间的支持（仅 H8SX）

H8SX 根据产品支持了小端法空间。小端法空间的 2 或者 4 个字节的数据以数据长进行读写。为此，扩展了__evenaccess 关键字的功能。

（3）Ver.6.0 → Ver.6.01 主要追加/改善的功能

（a）AE5 的支持

支持了 AE5 的 CPU。

（b）ANSI 标准的对应扩展

通过 strict_ansi 选项，能将浮点运算的结合律以 ANSI 为基准输出。

（c）V4 码兼容目标的输出

在 H8S 的 CPU 中，为了输出与旧版本（Ver.4.0）兼容的目标，支持了 legacy=v4 选项。

（d）指定 legacy=v4 时的 cpuexpand=v6 规格扩展

在指定 legacy=v4 选项时，cpuexpand=v6 能输出符合 Ver.6.00 的 cpuexpand 输出规格的目标。

（e）register 存储类变量的优先分配

enable_register 选项优先将指定 register 存储类的变量分配到寄存器。

（f）优化范围的分割功能

scope/noscope 选项能选择是否在函数中分割优化范围。

（g）文件间的直接插入展开

支持了直接插入展开跨越文件的函数的 file_inline 选项和指定对象文件路径名的 file_inline_path 选项。

（h）内部函数的追加

支持了进行 VBR 设定的 set_vbr 内部函数。

（i）#pragma address

追加了能对变量指定的绝对地址进行分配的#pragma address 扩展指令。

（j）.stack 控制指令的输出

在指定 code=asmcode 时，能将.stack 控制指令输出到汇编源程序。

（k）环境变量的追加

支持了能设定 SBR 寄存器初始值的 CH38SBR 的环境变量。

（l）隐式声明的追加

追加了以下的隐式声明：
_ _AE5_ _
_ _ABS16_ _

（4）有关 Ver.6.01 优化功能的注意事项

有关优化的注意事项是以 Ver.6.01 生成 H8SX 和 H8S（没有指定 legacy=v4 选项）目标的情况为对象。对于其它情况的优化处理与 Ver.4.0 以前的版本相同。

Ver.6.01 的优化处理使用最新编译程序优化技术，实现了原版本（Ver.4.0）无法实现的指针或外部变量的别名分析、包含控制流程的数据生存区间的分析。因此，在语言规格允许的范围内进行了比 V4 更大范围的优化。

另一方面，原来没优化而能运行的程序有可能在优化后不能运行。

原来没有被优化而在 Ver.6.01 中成为优化对象的例子如下所示：

（a）没有 volatile 声明的外部变量或者指针变量的存取处理

因为在程序中的逐次处理以外的位置更改数据值，所以 volatile 声明必须保证在参照时存取实际分配数据的区域。例如：通过中断处理或者硬件处理更改数据值。

在编译程序处理中，对于没有 volatile 声明的变量，解释为不在程序中的逐次处理以外的位置和函数调用以外的位置更改数据值。

在 Ver.4.0 为止的版本中，对于下例中没有 volatile 声明的外部变量进行优化。

例：

```
int a;
f() {
    int *ptr=&a;
    *ptr=1;         // <------ 只删除此赋值表达式
    *ptr=2;
}
```

在 Ver.6.01 中，对以下情况进行了进一步的优化。
要抑制这些优化时，必须对相应的变量进行 volatile 声明。

例 1：

```
int a;
f() {
    int *ptr=&a;
    *ptr &= ~( (0x0080) );          // <-- (1)
    while( !( *ptr & (0x0080) ) )   // <-- (2)
    {
       :
    }
}
```

在此例中，优化结果：(2)的 while 语句进入无限循环。
- 根据指针的别名分析，将(1)的*ptr 和(2)的*ptr 作为同一数值处理。
- 将(1)式传到(2)。其结果，(2)式被转换为：

```
  while( !( (*ptr & ~( (0x0080) ))   & (0x0080) ) )  // <-- (2)
-> while(!(*ptr & 0))
-> while(!(0))
-> while(1)
```

因此，该表达式总是为真，并且判断语句被删除，上述的 while 语句进入无限循环。

例 2：

```
int a,b;
f() {
    a=1;        // <-- (1)
    if(a)       // <-- (2)
    {
        b=1;    // <-- (3)
    }
}
```

在此例中，优化结果：(2)的 if 语句判断式被删除，总是执行(3)。

- 根据外部变量的别名分析，将(1)和(2)的 a 作为同一数值处理。
- 将(1)的常数值传到(2)。其结果，(2)式被转换为：
  -> if(1)

因此，该表达式总是为真，并且判断语句被删除，总是执行上述(3)式。

例 3：

```
    int a,b,c;
    f() {
        a=1;            // <-- (1)
        if(c)           // <-- (2)
        {
            b=1;        // <-- (3)
        }
        a=2;            // <-- (4)
    }
```

在此例中，优化结果：(1)式被删除。
- 得到包含 if 语句控制表达式的控制流程。
- 根据控制流程和外部变量的别名分析，知道没有使用在(1)给 a 设定的值。

因此，上述(1)式作为没有被参照的冗余表达式，被删除。

例 4：

```
    int a;
    int b[10];
    f() {
        int i;                      // <-- (1)
        for(i=0; i<10; i++)         // <-- (2)
        {
            b[i]=a;                 // <-- (3)
        }
    }
```

在此例中，优化结果：(3)的 a 只在循环前被参照了一次，而在循环内总是作为定值处理。
- 得到包含 for 循环控制表达式的控制流程。
- 根据控制流程和外部变量的别名分析，(3)的 a 在循环内作为定值处理。
- 将(3)的 a 的参照表达式移动到(2)的 for 循环外。

```
        temp=a;
        for(i=0; i<10; i++)     // <-- (2)
        {
            b[i]=temp;          // <-- (3)
        }
```

因此，(3)式的变量 a 在循环中为定值。

例 5：

```
    int a;
    f() {
        a = 0;          // <-- (1)
        while (1);      // <-- (2)
    }
```

在此例中，优化结果： (1)的语句被视为不需要的语句而被删除。
- 因为(2)为无限循环，所以判断为本函数没有出口。
- 因为在无限循环内没有参照 a，所以(1)的设定被视为不需要的语句而被删除。

（b）volatile_loop 选项

在循环控制变量为非 volatile 外部变量并且为纯判断式时，volatile_loop 选项将循环控制变量作为 volatile 处理，抑制无限循环。但是，如果循环控制变量在循环内变化，就不作为 volatile 处理。

在 Ver.6.01 中，如有这种情况，必须对相应的变量进行 volatile 声明。

如下例所示：

例：

```
struct {
    unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
    while (ST.a) {       // <-- (1)
        if (a) {         // <-- (2)
            f();         // <-- (3)
        }
    }
}
```

在此例中，ST.a 有可能被 f()改写，被视为在循环内变化。因此，即使指定 volatile_loop 选项，也不对 ST 进行 volatile 处理。

- 如果(2)的条件成立，ST.a 的值有可能被(3)的 f()改写，所以在调用后再次装入 ST.a。
- 如果(2)的条件不成立，因为 ST.a 没有被改写，所以继续使用前一次(1)判断时使用的 ST.a 的值。

（5）有关 Ver.4.0 目标和 Ver.6.01 目标的兼容性

连接 Ver.4.0 目标和 Ver.6.01 目标时，需要满足以下条件：

（a） C 源程序

对以下函数接口有影响的选项必须相同：

- regparam
- longreg/nolongreg
- double=float
- structreg/nostructreg
- stack
- byteenum
- pack/unpack

（b）汇编程序

符合“**9.3.2 函数调用的接口**”的函数调用规则。

【注 1**】**

有关手册中没有记述的内容，不能保证版本升级后的兼容性。

在记述了取决于寄存器的保存/恢复顺序等编译程序的输出代码的汇编代码时，不能将 Ver.4.0 目标和 Ver.6.01 目标连接。

【注 2**】**

有关 OS 和中间件等的连接，请向销售部门询问。

18.2.3　汇编程序的追加/改善的功能

（1）Ver.4.0 主要追加/改善的功能

（a）BEQU 的外部定义/参照

能使用.BIMPORT 和.BEXPORT 进行.BEQU 符号的外部定义和参照。

（2）Ver.4.0 → Ver.6.0 主要追加/改善的功能

（※ 不存在 Ver.5.x，缺号）

（a）新 CPU 的支持

支持了 CPU 种类为 H8SX 的目标文件的生成。

（b）寄存器使用方法的检查强化

在 CPU 种类为 H8SX、H8S 或者 H8/300H 时，如果在程序中记述了@Rn+、@-Rn、@+Rn、@Rn-、@(d,Rn)或者@Rn，就发生以下警告：

```
    819  (W) @Rn+, @-Rn, @+Rn, @Rn-, @(d,Rn) OR @Rn USED
```

在这些寻址方式中，必须将 Rn 改写为 ERn。

（3）Ver.6.0 → Ver.6.01 主要追加/改善的功能

（a）新 CPU 的支持

支持了 AE5 的 CPU 种类。

（b）限制值的放宽

DEFINE 选项/控制指令的置换符号名长度：32 字符 →无限制
（但是，置换后的字符串长度为 255 个字符，无变更）

（c）DEFINE 选项/.DEFINE 控制指令的置换对象外指令

不能用 DEFINE 选项和.DEFINE 控制指令置换下述控制指令：
.AENDI、.AENDR、.AENDW、.AIFDEF、.END、.ENDM、.ENDF、.ENDI、.ENDS、.ENDW

（d）.STACK 控制指令的支持

对符号，支持了能指定使用堆栈量的控制指令。

18.2.4　优化连接编辑程序追加/改善的功能

（1）Ver.7.0、Ver.7.01 主要追加/改善的功能

（a）通配符的支持

能用输入文件或 start 选项的段名指定通配符。

（b）检索路径

能通过环境变量（HLNK_DIR）指定多个输入文件或者库文件的检索路径。

（c）装入模块的分割输出

能分割输出绝对装入模块文件。

（d）错误级的变更

能对信息、警告和错误级的信息个别进行错误级或者是否输出的更改。

（e）二进制和 HEX 的支持

能输入/输出二进制文件。另外，也能选择 Intel® HEX 型的输出。

（f）stack 使用量信息的输出

能通过 stack 选项输出用于栈解析工具的信息文件。

（g）optimize=variable_access 优化的改善

根据优化，还能将分配在 16 位绝对地址空间的变量移动到 8 位地址空间。

（h）optimize=register 优化的改善

没有指定 optimize=speed 选项时，在进行了函数间的寄存器保存/恢复的优化后，将多个寄存器的保存/恢复置换为函数调用，进行容量压缩。

（i）汇编程序优化的改善

还能优化包含.org、.align 和.data 控制指令的段。

（j）调试信息的删除功能

通过 strip 选项，能只删除装入模块文件或者库文件内的调试信息。

（k）调试信息的压缩功能

通过指定 compress 选项，能压缩调试信息。

（2）Ver.7.0 -> Ver.8.0 主要追加/改善的功能

（a）新 CPU 的支持

支持了 CPU 种类为 H8SX 的目标文件的输入。

（b）空区域的输出指定

通过指定 space 选项，能将指定值填入到空区域。

（c）存储器使用量的指定

通过指定 memory 选项，能指定内部存储器的使用量。

（d）8 位绝对区域的地址指定

通过指定 sbr 选项，能指定 8 位绝对区域的分配地址。

（e）段地址重复时的错误级变更

在连接时，段地址重叠时的错误级在 Ver.7 中更改为 Fatal，而在 Ver.8 中被更改为 Error。因此，即使在段地址重叠的情况下，也能通过指定 change_message 选项继续进行处理。

（3）Ver.8.0→ Ver.9.0 主要追加/改善的功能

（a）对 binary 选项输入段的边界调整数的指定

能对 binary 选项指定的段指定边界调整数。

（b）对照表的信息输出

通过指定 show=xreference 选项，能将对照表信息输出到连接列表。因此，可得知参照变量/函数的位置。

（c）没有被参照的符号的通知选项

通过指定 msg_unused 选项，即使在不使用优化的状况下也可得知没有被参照的符号的存在。

（d）编译单位段之间的空区域的削减功能

通过指定 data_stuff 选项，能紧凑编译单位中产生的空区域，进行数据的分配。

18.3　格式转换程序的操作方法

18.3.1　目标文件的格式

目标文件的格式符合标准格式的 ELF 格式。另外，调试信息的格式符合标准格式的 DWARF2 格式。

18.3.2　与旧版本的兼容性

（1）目标文件和库文件

将 Ver.3.x 及以前版本的编译程序或者汇编程序输出的目标文件和库文件输入到优化连接编辑程序时，必须使用格式转换程序，将目标文件和库文件转换为 ELF 格式。但是，在转换时调试信息被删除。

另外，不能转换连接编辑程序输出的可再定位文件（扩展名 rel）和含有可再定位文件的库文件。

格式转换程序用和输入文件相同的文件名输出转换目标格式的文件。输入文件保存为<输入文件名.扩展名>.bak。

不能将 ELF 格式的目标文件和库文件转换为旧版本（Ver.3.x 及以前版本）的编译程序或者汇编程序输出的目标格式。

（2）装入模块文件

能使用格式转换程序将 ELF 格式的装入模块文件转换为连接编辑程序 Ver.6.x 及以前版本的输出格式。可转换的目标文件格式如表 18.3 所示。不支持 H8SX 的 ELF 格式的装入模块文件。

表 18.3　可从 ELF 格式转换的目标文件格式

	编译程序、汇编程序版本	连接编辑程序指定选项		目标文件格式		可否转换
				目标	调试信息	
1	1.0.x	debug		SYSROF	SYSROF	○
2	2.0.x	sdebug		SYSROF	SYSROF	×
3	3.0.x	sysrof	debug	SYSROF	SYSROF	○
4			sdebug	SYSROF	DWARF1	×
5		elf	debug	ELF	DWARF1	○
6			sdebug	ELF	DWARF1	×

格式转换程序用和输入文件相同的文件名输出转换目标格式的文件。输入文件保存为<输入文件名.扩展名>.bak。

不能将连接编辑程序 Ver.6.x 及以前版本的输出装入模块文件转换为 ELF 格式。

如果使用由 Ver.4 版本以后的编译程序、汇编程序和优化连接编辑程序新追加的功能，就不能从 ELF 格式转换为旧格式。

18.3.3　选项指定规则

命令行的格式如下：

```
helfcnv [△<选项> … ]△<文件名>[…] [△<选项> …]
  <选项>: - <选项>[=<子选项>]
  <文件名>: 也能指定通配符(*,?)。
```

18.3.4　选项解说

在命令行格式的情况下，英文大写字母表示指定缩短型时的字符。下划线表示省略时的解释。在使用综合开发环境时，通过优化连接编辑程序的选项窗口指定。用选项卡名<种类名>[项目]…表示对应的对话菜单。

转换对象文件的种类（目标文件、库文件、装入模块）由格式转换程序自动判断。

（1）目标文件和库文件的转换

将 Ver.3.x 及以前版本的编译程序或者汇编程序建立的目标文件或者库文件转换为 ELF 格式。删除目标文件或者库文件中包含的调试信息。

因为综合开发环境不支持本功能，所以必须通过命令行使用。

表 18.4　目标文件、库文件转换的选项一览表

	项目	选项	指定内容
1	地址空间的指定	Address_space=<地址空间大小> <地址空间大小>:{ 20 | 24 | 28 | 32 }	地址空间的指定
2	fpu	Fpu	有 FPU *1
3	dsp	Dsp	有 DSP *1

*1　用于 SuperH 的选项。在 H8S、H8/300 系列中无效。

地址空间的指定

Address_space

—

格　式　Address_space=<地址空间大小>
<地址空间大小>:{ 20 | 24 | 28 | 32 }

说　明　指定 cpu=300ha、2000a、2600a 时的地址空间大小。
省略本选项时解释为 24。

例

```
helfcnv  -a=20 *.obj  ; 将目录内的全部*.obj 转换为地址空间大小为
                      ; 20 位的 elf 格式。
```

备　考　不能转换连接编辑程序输出的可再定位文件（扩展名 rel）和含有可再定位文件的库文件。

（2）装入模块文件的转换

将 ELF 格式装入模块转换为连接编辑程序的旧版本（Ver.6.x 及以前版本）输出的目标文件格式。装入模块文件中包含调试信息时，也转换调试信息。不能转换 H8SX 的装入模块。

表 18.5　装入模块文件转换的选项一览表

	项目	选项	对话菜单	指定内容
1	转换格式的指定	Sysrof Dwarf1	编译程序 <输出> [输出格式:] [绝对 (SYSROF)]	转换为 SYSROF 格式 转换为 ELF/DWARF1 格式

转换格式的指定

Sysrof
Dwarf1

编译程序<输出>[输出格式:][绝对(SYSROF)]

格　式　`Sysrof`

`Dwarf1`

说　明　指定转换后的目标格式。

在指定 `sysrof` 时，将 ELF/DWARF2 格式的装入模块文件转换为 SYSROF 格式。

在指定 `dwarf1` 时，将 ELF/DWARF2 格式的装入模块文件转换为 ELF/DWARF1 格式。

如果对优化连接编辑程序指定 sdebug 选项，转换后的文件就不包含调试信息。

例

```
helfcnv  test.abs                   ; 将 test.abs 转换为 SYSROF 格式
helfcnv  -d test.abs                ; 将 test.abs 转换为 ELF/DWARF1 格式
```

备　考　如果在同一文件中存在用#pragma option 指定有或者无优化的函数，转换后的文件就不包含调试信息。

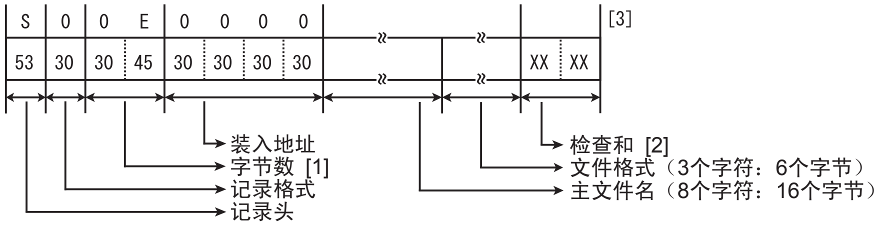
第 19 章　附录

19.1　S 型、HEX 文件格式

本节说明根据连接编辑程序输出的 S 型文件和 HEX 文件。

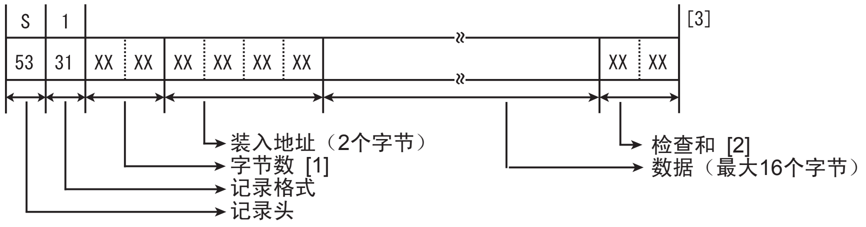
19.1.1　S 型文件格式

(a) 头记录（SO记录）

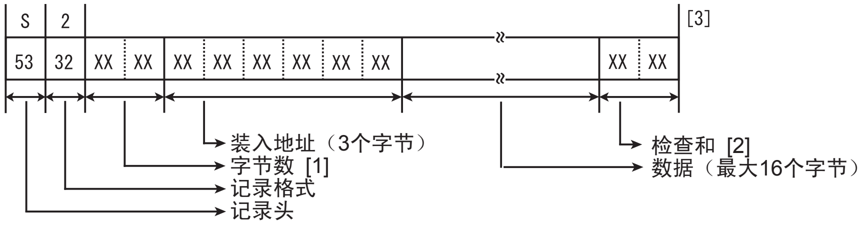


(b) 数据记录（S1、S2、S3记录）

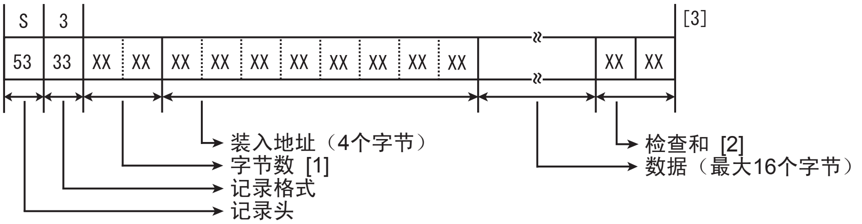
(i) 装入地址为0～FFFF时



(ii) 装入地址为10000～FFFFFF时

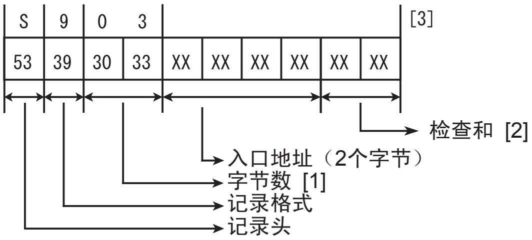


(iii) 装入地址为1000000～FFFFFFFF时

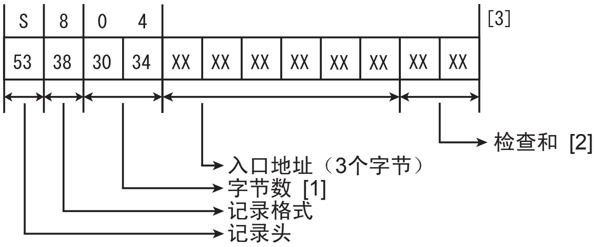


(c) 结束记录（S9、S8、S7记录）

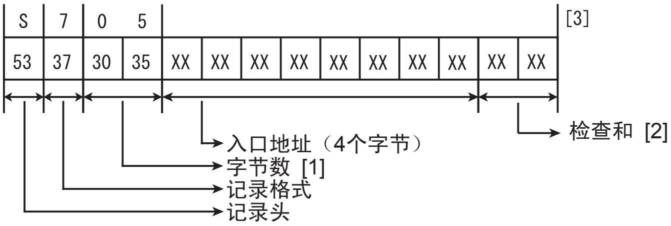
(i) 入口地址为0～FFFF时



(ii) 入口地址为10000～FFFFFF时



(iii) 入口地址为1000000～FFFFFFFF时



【注】 [1] 从装入地址（或者入口地址）到检查和的字节数
[2] 从字节数到检查和前的数据值的总和（字节单位）的1的补数
[3] 在检查和的后面加换行码

19.1.2　HEX 文件格式

各数据记录的执行地址按以下方法计算：

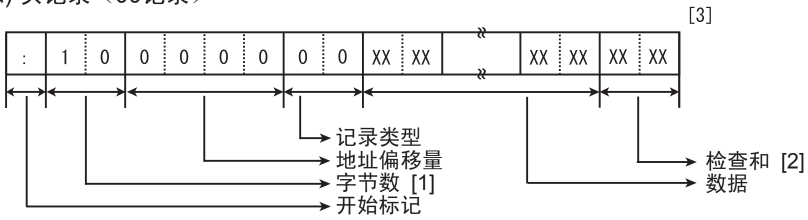
（1）段地址

（段基地址 << 4）+（数据记录的地址偏移量）

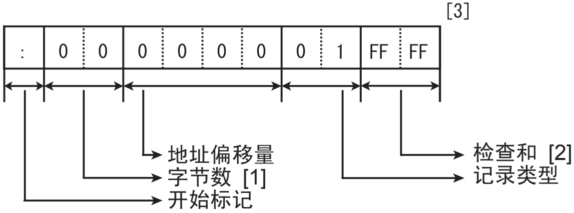
（2）线性地址

（线性基地址 << 16）+（数据记录的地址偏移量）

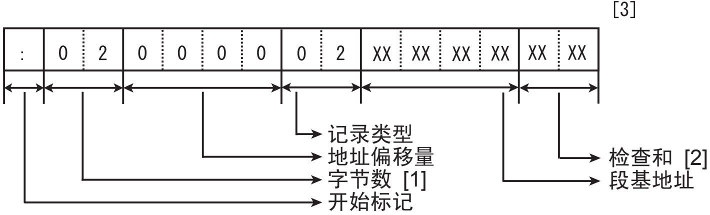
(a) 头记录（00记录）



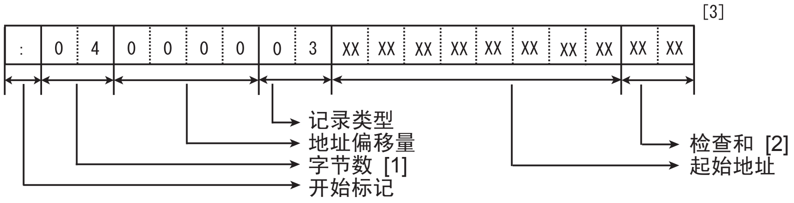
(b) 结束记录（01记录）



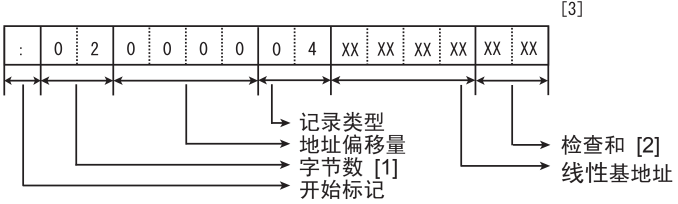
(c) 扩展段地址记录（02记录）



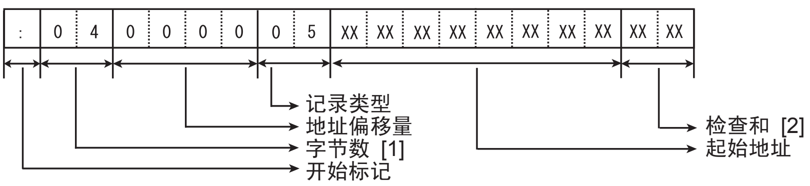
(d) 起始地址记录（03记录）



(e) 扩展线性地址记录（04记录）



(f) 32位起始线性地址记录（05记录）



【注】 [1]　从记录类型到检查和前的字节数
[2]　从字节数到检查和前的数据值的总和（16进制数）的2的补数
[3]　在检查和的后面加换行码

19.2　ASCII码一览表

表19.1　ASCII码一览表

低4位	高4位							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	＃	3	C	S	c	s
4	EOT	DC4	＄	4	D	T	d	t
5	ENQ	NAK	％	5	E	U	e	u
6	ACK	SYN	＆	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	（	8	H	X	h	x
9	HT	EM	）	9	I	Y	i	y
A	LF	SUB	*	：	J	Z	j	z
B	VT	ESC	＋	；	K	[	k	｛
C	FF	FS	，	＜	L	＼	l	｜
D	CR	GS	－	＝	M	]	m	｝
E	SO	RS	．	＞	N	^	n	～
F	SI	US	／	？	O	_	o	DEL

19.3　短绝对地址的存取范围

各 CPU/运行模式中的 8 位绝对地址和 16 位绝对地址的存取范围如表 19.2 所示：

表 19.2　短绝对地址的存取范围

CPU/运行模式	8 位绝对地址 （@aa:8）的存取范围	16 位绝对地址 （@aa:16）的存取范围
H8SXA:32 H8SXX [:32] 2600a:32 2000a:32	0xFFFFFF00～0xFFFFFFFF	0x0～0x7FFF、 0xFFFF8000～0xFFFFFFFF
H8SXA:28 H8SXX:28 2600a:28 2000a:28	0xFFFFF00～0xFFFFFFF	0x0～0x7FFF、 0xFFF8000～0xFFFFFFF
H8SXA[:24] H8SXM[:24] 2600a[:24] 2000a[:24] 300ha[:24]	0xFFFF00～0xFFFFFF	0x0～0x7FFF、 0xFF8000～0xFFFFFF
H8SXA:20 H8SXM:20 2600a:20 2000a:20 300ha:20	0xFFF00～0xFFFFF	0x0～0x7FFF、 0xF8000～0xFFFFF
H8SXN 2600n 2000n 300hn 300 300l	0xFF00～0xFFFF	–

【注】 在CPU种类为H8SX时，可根据指定 sbr选项更改8位绝对地址区。

索　引

符号

#pragma abs16 250
#pragma abs16 section 249
#pragma abs8 250
#pragma abs8 section 249
#pragma address 276
#pragma asm 267
#pragma bit_order 253
#pragma entry 258
#pragma global_register 271
#pragma indirect 260
#pragma indirect section 249
#pragma inline 262
#pragma inline_asm 263
#pragma interrupt 254
#pragma noregsave 264
#pragma option 266
#pragma pack 1 272
#pragma pack 2 272
#pragma regsave 264
#pragma section 249
#pragma stacksize 248
#pragma unpack 272
$1 155
$4 155
$ABS16B 28, 154, 155
$ABS16C 28, 154, 155
$ABS16D 28, 154, 155
$ABS8B 28, 154
$ABS8C 28, 154
$ABS8D 28, 154
$ADDRESS 155
$EXINDIRECT 154
$INDIRECT 154
$VECT 154
.ABS8 552
.AELIF 578
.AELSE 578, 579
.AENDI 578, 579
.AENDR 580
.AENDW 581
.AERROR 583

.AIF....578
.AIFDEF....579
.ALIGN....532
.ALIMIT....583
.AREPEAT....580
.ASSIGN....534
.ASSIGNA....575
.ASSIGNC....576
.AWHILE....581
.BEQU....536
.BEXPORT....550
.BIMPORT....551
.BREAK....602, 612
.CASE....602
.CONTINUE....613
.CPU....524
.DATA....537
.DATAB....538
.DEBUG....554
.DEFINE....577
.DISPSIZE....556
.ELSE....600
.END....566
.ENDF....605
.ENDI....600
.ENDM....587
.ENDW....608
.EQU....533
.EXITM....582
.EXPORT....547
.FOR[U]....605
.FORM....560
.GLOBAL....549
.HEADING....561
.IF....600
.IMPORT....548
.INCLUDE....568
.INSTR....595
.LEN....594
.LINE....555
.LIST....559
.MACRO....587
.NOABS8....552
.ORG....530
.OTHERS....602
.OUTPUT....553
.PAGE....562
.PRINT....558

.PROGRAM 564
.RADIX 565
.REG 535
.REPEAT 610
.RES 543
.SBR 526
.SDATA 539
.SDATAB 540
.SDATAC 541
.SDATAZ 542
.SECTION 527
.SPACE 563
.SRES 544
.SRESC 545
.SRESZ 546
.STACK 566
.SUBSTR 596
.SWITCH 602
.UNTIL 610
.WHILE 608
_ _abs16 ....28, 250
_ _abs8 ....28, 250
_ _asm 268
_ _entry 258
_ _evenaccess 274
_ _global_register 271
_ _indirect 260
_ _indirect_ex 261
_ _inline 262
_ _interrupt 254
_ _near16 251
_ _near8 251
_ _noregsave 264
_ _ptr16 252
_ _regparam2 265
_ _regparam3 265
_ _regsave 264
_ _secend 279
_ _sectop 279
_CALL_END 170
_CALL_INIT 170, 175
_CLOSEALL 171
_ec2p_new_handler 460
_file_Ptr 458
_INITLIB 171, 175
_INITSCT 170
_IOFBF 361
_IOLBF 361

_IONBF 361

英文

A
abort..80, 171, 198, 738, 739
abs...407, 468, 475
abs16...28, 72
abs 8...28, 72
absolute...91
absolute_forbid...104
acos..330
acosf...344
add 739
address_space 754
align 17
ALIGN..527
align_section..738
all 16
ALL 556
allocation 21
and_ccr 285
and_exr 288
arg...468, 475
asin...330
asinf 344
asmcode 10
assert..314
assert.h...314
assigna 58
assignc 59
atan 331
atan2 332
atan2f...346
atanf 345
atexit 171, 196
atof...399
atoi 399
atol 400
auto 26
auto 16

B
B 154
binary..88, 91
bit_order 51
boolalpha 433
br_relative..64

branch 101
bsearc......406
bss......11
BTBL......170, 173
BUFSIZ 361
byteenum 39

C

C 154, 230
C$BSEC 155
C$DSEC 155
C$INIT 155
C$VTBL......155
cachesize......104
calloc......404
calls......69
case 26
ceil 340
ceilf......354
CH38......127
CH38SBR......128
CH38TMP 128
change_message 41, 113
CHAR_BIT......326
CHAR_MAX......326
CHAR_MIN 326
check_section......738
chgincpath......735
clearerr......395
close......181
cmncode......40
cnvs......737
code......10, 69, 153
CODE 527
columns......83
comment 37
complex 423, 462
compress......110
conditionals......69
conj 468, 475
const......11, 12
const_var_propagate......34
cos......333, 468, 475
cosf 347
cosh......334, 468, 475
coshf 348
cpp 52
cpu 45, 74, 107, 732, 728, 738

cpucheck 738
cpuexpand...13
cross_reference 70
ctype.h 315

D
D 154
dadd 305
data 11, 12, 153
DATA 527
DBL_DIG 324
DBL_EXP_DIG..324
DBL_MANT_DIG 324
DBL_MAX 323
DBL_MAX_10_EXP 323
DBL_MAX_EXP 323
DBL_MIN 323
DBL_MIN_10_EXP 324
DBL_MIN_EXP 323
DBL_NEG_EPS 324
DBL_NEG_EPS_EXP...325
DBL_POS_EPS 324
DBL_POS_EPS_EXP..324
debug 11, 61, 93, 738
dec..435
defbool.h 739
define 7, 57, 88, 738
definitions 69
del_vacant_loop 31
delete..111, 738, 739
directory 738, 739
div 408
div_t 398
double_complex
    _re 470
    _im...470
    double_complex 470
    imag 470
    operator= 471
    operator+= 471
    operator-= 471
    operator*= 471
    operator/= 471
    real 470
    value_type 470
double=float..48
dsub..306
DTBL...170, 173

DUMMY 527
dwarf1....755
DWARF1....753
DWARF2....753

E
EBADF....327, 688
ECBASE....327, 687
echo....738
ecpp....37
ECSPEC....327, 688
EDBLO....327
EDBLU....327
EDIV....327, 687
EDOM....327, 328, 342, 687
eepmov....40
EEPMOV/P.W....728, 732
eepmovb....296
eepmovi....297
eepmovw....296
EEPROM....728, 732
eepromb....730
eepromb_epr....731
eepromw....730
eepromw_epr....731
EEPROM....728, 732
EEXP....327, 687
EEXPN....327, 687
EFLOATO....327, 687
EFLOATU....327, 687
ELDBLO....327
ELDBLU....327
elf....738
ELF....753
enable_register....42
end....118, 455
endl....455
ends....455
entry....89
EOF....309, 361
EOVER....687
ERANGE....327, 328, 342, 687
errno....313, 327
errno.h....307, 327
errno_addr....183
error....41, 80, 113
ESTRN....327, 687
ETLN....327, 687

euc....................53, 82
EUNDER....................687
exception....................49
exchange....................738
exclude....................73, 738
exit....................118, 171, 197, 738, 739
exp....................336, 469, 476
expand....................61
expansion....................21
expansions....................69
expf....................350
expression....................25
extract....................112

F
fabs....................340
fabsf....................354
FBR....................556
fclose....................365
feof....................395
ferror....................396
fflush....................365
fgetc....................383
fgets....................384
FILE....................361
file_inline....................35
file_inline_path....................8
fixed....................435
float.h....................307, 323
float_complex
  _re....................462
  _im....................462
  imag....................463
  operator=....................463
  operator+=....................463
  operator-=....................463
  operator*=....................463
  operator/=....................463
  real....................463
  value_type....................462
floor....................341
floorf....................355
FLT_DIG....................324
FLT_EXP_DIG....................324
FLT_GUARD....................323
FLT_MANT_DIG....................324
FLT_MAX....................323
FLT_MAX_10_EXP....................323

FLT_MAX_EXP 323
FLT_MIN 323
FLT_MIN_10_EXP 324
FLT_MIN_EXP 323
FLT_NEG_EPS 324
FLT_NEG_EPS_EXP 325
FLT_NORMALIZE 323
FLT_POS_EPS 324
FLT_POS_EPS_EXP 324
FLT_RADIX 323
FLT_ROUNDS 323
flush 455
fmod 341
fmodf 355
fopen 366
form 91, 738
fprintf 370
fputc 385
fputs 386
fread 391
free 404
freopen 367
frexp 336
frexpf 350
FRG 556
fscanf 375
fseek 393
fsymbol 106
ftell 394
function_call 101
function_forbid 104
FWD 556
fwrite 392

G
get_ccr 284
get_exr 288
get_imask_ccr 283
get_imask_exr 287
getc 387
getchar 387
getline 492
gets 388
global_alloc 33
goptimize 24, 64

H
H16 93

H20 93
H32 93
H38CPU 127
H8/300 525
H8/300L 525
H8SXA 45, 524
H8SXM 45, 524
H8SXN 45, 524
H8SXX 45, 524
head 120
hex 435
hexadecimal 91
hide 114
HIGH 507
HLNK_DIR 128
HLNK_LIBRARY1 128
HLNK_LIBRARY2 128
HLNK_LIBRARY3 128
HLNK_TMP 128
HUGE_VAL 328, 342
HWORD 507

I
ifthen 26
imag 468, 475
include 6, 56
indirect 27
infinite_loop 32
information 41, 113, 738
inline 25
input 87
INT_MAX 326
INT_MIN 326
int_type 424
internal 434
iomanip 423, 424
ios 423, 424, 430
    ~ios 430
    bad 431
    clear 431
    copyfmt 432
    eof 431
    fail 431
    good 431
    init 430
    operator void* 431
    operator! 431
    rdstate 431

  rdbuf 432
  sb430
  state....................430
  tie 432
  tiestr 430
ios_base
  _ec2p_init_base 428
  _ec2p_copy_base....................428
  _fmtmask 427
  _statemask 427
  ~ios_base 428
  adjustfield 427
  app 427
  ate 427
  badbit 427
  basefield....................427
  beg 428
  binary....................427
  boolalpha 427
  cur 428
  dec 427
  end 428
  eofbit....................427
  failbit 427
  fill 428, 429
  fillch 426
  fixed....................427
  flags 428
  floatfield 427
  fmtf 426
  fmtflags....................426, 427
  goodbit....................427
  hex 427
  in 427
  Init
    Init 425
  Init
    ~Init 425
  init_cnt....................425
  internal....................427
  ios_base 428
  iostate....................426, 427
  left....................427
  oct 427
  openmode 426, 427
  out....................427
  prec 426
  precision 429

  - right 427
  - scientific 427
  - seekdir 426, 428
  - setf 428
  - skipws 427
  - showbase 427
  - showpoint 427
  - showpos 427
  - trunc 427
  - unitbuf 427
  - unsetf 428
  - uppercase 427
  - wide 426
  - width 429
- iostream 423
- isalnum 316
- isalpha 317
- iscntr 317
- isdigit 318
- isgraph 318
- islower 319
- isprint 319
- ispunct 320
- isspace 320
- istream 423, 424
  - _ec2p_getistr 444
  - ~istream 445
  - chcoun 443
  - gcount 445
  - get 445
  - getline 446
  - ignore 446
  - istream 444
  - operator>> 445
  - peek 446
  - putback 447
  - read 447
  - readsome 447
  - seekg 447, 448
  - sentry
    - ~sentry 442
    - operator bool 442
    - sentry 442
  - sentry ok_ 442
  - sync 447
  - tellg 447
  - unget 447
- isupper 321

isxdigit 321

J
jmp_buf....356

L
L_tmpnam....361
labs....408
lang 52
large 48
latin1 ....53, 81
lbr....737
LDBL_DIG....324
LDBL_EXP_DIG 324
LDBL_MANT_DIG 324
LDBL_MAX 323
LDBL_MAX_10_EXP 323
LDBL_MAX_EXP 323
LDBL_MIN 323
LDBL_MIN_10_EXP....324
LDBL_MIN_EXP 324
LDBL_NEG_EPS....324
LDBL_NEG_EPS_EXP 325
LDBL_POS_EPS....324
LDBL_POS_EPS_EXP 324
ldexp 337
ldexpf....351
ldiv 409
ldiv_t....398
left....51, 434
legacy....19
length 21
library 34, 87, 91, 738
limits.h 307, 326
lines....83
list 20, 67, 98, 738, 739
lnk 737
LOCATE 527
log....337, 469, 476
log10 ....338, 352, 469, 476
log10f....352
logf....351
logo 53, 84, 117
LONG_MAX....326
LONG_MIN 326
longjmp....357
longreg....47
loop 25

LOW 507
lseek....182
LWORD 507

M
mac 291
machinecode 10
macl 291
macsave 38
malloc 405
map 95
math.h 328
mathf.h....342
max_unroll....32
medium 48
memchr 416
memcmp 414
memcpy 412
memmove 422
memory....110
memset....421
message....7, 96
mlist 738
modf....338
modff 352
movfpe....294
movmdb 298
movmdl....298
movmdw 298
movsd 299
movtpe 295
msg_unused 96
mulsu 292
muluu....292

N
new 423, 460
new_handler....460
no_float.h....397
noalign 17
noallocation 21
noboolalpha 433
nocmncode....266
nocompress 110
nocpuexpand....13
nocross_reference 70
nodebug 11, 61, 93, 738
noecho....738

noexception..49
noexclude...73, 738
noexpansion...21
nolibrary 738
noline 41
nolist 20, 67
nologo 53, 84, 117
nolongreg 47
noloop 266
nomacsave 266
nomessage 7, 96
none 16
noobject 15, 21, 65
nooptimize 62, 101
nooutput 738
nop 299
noprelink 89
noprint..738
noregexpansion...39
noregister 266
norm...468, 475
noscope 35
nosection 71
noshow..68
noshowbase..433
noshowpoint...434
noshowpos 434
noskipws 434
nosource 21, 68
nostatistics 21
nostructreg 47
NOTOPN 327, 688
noudf..738
nouppercase 434
novolatile 29
NULL 310, 313

O
object 15, 21, 65, 91
oct 435
off_type..424
offsetof 313
onexit_t 398
open 180
operator..445
operator- 467, 474
operator delete 461
operator delete [] 461

operator new 460, 461
operator new [] 460
operator!= 467, 474, 491
operator* 467, 474
operator/ 467, 474
operator+ 467, 474, 491
operator< 491
operator<< 456, 468, 475, 492
operator<= 491
operator= = ...467, 474, 491
operator> 491
operator>= 492
operator>> 450, 468, 475, 492
opt_range 30
optimize 24, 62, 101
optlnk38 737
or_ccr 285
or_exr 289
ostream 423, 424
  ~ostream 453
  flush 453
  operator<< 453
  ostream 453
  put 453
  seekp 454
  sentry
    ~sentry 451
    ok_ 451
    operator bool 451
    sentry 451
  tellp 453
  write 453
outcode 54, 82
output 94, 120, 738, 739
ovfaddc 300
ovfaddl 300
ovfadduc 300
ovfaddul 300
ovfadduw 300
ovfaddw 300
ovfnegc 304
ovfnegl 304
ovfnegw 304
ovfshalc 302
ovfshall 302
ovfshalw 302
ovfshllu 303
ovfshllul 303

ovfshlluw 303
ovfsubc 301
ovfsubl 301
ovfsubuc 301
ovfsubul 301
ovfsubuw 301
ovfsubw 301

P
P 154
pack 50
path 127
perror 396
polar 468, 475
pos_type 424
pow 339, 469, 476
PowerON_Reset 170, 172
powf 353
preinclude 6
preprocessor 10
print 738
printf 378
profile 103
program 11
ptr16 27
ptrdiff_t 313
PTRERR 327, 687
putc 388
putchar 389
puts 389

Q
qsort 407

R
rand 403
RAND_MAX 398
read 181
real 468, 475
realloc 405
record 93
reent 121
reference 99
regexpansion 39
register 25, 101
regparam 46
relocate 91
rename 111, 738

replace....112
resetiosflags....457
rewind....394
right....51, 434
rom....94, 97, 738
rotlc....292
rotll....292
rotlw....292
rotrc....293
rotrl....293
rotrw....293
rtti....49

S

S....155
S1....93
S2....93
S3....93
S9....109
safe....102
same_code....101
samecode_forbid....104
samesize....102
sbr....51, 116
sbrk....183
scanf....378
SCHAR_MAX....326
SCHAR_MIN....326
scientific....435
scope....35
sdebug....93
section....11, 71, 99
SEEK_CUR....361, 393
SEEK_END....361, 393
SEEK_SET....361, 393
set_ccr....284
set_exr....287
set_imask_ccr....283
set_imask_exr....286
set_new_handler....461
set_vbr....290
setbase....457
setbuf....368
setfill....457
setiosflags....457
setjmp....357
setjmp.h....356
setprecision....457

setstate 431
setvbuf 369
setw 457
shift 25
show 21, 68, 99
showbase 433
showpoint 433
showpos 434
SHRT_MAX 326
SHRT_MIN 326
signal_sem 184
sin 333, 469, 476
sinf 347
sinh 335, 469, 476
sinhf 349
size_t 313
SIZEOF 506
sjis 53, 81
skipws 434
sleep 294
slist 739
small 48
source 21, 68
sp 258
space 95
speed 25, 102
sprintf 379
sqrt 339, 469, 476
sqrtf 353
srand 403
sscanf 379
stack 48, 109, 153
STACK 527
start 27, 105, 738
STARTOF 506
static 16
statistics 21
stdarg.h 358
stddef.h 307, 313
stderr 310, 361
stdin 310, 361
stdio.h 361
stdlib.h 398
stdout 310, 361
strcat 413
strchr 416
strcmp 415
strcpy 412

strcspn 417
streambuf 423, 424
  _B_cnt_ptr 436
  _B_len_ptr 436
  ~streambuf 438
  B_beg_ptr 436
  B_next_ptr 436
  B_end_ptr 436
  B_beg_pptr 436
  B_next_pptr 436
  C_flg_ptr 436
  eback 439
  egptr 440
  eof 436
  epptr 440
  gbump 440
  gptr 439
  in_avail 438
  overflow 441
  pbackfail 441
  pbase 440
  pbump 440
  pptr 440
  pubseekoff 438
  pubseekpos 438
  pubsetbuf 438
  pubsync 438
  sbumpc 438
  seekoff 440
  seekpos 441
  setbuf 440
  setg 440
  setp 440
  sgetc 439
  sgetn 439
  showmanyc 441
  snextc 438
  sputbackc 439
  sputc 439
  sputn 439
  streambuf 438
  sungetc 439
  sync 441
  uflow 441
  underflow 441
  xsgetn 441
  xsputn 441
streamoff 424

streamsize 424
strerror 421
strict_ansi....42
string....12, 423, 477
  ~string....482
  append 484
  assign....485
  at 484
  begin....482
  c_str 486
  capacity....483
  clear 483
  compare 488
  compare 489
  const_iterator....477
  copy 486
  data 486
  empty....483
  end....483
  erase....485
  find 487
  find_first_not_of....488
  find_first_of....487
  find_last_not_of....488
  find_last_of....487
  insert....485
  iterator 477
  max_size....483
  npos 477
  operator= 482
  reserve 483
  resize....483
  operator[]....484
  operator+=....484
  replace 486
  s_ptr....477
  s_len 477
  s_res....477
  size....483
  string....482
  substr 488
  swap....486
string.h....410
string_unify....101
strip....113
strlen 422
strncat 414
strncmp 415

strncpy 413
strpbrk..........417
strrchr..........418
strspn..........418
strstr 419
strtod..........401
strtok..........420
strtol..........402
struct 25
struct_alloc 33
structreg 47
structured 69
stype..........91
subcommand..........54, 84, 115
swap..........492
switch..........25
symbol 99
symbol_delete..........101
symbol_forbid..........104
SYS_OPEN 361
sysrof 738, 755
SYSROF..........753
sysrofplus..........738

T
tab 21
table 26
tan 334, 469, 476
tanf..........348
tanh 335, 469, 476
tanhf..........349
tas..........295
template 16
TMP_MAX..........361
tolower..........322
toupper..........315, 322
trapa 293

U
UCHAR_MAX..........326
udf..........738
udfcheck 738
UINT_MAX 326
ULONG_MAX..........326
ungetc 390
uppercase 434
used..........16
USHRT_MAX..........326

V
va_arg 358, 359
va_end....................358, 360
va_list 358
va_start 358, 359
variable_access 101
variable_forbid....................104
VEC_TBL....................170, 171
vect 260
vfprintf 380
volatile 29
volatile_loop 38
vprintf 381
vsprintf....................382

W
wait_sem....................184
warning 41, 80, 113
width 21
write 182
ws....................449

X
XBR....................556
xor_ccr 286
xor_exr....................289
XRG....................556
XTN....................556

修订记录	H8S、H8/300 系列 C/C++编译程序、 汇编程序、优化连接编辑程序 用户手册

Rev.	发行日	修订内容	
		页	修订处
1.00	2005.10.28	—	初版发行

瑞萨单片机开发环境系统
用户手册
H8S、H8/300系列C/C++编译程序、汇编程序、优化连接编辑程序

Publication Date: 1st Edition, October, 2005
Rev.1.00, October 28, 2005
Published by: Sales Strategic Planning Div.
Renesas Technology Corp.
Edited by: Customer Support Department
Global Strategic Communication Div.
Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, United Kingdom
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit2607 Ruijing Building, No.205 Maoming Road (S), Shanghai 200020, China
Tel: <86> (21) 6472-1001, Fax: <86> (21) 6415-2952

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> 2-796-3115, Fax: <82> 2-796-2145

Renesas Technology Malaysia Sdn. Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

Colophon 4.0

H8S、 H8/300 系列 C/C++编译程序、
汇编程序、优化连接编辑程序

瑞萨电子株式会社