

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.

SuperH™ RISC engine C/C++ 编译程序、 汇编程序、优化连接编辑程序 编译程序包 V.9.01

瑞萨单片机开发环境系统

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, 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 products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (<http://www.renesas.com>)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
 - (1) artificial life support devices or systems
 - (2) surgical implantations
 - (3) healthcare intervention (e.g., excision, administration of medication, etc.)
 - (4) any other purposes that pose a direct threat to human lifeRenesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas 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 applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.

注意

本文只是参考译文，前页所载英文版“Cautions”具有正式效力。

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本公司产品的参考资料，对于本资料中所记载的技术信息，并非意味着对本公司或者第三者的知识产权及其他权利做出保证或对实施权力进行的承诺。
2. 对于因使用本资料所记载的产品数据、图、表、程序、算法及其他应用电路例而引起的损害或者对第三者的知识产权及其他权利造成侵犯，本公司不承担任何责任。
3. 不能将本资料所记载的产品和技术用于大规模破坏性武器的开发等目的、军事目的或其他的军需用途方面。另外，在出口时必须遵守日本的《外汇及外国贸易法》及其他出口的相关法令并履行这些法令中规定的必要手续。
4. 本资料所记载的产品数据、图、表、程序、算法以及其他应用电路例等所有信息均为本资料发行时的内容，本公司有可能在未做事先通知的情况下，对本资料所记载的产品或者产品规格进行更改。所以在购买和使用本公司的半导体产品之前，请事先向本公司的营业窗口确认最新的信息并经常留意本公司通过公司主页（<http://www.renesas.com>）等公开的最新信息。
5. 对于本资料中所记载的信息，制作时我们尽力保证出版时的精确性，但不承担因本资料的叙述不当而致使顾客遭受损失等的任何相关责任。
6. 在使用本资料所记载的产品数据、图、表等所示的技术内容、程序、算法及其他应用电路例时，不仅要对所使用的技术信息进行单独评价，还要对整个系统进行充分的评价。请顾客自行负责，进行是否适用的判断。本公司对于是否适用不负任何责任。
7. 本资料中所记载的产品并非针对万一出现故障或是错误运行就会威胁到人的生命或给人体带来危害的机器、系统（如各种安全装置或者运输交通用的、医疗、燃烧控制、航天器械、核能、海底中继用的机器和系统等）而设计和制造的，特别是对于品质和可靠性要求极高的机器和系统等（将本公司指定用于汽车方面的产品用于汽车时除外）。如果要用于上述的目的，请务必事先向本公司的营业窗口咨询。另外，对于用于上述目的而造成的损失等，本公司概不负责。
8. 除上述第7项内容外，不能将本资料中记载的产品用于以下用途。如果用于以下用途而造成的损失，本公司概不负责。
 - 1) 生命维持装置。
 - 2) 植埋于人体使用的装置。
 - 3) 用于治疗（切除患部、给药等）的装置。
 - 4) 其他直接影响到人的生命的装置。
9. 在使用本资料所记载的产品时，对于最大额定值、工作电源电压的范围、放热特性、安装条件及其他条件请在本公司规定的保证范围内使用。如果超出了本公司规定的保证范围使用时，对于由此而造成的故障和出现的事故，本公司将不承担任何责任。
10. 本公司一直致力于提高产品的质量和可靠性，但一般来说，半导体产品总会以一定的概率发生故障、或者由于使用条件不同而出现错误运行等。为了避免因本公司的产品发生故障或者错误运行而导致人身事故和火灾或造成社会性的损失，希望客户能自行负责进行冗余设计、采取延烧对策及进行防止错误运行等的安全设计（包括硬件和软件两方面的设计）以及老化处理等，这是作为机器和系统的出厂保证。特别是单片机的软件，由于单独进行验证很困难，所以要求在顾客制造的最终的机器及系统上进行安全检验工作。
11. 如果把本资料所记载的产品从其载体设备上卸下，有可能造成婴儿误吞的危险。顾客在将本公司产品安装到顾客的设备上时，请顾客自行负责将本公司产品设置为不容易剥落的安全设计。如果从顾客的设备上剥落而造成事故时，本公司将不承担任何责任。
12. 在未得到本公司的事先书面认可时，不可将本资料的一部分或者全部转载或者复制。
13. 如果需要了解关于本资料的详细内容，或者有其他关心的问题，请向本公司的营业窗口咨询。

前 言

本手册叙述了 "SuperH RISC engine 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 语言、C++ 语言、DSP-C 语言*1 以及汇编语言记述的源程序转换成 SuperH RISC engine 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*2、PC-9801*3 系列、IBM PC*4 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*5。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 DSP-C 语言是荷兰 ACE 公司（Associated Compiler Experts）在对安装 DSP 编译程序所需的语言扩展进行了探讨的基础上，于 1998 年向 ISO 标准委员会提案的规格。

*2 UNIX 是 X/Open 公司在美国及其他国家的注册商标。

*3 PC-9801 是日本电气株式会社的商标。

*4 IBM PC 是美国 International Business Machines Corporation 的注册商标。

*5 Microsoft®、Windows®、Windows NT® 是美美国 Microsoft Corporation 在美国及其他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	4
2.1	选项的指定规则	4
2.2	选项解说	4
2.2.1	源选项	4
2.2.2	目标选项	8
2.2.3	列表选项	20
2.2.4	优化选项	22
2.2.5	其他选项	40
2.2.6	单片机选项	53
2.2.7	其余选项	60
第 3 章	汇编程序的操作方法	63
3.1	选项的指定规则	63
3.2	选项解说	63
3.2.1	源选项	63
3.2.2	目标选项	67
3.2.3	列表选项	70
3.2.4	其他选项	75
3.2.5	单片机选项	81
3.2.6	剩余选项	84
第 4 章	优化连接编辑程序的操作方法	89
4.1	选项的指定规则	89
4.1.1	命令行的格式	89
4.1.2	子命令文件的格式	89
4.2	选项解说	90
4.2.1	输入选项	90
4.2.2	输出选项	94
4.2.3	列表选项	103
4.2.4	优化选项	105
4.2.5	段选项	110
4.2.6	验证选项	113
4.2.7	其他选项	115
4.2.8	子命令文件选项	122
4.2.9	单片机选项	123
4.2.10	剩余选项	124
第 5 章	标准库生成工具的操作方法	126
5.1	选项指定规则	126
5.1.1	命令行格式	126

5.2	选项解说	126
5.2.1	追加选项	126
5.2.2	不可指定的选项	129
5.2.3	指定选项时的注意事项	130
第 6 章	栈解析工具的操作方法	131
6.1	栈解析工具的启动	131
第 7 章	环境变量	132
7.1	环境变量一览表	132
7.2	预定义宏	135
第 8 章	文件规格	136
8.1	文件名的命名方法	136
8.2	编译列表的参照方法	137
8.2.1	编译列表的结构	137
8.2.2	源列表信息	137
8.2.3	目标信息	139
8.2.4	统计信息	140
8.2.5	命令指定的信息	140
8.3	汇编列表的参照方法	141
8.3.1	汇编列表的结构	141
8.3.2	源列表信息	141
8.3.3	对照表	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	变量存取优化对象的符号信息	147
8.4.8	函数存取优化对象的符号信息	147
8.4.9	对照表信息	148
8.5	库列表的参照方法	149
8.5.1	库列表的结构	149
8.5.2	选项信息	149
8.5.3	错误信息	150
8.5.4	库信息	150
8.5.5	库内模块、段、符号信息	151
第 9 章	编程	152
9.1	程序结构	152
9.1.1	段	152
9.1.2	C/C++ 程序的段	152
9.1.3	汇编程序的段	155
9.1.4	段的连接	156
9.2	初始设定程序的建立	159
9.2.1	存储区的分配	159
9.2.2	执行环境的设定	166

9.3	C/C++ 程序和汇编程序的连接	198
9.3.1	外部名的相互参照方法	198
9.3.2	函数调用的规则	200
9.3.3	参数分配的具体例子	206
9.3.4	寄存器和堆栈区的使用方法	208
9.4	程序建立时的注意事项	209
9.4.1	编程时的注意事项	209
9.4.2	用 C++ 编译程序对 C 程序进行编译时的注意事项	211
9.4.3	程序开发时的注意事项	212
第 10 章	C/C++ 语言规格	213
10.1	语言规格	213
10.1.1	编译程序的规格	213
10.1.2	数据的内部表示	218
10.1.3	浮点型的规格	227
10.1.4	运算符的评价顺序	232
10.2	DSP-C 规格	233
10.2.1	定点型	233
10.2.2	型限定词	233
10.2.3	常数	235
10.2.4	型转换	235
10.2.5	算术转换	236
10.2.6	指针转换	237
10.2.7	运算符	237
10.2.8	库	237
10.3	扩展功能	239
10.3.1	#pragma	239
10.3.2	段地址运算符	265
10.3.3	内部函数	266
10.4	C/C++ 库	320
10.4.1	标准 C 库	320
10.4.2	EC++ 类库	444
10.4.3	可重入库	511
10.4.4	不支持的库	513
10.4.5	DSP 库	514
第 11 章	汇编程序的语言规格	561
11.1	程序的元素	561
11.1.1	源语句	561
11.1.2	关键字	563
11.1.3	符号	563
11.1.4	常数	565
11.1.5	定位计数器	572
11.1.6	表达式	573
11.1.7	字符串	579
11.1.8	局部标号	579
11.2	执行指令	581
11.2.1	执行指令的概要	581
11.2.2	有关执行指令的注意事项	585
11.3	DSP 指令	605
11.3.1	程序的元素	605
11.3.2	DSP 指令	608

11.4	汇编程序控制指令	615
11.5	文件包含功能	660
11.6	条件汇编功能	662
11.6.1	条件汇编功能的概要	662
11.6.2	有关条件汇编功能的控制语句	666
11.7	宏功能	678
11.7.1	宏功能的概要	678
11.7.2	有关宏功能的控制语句	680
11.7.3	宏体	683
11.7.4	宏调用	686
11.7.5	字符串操作函数	688
11.8	文字池的自动生成功能	691
11.8.1	文字池的自动生成功能的概要	691
11.8.2	有关文字池自动生成功能的扩展指令	691
11.8.3	文字池自动生成功能的大小模式	692
11.8.4	文字池的输出	693
11.8.5	文字的共享	695
11.8.6	文字池输出的抑止	696
11.8.7	有关文字池自动生成的注意事项	697
11.9	重复循环指令的自动生成功能	699
11.9.1	重复循环指令自动生成功能的概要	699
11.9.2	有关重复循环指令的自动生成功能的扩展指令	700
11.9.3	REPEAT 的记述方法	700
11.9.4	编程例	701
11.9.5	有关 REPEAT 扩展指令的注意事项	703
11.10	扩展重复循环指令的自动生成功能	704
11.10.1	扩展重复循环命令的自动生成功能的概要	704
11.10.2	有关扩展重复循环指令的自动生成功能的扩展指令	704
11.10.3	EREPEAT 的记述方法	705
11.10.4	编程例	705
11.10.5	有关扩展 REPEAT 指令（EREPEAT 指令）的注意事项	706
第 12 章	编译程序的错误信息	708
12.1	错误格式和错误级	708
12.2	信息一览	708
12.3	C 标准库函数的错误信息	775
第 13 章	汇编程序的错误信息	779
13.1	错误格式和错误级	779
13.2	信息一览	779
第 14 章	优化连接编辑程序的错误信息	798
14.1	错误格式和错误级	798
14.2	信息一览	798
第 15 章	标准库生成工具 / 格式转换程序的错误信息	811
15.1	错误格式和错误级	811
15.2	信息一览	811
第 16 章	编译限制	813
16.1	编译程序的编译限制	813
16.2	汇编程序的汇编限制	814

第 17 章 版本升级的注意事项	815
17.1 版本升级时的注意事项	815
17.1.1 程序的运行保证	815
17.1.2 与旧版本的兼容性	816
17.1.3 与旧版本的目标文件的兼容性	817
17.1.4 命令行接口	818
17.1.5 提供的内容	819
17.1.6 列表文件规格	819
17.2 追加和改善的内容	820
17.2.1 共同的追加和改善（程序包 Ver.6）	820
17.2.2 编译程序的追加和改善	820
17.2.3 汇编程序的追加和改善的功能	825
17.2.4 优化连接编辑程序追加和改善的功能	825
17.3 格式转换程序的操作方法	828
17.3.1 目标文件的格式	828
17.3.2 与旧版本的兼容性	828
17.3.3 选项指定规则	828
17.3.4 选项解说	829
第 18 章 附录	832
18.1 Motorola S 格式、Intel HEX 格式文件	832
18.1.1 Motorola S 格式文件	832
18.1.2 Intel HEX 格式文件	834
18.2 ASCII 码一览表	836
索引	837

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。阴影部分表示本编辑程序包提供的软件。

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。

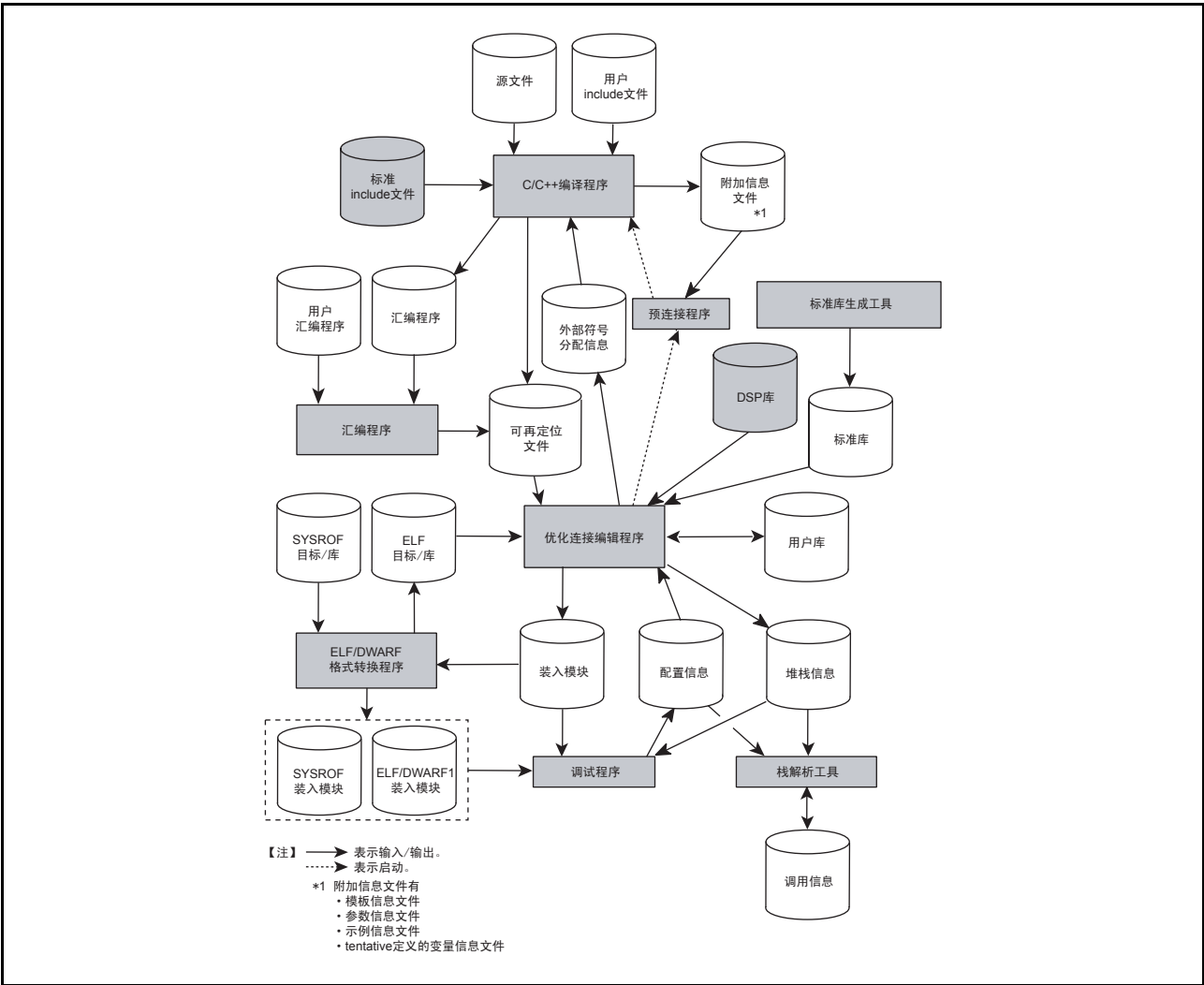


图 1.1 程序的开发步骤

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

“C/C++ 编译程序”以 C 语言或者 C++ 语言记述的源程序为输入，生成可再定位目标程序或者汇编源程序。

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. 支持优化功能，可提高目标程序的执行速度以及减小程序的 ROM/RAM 容量。
3. 作为程序描述语言，支持 C 语言和 C++ 语言。
4. 支持非 C/C++ 语言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。
5. 能指定由调试程序进行 C/C++ 源级调试的调试信息输出。
6. 能选择输出汇编源程序或者输出可再定位目标程序。
7. 能指定用于优化连接编辑程序进行优化的模块间的优化信息输出。

1.3 汇编程序的概要

“汇编程序”以汇编语言记述的源程序为输入，生成可再定位目标程序。

本汇编程序有以下特点：

1. 能通过以下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
2. 执行指令和汇编程序控制指令的助记符（名称）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”以编译程序和汇编程序输出的多个目标程序为输入，生成装入模块或者库文件。

本优化连接编辑程序有以下特点：

1. 对无法通过编译器进行的存储器配置和函数调用的优化，跨目标程序进行优化。
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
 - Motorola S 格式
 - Intel（扩展）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的列表。
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++ 程序的 `template`、`typeid`、`dynamic_cast` 功能时，启动编译程序，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++ 程序的 `template`、`typeid`、`dynamic_cast` 功能时，能通过指定优化连接编辑程序的 `noprelink` 选项抑止预连接程序的启动，提高连接速度。

1.6 标准库生成工具的概要

“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库群（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，有时也需要运行库。

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 C/C++ 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 选项的指定规则

启动编译程序的命令行的格式如下：

```
shc [ △ <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，带有下划线的字符表示默认值。
另外，用选项卡 <种类名> [项目] 等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。

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 文件的 文件夹	Include = <路径名>[,...]	编译程序 <源> [选项项目:] [include 文件的目录]	指定 include 文件的路径名
2 默认 include 文件	PREInclude = <文件名>[,...]	编译程序 <源> [选项项目:] [默认 include 文件]	在编译单位的起始位置包含指 定的文件
3 宏名的定义	DEFine = <sub>[,...] <sub> : <宏名> [= <字符串>]	编译程序 <源> [选项项目:] [宏定义]	<字符串> 定义为 <宏名>
4 信息消息	MESsage NOMESsage [= <错误号> [- <错误号>[,...]]	编译程序 <源> [选项项目:] [信息消息] [信息级消息的表示]	输出 不输出
5 文件间 inline 展开 的文件夹指定	FILE_INLINE_PATH= <路径名>[,...]	编译程序 <源> [选项项目:] [文件间 inline 展开的目录]	指定文件间 inline 展开文件的 路径名
6 消息级	CHAnge_message =<sub>[,...] <sub>:<level> [=<n>[-m],...] <level>:{Information Warning Error }	编译程序 <源> [消息级:]	更改消息级

include 文件的文件夹

<i>Include</i>	
编译程序 <源>[选项项目 :][include 文件的目录]	
格 式	Include = <路径名>[, ...]
说 明	指定 include 文件的路径名。 当有多个路径名时，能用逗号 (,) 分开指定。 按照 include 选项的指定文件夹、环境变量 SHC_INC 的指定文件夹和环境变量 SHC_LIB 的指定文件夹的顺序进行系统 include 文件的检索。 按照当前文件夹、include 选项的指定文件夹、环境变量 SHC_INC 的指定文件夹和环境变量 SHC_LIB 的指定文件夹的顺序进行用户 include 文件的检索。
例	shc -include=c:\usr\inc,c:\usr\shc test.c 将文件夹 c:\usr\inc 和 c:\usr\shc 作为 include 文件的路径进行检索。

默认 include 文件

<i>PREInclude</i>	
编译程序 <源>[选项项目 :][include 文件目录]	
格 式	PREInclude = <文件名>[, ...]
说 明	将指定的文件内容取到编译单位的起始位置。当有多个文件名时，能用逗号 (,) 分开指定。
例	shc -preinclude=a.h test.c <test.c> 的内容 <pre>int a; main() { ... }</pre> 编译时的解释 <pre>#include "a.h" int a; main() { ... }</pre>

宏名的定义

DEFine

编译程序 < 源 > [选项项目 :] [宏的定义]

格 式 **DEFine** = <sub>[, ...]
 <sub> : < 宏名 > [= < 字符串 >]

说 明 获得与源文件中记述的 **#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
 整数型常数。

信息消息

MMessage

NOMessage

编译程序 < 源 > [选项项目 :] [信息消息]
 [信息级消息的表示]

格 式 **MMessage**
 NOMessage [= < 错误序号 > [- < 错误序号 >] [, ...]]

说 明 指定是否输出信息级的消息。
 当指定 **message** 选项时，输出信息级的消息。
 当指定 **nomessage** 选项时，不输出信息级的消息。另外，如果通过子选项指定错误号，
 就只抑止指定信息级消息的输出。也能用如 < 错误号 > - < 错误号 > 的连字符 (-) 指定要
 抑止的错误号范围。
 在省略本选项时解释为 **nomessage**。

例 **shc -message test.c**
 输出信息级消息。

文件间 *inline* 展开的文件夹指定

<i>FILE_INLINE_PATH</i>	
编译程序 <源>[选项项目 :][文件间 <i>inline</i> 展开的目录]	
格 式	<code>FILE_INLINE_PATH = <路径名>[, ...]</code>
说 明	指定文件间 <i>inline</i> 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 <code>file_inline_path</code> 选项的指定文件夹和当前文件夹的顺序，检索文件间 <i>inline</i> 展开的对象文件。
例	<code>shc -file_inline_path=c:\usr\file -file_inline=test2.c test.c</code> 将文件夹 <code>c:\usr\file</code> 作为文件间 <i>inline</i> 展开的指定文件夹，检索由 <code>file_inline</code> 选项指定的 <code>test2.c</code> 。

消息级

<i>CHAnge_message</i>	
编译程序 <源>[消息级 :]	
格 式	<code>CHAnge_message = <sub>[, ...]</code> <code><sub></code> : <错误级>[=<错误号>[- <错误号>][, ...]] <code><错误级></code> : { Information Warning Error }
说 明	更改信息和警告的消息级。
例	<code>change_message=information= 错误号</code> 只将警告级的指定错误号更改为信息级。 <code>change_message=warning= 错误号</code> 只将信息级的指定错误号更改为警告级。 <code>change_message=error= 错误号</code> 只将信息级和警告级的指定错误号更改为错误级。 <code>change_message=information</code> 将全部警告消息更改为信息级。 <code>change_message=warning</code> 将全部信息消息更改为警告级。 <code>change_message=error</code> 将全部信息消息和警告消息更改为错误级。
备 注	能通过指定 <code>nomessage</code> 选项，抑止已更改为信息级消息的输出。

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 >] NOLINE	编译程序 < 目标 > [输出文件格式 :] [预处理器展开的程序] [预处理器展开的程序 (抑止 #line 的输出)]	输出预处理器展开后的源程序 在预处理器展开时, 抑止 #line 的输出
2 目标格式	Code = { <u>Machinecode</u> Asmcode }	编译程序 < 目标 > [输出文件格式 :] [机器语言程序] [汇编程序]	 输出机器语言程序 输出汇编程序
3 调试信息	DEBug <u>NODEBug</u>	编译程序 < 目标 > [调试信息的输出]	输出 不输出
4 段名	S ~~E~~ ction = <sub>[,...] <sub>:{ Program=< 段名 > Const = < 段名 > Data = < 段名 > Bss = < 段名 > } }	编译程序 < 目标 > [详细……] [段名 :] [程序区 (P)] [常数区 (C)] [初始化的数据区 (D)] [未初始化的数据区 (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	SString = { <u>Const</u> Data }	编译程序 < 目标 > [详细…] [字符串数据的保存 :]	输出到常数区 输出到初始化的数据区
6 目标文件名的指定	O ~~B~~ jectfile = < 文件名 >	编译程序 < 目标 > [目标输出的目录 :]	输出指定文件名的目标文件
7 模板示例生成功能	Template = { None Static Used ALI <u>AUto</u> }	编译程序 < 目标 > [详细…] [模板生成 :]	不生成模板示例 作为内部连接, 只生成参照的模板示例 作为外部连接, 只生成参照的模板示例 生成声明和参照的模板示例 生成连接时所需的模板示例
8 ABS16/20/28/32 声明	<ABS>=<sub>[,...] <ABS>: { ABs16 ABS20 ABS28 <u>ABS32</u> } <sub>: { Program Const Data Bss Run <u>All</u> }	编译程序 < 目标 > [详细 2…] [地址声明 :]	指定属于指定段的标号地址或者运行库的分配存储空间

项目	命令行格式	对话菜单	指定内容
9 除法计算方式的选择 (SH-1 除外)	DIvision = <u>Cpu</u> = {Inline Runtime }	编译程序 < 目标 > [详细…] [除法计算方式的选择 :]	使用单片机的除法指令 转换为乘法运算, inline 展开调用运行库
10 抑止浮点寄存器的保护和恢复 (SH-2E、SH2A-FPU、SH-4、SH-4A)	IFUnc	编译程序 < 目标 > [详细…] [抑止浮点寄存器的保护和复]	抑止浮点寄存器的保护和恢复
11 标号的 16/32 字节调整	ALIGN16 ALIGN32 <u>NOAlign</u>	编译程序 < 目标 > [详细…] [标号的 16/32 字节调整 :]	对于程序段内的标号, 将紧接在无条件转移指令后的标号全部调整为 16 字节 对于程序段内的标号, 将紧接在无条件转移指令后的标号全部调整为 32 字节 不对标号进行 16/32 字节的调整
12 TBR 相对函数调用的指定 (SH-2A、SH2A-FPU)	TBR [= < 段名 >]	编译程序 < 目标 > [详细 2…] [TBR 指定 :]	通过 TBR 相对进行函数调用
13 未初始化变量的输出顺序	BSs_order = { <u>DEClaration</u> DEFinition }	编译程序 < 目标 > [详细 2…] [未初始化变量的输出顺序 :]	按声明顺序输出 按定义顺序输出
14 变量的配置指定	STuff [= {Bss Data Const} [,…]] <u>NOSTuff</u>	编译程序 < 目标 > [详细 2…] [变量的配置指定 :]	配置到对应变量长度的段 与变量长度无关配置到相同段

预处理器展开

PREProcessor

NOLINE

编译程序 < 目标 > [输出文件形式 :] [预处理器展开的程序]
[预处理器展开的程序 (抑止 #line 的输出)]

格 式	PREProcessor [= < 文件名 >] NOLINE
说 明	输出预处理器展开后的源程序。 当不指定 < 文件名 > 时，建立与源文件相同的文件名且扩展名为 “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。 当指定本选项时，不输出目标文件。 如果指定 noline ，就在预处理器展开时抑止 #line 的输出。
备 注	当指定本选项时，只有以下的选项才有效： define 、 include 、 comment 、 euc 、 sjis 、 latin1 、 subcommand 、 preinclude 、 message 、 lang 、 logo 、 cpu 、 change_message

目标格式

Code

编译程序 < 目标 > [输出文件格式 :] [机器语言程序] [汇编程序]

格 式	Code = { <u>Machinecode</u> Asmcode }
说 明	指定目标程序的输出格式。 当指定 code=machinecode 时，输出可再定位的目标（机器语言）程序。 当指定 code=asmcode 时，输出汇编程序。 在省略本选项时解释为 code=machinecode 。
备 注	如果指定 code=asmcode ， show=object 、 goptimize 选项就无效。

调试信息

<i>DEBug</i> <i>NODEBug</i>	
编译程序 < 目标 > [调试信息输出]	
格 式	DEBug NODEBug
说 明	当指定 debug 选项时，将 C 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， debug 选项也有效。 当指定 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

S ection	
	编译程序 < 目标 > [详细…] [段名 :] [程序区 (P)] [常数区 (C)] [初始化的数据区 (D)] [未初始化的数据区 (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。段名最多为 8192 个字符。 在省略本选项时解释为 section=program=P, const=C, data=D, bss=B。
备 注	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”。 不能给不同区的段指定相同的段名。

字符串输出区域

<i>SString</i>	
编译程序 < 目标 > [详细…] [字符串数据保存 :]	
格 式	SString = { <u>Const</u> Data }
说 明	指定字符串的输出目标。 当指定 <code>string=const</code> 时，输出到常数区。 当指定 <code>string=data</code> 时，输出到初始化的数据区。 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需要将 ROM 的数据传送到 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 <code>string=const</code>。

目标文件名指定

<i>Objectfile</i>	
编译程序 < 目标 > [目标输出目录 :]	
格 式	Objectfile = < 文件名 >
说 明	指定要输出的目标文件。 如果不指定本选项，就输出与源文件相同的文件名且扩展名为“<code>obj</code>”（输出文件为可再定位目标程序时）或者“<code>src</code>”（输出文件为汇编源程序时）的目标文件。 文件扩展名由 <code>code</code> 选项决定为“<code>obj</code>”或者“<code>src</code>”。

模板示例生成功能

Template		编译程序 < 目标 > [详细…] [模板生成 :]	
格 式	Template = { None Static Used All Auto }		
说 明	指定模板的示例生成方法。 当指定 <code>template=none</code> 时，不生成示例。 当指定 <code>template=static</code> 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。 当指定 <code>template=used</code> 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。 当指定 <code>template=all</code> 时，生成在编译单位中被声明或被参照的全部模板示例。 <code>templete=auto</code> 时，生成连接时所需的示例。		
备 注	当指定 <code>code=asmcode</code> 时，总是 <code>template=static</code> 。		

ABS16/20/28/32 声明

ABs16
ABS20
ABS28
ABS32

编译程序 < 目标 > [详细...] [地址声明 :]

格 式 ABs16 = { Program | Const | Data | Bss | Run | All }[, ...]
 ABS20 = { Program | Const | Data | Bss | Run | All }[, ...]
 ABS28 = { Program | Const | Data | Bss | Run | All }[, ...]
 ABS32 = { Program | Const | Data | Bss | Run | All }[, ...]

说 明 指定属于子选项指定的段标号地址或者运行库的分配存储空间。
 在省略本选项时解释为 abs32=all。

表 2.3 地址范围

选项名	地址范围	
	低位	高位
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

【注】 * 注意 0x07FFFF7F 的情况。

表 2.4 子选项的说明

子选项名	含义
program	以程序区为对象
const	以常数区为对象
data	以初始化的数据区为对象
bss	以未初始化的数据区为对象
run	以运行库为对象
all	以全部区域为对象

例

-abs20=program -abs28=const,data
 → 与 -abs20=program -abs28=const,data -abs32=bss,run 相同
-abs20=data -abs16=data
 → 输出警告, -abs16=data 有效

备 注 当同时指定本选项和 #pragma abs16|abs20|abs28|abs32 时, #pragma 的指定有效。
 当同时指定本选项和 #pragma gbr_base|gbr_base1 时, 对于 #pragma gbr_base|gbr_base1 指定的变量, 不适用本选项的指定。
 只在指定 cpu=sh2a|sh2afpu 时, abs20|abs28 选项才有效。

除法计算方式的选择

DDivision	
编译程序 < 目标 >[详细…][除法计算方式的选择 :]	
格 式	DDivision = {Cpu [= { Inline Runtime }] Peripheral Nomask }
说 明	选择程序中的整数型除法和余数的计算方式。 当指定 division=cpu=inline 时，将常数除法转换为乘法，进行 inline 展开；对于变量除法，在单片机为 SH-2A 或者 SH2A-FPU 时选择 DIVS 或者 DIVU 指令，而在其他单片机时选择 DIV1 指令的运行库。在指定 CPU=sh1 时本选项无效。 当指定 division=cpu=runtime 时，对于不能用移位运算进行的除法，在单片机为 SH-2A 或者 SH2A-FPU 时选择 DIVS 或者 DIVU 指令，而在其他单片机时选择 DIV1 指令的运行库。在指定 CPU=sh1 时本选项无效。 当只指定 division=cpu 时，就在指定 size 选项时视为 division=cpu=runtime、speed，而在指定 nospeed 选项时视为 division=cpu=inline。 当指定 division=peripheral 时，选择使用除法器的运行库（中断屏蔽设定为 15），只在单片机为 SH-2（SH7604） 时可执行。 当指定 division=nomask 时，选择使用除法器的运行库（中断屏蔽不变），只在单片机为 SH-2（SH7604） 时可执行。 在指定 peripheral、nomask 时，必须注意以下几点： (1) 不检查被零除，并且不设定 errno。 (2) 在指定 nomask 时，如果在除法器运行中发生中断或者在中断处理例程中使用除法器，就不保证运行。 (3) 不支持上溢中断。 (4) 被零除、上溢等的运算结果按照除法器的规格，可能与指定 cpu 子选项时不同。

浮点寄存器的保存和恢复抑止

IFUnc	
编译程序 < 目标 >[详细…][浮点寄存器的保存和恢复抑止]	
格 式	IFUnc
说 明	抑止浮点寄存器的保存和恢复。
备 注	也能用 #pragma ifunc 以函数单位指定。 在指定 ifunc 选项时，如果对生成浮点指令的源程序进行编译，就发生错误。 本选项只在指定 cpu=sh2e sh2afpu sh4 sh4a 时才有效。

标号的 16/32 字节调整

ALIGN16	
ALIGN32	
NOALign	
编译程序 < 目标 >[详细…][标号的 16/32 字节调整 :]	
格 式	ALIGN16 ALIGN32 NOALign
说 明	当指定 align16 选项时，对于程序段内的标号，将紧接在无条件转移指令后的标号全部调整为 16 字节。 当指定 align32 选项时，对于程序段内的标号，将紧接在无条件转移指令后的标号全部调整为 32 字节。 当指定 noalign 选项时，不对紧接在无条件转移指令后的标号进行 16/32 字节的调整。在省略本选项时解释为 noalign。
备 注	不能同时指定 align16 选项和 align32 选项。 当指定 noalign16 选项时，视为被指定了 noalign 选项。

TBR 相对函数调用的指定

TBR	
编译程序 < 目标 >[详细…][TBR 指定 :]	
格 式	TBR [= < 段名 >]
说 明	通过 TBR 相对进行函数调用。 当指定 < 段名 > 时，对于函数定义，将函数地址表输出到 \$TBR< 段名 > 段。 当不指定 < 段名 > 时，对于函数定义，将函数地址表输出到 \$TBR 段。 详细内容请参照 “10.3.1(2) #pragma tbr”。
备 注	本选项只在指定 cpu=sh2a sh2afpu 时才有效。 在同时指定本选项和 #pragma tbr 时，#pragma tbr 指定有效；在同时指定本选项和 pic=1 时，本选项无效。 当建立函数地址表的函数超过 255 个时，就发生错误。

未初始化变量的输出顺序

BSs_order	
编译程序 < 目标 > [详细…] [未初始化变量的输出顺序 :]	
格 式	BSs_order = { <u>declaration</u> definition }
说 明	当指定 bss_order=declaration 时，按声明的顺序配置未初始化的变量。 当指定 bss_order=definition 时，按定义的顺序配置未初始化的变量。 在省略本选项时解释为 bss_order=declaration。
例	<pre>extern int a1; extern int a2; int a3; extern int a4; int a5; int a2; int a1; int a4;</pre> <指定 bss_order=declaration 时> <pre> .SECTION B,DATA,ALIGN=4 _a1: .RES.L 1 _a2: .RES.L 1 _a3: .RES.L 1 _a4: .RES.L 1 _a5: .RES.L 1</pre> <指定 bss_order=definition 时> <pre> .SECTION B,DATA,ALIGN=4 _a3: .RES.L 1 _a5: .RES.L 1 _a2: .RES.L 1 _a1: .RES.L 1 _a4: .RES.L 1</pre>
备 注	在指定 stuff 选项时，与本选项的指定无关，总是按定义的顺序进行未初始化变量的配置。

STuff

NOSTuff

编译程序 < 目标 > [详细…] [未初始化变量的输出顺序 :]

格 式 *STuff* [= < 段类 > [, ...]]
 NOSTuff
 < 段类 > : { Bss | Data | Const }

说 明 当指定 *stuff* 选项时，按数据长度，将属于指定 < 段类 > 的变量配置到调整数为 4、2、1 的段（表 2.5）。
 在省略 < 段类 > 时，对象为全部变量。
 C、D、B 为 *section* 选项或者用 *#pragma section* 指定的段名。与 *bss_order* 选项的指定无关，总是按定义的顺序输出各段内的数据。

表 2.5 变量长度和段名的关系

		变量长度 (byte)		
		4n	4n-2	2n-1
const 型变量	const	C\$4	C\$2	C\$1
有初始值的变量	data	D\$4	D\$2	D\$1
无初始值的变量	bss	B\$4	B\$2	B\$1

当指定 *nostuff* 选项时，将全部变量配置到调整数为 4 的段。在输出各段内的数据时，C、D 段按定义的顺序，B 段取决于 *bss_order* 选项。
在省略本选项时解释为 *nostuff*。

例 `int a;
char b=0;
const short c=0;
struct {
 char x;
 char y;
} ST;

<指定 stuff 选项时>
 .SECTION C$2, DATA, ALIGN=2
_c:
 .DATA.W H'0000
 .SECTION D$1, DATA, ALIGN=1
_b:
 .DATA.B H'00
 .SECTION B$4, DATA, ALIGN=4
_a:
 .RES.L 1
 .SECTION B$2, DATA, ALIGN=2`

```
_ST:
    .RES.B 2

<指定 nostuff 选项时>
    .SECTION C, DATA, ALIGN=4
_c:
    .DATA.W H'0000
    .SECTION D, DATA, ALIGN=4
_b:
    .DATA.B H'00
    .SECTION B, DATA, ALIGN=4
_a:
    .RES.L 1
_ST:
    .RES.B 2
```

备 注 指定了 #pragma gbr_base|gbr_base1 或者 #pragma global_register 的变量不
为本选项的对象。

2.2.3 列表选项

表 2.6 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	Listfile [= < 文件名 >] NOListfile	编译程序 < 列表 > [编译列表输出]	输出 不输出
2 列表内容和格式	SHow = <sub>[,...] <sub>:{ SOurce <u>NOSource</u> <u>Object</u> NOObject <u>Statistics</u> NOSTatistics Include <u>NOInclude</u> Expansion <u>NOExpansion</u> Width = < 数值 > Length = < 数值 > Tab = {4 8} }	编译程序 < 列表 > [列表内容 :]	源列表的有无 目标列表的有无 统计信息的有无 include 展开后的列表的有无 宏展开后的列表的有无 1 行的最大字符数 : 0, 80 ~ 132 页内的最大行数 : 0, 40 ~ 255 使用制表符时的列数 : 4 或 8

列表文件

Listfile		编译程序 < 列表 > [编译列表输出]
<u>NOListfile</u>		
格 式	Listfile [= < 文件名 >] <u>NOListfile</u>	
说 明	指定是否输出列表文件。 当指定 listfile 选项时，输出列表文件。也能指定 < 文件名 >。 当指定 nolistfile 选项时，不输出列表文件。 能按照 “8.1 文件名的命名方法” 指定 < 文件名 >。 如果不用 listfile 选项指定 < 列表文件名 >，就建立与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 ”（输入源为 C 程序时）或者 “ lpp ”（输入源为 C++ 程序时）。 在省略本选项时解释为 nolistfile 。	

列表内容和格式

SHow		编译程序 < 列表 >[列表内容 :]		
格 式	SHow = <sub>[, ...] <sub>:{ SOurce <u>NOSource</u> <u>Object</u> <u>NObject</u> <u>Statistics</u> <u>NStatistics</u> Include <u>NInclude</u> Expansion <u>NExpansion</u> 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，并且解除列表的输出。 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=nosource,object,statistics,noinclude, noexpansion,width=0,length=0,tab=8。			
备 注	子选项的一览表如表 2.7 所示。			

表 2.7 选项的子选项一览表

子选项		含义
1	source	输出源程序的列表。
2	nosource	不输出源程序的列表。
3	object	输出目标程序的列表。
4	noobject	不输出目标程序的列表。
5	statistics	输出统计信息的列表。
6	nostatistics	不输出统计信息的列表。
7	include	输出 include 文件展开后的源程序列表。如果同时指定 nosource 子选项，include 子选项就无效，不输出源程序列表。
8	noinclude	输出 include 文件展开前的源程序列表。如果同时指定 nosource 子选项，noinclude 子选项就无效，不输出源程序列表。
9	expansion	输出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。
10	noexpansion	输出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。
11	width = < 数值 >	< 数值 > 指定的字符数为列表行的最大字符数。< 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，不规定列表行的最大字符数。
12	length = < 数值 >	< 数值 > 指定的行数为列表页的最大行数。< 数值 > 用 10 进制数指定，能指定 0 或者从 40 到 255 之间的数值。 < 数值 > 为 0 时，不规定列表页的最大行数。
13	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 2.8 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 Debug_only }	编译程序 < 优化 > [优化]	不优化 优化 生成不影响调试信息的代码
2 执行速度和大小的 优化	SPeed SSize NOSPeed	编译程序 < 优化 > [优化方法 :] [速度优先] [大小优先] [大小和速度]	优化项目的选择
3 模块间的优化	Goptimize	编译程序 < 优化 > [模块间的优化]	用于模块间优化的附加信息输出
4 外部变量存取的 优化	MAP = < 文件名 >	编译程序 < 优化 > [外部变量存取的优化 :] [模块间]	进行外部变量存取的优化
5 外部变量存取的 优化	SMap	编译程序 < 优化 > [外部变量存取的优化 :] [模块内]	对于在编译对象文件内定义的外部变量, 进行外部变量存取的优化
6 GBR 相对存取 代码的自动生成	GBr = { Auto User }	编译程序 < 优化 > [GBR 相对存取代码的自动生成 :]	进行 GBR 相对存取代码的自动生成 不进行 GBR 相对存取代码的自动生成
7 switch 语句的展开 方式	CAse = { Ifthen Table }	编译程序 < 优化 > [switch 语句的展开 :]	以 if_then 方式进行展开 以表转移方式进行展开
8 移位运算的展开	SHift = { Inline Runtime }	编译程序 < 优化 > [移位运算的展开 :]	inline 展开 如果移位运算的展开指令数多, 就调用运行库
9 传送代码的展开	BLOckcopy = { Inline Runtime }	编译程序 < 优化 > [传送代码的展开 :]	inline 展开 如果传送容量大, 就调用运行库
10 非调整数据的传送	Unaligned = { Inline Runtime }	编译程序 < 优化 > [非边界调整数据的传送 :]	inline 展开 设定为调用运行库
11 自动 inline 展开	INLine[= < 数值 >] NOINLine	编译程序 < 优化 > [详细...] [inline 展开] [自动 inline 展开]	自动 inline 展开 不自动 inline 展开
12 文件间的 inline 展 开	FILE_inline= < 文件名 >[,...]	编译程序 < 优化 > [详细...] [inline 展开] [inline 展开文件]	指定文件间的 inline 展开对象文件
13 外部变量的 volatile 化	GLOBAL_Volatile={ 0 1 }	编译程序 < 优化 > [详细...] [外部变量] [外部变量 volatile 化]	对于外部变量, 不作为 volatile 声明的变量处理 (volatile 声明的变量除外) 对于全部外部变量, 作为 volatile 声明的变量处理

项目	命令行格式	对话菜单	指定内容
14 外部变量优化范围的指定	OPT_Range={All NOLoop NOBlock }	编译程序 < 优化 > [详细…] [外部变量] [外部变量的优化范围 :]	将函数内的全部外部变量指定为优化对象 抑止循环控制变量和循环内的外部变量向循环外的移动 抑止对跨过循环或分支的外部变量的优化
15 空循环的删除	DEL_vacant_loop={ 0 1 }	编译程序 < 优化 > [详细…] [其他] [空循环删除]	不删除空循环 删除空循环
16 循环最大展开数的指定	MAX_unroll=< 数值 > < 数值 >:1-32	编译程序 < 优化 > [详细…] [其他] [循环展开最大数 :]	指定展开循环时的最大展开数 default:1(指定 speed,loop 时为 2)
17 无限循环前的赋值表达式的删除	INFinite_loop={ 0 1 }	编译程序 < 优化 > [详细…] [外部变量] [无限循环前的外部变量赋值式的删除]	不删除无限循环前的外部变量的常数 删除无限循环前的外部变量常数
18 外部变量的寄存器分配	GLOBAL_Alloc={ 0 1 }	编译程序 < 优化 > [详细…] [外部变量] [外部变量的寄存器分配 :]	抑止外部变量的寄存器分配 进行外部变量的寄存器分配
19 结构体 / 联合体成员的寄存器分配	STRUCT_Alloc={ 0 1 }	编译程序 < 优化 > [详细…] [其他] [结构体 / 联合体成员的寄存器分配 :]	抑止结构体 / 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
20 const 常数的传送	CONST_Var_propagate={ 0 1 }	编译程序 < 优化 > [详细…] [外部变量] [外部常数的常数传送 :]	抑止由 const 限定型外部变量的常数传送 进行由 const 限定型外部变量的常数传送
21 常数装入的指令展开	CONST_Load={ Inline Literal }	编译程序 < 优化 > [详细…] [其他] [常数装入的指令展开 :]	对常数装入进行指令展开 对常数装入进行文字存取 default:size 指定时最多展开 2-3 条指令。除此以外, 进行文字存取
22 指令的顺序更改	SCchedule={ 0 1 }	编译程序 < 优化 > [详细…] [外部变量] [指令的顺序变更 :]	不更改指令的顺序 更改指令的顺序
23 软件流水线 (SH-2A、SH2A-FPU、SH-4、SH-4A、SH4AL-DSP)	SOftpipe	编译程序 < 优化 > [详细…] [其他] [软件流水线 :]	软件流水线有效

项目	命令行格式	对话菜单	指定内容
24 优化范围的分割	<u>SCOpe</u> NOScope	编译程序 < 优化 > [详细…] [其他] [不分割优化范围 :]	分割优化范围 不分割优化范围
25 GBR 相对逻辑 运算的生成	LOGlc_gbr	编译程序 < 优化 > [GBR 相对存取代码的自动生 成 :]	生成用 GBR 相对进行外部变 量逻辑运算的代码

优化级

<i>OPTimize</i>	
编译程序 < 优化 > [优化]	
格 式	OPTimize = { 0 1 Debug_only }
说 明	指定目标程序的优化级。 当指定 optimize=0 时，不进行目标程序的优化。 当指定 optimize=1 时，进行优化。 当指定 optimize=Debug_only 时，抑止对调试信息有影响的优化处理，并且总是能参 照局部变量的信息。因此，和当指定 optimize=0 时相比，有时会增加代码大小。 在省略本选项时解释为 ptimize=1。

执行速度和大小的优化

<i>SPeed</i>	
<i>SIZe</i>	
<i>NOSPeed</i>	
编译程序 < 优化 > [优化方法 :] [速度优先] [大小优先] [大小和速度]	
格 式	<i>SPeed</i> <i>SIZe</i> <u><i>NOSPeed</i></u>
说 明	对于 <i>speed</i> 、 <i>size</i> 、 <i>nospeed</i> 选项的主要优化项目如表 2.9 所示。 可通过选项控制这些优化项目。 在省略本选项时解释为 <i>nospeed</i> 。

表 2.9 优化项目一览

	自动直接 插入展开	循环展开 优化	移位代码 展开	传送代码 展开	整数常数 除法展开	非调整 数据传送
<i>speed</i>	○	○	<i>inline</i>	<i>inline</i>	<i>inline</i>	<i>inline</i>
<i>size</i>	×	×	<i>runtime</i>	<i>runtime</i>	<i>runtime</i>	<i>runtime</i>
<i>nospeed</i>	×	×	<i>inline</i>	<i>inline</i>	<i>inline</i>	<i>inline</i>

模块间的优化

<i>Goptimize</i>	
编译程序 < 优化 > [模块间的优化]	
格 式	<i>Goptimize</i>
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。

外部变量存取的优化

MAP

编译程序 < 优化 > [外部变量存取的优化 :] [模块间]

格 式 MAP = < 文件名 >

说 明 根据优化连接编辑程序生成的外部符号分配信息，设定基地址，生成以基地址相对存取外部变量或者静态变量的代码。当指定 **gbr=auto** 时，根据条件将基地址设定到 **GBR** 寄存器，生成以 **GBR** 相对存取外部变量或者静态变量的代码。

 在使用本选项时，首先不指定本选项进行程序的编译，而在连接时指定 **map=< 文件名 >**，建立外部符号分配信息文件，然后再次指定 **map=< 文件名 >** 进行编译。

例 程序 (test.c)

```
int A,B,C;
void main()
{
    A = 0;
    B = 0;
    C = 0;
}
```

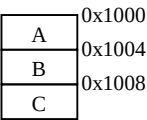
(1) 命令: shc test.c

<输出代码>

```
_main:
    MOV.L    L11,R6      ;_A
    MOV      #0,R2
    MOV.L    R2,@R6
    MOV.L    L11+4,R6    ;_B
    MOV.L    R2,@R6
    MOV.L    L11+8,R6    ;_C
    RTS
    MOV.L    R2,@R6
L11:
    .DATA.L  _A
    .DATA.L  _B
    .DATA.L  _C
```

(2) 命令: shc test.c
 optlnk -map=test.bls -start=P/400,B/1000 test.obj
 shc -map=test.bls test.c

连接后的数据配置



```
<输出代码>
_main:
    MOV.W    L11,R1      ;_A      ※将 A 的地址设定为基地址
    MOV      #0,R0
    MOV.L    R0,@R1
    MOV.L    R0,@(4,R1)
    RTS
    MOV.L    R0,@(8,R1)

L11:
    .DATA.W  _A          ※ A 的地址为 2 字节
```

备 注 在更改外部变量或者静态变量的定义顺序时，需要重新建立外部符号地址信息文件。
如果指定的选项与不使用本选项而进行第 1 次编译时指定的选项不同，就在追加函数内的
处理时不保证运行。在这些情况下，必须重新生成外部符号地址信息文件。

外部变量存取的优化

SMap

编译程序 < 优化 > [外部变量存取优化 :] [模块内]

格 式	SMap
说 明	对在编译对象文件内定义的外部变量或者静态变量设定基地址，生成以基地址相对进行存 取的代码。 当指定 gbr=auto 时，根据条件，将基地址设定到 GBR 寄存器，生成以 GBR 相对存取外 部变量或者静态变量的代码。
例	<pre>int A,B,C; void main() { A = 0; B = 0; C = 0; } MOV.L L11,R6 ; _A MOV #0,R2 ; H'00000000 MOV.L R2,@R6 MOV.L R2,@(4,R6) RTS MOV.L R2,@(8,R6)</pre>

备 注 当和 map=< 文件名 > 同时指定时， map 选项有效。

GBR 相对存取代码的自动生成

GBr		编译程序 < 优化 > [GBR 相对存取代码的自动生成 :]
格 式	GBr = { <u>Auto</u> User }	
说 明	当指定 gbr=auto 时，编译程序根据条件自动生成 GBR 相对的逻辑运算代码。当指定 gbr=auto 和 map=< 文件名 > 时，根据条件给 GBR 设定基地址，生成以 GBR 相对进行外部变量存取的代码。 当指定 gbr=user 时，用户可通过 #pragma gbr_base、#pragma gbr_base1 或者 GBR 关联的内部函数，指定 GBR 的设置、参照和 GBR 相对的存取。 在省略本选项时解释为 gbr=auto。	
例	程序 <pre>char A,B,C; void main() { A = 1; B &= 1; C ^= 1; }</pre> <输出代码 (gbr=auto) > <pre>_main: STC GBR,@-R15 ; 保存 GBR MOV #0,R0 LDC R0,GBR ; 给 GBR 设定 0 MOV.L L11+2,R0 ; R0 <- A 的地址 OR.B #1,@(R0,GBR) ; A = 1 MOV.L L11+6,R0 ; R0 <- B 的地址 AND.B #1,@(R0,GBR) ; B &= 1 MOV.L L11+10,R0 ; R0 <- C 的地址 XOR.B #1,@(R0,GBR) ; C ^= 1 RTS LDC @R15+,GBR ; 恢复 GBR L11: .RES.W 1 .DATA.L _A .DATA.L _B .DATA.L _C</pre>	
备 注	如果指定 gbr=auto 对使用 #pragma gbr_base、#pragma gbr_base1 的程序进行编译，就输出警告信息，忽视 #pragma 的指定。另外，当指定 gbr=auto 对使用 gbr 关联内部函数的程序进行编译时，就发生错误。 当指定 gbr=auto 时，在函数的调用前后保证 GBR 寄存器的内容。	

switch 语句的展开方式

CAsE

编译程序 < 优化 > [switch 语句展开 :]

格 式	CAsE = { Ifthen Table }
说 明	指定 switch 语句的代码展开方式。 当指定 case=ifthen 时，以 if_then 方式展开 switch 语句。if_then 的展开方式将 switch 语句的评价式的值和 case 标号的值进行比较，如果一致就重复进行数次（case 标号的次数）跳转到 case 标号语句处理。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。 当指定 case=table 时，以表格的方式展开 switch 语句。表格的展开方式将 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。此方式与 switch 语句所包含的 case 标号个数成正比，增大确保在文字池的转移表容量，但是执行速度总是固定。 在省略本选项时，编译程序自动选择其中之一的展开方式。

移位运算的展开

SHIfT

编译程序 < 优化 > [移位运算的展开 :]

格 式	SHIfT = { Inline Runtime }
说 明	当指定 shift=inline 时，对全部移位运算进行指令展开。 当指定 shift=runtime 时，如果要展开的指令多，就调用运行库。 指定 size 选项时的默认值是 shift=runtime，指定 speed、nospeed 选项时的默认值是 shift=inline。

传送代码的展开

BLOckcopy

编译程序 < 优化 > [传送代码的展开 :]

格 式	BLOckcopy = { Inline Runtime }
说 明	当指定 blockcopy=inline 时，对全部存储器间的传送代码进行指令展开。 当指定 blockcopy=runtime 时，如果传送容量大，就调用运行库。 指定 size 选项时的默认值是 blockcopy=runtime ，指定 speed 、 nospeed 选项时的默认值是 blockcopy=inline 。

非调整的数据传送

Unaligned

编译程序 < 优化 > [非边界调整的数据传送 :]

格 式	Unaligned = { Inline Runtime }
说 明	当指定 unaligned=inline 时，对非调整的数据传送进行指令展开。 当指定 unaligned=runtime 时，如果非调整的数据传送容量大，就调用运行库。 指定 size 选项的默认值是 unaligned=runtime ，指定 speed 、 nospeed 选项的默认值是 unaligned=inline 。
备 注	本选项以调整数为 1 的结构体传送为对象。

自动 inline 展开

<i>INLine</i>	
<i>NOINLine</i>	
编译程序 < 优化 > [详细…] [inline 展开] [自动 inline 展开]	
格 式	INLine [= < 数值 >] NOINLine
说 明	指定是否对函数进行自动 inline 展开。 当指定 inline 选项时，进行自动 inline 展开。 通过 inline=< 数值 >，能指定程序大小增加到几个 % 为止的 inline 展开。例如：指定 inline=50 时，inline 展开到程序大小增加 1.5 倍为止（50% 为止）。 当指定 noinline 选项时，不进行自动 inline 展开。 指定 speed 选项时的默认值是 inline=20，指定 nospeed、size、optimize=0 或者 optimize=debug_only 选项时的默认值是 noinline。

文件间的 inline 展开

<i>File_inline</i>	
编译程序 < 优化 > [详细…] [inline 展开] [inline 展开的文件]	
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 inline 展开。
例	<pre><a.c> func(){ g(); } <b.c> g(){ h(); }</pre> 如果指定 shc -file_inline=b.c a.c 进行编译，a.c 中的函数 g() 调用就进行以下的展开。 <pre>func(){ h(); }</pre>
备 注	如果同时指定 file_inline 选项和 noinline 选项，就只对 #pragma inline 指定的函数进行 inline 展开。只对指定的函数进行 inline 展开。 如果在 file_inline 选项指定的多个文件中定义了相同名字的 extern 函数，就不保证运行（使用任选的 1 个函数定义进行 inline 展开）。 不能省略 < 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 (*、?)。

外部变量的volatile 化

GLOBAL_Volatile	
编译程序 < 优化 > [详细…] [外部变量] [外部变量的 volatile 化]	
格 式	GLOBAL_Volatile = { 0 1 }
说 明	当指定 global_volatile=0 时，对没有 volatile 声明的外部变量进行优化。因此，外部变量的存取次数和存取顺序可能与 C/C++ 程序记述的情况不同。 当指定 global_volatile=1 时，将全部外部变量作为 volatile 声明的变量处理。因此，外部变量的存取次数和存取顺序为 C/C++ 程序记述的情况相同。 在省略本选项时解释为 global_volatile=0。

外部变量优化范围的指定

OPT_Range	
编译程序 < 优化 > [详细...] [外部变量] [外部变量的优化范围 :]	
格 式	OPT_Range = { All NOLoop NOBlock }
说 明	当指定 <code>opt_range=all</code> 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 <code>opt_range=noloop</code> 时，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 <code>opt_range=noblock</code> 时，抑止跨分支的外部变量的优化（包含循环）。 在省略本选项时解释为 <code>opt_range=all</code>。
例	(1) 跨分支的优化例子（在指定 <code>opt_range=all/noloop</code> 时进行） <pre>int A,B,C; void f(int a) { A = 1; if (a) { B = 1; } C = A; }</pre> <优化后的源程序> <pre>int A,B,C; void f(int a) { A = 1; if (a) { B = 1; } C = 1; /* 删除 A 的参照，传送 A=1 */ }</pre> (2) 循环的优化例子（在指定 <code>opt_range=all</code> 时进行） <pre>int A,B,C[100]; void f() { int i; for (i=0;i<A;i++) { C[i] = B; } }</pre> <优化后的源程序> <pre>int A,B,C[100]; void f() { int i; int temp_A, temp_B; 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 */ } }</pre>
备 注	当指定 <code>opt_range=noloop</code> 时，<code>max_unroll=1</code> 总是为默认值。 当指定 <code>opt_range=noblock</code> 时，<code>max_unroll=1</code>、<code>const_var_propagate=0</code>、<code>global_alloc=0</code> 总是为默认值。

空循环的删除

<i>DEL_vacant_loop</i>	
编译程序 < 优化 > [详细...] [其他] [空循环的删除]	
格 式	DEL_vacant_loop = { 0 1 }
说 明	当指定 del_vacant_loop=0 时，即使循环内没有处理，也不删除循环。 当指定 del_vacant_loop=1 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 注	请注意：默认值和 SHC/C++V7 版本（V7.0.04 为止）的编译程序不同。 V7.0.04 为止：删除循环 V7.0.06 开始：不删除循环

循环最大展开数的指定

<i>MAX_unroll</i>	
编译程序 < 优化 > [详细...] [其他] [循环展开最大数 :]	
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，如果指定其他值，就发生错误。 省略本选项时解释为：在指定 speed 或者 loop 选项时为 max_unroll=2，否则为 max_unroll=1。
备 注	当指定 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	
编译程序 < 优化 > [详细...] [外部变量] [无限循环前的外部变量赋值式的删除]	
格 式	INFinite_loop = { 0 1 }
说 明	当指定 infinite_loop=0 时，不删除无限循环前一行的外部变量赋值表达式。 当指定 infinite_loop=1 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	<pre>int A; void f() { A = 1; /* 外部变量 A 的赋值表达式 */ while(1) {} /* 没有参照 A */ }</pre> <指定 infinite_loop=1 时的源程序> <pre>void f() { /* 删除外部变量 A 的赋值表达式 */ while(1) {} }</pre>
备 注	请注意：默认值和 SHC/C++V7 版本（V7.0.04 为止）的编译程序不同。 V7.0.04 为止：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 V7.0.06 开始：不删除无限循环前一行的外部变量赋值表达式

外部变量的寄存器分配

GLOBAL_Alloc	
编译程序 < 优化 > [详细...] [外部变量] [外部变量的寄存器分配 :]	
格 式	GLOBAL_Alloc = { 0 1 }
说 明	当指定 global_alloc=0 时，抑止外部变量的寄存器分配。 当指定 global_alloc=1 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 注	当指定 opt_range=noblock 和 optimize=0 时，global_alloc=0 为默认值。 在指定 optimize=0 时，请注意：默认值和 SHC/C++V7 版本（V7.0.04 为止）的编译程序不同。 V7.0.04 为止：抑止外部变量的寄存器分配 V7.0.06 开始：进行外部变量的寄存器分配

结构体 / 联合体成员的寄存器分配

STRUCT_Alloc	
编译程序 < 优化 > [详细...] [其他] [结构体 / 联合体成员的寄存器分配 :]	
格 式	STRUCT_Alloc = { 0 1 }
说 明	当指定 struct_alloc=0 时，抑止结构体 / 联合体成员的寄存器分配。 当指定 struct_alloc=1 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 注	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。 在指定 optimize=0 时，请注意：默认值和 SHC/C++V7 版本（V7.0.04 为止）的编译 程序不同。 V7.0.04 为止：进行结构体 / 联合体成员的寄存器分配 V7.0.06 开始：抑止结构体 / 联合体成员的寄存器分配

const 常数传送

CONST_Var_propagate	
编译程序 < 优化 > [详细...] [外部变量] [外部常数的常数传送 :]	
格 式	CONST_Var_propagate = { 0 1 }
说 明	当指定 const_var_propagate=0 时，抑止 const 限制型外部变量的常数传送。 当指定 const_var_propagate=1 时，进行 const 限制型外部变量的常数传送。 在省略本选项时解释为 const_var_propagate=1。
例	<pre>const int X = 1; int A; void f() { A = X; }</pre> <指定 const_var_propagate=1 时的源程序> <pre>void f() { A = 1; /* 传送 X=1 */ }</pre>
备 注	在指定 opt_range=noblock 时，const_var_propagate=0 为默认值。 本选项不能抑止 C++ 程序的 const 限制型变量（总是进行常数传送）。

常数装入的指令展开

CONST_Load	
编译程序 < 优化 > [详细…] [其他] [常数装入的指令展开 :]	
格 式	CONST_Load = { Inline Literal }
说 明	当指定 const_load=inline 时，对全部 2 字节常数和一部分 4 字节常数的装入进行指令展开。 当指定 const_load=literal 时，文字存取全部 2 字节以上（包括 2 字节）的常数装入（在 SH-2A/SH2A-FPU 时，有可能进行指令展开）。 在省略本选项时解释为：在指定 speed 选项时为 const_load=inline；在指定 size 或者 nospeed 选项时，如果在 2 字节常数时能用 2 条指令展开而在 4 字节常数时能用 3 条指令，就为 const_load=inline，否则为 const_load=literal。
例	<pre>int f() { return (257); }</pre> (1) 指定 const_load=inline 或者 speed 时 <pre>MOV #1,R0 ; R0 <- 1 SHLL8 R0 ; R0 <- 256 (1<<8) RTS ADD #1,R0 ; R0 <- 257 (256+1)</pre> (2) 指定 const_load=literal 或者 size/nospeed 时 <pre>MOV.W L11,R0 RTS NOP L11: .DATA.W H'0101</pre>

指令的顺序更改

Schedule	
编译程序 < 优化 > [详细…] [外部变量] [指令的顺序更改 :]	
格 式	Schedule = { 0 1 }
说 明	当指定 schedule=0 时，不进行指令的顺序更改。因此，以 C/C++ 程序记述的顺序进行处理。 当指定 schedule=1 时，进行考虑到流水线处理和超标量（SH-2A、SH2A-FPU、SH-4、SH-4A、SH4AL-DSP）的指令顺序更改。 在省略本选项时解释为 schedule=1。

软件流水线

<i>SOftpipe</i>	
编译程序 < 优化 > [详细…] [其他] [软件流水线 :]	
格 式	SOftpipe
说 明	将软件流水线置为有效。
备 注	本选项只在指定 <code>cpu=sh2a sh2afpu sh4 sh4a sh4aldsp</code> 和 <code>optimize=1</code> 时才有效。

优化范围的分割

<i>SCOpe</i> <i>NOSCOpe</i>	
编译程序 < 优化 > [详细…] [其他] [不分割优化范围 :]	
格 式	<u>SCOpe</u> NOSCOpe
说 明	当指定 <code>scope</code> 选项时，对于大容量函数，将优化范围分割为多个范围进行编译。 当指定 <code>noscope</code> 选项时，不分割优化范围而进行编译。由于扩大优化范围，编译速度变慢，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。 根据程序，本选项会影响目标性能，所以请在调整性能时试一试。

GBR 相对逻辑运算的生成

LOGIc_gbr	
编译程序 < 优化 > [GBR 相对存取代码的自动生成 :]	
格 式	LOGIc_gbr
说 明	生成以 GBR 相对进行外部变量逻辑运算的代码。
备 注	在指定 gbr=auto 时，本选项无效。 在使用本选项时，必须通过内部函数 set_gbr() 给 GBR 设定 \$G0 段的起始地址。

2.2.5 其他选项

表 2.10 其他种类选项一览表

项目	命令行格式	对话框菜单	指定内容
1 嵌入式 C++ 语言	ECpp	编译程序 < 其他 > [其他选项 :] [基于 EC++ 语言的检查]	基于 EC++ 语言的语法检查
2 DSP-C 语言 (SH2-DSP、 SH3-DSP、 SH4AL-DSP)	DSpc	编译程序 < 其他 > [其他选项 :] [基于 DSP-C 语言的检查]	基于 DSP-C 语言的语法检查
3 注释的嵌套	COMment = { Nest NONest }	编译程序 < 其他 > [其他选项 :] [允许注释 (/ * */) 的嵌套]	允许注释 (/ * */) 的嵌套 不允许注释 (/ * */) 的嵌套
4 MAC 寄存器的 保证	Macsave = { 0 1 }	编译程序 < 其他 > [其他选项 :] [保证 MAC 寄存器]	在函数调用前后不保证 MAC 寄存器的 内容 在函数调用前后保证 MAC 寄存器的内 容
5 SSR/SPC 寄存器 的保存和恢复 (SH-3 ~ SH-4)	SAve_cont_reg={ 0 1 }	编译程序 < 其他 > [其他选项 :] [SSR/SPC 的保存和恢复]	不进行 SSR/SPC 寄存器的保存和恢复 进行 SSR/SPC 寄存器的保存和恢复
6 返回值的扩展	RTnext NORTnext	编译程序 < 其他 > [其他选项 :] [进行返回值的扩展]	循环返回值的符号 / 零扩展 不进行返回值的符号 / 零扩展
7 循环展开的优化	LOop NOLOp	编译程序 < 其他 > [其他选项 :] [循环展开]	循环循环展开的优化 不循环循环展开的优化
8 浮点常数除法的 乘法化	APproxdiv	编译程序 < 其他 > [其他选项 :] [浮点常数除法的乘法化]	进行浮点常数除法的乘法化
9 SH7055 不正常 运行的避免 (SH-2E)	PArch=7055	编译程序 < 其他 > [其他选项 :] [SH7055 不正常运行的避免]	不生成使 SH7055 不正常运行的指令 行
10 FPSCR 寄存器的 转换 (SH2A-FPU, SH-4,SH-4A)	FPScr = { Safe Aggressive }	编译程序 < 其他 > [其他选项 :] [FPSCR 寄存器的转换]	当发生 double 运算时, 转换 FPSCR 寄存器 尽量抑止 FPSCR 寄存器的转换
11 循环判断表达式 的优化抑止	Volatile_loop	编译程序 < 其他 > [其他选项 :] [循环判断表达式的优化止]	抑止循环判断表达式的优化
12 枚举型大小	AUto_enum	编译程序 < 其他 > [其他选项 :] [枚举型大小的自动选择]	枚举型大小的自动选择
13 register 存储类 变量的优先分配	ENable_register	编译程序 < 其他 > [其他选项 :] [register 指定变量的优先 寄存器分配]	将指定 register 存储类的变量优先进行 寄存器分配

项目	命令行格式	对话框菜单	指定内容
14 ANSI 的依据	STRlct_ansi	编译程序 < 其他 > [其他选项 :] [依照 ANSI 的对应扩展]	跟据 ANSI 进行以下处理: • unsigned int 和 long 型运算 • 浮点运算的结合律
15 浮点除法的转换 (SH-2E、 SH2A-FPU、 SH-4、SH-4A)	FDIv	编译程序 < 其他 > [其他选项 :] [整数除法到浮点除法的转换]	整数除法转换为浮点除法
16 浮点 → 定点的转换 (SH2-DSP、 SH3-DSP、 SH4AL-DSP)	FIXED_Const	编译程序 < 其他 > [其他选项 :] [浮点 → 定点的转换]	将浮点常数作为定点常数处理
17 1.0r→__fixed 型的最大值转换 (SH2-DSP、 SH3-DSP、 SH4AL-DSP)	FIXED_Max	编译程序 < 其他 > [其他选项 :] [1.0r→__fixed 型的最大值转换]	将 1.0r(R) 作为 (long)__fixed 型的最大值处理
18 省略 __fixed 乘法结果的型转换 (SH2-DSP、 SH3-DSP、 SH4AL-DSP)	FIXED_Noround	编译程序 < 其他 > [其他选项 :] [省略 __fixed 型乘法结果的型转换]	省略对 __fixed 型乘法的运算结果的型转换
19 DSP 重复循环 (SH3-DSP、 SH4AL-DSP)	REPeat	编译程序 < 其他 > [其他选项 :] [DSP 扩展的重复循环]	使用 DSP 扩展的重复循环
20 省略浮点数 - 整数转换时的范围检查 (SH-2E、 SH2A-FPU、 SH-4、SH-4A)	SIMple_float_conv	编译程序 < 其他 > [其他选项 :] [省略浮点数 - 整数转换时的范围检查]	对无符号整数型和浮点型之间的型转换, 生成不检查转换对象值范围的代码。

嵌入式 C++ 语言

ECpp		编译程序 < 其他 >[其他选项 :][基于 EC++ 语言的检查]
格 式	ECpp	
说 明	根据 Embedded C++ 语言规格进行程序的语法检查。 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, 就输出错误消息。	
备 注	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, 就输出错误消息 "C5882 (E) Embedded C++ does not support multiple or virtual inheritance". 本选项不能和 exception 选项同时指定。	

DSP-C 语言

DSpc	
编译程序 < 其他 > [其他选项 :] [基于 DSP-C 语言的检查]	
格 式	DSpc
说 明	根据 DSP-C 语言规格进行 DSP-C 程序的语法检查。 有关 DSP-C 语言规格的详细内容，请参照 “10.2 DSP-C 规格”。
备 注	只在 cpu=sh2dsp sh3dsp sh4aldsp 时，才能指定本选项。 不能对 C++ 程序指定本选项。

注释的嵌套

COMment	
编译程序 < 其他 > [其他选项 :] [允许注释 (/* */) 的嵌套]	
格 式	COMment = { Nest <u>NONest</u> }
说 明	当指定 comment=nest 时，可进行嵌套的注释记述。 在指定 comment=nonest 时，如果进行注释嵌套的记述，就发生错误。 在省略本选项时解释为 comment=nonest。
例	<pre>/* This is an example of /* nested */ comment */ ↑ [1]</pre> 如果指定 comment=nest，就全部解释为注释，但是，如果指定 comment=nonest，就解释为在 [1] 处注释结束。

MAC 寄存器的保证

Macsave	
编译程序 < 其他 > [其他选项 :] [保证 MAC 寄存器]	
格 式	Macsave = { 0 1 }
说 明	指定是否在函数调用前后保证 MACH 和 MACL 寄存器的内容。 当指定 macsave=0 时，在函数调用前后不保证 MACH 和 MACL 寄存器的内容。 当指定 macsave=1 时，在函数调用前后保证 MACH 和 MACL 寄存器的内容。 不能从用 macsave=1 编译的函数调用用 macsave=0 编译的函数，但是能从用 macsave=0 编译的函数调用用 macsave=1 编译的函数。 在省略本选项时解释为 macsave=1。

SSR/SPC 寄存器的保存和恢复

SAve_cont_reg	
编译程序 < 其他 > [其他选项 :] [SSR/SPC 的保存和恢复]	
格 式	SAve_cont_reg = { 0 1 }
说 明	指定是否进行 SSR/SPC 寄存器的保存和恢复。 当指定 save_cont_reg=0 时，不进行 SSR/SPC 寄存器的保存和恢复。 当指定 save_cont_reg=1 时，进行 SSR/SPC 寄存器的保存和恢复。 在指定 cpu=sh3 sh3dsp sh4 sh4a sh4aldsp 时，只对 #pragma interrupt 指定的函数，本选项才有效。 在省略本选项时解释为 save_cont_reg=1。

返回值的扩展

<i>RTnext</i>	
<i>NORTnext</i>	
编译程序 < 其他 > [其他选项 :] [进行返回值的扩展]	
格 式	<i>RTnext</i> <i>NORTnext</i>
说 明	指定对于返回 (unsigned)char 型、(unsigned)short 型的 return 语句，是否进行函数返回值寄存器 R0 的符号扩展或者零扩展。当有函数原型声明时，不需要指定本选项。 当指定 rtnext 选项时，进行函数返回值的符号 / 零扩展。 当指定 nortnext 选项时，不进行函数返回值的符号 / 零扩展。 在省略本选项时解释为 nortnext 。

循环展开的优化

<i>L0op</i>	
<i>NOL0op</i>	
编译程序 < 其他 > [其他选项 :] [循环展开]	
格 式	<i>L0op</i> <i>NOL0op</i>
说 明	指定是否进行循环展开的优化。 当指定 loop 选项时，展开循环语句（for、while、do-while）。 当指定 no0op 选项时，不展开循环语句。 在省略本选项时解释为 no0op 。

浮点常数除法的乘法化

AApproxdiv	
编译程序 < 其他 > [其他选项 :] [浮点常数除法的乘法化]	
格 式	AApproxdiv
说 明	将浮点常数除法转换为常数的倒数乘法。
备 注	当指定本选项时，提高浮点常数除法的执行速度，但是可能会改变运算精度，必须注意。

SH7055 不正常运行的避免

PAth	
编译程序 < 其他 > [其他选项 :] [SH7055 不正常运行的避免]	
格 式	PAth = 7055
说 明	编译程序不输出使 SH7055 不正常运行的指令行。
备 注	本选项只在指定 cpu=sh2e 时才有效。

FPSCR 寄存器精度模式的转换

FPScr	
编译程序 < 其他 > [其他选项 :] [FPSCR 寄存器的转换]	
格 式	FPScr = { Safe <u>Aggressive</u> }
说 明	指定是否在函数调用前后保证 FPSCR 寄存器的精度模式。 在 SH2A-FPU、SH-4、SH-4A 中执行 float 运算和 double 运算时，将 FPSCR 寄存器的精度模式设定为单精度或者双精度。 如果指定 fpscr=safe，就在出现 double 运算时将 FPSCR 寄存器的精度模式转换为双精度模式。当双精度运算结束时，就转换为单精度模式。此情况在从函数调用返回时，FPSCR 寄存器的精度模式总是为单精度模式。 如果指定 fpscr=aggressive，就在继续进行 double 运算时尽可能不转换 FPSCR 寄存器的精度模式。此情况在从函数调用返回时不保证 FPSCR 寄存器的精度模式的值。 在 cpu=sh2afpu sh4 sh4a 并且不为 fpu=single 或者 fpu=double 的指定时，本选项有效。 在省略本选项时解释为 fpscr=aggressive。

循环判断表达式的优化抑止

Volatile_loop	
其他 [其他选项 :] [循环判断表达式的优化抑止]	
格 式	Volatile_loop
说 明	在循环判断表达式包含外部变量时，为非优化对象。 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，可能不为优化抑止对象。
备 注	在不指定本选项时，如果在循环内循环判断表达式不变，就有可能被删除。

枚举型大小

AUto_enum

编译程序 < 其他 >[其他选项 :] [枚举型大小的自动选择]

- 格 式AUto_enum
- 说 明

将 enum 声明的枚举型数据作为枚举值范围内的最小数据类型处理。
在省略本选项时解释为枚举型大小作为 int 型处理。
枚举型能取得的值和数据型的关系如表 2.11 所示。

表 2.11 枚举型能取得的值和数据型的关系

枚举值		数据类型
最小值	最大值	
-128	127	signed char
0	255	unsigned char
-32768	32767	signed short
0	65535	unsigned short
上述以外		int

register 存储类变量的优先分配

ENAbLe_register

编译程序 < 其他 >[其他选项 :] [register 指定变量的优先寄存器分配]

- 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 注

如果不能分配到寄存器，就输出信息消息：
C0102 (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，就不输出本消息。

ANSI 的依据

STRICT_ansi	
编译程序 < 其他 > [其他选项 :] [依照 ANSI 的对应扩展]	
格 式	STRICT_ansi
说 明	根据 ANSI 进行以下处理: (1) unsigned int 和 long 型的运算 例: <pre>long sl; unsigned int ui; sl /= ui; /* 未指定 strict_ansi 时以 long 运算, 指定时以 unsigned int 运算 */</pre> (2) 浮点运算的结合律
备 注	在指定本选项时, 运算结果有可能与旧版本的编译程序不同。

浮点除法的转换

FDIV	
编译程序 < 其他 > [其他选项 :] [整数除法到浮点除法的转换]	
格 式	FDIV
说 明	将整数除法转换为浮点除法。从而能提高除法的执行速度。
备 注	本选项只在指定 <code>cpu=sh2e sh2afpu sh4 sh4a</code> 时才有效。 在指定 <code>ifunc</code> 选项时, 本选项指定对指定 <code>#pragma ifunc</code> 的函数无效。 如果指定 <code>cpu=sh2afpu sh4 sh4a</code> 和 <code>fpu=double</code>, 就在除数和被除数都不超过 4byte 时进行转换, 否则就在除数和被除数都不超过 2byte 时进行转换。

浮点 → 定点的转换

<i>FIXED_Const</i>	
编译程序 < 其他 > [其他选项 :] [浮点 → 定点的转换]	
格 式	FIXED_Const
说 明	当指定本选项时，将浮点型常数作为定点型常数生成目标。
备 注	本选项只在指定 <code>cpu=sh2dsp sh3dsp sh4aldsp</code> 和 <code>dspc</code> 选项时才有效。 如果通过浮点型常数的表示格式明确地表示，即使指定本选项也作为浮点型常数生成目标。

1.0r → __fixed 型的最大值转换

<i>FIXED_Max</i>	
编译程序 < 其他 > [其他选项 :] [1.0r → __fixed 型的最大值转换]	
格 式	FIXED_Max
说 明	当指定本选项时，将 <code>1.0r</code> 作为 <code>__fixed</code> 型的最大值、 <code>1.0R</code> 作为 <code>long __fixed</code> 型的最大值生成目标。 最大值的详细内容请参照 “10.4.1(8) <fixed.h>”。
备 注	本选项只在指定 <code>cpu=sh2dsp sh3dsp sh4aldsp</code> 和 <code>dspc</code> 选项时才有效。

省略 `__fixed` 乘法结果的型转换

FIXED_Noround

编译程序 < 其他 > [其他选项 :] [省略 `__fixed` 型乘法结果的型转换]

格 式	<code>FIXED_Noround</code>
说 明	当指定本选项时，对于 <code>__fixed</code> 型乘法的 <code>long __fixed</code> 型运算结果，省略将该结果转换为 <code>__fixed</code> 型的型转换。
备 注	如果指定本选项，就有可能改变运算结果的精度。 本选项只在指定 <code>cpu=sh2dsp sh3dsp sh4aldsp</code> 和 <code>dspc</code> 选项时才有效。

DSP 扩展的重复循环

REPEAT

编译程序 < 其他 > [其他选项 :] [DSP 扩展的重复循环]

格 式	<code>REPEAT</code>
说 明	当指定 <code>repeat</code> 选项时，有可能将循环展开为使用 DSP 扩展重复循环的代码。
备 注	扩展重复循环只能用于支持 LDRC 指令的单片机。 本选项只在指定 <code>cpu=sh3dsp sh4aldsp</code> 时才有效。

省略浮点数 - 整数转换时的范围检查

SIMple_float_conv

编译程序 < 其他 > [其他选项 :] [省略浮点数 - 整数转换时的范围检查]

格 式 SIMple_float_conv

说 明 对无符号整数型和浮点型之间的型转换，生成不检查转换对象值范围的代码。

例 1 unsigned long func(float f)
 {
 return ((unsigned int)f);
 }

[未使用选项时]

```
MOV      #79,R2          ; 0x0000004F
SHLL8    R2
SHLL16   R2              ; 0x4F000000
LDS      R2,FPUL
FSTS     FPUL,FR8
FCMP     FR4,FR8
BT       L12
FADD     FR8,FR8          ; 在 f ≥ 0x4F000000 时,
FSUB     FR8,FR4          ; 转换前的值为 (f-0x4F800000)。
L12:
FTRC     FR4,FPUL        ; float 转换为 signed long
STS      FPUL,R0

[ 使用选项时 ]
FTRC     FR4,FPUL        ; float 型转换为 signed long 型
STS      FPUL,R0
```

例 2 float func2(unsigned long u)
 {
 return ((float)u);
 }

[未使用选项时]

```
LDS      R4,FPUL
CMP/PZ   R4
BT/S     L12
FLOAT    FPUL,FR0        ; signed long 转换为 float
MOVA     L13+2,R
FMOV.S   @R0,FR9          ; 在 u ≥ 0x80000000u 时,
FADD     FR9,FR0          ; 转换后的值加上 0x4F800000。
L12:
```

```

        RTS
        NOP
L13:
        RES.W      1
        DATA.L    H'4F800000

[ 使用选项时 ]
        LDS        R4,FPUL
        RTS
        FLOAT      FPUL,FR0      ; signed long 转换为 float

```

备 注 本选项只在指定 `cpu=sh2e|sh2afpu|sh4|sh4a` 时才有效。
 如果型转换前的值为 `0 ~ 2147483647` 以外的整数或者 `0.0 ~ 2147483647.0` 以外的浮
 点数，就不保证运行。在使用这些范围外的值时，不能使用本选项。

2.2.6 单片机选项

表 2.12 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 单片机 / 运行模式	CPU = { <u>SH1</u> SH2 SH2E SH2A SH2AFPU SH2DSP SH3 SH3DSP SH4 SH4A SH4ALDSP }	CPU [CPU 种类 :]	生成 SH-1 的目标 生成 SH-2 的目标 生成 SH-2E 的目标 生成 SH-2A 的目标 生成 SH2A-FPU 的目标 生成 SH2-DSP 的目标 生成 SH-3 的目标 生成 SH3-DSP 的目标 生成 SH-4 的目标 生成 SH-4A 的目标 生成 SH4AL-DSP 的目标
2 存储器的字节排列顺序 (SH-2 ~ SH-4)	ENdian = { <u>Big</u> Little }	CPU [Endian 选择 :]	Big Endian Little Endian
3 浮点运算模式 (SH2A-FPU、SH-4、SH-4A)	FPu = { Single Double }	CPU [浮点运算模式 :]	全部以单精度浮点进行浮点运算 全部以双精度浮点进行浮点运算
4 舍入方式 (SH2A-FPU、SH-4、SH-4A)	Round = { <u>Zero</u> Nearest }	CPU [舍入方式 :]	以 Round to Zero 舍入 以 Round to Nearest 舍入
5 非规范化数的处理 (SH-4、SH-4A)	DENormalize = { <u>OFF</u> ON }	CPU [非规范化数作为非规范化数处理]	非规范化数作为 0 处理 非规范化数作为非规范化数处理
6 程序段的位置独立 (SH-1 除外)	Pic = { <u>0</u> 1 }	CPU [位置独立的代码生成]	不生成程序段的位置独立代码 生成程序段的位置独立代码
7 double→float 的转换 (SH2A-FPU、SH-4、SH-4A 除外)	DOuble = Float	CPU [double→float 转换]	将 double 型的变量作为 float 型处理
8 位域排列顺序的指定	Blt_order={ <u>Left</u> Right }	CPU [位域成员从低位开始保存]	位域成员从高位开始保存 位域成员从低位开始保存
9 结构体、联合体、类成员的调整数	PACK={1 4 }	CPU [将结构体成员的边界调整数设定为 1]	将调整数设定为 1 按照数据的调整
10 异常处理功能	EXception <u>NOEXception</u>	CPU [将 C++ 的 try、throw、catch 设定为有效]	将异常处理功能设定为有效 将异常处理功能设定为无效
11 运行时的型信息	RTTI = { ON OFF }	CPU [将 C++ 的 dynamic_cast、typeid 设定为有效]	dynamic_cast、typeid 有效 dynamic_cast、typeid 无效

项目	命令行格式	对话菜单	指定内容
12* 除法方式的选择 (只限 SH-2)	Division = { <u>C</u> pu	CPU	使用单片机的除法指令
	Peripheral	[除法方式的选择 :]	使用除法器 (有中断屏蔽)
	Nomask		使用除法器 (没有中断屏蔽)
	}		

【注】 * 选项的详细内容请参照 “2.2.2 目标选项”。

单片机种类

CPu

CPU[CPU 种类 :]

格 式 CPu = { SH1
 SH2
 SH2E
 SH2A
 SH2AFPU
 SH2DSP
 SH3
 SH3DSP
 SH4
 SH4A
 SH4ALDSP }

说 明 指定要建立的目标程序的单片机种类。子选项的一览表如表 2.13 所示。
 在省略本选项时解释为 cpu=sh1。

表 2.13 cpu 选项的子选项一览表

子选项名	含 义
1 sh1	建立 SH-1 的目标。
2 sh2	建立 SH-2 的目标。
3 sh2e	建立 SH-2E 的目标。
4 sh2a	建立 SH-2A 的目标。
5 sh2afpu	建立 SH2A-FPU 的目标。
6 sh2dsp	建立 SH2-DSP 的目标。
7 sh3	建立 SH-3 的目标。
8 sh3dsp	建立 SH3-DSP 的目标。
9 sh4	建立 SH-4 的目标。
10 sh4a	建立 SH-4A 的目标。
11 sh4aldsp	建立 SH4AL-DSP 的目标。

存储器的字节排列顺序

ENdian

CPU[Endian 的选择 :]

- 格 式
- ENdian = { Big | Little }
- 说 明
- 当指定 endian=big 时，数据的字节排列顺序为 Big Endian。
当指定 endian=little 时，数据的字节排列顺序为 Little Endian。Little Endian 的目标程序不能执行于 SH-1、SH-2E、SH-2A、SH2A-FPU、SH2-DSP。
在省略本选项时解释为 endian=big。

浮点运算模式

FPu

CPU[浮点运算模式 :]

- 格 式
- FPu = { Single | Double }
- 说 明
- 当指定 fpu=single 时，以单精度浮点进行全部浮点运算。
当指定 fpu=double 时，以双精度浮点进行全部浮点运算。
在程序中没有浮点运算时，必须指定 fpu=single。
本选项只在指定 cpu=sh2afpu|sh4|sh4a 时才有效。

舍入方式

Round	
CPU[舍入方式 :]	
格 式	Round = { <u>Zero</u> Nearest }
说 明	指定将浮点常数转换为目标代码时的舍去方式。 当指定 round=zero 时，以 Round to Zero 舍入。 当指定 round=nearest 时，以 Round to Nearest 舍入。 本选项只在指定 cpu=sh2afpu sh4 sh4a 时才有效。 在省略本选项时解释为 round=zero。

非正规化数的处理

DENormalize	
CPU[非正规化数作为非正规化数处理]	
格 式	DENormalize = { <u>OFF</u> ON }
说 明	指定在对浮点常数记述了非正规化数时的处理。 当指定 denormalize=off 时，将非正规化数作为 0 处理。 当指定 denormalize=on 时，将非正规化数作为非正规化数处理。 本选项只在指定 cpu=sh4 sh4a 时才有效。 在省略本选项时解释为 denormalize=off。

位置独立的代码

<i>Pic</i>	
CPU[位置独立的代码生成]	
格 式	<code>Pic = { 0 1 }</code>
说 明	当指定 <code>pic=1</code> 时，能将连接后的程序段配置到任意的地址后进行执行，而数据段只能配置到连接时决定的地址。当作为位置独立代码进行执行时，不能将函数的地址指定为初始值。在编译 C++ 时，因为指向虚拟函数或者函数成员的指针需要将函数的地址指定为初始值，所以包含指向虚拟函数或者成员函数的指针的 C++ 程序不能作为位置独立代码进行执行。 例 1 <pre>extern int f(); int (*fp)() = f; ← 不能指定</pre> 例 2 <pre>struct A{virtual void f();}; ← 不能指定 void (A::*ap)() = &A::f; ← 不能指定</pre> 当指定 <code>cpu=sh1</code> 时，忽视 <code>pic=1</code> 的指定。 在省略本选项时解释为 <code>pic=0</code>。

double→float 的转换

<i>DObble = Float</i>	
CPU[double→float 的转换]	
格 式	<code>DObble = Float</code>
说 明	将 <code>double</code> （双精度浮点）型的数值作为 <code>float</code> （单精度浮点）型生成目标。
备 注	当指定 <code>cpu=sh2afpu sh4 sh4a</code> 时，如果指定本选项，本选项就无效，被视为指定了 <code>fpu=single</code> 。

位域排列顺序的指定

<i>Bit_order</i>	
CPU[位域从低位开始保存]	
格 式	<code>Bit_order = { <u>Left</u> Right }</code>
说 明	指定位域成员的排列顺序。 当指定 <code>bit_order=left</code> 时，从高位开始分配成员。 当指定 <code>bit_order=right</code> 时，从低位开始分配成员。 在省略本选项时解释为 <code>bit_order=left</code> 。
备 注	有关位域成员的分配，请参照“10.1.2 数据的内部表示”和“10.3.1 #pragma”的 <code>#pragma bit_order</code> 。

结构体、联合体和类成员的调整数

<i>PACK</i>	
CPU[将结构体成员的边界调整数设定为 1]	
格 式	<code>PACK = { 1 <u>4</u> }</code>
说 明	指定结构体、联合体和类成员的调整数。 也能用 <code>#pragma pack</code> 扩展名指定结构体成员的调整数。如果指定选项和 <code>#pragma</code> ，就优先扩展名的指定。 结构体、联合体和类成员的调整数与成员的最大调整数相同 详细内容请参照“10.1.2(2) 结构体 / 联合体（C 语言）和 class 型（C++ 语言）”。 在省略本选项时解释为 <code>pack=4</code> 。
备 注	指定 <code>pack</code> 选项时的结构体成员的调整数如表 2.14 所示。

表 2.14 指定 pack 选项时的结构体、联合体和类成员的调整数

成员的型	pack=1	pack=4	无指定
(unsigned)char	1	1	1
(unsigned)short、__fixed	1	2	2
(unsigned)int、(unsigned)long、 (unsigned)long long、long __fixed、__accum、 long __accum、浮点型、指针型	1	4	4
调整数为 1 的结构体、联合体和类成员	1	1	1
调整数为 2 的结构体、联合体和类成员	1	2	2
调整数为 4 的结构体、联合体和类成员	1	4	4

异常处理功能

<i>EXception</i>	
<i>NOEXception</i>	
CPU[将 C++ 的 try、throw、catch 设定为有效]	
格 式	EXception <u>NOEXception</u>
说 明	当指定 exception 选项时，将 C++ 异常处理功能（try、catch、throw）设定为有效。 当指定 noexception 选项时，将 C++ 异常处理功能（try、catch、throw）设定为无效。 如果指定 exception 选项，就有可能降低代码性能。 在省略本选项时解释为 noexception。
备 注	要在文件间将例外处理功能设定为有效时，必须进行以下的指定： (1) 指定 rtti=on (2) 不能用优化连接编辑程序指定 noprelink 选项 不能同时指定 exception 选项和 ecpp 选项。

运行时的型信息

<i>RTTI</i>	
CPU[将 C++ 的 dynamic_cast、typeid 设定为有效]	
格 式	RTTI = { ON <u>OFF</u> }
说 明	指定运行时的型信息的有效或者无效。 当指定 rtti=on 时，将 dynamic_cast、typeid 设定为有效。 当指定 rtti=off 时，将 dynamic_cast、typeid 设定为无效。 在省略本选项时解释为 rtti=off。
备 注	不能将由本选项的指定而建立的目标文件登录到库，也不能通过优化连接编辑程序以可再定位形式输出文件，否则就有可能产生符号双重定义错误或者符号未定义错误。

2.2.7 其余选项

表 2.15 其余选项一览表

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 }	— (用扩展名判断)	作为 C 程序编译 作为 C++ 程序编译
2 版权输出的抑止	LOGO NOLOGO	— (nologo 总是有效)	输出版权 抑止版权的输出
3 字符串中的字符码	Euc SJis LATin1	编译程序 < 其它 > [用户的指定选项 :]	选择 euc 码 选择 sjis 码 选择 latin1 码
4 目标码中汉字码的指定	OUTcode = { EUc SJis }	编译程序 < 其它 > [用户的指定选项 :]	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		无 (总是通过扩展名判断)
格 式	LAng = { C Cpp }	
说 明	指定源程序的语言。 当指定 lang=c 时, 作为 C 程序编译。 当指定 lang=cpp 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	shc test.c shc test.cpp shc -lang=cpp test.c shc test	作为 C 程序编译。 作为 C++ 程序编译 作为 C++ 程序编译 假设为 test.c, 作为 C 程序编译。
备 注	在指定 lang=c 时, ecpp 选项无效。	

版权输出抑止

LOGO	
NOLOGO	
无（nologo 总是有效）	
格 式	LOGO NOLOGO
说 明	抑止版权的输出。 当指定 logo 选项时，输出版权显示。 当指定 nologo 选项时，抑止版权显示的输出 在省略本选项时解释为 logo 。

字符串中的字符码

Euc	
SJis	
LATin1	
编译程序 < 其他 >[用户的指定选项 :]	
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 2.16 所示。

表 2.16 主机和字符串中的代码

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

备 注 在指定 **latin1** 选项时，**outcode** 选项无效。

目标码中的汉字码指定

OUnicode	
编译程序 < 其他 >[用户的指定选项 :]	
格 式	OUnicode = { 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。 当指定 outcode=euc 时，用 euc 码输出汉字码。 当指定 outcode=sjis 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。

子命令文件的指定

Subcommand	
无	
格 式	Subcommand = < 文件名 >
说 明	当指定 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 : -listfile -show=object -debug 命令行指定 : shc -cpu=sh4 -subcommand=opt.sub test.c 编译程序解释 : shc -cpu=sh4 -listfile -show=object -debug test.c

第 3 章 汇编程序的操作方法

3.1 选项的指定规则

启动汇编程序的命令行的格式如下：

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

【注】* 当指定多个源文件名时，按指定的顺序连接这些文件后的内容为汇编处理单位。此时，只能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，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 文件的 文件夹	Include = < 路径名 > [, ...]	汇编程序 < 源 > [选项项目 :] [include 文件目录]	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 >	汇编程序 < 源 > [选项项目 :] [符号定义]	定义字符串的置换。
3 整数型预处理器变 量的定义	ASsignA = < sub > [, ...] < sub > : < 变量名 > = < 整数常数 >	汇编程序 < 源 > [选项项目 :] [预处理器变量的定义]	定义整数型的预处理器变量。
4 字符型预处理器变 量的定义	ASsignC = < sub > [, ...] < sub > : < 变量名 > = < 字符串 >	汇编程序 < 源 > [选项项目 :] [预处理器变量的定义]	定义字符型的预处理器变量。

*include 文件的文件夹****Include***

汇编程序 <源>[选项项目 :][include 文件目录]

格 式 Include=< 路径名 > [, ...]

说 明 指定 **include** 文件的文件夹名。
 文件夹名的指定取决于主机的标准指定方法。
 能输入多个文件夹名，但是必须在 1 行的命令行中。
 首先检索当前文件夹，然后按照指定的顺序检索 **include** 选项指定的文件夹。

例 `asmsh aaa.src -include=C:\common, C:\local`
 如果在 `aaa.src` 中指定 `.INCLUDE "file.h"`，就按照当前文件夹、`C:\common`、`C:\local` 的顺序检索 `file.h`。

备 注 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 <code>.INCLUDE</code> 指定的文件夹 由 <code>include</code> 选项指定的文件夹 *
(无指定)	<code>.INCLUDE < 文件名 ></code>	由 <code>.INCLUDE</code> 指定的文件夹

【注】* 使用附加在 `.INCLUDE` 指定的文件夹字符串前的 `include` 选项指定的目录字符串的文件夹名。

*置换符号的定义****DEFine***

汇编程序 <源>[选项项目 :][符号定义]

格 式 `DEFine = <sub>[, ...]`
`<sub> : < 置换符号 >=< 字符串 >`

说 明 通过预处理器将置换符号替换为对应的字符串。
`define` 和 `assignc` 选项的不同功能对应 `.DEFINE` 和 `.ASSIGNC` 的不同功能。

备 注 与汇编程序控制语句的关系

选项	控制语句	结 果
define	<code>.DEFINE*</code>	由 <code>define</code> 选项指定的字符串
	(无指定)	由 <code>define</code> 选项指定的字符串
(无指定)	<code>.DEFINE</code>	由 <code>.DEFINE</code> 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。另外，不能通过 `define` 选项置换下述 6 种控制指令：

`.AENDI`、`.AENDR`、`.AENDW`、`.AIFDEF`、`.END`、`.ENDM`

整数型预处理器变量的定义

ASsignA		汇编程序 <源>[选项项目 :][预处理器变量定义]
格 式	ASsignA= <sub>[, ...] <sub> : < 预处理器变量名 >=< 整数常数 >	
说 明	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常数 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，但是在设定负值时不 能使用 10 进制数的基数。	
例	asmsh aaa.src -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 注	如果主机 OS 为 UNIX 并且在预处理器变量名中存在 \$ 或者表示基数的单引号（'），就在 前面指定反斜杠（\）。	

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 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		汇编程序 < 源 > [选项项目 :] [预处理器变量定义]
格 式	ASsignC= <sub>[, ...] <sub> : < 预处理器变量名 >=< 字符串 >	
说 明	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	asmsh aaa.src -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 注	如果主机 OS 为 UNIX 并且在字符串中指定下列字符时，就在前面指定反斜杠 (\) 。 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)	.ASSIGNC	由 .ASSIGNC 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	汇编程序 < 目标 > [调试信息输出 :]	控制调试信息的输出
2 预处理器的展开结果输出	EXPand [= < 输出文件名 >]	汇编程序 < 目标 > [预处理器的展开结果输出]	输出预处理器的展开结果
3 文字池输出位置的指定	LITERAL = <point> [, ...] <point> : { Pool Branch Jump Return }	汇编程序 < 目标 > [文字池的输出位置指定 :]	指定文字池的输出位置。
4 目标模块的输出控制	Object [= < 输出文件名 >] NOObject	汇编程序 < 目标 > [目标输出的目录 :]	控制目标模块的输出。
5 未解决符号的长度指定 (SH-2A、SH2A-FPU)	Dlspsize={4 12}	汇编程序 < 目标 > [未解决符号的长度指定]	指定未解决符号的长度。

调试信息

Debug

NODebug

汇编程序 < 目标 > [调试信息输出 :]

格 式 Debug
 NODebug

说 明 当指定 debug 选项时，输出调试信息。
 当指定 nodebug 选项时，不输出调试信息。
 只在输出目标模块时，debug 和 nodebug 各选项的指定才有效。
 在省略本选项时解释为 nodebug。

备 注 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

汇编程序 < 目标 > [输出预处理器的展开结果]

格 式	EXPand [= < 输出文件名 >]
说 明	expand 选项输出进行宏展开、条件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 注	不能给输入的源文件和输出的文件指定相同的文件名。

文字池的输出位置指定

LITERAL

汇编程序 < 目标 > [文字表输出位置的指定 :]

格 式	LITERAL = <point>[, ...] <point> : { Pool Branch Jump Return }
说 明	指定由文字池自动生成功能生成的文字池的输出位置。 • pool 输出到 .POOL 的位置 • branch 输出到 BRA/BRAF 指令后 • jump 输出到 JMP 指令后 • return 输出到 RTS/RTE 指令后 在省略本选项时解释为 literal=pool,branch,jump,return。

目标模块的输出控制

Object
NOObject

汇编程序 < 目标 > [目标输出的目录 :]

格 式 **Object** [= < 输出目标文件名 >]
 NOObject

说 明 当指定 **object** 选项时，输出目标文件。
 当指定 **noobject** 选项时，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**。

在省略本选项时解释为 **object**。

备 注 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。

不能给输入的源文件和输出的目标文件指定相同的文件名。如果指定相同的文件名，输入的源文件就会被盖写。

未解决符号的长度指定

DIspsize

汇编程序 < 目标 > [未解决符号的长度指定 :]

格 式 **DIspsize** = {4|12}

说 明 指定外部参照符号和值没解决的符号长度。
 指定对象是对于寻址方式其位移量长度为 4 和 12 的指令。
 移量长度只为 4 的指令为对象外。
 在省略本选项时解释为 **dispsize=12**。

备 注 本选项只在单片机种类为 **SH-2A** 和 **SH2A-FPU** 时才有效。
 即使指定本选项也优先确保长度（:12）的指定。

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	汇编程序 < 列表 > [汇编程序列表输出]	控制汇编列表的输出。
2 源程序列表的输出控制 *	SOUrce NOSOUrce	汇编程序 < 列表 > [汇编程序列表输出] [源程序 :]	控制源程序列表的输出。
3 源程序列表的部分输出控制和制表符长度的设定 *	SHoW [= < 输出种类 >[, ...]] NOSHoW [= < 输出种类 >[, ...]] < 输出种类 > : {CONditionals Definitions CALLs Expansions CODE TAB={ 4 8 } }	汇编程序 < 列表 > [源程序列表的部分输出 :] [附带条件不成立] [定义] [调用] [展开] [目标码 :] [制表符长度 :]	控制源程序列表的部分输出以及设定制表符长度。
4 对照表的输出控制 *	CRoss_reference NOCross_reference	汇编程序 < 列表 > [汇编程序列表输出] [对照表 :]	控制对照表的列表输出。
5 段信息列表的输出控制 *	SEction NOSEction	汇编程序 < 列表 > [汇编列表输出] [段 :]	控制段信息列表的输出。

【注】 * 这些选项只在指定 list 选项时才有效。

汇编列表的输出控制

LISt**NOLISt**

汇编程序 < 列表 > [汇编程序列表输出]

格 式 **LISt** [= < 输出文件名 >]
NOLISt

说 明 当指定 **list** 选项时，输出汇编列表。
 当指定 **nolist** 选项时，不输出汇编列表。
 在省略输出文件名的指定时，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**。
 在省略本选项时解释为 **nolist**。

备 注 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。

不能给输入的源文件和输出的列表文件指定相同的文件名。如果指定相同的文件名，输入的源文件就会被盖写。

源程序列表的部分输出控制和制表符长度的设定

SHow**NOSHow**

汇编程序 < 列表 > [源程序列表部分输出 :]

[附带条件不成立] [定义] [调用] [展开] [目标码 :] [制表符长度 :]

格 式

SHow [= < 输出种类 > [, ...]]**NOSHow** [= < 输出种类 > [, ...]]

< 输出种类 > : {	CONDitionals		Definitions	
	CALLs		Expansions	
	CODe		TAB = { 4 8 }	}

说 明

当指定 **show** 选项时，用指定的制表符长度，输出源程序列表的预处理器功能的源语句和目标码的显示行。

当指定 **show**=< 输出种类 > 时，只输出指定的项目。在不指定制表符长度时，用默认的长度输出。

当指定 **noshow** 选项时，不输出源程序列表的预处理器功能的源语句和目标码的显示行。

当指定 **noshow**=< 输出种类 > 时，不输出指定的项目。

只在输出汇编列表时，**show** 和 **noshow** 选项的指定才有效。

输出种类的内容如下。

在省略本选项时解释为 **show**。

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 的定义部分 .INCLUDE .ASSIGNA、.ASSIGNC
calls	调用	宏调用语句 .AIF、.AIFDEF、.AENDI
expansions	展开	宏展开部分 .AREPEAT、.AWHILE 展开部分
code	目标码的显示行	控制指令的目标码显示超过源语句行数的部分
tab={4 8}	制表符长度	列表显示时的制表符长度

备 注

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）	抑止输出
（无指定）	.LIST 输出种类（输出）	输出
	.LIST 输出种类（输出抑止）	抑止输出
	（无指定）	输出

对照表的输出控制

Cross_reference***NOCross_reference***

汇编程序 < 列表 > [汇编列表输出] [对照表 :]

格 式 ***Cross_reference***
 NOCross_reference

说 明 当指定 **cross_reference** 选项时，将对照表附加到汇编列表。
 当指定 **nocross_reference** 选项时，不将对照表附加到汇编列表。
 只在输出汇编列表时，**cross_reference** 和 **nocross_reference** 的指定才有效。
 在省略本选项时解释为 **cross_reference**。

备 注 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***

汇编程序 < 列表 > [汇编列表输出] [段 :]

格 式 ***Section***
 NOSection

说 明 当指定 **section** 选项时，将段信息列表附加到汇编列表。
 当指定 **nosection** 选项时，不将段信息列表附加到汇编列表。
 只在输出汇编列表时，**section** 和 **nosection** 的指定才有效。
 在省略本选项时解释为 **section**。

备 注 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 其他选项

表 3.4 其他种类选项一览表

项目	命令行格式	对话菜单	指定内容
1 指定文字池自动生成功能的长度模式	AUTO_literal	汇编程序 < 其他 > [其他选项 :] [将文字池自动生成功能指定为长度选择模式]	将文字池自动生成功能设定为长度选择模式。
2 未参照的外部参照符号信息的输出抑止	Exclude NOExclude	汇编程序 < 其他 > [其他选项 :] [未定义外部参照符号信息的输出抑止]	指定输出或者不输出未参照外部参照符号的符号信息。
3 指定特权模式指令的检查	CHKMd	汇编程序 < 其他 > [其他选项 :] [指定特权模式指令的检查]	指定特权模式指令的检查。
4 指定 LDTLB 指令的检查	CHKTlb	汇编程序 < 其他 > [其他选项 :] [指定 LDTLB 指令的检查]	指定 LDTLB 指令的检查。
5 指定高速缓存关联指令的检查	CHKCache	汇编程序 < 其他 > [其他选项 :] [指定高速缓存关联指令的检查]	指定高速缓存关联指令的检查。
6 指定 DSP 关联指令的检查	CHKDsp	汇编程序 < 其他 > [其他选项 :] [指定 DSP 关联指令的检查]	指定 DSP 关联指令的检查。
7 指定 FPU 关联指令的检查	CHKFpu	汇编程序 < 其他 > [其他选项 :] [指定 FPU 关联指令的检查]	指定 FPU 关联指令的检查。
8 指定 .FDATA 的 8 字节 ALIGN 检查	CHKAlign8	汇编程序 < 其他 > [其他选项 :] [指定 .FDATA 的 8 字节 ALIGN 检查]	指定 .FDATA 的 8 字节 ALIGN 检查。

文字池自动生成功能的长度模式设定

AUTO_literal

汇编程序 < 其他 > [将文字池自动生成功能指定为长度选择模式]

格 式 AUTO_literal

说 明 指定文字池自动生成功能的长度模式。
当指定本选项时，文字池自动生成功能为长度选择模式，对于没有操作长度指定的数据传送指令（MOV #imm,Rn），汇编程序判断 imm 值的范围，如果需要就自动生成文字池。
当不指定本选项时，文字池自动生成功能为长度指定模式，对于没有操作长度指定的数据传送指令，作为 1 字节的数据传送指令进行汇编。
长度选择模式中，对没有操作长度指定的数据传送指令进行带符号的范围判断，所以 H'00000080 ~ H'000000FF（128 ~ 255）作为字长处理。

Imm 值的范围	选择的长度或者错误	
	长度选择模式	长度指定模式
H'80000000 ~ H'FFFF7FFF (-2,147,483,648 ~ -32,769)	长字	警告 835
H'FFFF8000 ~ H'FFFFFF7F (-32,768 ~ -129)	字	警告 835
H'FFFFFF80 ~ H'0000007F (-128 ~ 127)	字节	字节
H'00000080 ~ H'000000FF (128 ~ 255)	字	字节
H'00000100 ~ H'00007FFF (256 ~ 32,767)	字	警告 835
H'00008000 ~ H'7FFFFFFF (32,768 ~ 2,147,483,647)	长字	警告 835

【注】 () 内为 10 进制表示

未参照符号信息的输出抑止

Exclude**NOExclude**

汇编程序 < 其他 > [其他选项 :] [未定义外部参照符号信息的输出抑止]

格 式

Exclude
NOExclude

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。
 能通过抑止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

asmsb aaa.src -exclude
 不输出未参照的外部参照符号的符号信息。

asmsb aaa.src -noexclude
 输出未参照的外部参照符号的符号信息。

特权模式指令的检查

CHKMd

汇编程序 < 其他 > [指定特权模式指令的检查]

格 式

CHKMd

说 明

在单片机种类为 **SH-3**、**SH3-DSP**、**SH-4**、**SH-4A** 或者 **SH4AL-DSP** 的情况下指定本选项时，只有各单片机的用户模式指令才有效，如果记述了特权模式指令就输出警告 **704**。

备 注

单片机种类为 **SH3-DSP** 或者 **SH4AL-DSP** 的情况下没有 **CHKDSP** 选项的指定时，以下的特权模式指令就作为用户模式处理。

LDC	Rm, SR
LDC.L	@Rm+, SR
STC	SR, Rm
STC.L	SR, @-Rn

LDTLB 指令的检查**CHKTLb**

汇编程序 < 其他 > [指定 LDTLB 指令的检查]

格 式	CHKTLb
说 明	在单片机种类为 SH-3、SH3-DSP、SH-4、SH-4A 或者 SH4AL-DSP 的情况下指定本选项时，如果记述了 LDTLB 指令就输出警告 705。

高速缓存关联指令的检查**CHKCache**

汇编程序 < 其他 > [指定高速缓存关联指令的检查]

格 式	CHKCache
说 明	在单片机种类为 SH-3、SH3-DSP、SH-4、SH-4A 或者 SH4AL-DSP 的情况下指定本选项时，如果记述了高速缓存关联指令就输出警告 706。

DSP 关联指令的检查**CHKDsp**

汇编程序 < 其他 > [指定 DSP 关联指令的检查]

格 式	CHKDsp
说 明	在单片机种类为 SH3-DSP 或者 SH4AL-DSP 的情况下指定本选项时，如果记述了 DSP 关联指令就输出警告 707。

FPU 关联指令的检查**CHKFpu**

汇编程序 < 其他 > [指定 FPU 关联指令的检查]

格 式	CHKFpu
说 明	在单片机种类为 SH2A-FPU、SH-4 或者 SH-4A 的情况下指定本选项时，如果记述了 FPU 关联指令就输出警告 708。

FDATA 的 8 字节 ALIGN 检查***CHKAlign8***

汇编程序 < 其他 > [指定 .FDATA 的 8 字节 ALIGN 检查]

格 式 CHKAlign8

说 明 如果在单片机种类为 SH-4A 或者 SH4AL-DSP 的情况下指定本选项，就进行 .FDATA.D 的 8 字节 ALIGN 检查。

当用 .FDATA.D 指定的双精度浮点常数数据不在 8 字节 ALIGN 边界时，就输出警告 816。

3.2.5 单片机选项

表 3.5 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 单片机种类的指定	CPU = < 单片机种类 >	CPU [CPU 种类 :]	指定单片机种类
2 指定存储器的字节排列顺序	ENdian = { Big Little }	CPU [Endian 选择 :]	指定存储器的字节排列顺序。
3 指定浮点常数的舍入方式	Round = { Nearest Zero }	CPU [舍入方式 :]	指定浮点常数的舍入方式。
4 指定非正规化数的浮点常数的处理	DENormalize = { ON OFF }	CPU [非正规化数作为非正规化数处理 :]	指定浮点常数为非正规化数时的处理。

单片机种类

CPU

CPU[CPU 种类 :]

格 式 CPU = < 单片机种类 >

说 明 指定源程序汇编对象的单片机种类。
单片机种类的内容如下：

单片机种类	对象单片机
SH1	用于 SH-1
SH2	用于 SH-2
SH2E	用于 SH-2E
SHDSP	用于 SH2-DSP
SH2A	用于 SH-2A
SH2AFPU	用于 SH2A-FPU
SH3	用于 SH-3
SH3DSP	用于 SH3-DSP
SH4	用于 SH-4
SH4A	用于 SH-4A
SH4ALDSP	用于 SH4AL-DSP

备 注 和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=< 单片机种类 >	（与指定无关）	（与指定无关）	由 cpu 选项指定的单片机种类
（无指定）	.CPU < 单片机种类 >	（与指定无关）	由 .CPU 指定的单片机种类
	（无指定）	SHCPU=< 单片机种类 >	环境变量的单片机种类
		（无指定）	用于 SH-1

存储器的字节排列顺序

Endian

CPU[Endian 选择 :]

格 式 Endian = { **Big** | Little }

说 明 指定目标单片机的字节排列方法是 **Big Endian** 还是 **Little Endian**。
在省略本选项时解释为 **endian=big**。

备 注 和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）
endian=big	（与指定无关）	用 Big Endian 进行汇编。
endian=little	（与指定无关）	用 Little Endian 进行汇编。
（无指定）	.ENDIAN BIG	用 Big Endian 进行汇编。
	.ENDIAN LITTLE	用 Little Endian 进行汇编。
	（无指定）	用 Big Endian 汇编。

舍入方式

Round

CPU[舍入方式 :]

格 式 Round = { Nearest | Zero }

说 明 指定在将浮点数据控制指令记述的常数转换为目标码时的舍入方式。
当指定 Round=nearest 时，以 round to NEAREST even 舍入。
当指定 Round=zero 时，以 round to ZERO 舍入。
在省略本选项时，根据单片机种类，舍入方式如下：

单片机种类	舍入方式
SH1	round to NEAREST even
SH2	round to NEAREST even
SH2E	round to ZERO
SH2A	round to NEAREST even
SH2AFPU	round to ZERO
SHDSP	round to NEAREST even
SH3	round to NEAREST even
SH4	round to ZERO
SH3DSP	round to NEAREST even
SH4A	round to ZERO
SH4ALDSP	round to NEAREST even

备 注 在单片机种类为 SH2E 的情况下，如果选择 round to NEAREST even 的舍入方式，就对源程序中最早出现的浮点数据控制指令发出警告 818，并且以 round to NEAREST even 输出目标码。

非正规化数的处理

DENormalize

CPU[非正规化数作为非正规化数处理 :]

格 式 DENormalize = { ON | OFF }

说 明 指定对浮点数据控制指令记述的非正规化数是否为有效值。在非正规化数为有效值 (ON) 或者无效值 (OFF) 时目标码不一样。

当指定 **denormalize=on** 时, 非正规化数为有效值。

当指定 **denormalize=off** 时, 非正规化数为无效值。

在非正规化数有效时, 发生警告 **842**, 并且输出目标码。

在非正规化数无效时, 发生警告 **841**, 并且输出 **0** 的目标码。

在省略本选项时, 根据单片机种类, 非正规化数的处理如下:

单片机种类	有效 / 无效
SH1	为有效值 (ON)
SH2	为有效值 (ON)
SH2E	为无效值 (OFF)
SH2A	为有效值 (ON)
SH2AFPU	为无效值 (OFF)
SHDSP	为有效值 (ON)
SH3	为有效值 (ON)
SH3DSP	为有效值 (ON)
SH4	为无效值 (OFF)
SH4A	为无效值 (OFF)
SH4ALDSP	为有效值 (ON)

备 注 在单片机种类为 **SH2E** 或者 **SH2AFPU** 的情况下, 如果将非正规化数处理为有效值, 就对源程序中最早出现的浮点数据控制指令发出警告 **818**, 并且将非正规化数处理为有效值, 输出目标码。

3.2.6 剩余选项

表 3.6 剩余选项

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 }	汇编程序 < 其它 > [用户的指定选项 :]	更改汇编程序异常结束的错误级
2 ISO-Latin1 代码字符	LATIN1	汇编程序 < 其它 > [用户的指定选项 :]	为了在源文件内能使用 LATIN1 代码字符。
3 shift JIS 码的指定	SJIS	汇编程序 < 其它 > [用户的指定选项 :]	源文件内的日语汉字码作为 shift JIS 码处理。
4 将日语汉字码指定为 EUC	EUC	汇编程序 < 其它 > [用户的指定选项 :]	源文件内的日语汉字码作为 EUC 码处理。
5 指定输出的日语汉字码	OUtcode = { SJIS EUC }	汇编程序 < 其它 > [用户的指定选项 :]	指定输出到目标文件的日语汉字码。
6 汇编列表的行数指定	LINEs = < 行数 >	汇编程序 < 其它 > [用户的指定选项 :]	设定汇编列表的行数
7 汇编列表的位数指定	COLumns = < 位数 >	汇编程序 < 其它 > [用户的指定选项 :]	设定汇编列表的位数
8 版权输出的抑止	LOGO NOLOGO	- (nologo 总是有效)	输出版权。 抑止版权的输出。
9 子命令文件的指定	SUBcommand = < 文件名 >	-	从文件输入命令行。

错误级的变更

ABort

汇编程序 < 其他 > [用户的指定选项 :]

格 式 ABort = { Warning | Error }

说 明 更改异常结束的错误级。

当指定 abort=warning 时，如果发生警告就中断处理。

当指定 abort=error 时，如果发生错误就中断处理。

在返回给 OS 的值 ≥ 1 时，抑止目标模块的输出。

只在输出目标模块时，abort 的指定才有效。

返回给 OS 的值如下。

在省略本选项时解释为 abort=error。

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	0	0	2	1	0	0
—	≥ 1	0	2	1	2	1
—	—	≥ 1	4	1	4	1

LATIN 代码字符

LATIN1

汇编程序 < 其他 > [用户的指定选项 :]

格 式 **LATIN1**

说 明 能在字符串和注释中记述 **LATIN** 代码字符。
 latin1 选项不能和 **sjis**、**euc**、**outcode** 同时指定。

shift JIS

SJIS

汇编程序 < 其他 > [用户的指定选项 :]

格 式 **SJIS**

说 明 将字符串和注释中的日语解释为 **shift JIS** 码。
 当本选项和 **euc** 选项都省略时，就将字符串和注释中的日语解释为取决于主机的日语码。
 sjis 选项不能和 **latin1**、**euc** 同时指定。

EUC**EUC**

汇编程序 < 其他 > [用户的指定选项 :]

格 式 EUC

说 明 将字符串和注释中的日语解释为 **EUC** 码。
 当本选项和 **euc** 选项都省略时，就将字符串和注释中的日语解释为取决于主机的日语码。
 本选项不能和 **latin1**、**sjis** 同时指定。

输出的日语汉字码**OUnicode**

汇编程序 < 其他 > [用户的指定选项 :]

格 式 **Unicode = { SJIS | EUC }**

说 明 当指定 **unicode=sjis** 时，将源文件内的日语记述转换为 **shift JIS** 码，输出到目标文件。
 当指定 **unicode=euc** 时，将源文件内的日语记述转换为 **EUC** 码，输出到目标文件。
 根据本选项和源文件内的日语汉字码的指定（**sjis** 和 **euc** 选项），输出到目标文件的日语汉字码如下：

outcode 选项	源文件中的日语汉字码		
	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	默认的日语汉字码

省略本选项时的日语汉字码如下：

主机	日语汉字码
SPARC 工作站	EUC 码
HP9000/700 系列	shift JIS 码
PC	shift JIS 码

汇编列表的行数设定

LINes

汇编程序 < 其他 > [用户的指定选项 :]

格 式 **LINes** = < 行数 >

说 明 设定汇编列表的页的行数。
 行数的有效值为 20 ~ 255。
 本选项只在输出汇编列表时才有效。

备 注 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 .FORM 指定的行数。
	（无指定）	每页为 60 行。

汇编列表的位数设定

Columns

汇编程序 < 其他 > [用户的指定选项 :]

格 式 **Columns** = < 位数 >

说 明 设定汇编列表的行的位数。
 位数的有效值为 79 ~ 255。
 本选项只在输出汇编列表时才有效。

备 注 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 .FORM 指定的位数。
	（无指定）	每行为 132 位。

版权输出的抑止

LOGO**NOLOGO**

无（nologo 总是有效）

格 式 **LOGO**
 NOLOGO

说 明 控制版权的输出。
 当指定 **logo** 选项时，输出版权显示。
 当指定 **nologo** 选项时，抑止版权显示的输出。
 在省略本选项时解释为 **logo**。

子命令文件的指定

SUBcommand

无

格 式 **SUBcommand** = < 文件名 >

说 明 从文件输入命令行。
 必须按和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 **asmsh 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：
 asmsh aaa.src,bbb.src -list -noobj

备 注 整个子命令文件的大小最多为 65,535 个字节。

第 4 章 优化连接编辑程序的操作方法

4.1 选项的指定规则

4.1.1 命令行的格式

命令行的格式如下：

```
opt lnk [ { △ < 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 > [, ...] ] [ △ & ] [ ; < 注释 > ]  
& : 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

选项和子选项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 > [, ...])]	连接程序 < 输入 > [选项项目 :] [可再定位文件 / 目标文件]	指定输入文件 (在命令行指定输入文件时不需要 input)
2 库文件	LiBrary = < 文件名 > [, ...]	连接程序 < 输入 > [选项项目 :] [库文件]	指定输入库文件
3 二进制文件	Binary = <sub> [, ...] <sub> : < 文件名 > (< 段名 > [: < 调整数 >] [, < 符号名 >)	连接程序 < 输入 > [选项项目 :] [二进制文件]	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 > }	连接程序 < 输入 > [选项项目 :] [符号定义]	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 执行开始的地址	ENTry = { < 符号名 > < 地址 > }	连接程序 < 输入 > [入口点 :]	指定入口符号 指定入口地址
6 预连接程序	NOPRElink	连接程序 < 输入 > [预连接程序的控制 :]	抑止预连接程序的启动

输入文件

Input

连接程序 < 输入 > [选项项目 :] [可再定位文件 / 目标文件]

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 (*、?)，按照字母顺序展开用通配符指定的字符串。数字先于英文字母，英文大写字母先于英文小写字母。
 作为输入文件，能指定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 注 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

连接程序 < 输入 > [选项项目 :] [库文件]

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 (*、?)，按照字母顺序展开用通配符指定的字符串。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”。
 当指定 **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，在被指定为输入文件的文件之间进行连接处理后，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

连接程序 < 输入 > [选项项目 :] [二进制文件]

格 式	Binary = < 子选项 > [, ...] < 子选项 > : < 文件名 > (< 段名 > [: < 调整数 >] [, < 符号名 >]) < 调整数 > : <u>1</u> 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 的选项指定段的地址, 段不能省略。另外, 也能指定符号, 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, 可指定调整数。能给调整数指定的值为 2 的乘方, 而不能指定其他值。 当没有指定调整数时, 默认值为 1。
例	<pre>input=a.obj start=P,D*/200 binary=b.bin(D1bin),c.bin(D2bin:4,_datab)</pre> 将 b.bin 作为 D1bin 段, 从地址 0x200 开始分配。 将 c.bin 作为 D2bin 段 (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 注	在指定 form={object relocate library} 或者 strip 时, 本选项无效。 另外, 当没有指定输入目标文件时, 不能指定本选项。

符号定义

DEFine

连接程序 < 输入 > [选项项目 :] [符号定义]

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的符号。 用 16 进制数指定数值。当前头为 A ~ F 时, 先检索符号, 如果没有相应的符号, 就解释为数值; 当前头附加 0 时, 总是解释为数值。 符号名为 C/C++ 变量名时, 在程序中的定义名前附加 “_”; 符号名为 C++ 函数名时 (main 函数除外), 用双引号将包含参数串的程序中的定义名括起来指定。但是, 参数为 void 时用 “函数名 ()” 指定。
例	<pre>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。</pre>
备 注	在指定 form={object relocate library} 时, 本选项无效。

执行开始的地址

ENTry

连接程序 <输入>[入口点 :]

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的地址。
 用 16 进制数指定地址。当前头为 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；当前头附加 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 “_”；符号名为 C++ 函数名时（main 函数除外），用双引号将包含参数串的程序中的定义名括起来指定。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的地址。
 entry="init()" ; 将 C++ 的 init 函数设定为执行开始的地址。
 entry=100 ; 将 0x100 设定为执行开始的地址。

备 注 在指定 form={object | relocate | library} 或者 strip 时，本选项无效。
 在指定未参照符号的删除优化（optimize=symbol_delete）时，必须指定执行开始的地址，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

连接程序 <输入>[预连接程序控制 :]

格 式 NOPRElink

说 明 抑止预连接程序的启动。
 预连接程序支持 C++ 模板示例自动生成功能和运行时的型检查功能。在不使用 C++ 模板功能和运行时的型检查功能时，必须指定 noprelink 选项，缩短连接时间。

备 注 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 = { <u>A</u> bsolute Relocate Object Library [= {S <u>U</u> }] Hexadecimal Stype Binary }	连接程序 < 输出 > [输出格式 :]	绝对格式 可再定位格式 目标格式 库格式 Intel HEX 格式 Motorola S 格式 二进制格式
2 调试信息	DEBug SDebug NODEBug	连接程序 < 输出 > [调试信息 :]	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 }	连接程序 < 输出 > [记录大小的统一 :]	Intel HEX 记录 Intel 扩展 HEX 记录 Intel 32 位 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 >	连接程序 < 输出 > [选项项目 :] [从 ROM 映像到 RAM 的段]	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 >[,...] }	连接程序 < 输出 > [选项项目 :] [输出文件 / 信息抑止] [输出文件的分割]	指定输出文件 (可指定范围和分割输出)
6 外部符号分配信息文件	MAp [= < 文件名 >]	连接程序 < 输出 > [映像文件的输出]	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 > Random}]	连接程序 < 输出 > [选项项目 :] [空区域的输出指定] [空区域的输出]	指定空区域的输出值
8 信息消息	Message_ NOMessage [= <sub>[,...]] <sub> : < 错误号 > [- < 错误号 >]	连接程序 < 输出 > [选项项目 :] [输出文件 / 信息的抑止] [信息级消息的抑止]	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	连接程序 < 输出 > [选项项目 :] [消息输出的指定] [没被参照定义符号的通知]	通过消息输出, 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	连接程序 < 输出 > [选项项目 :] [段内数据的紧凑分配]	在紧凑编译单位之间的空区域 后分配数据
11 数据记录的字节数指定	BYte_count= < 数值 >	连接程序 < 输出 > [数据记录的长度 :]	指定数据记录的最大字节数

连接程序 < 输出 > [输出格式 :]

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。

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。 当指定 library=s 时，将输出库文件作为系统库。 当指定 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 Intel HEX 格式文件。Intel HEX 格式请参照“18.1.2 Intel HEX 格式文件”。
6 stype	输出 Motorola S 格式文件。Motorola S 格式请参照“18.1.1 Motorola S 格式文件”。
7 binary	输出二进制文件。

表 4.4 输出格式与输入文件及其他选项的关系

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
	其他	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、ps_check、start、rom、entry、output、map、hide、optimize/nooptimize、samesize、symbol_forbid、samecode_forbid、variable_forbid、function_forbid、section_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
	其他	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,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、ps_check、start、rom、entry、output、map、space、optimize/nooptimize、samesize、symbol_forbid、samecode_forbid、variable_forbid、function_forbid、section_forbid、absolute_forbid、profile、cachesize、sbr、rename、delete、define、fsymbol、stack、noprelink、record、s9*2、byte_count*3、memory、msg_unused、data_stuff、show=symbol,reference,xreference
		绝对文件	input、output、record、s9*2、byte_count*3、show=symbol,reference,xreference
5 Library	有 strip	库文件	library、output、memory*4、show=symbol,section
	有 extract	库文件	library、output、show=symbol,section
	其他	目标文件 可再定位文件	input、library、output、hide、rename、delete、replace、noprelink、memory*4、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。

*3: 只能在输出格式为 form=hexadecimal 时指定 byte_count。

*4: 在指定 hide 时不能使用。

调试信息

DEBug
SDebug
NODEBug

连接程序 < 输出 > [调试信息 :]

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 注	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

连接程序 < 输出 > [记录大小的统一 :]

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，按照各地址输出各种数据记录。
备 注	在没有指定 form=hexadecimal 或者 stype 时，本选项无效。

ROM 化支持

ROm

连接程序 < 输出 > [选项项目 :] [从 ROM 映像到 RAM 的段]

- 格 式 **ROm** = < 子选项 > [, ...]
 < 子选项 > : < ROM 段名 >=< RAM 段名 >
- 说 明 确保初始化数据区的 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 注 在指定 form={object | relocate | library} 或者 strip 时，本选项无效。

输出文件

OUpout

连接程序 < 输出 > [选项项目 :] [输出文件 / 信息抑止] [输出文件的分割]

- 格 式 **OUpout** = < 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。当 form={absolute | hexadecimal | stype | binary} 时，能指定多个文件。用 16 进制数指定地址。当前头为 A ~ F 时先检索段，如果没有相应的段，就判断为地址；当前头附加 0 时，总是解释为地址。
 在省略本选项时解释为 < 开头的输入文件名 > . < 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=binaruy : “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

连接程序 < 输出 > [映像文件的输出]

- 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开头的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在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 注 本选项只在指定 **form={absolute | hexadecimal | stype | binary}** 时才有效。

空区域输出的指定

SPace

连接程序 < 输出 > [选项项目 :] [空区域输出的指定] [空区域输出]

- 格 式 SPace [= { < 数值 > | Random }]
- 说 明 将用户指定的数据填入输出范围的存储器空区域。
作为填充数据，可指定乱数或者 16 进制数值。
根据 **output** 选项指定时的输出范围指定方法，填入空区域的方法有以下不同点：
- 输出范围：段指定
如果在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
如果在指定的范围内存在空区域，就输出指定数据
- 输出数据的长度以 1、2、4 字节为有效单位，由 **space** 选项指定的 16 进制数的个数决定。如果指定 3 字节的数据，就对高位数字进行 0 扩展，用作 4 字节的数据。另外，如果指定奇数数字，也对高位数字进行 0 扩展，用作偶数数字的输入。
当空区域的容量不是输出数据长度的倍数时，尽量输出，并发出警告消息。
- 备 注 如果没有用本选项指定数值，就不输出到空区域。
本选项只在指定 **form={ binary | stype | hexadecimal }** 选项时才有效。
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**

连接程序 < 输出 > [选项项目 :] [输出文件 / 信息的抑止] [信息级消息的抑止]

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。

当指定 **message** 选项时, 输出信息级消息。当指定 **nomessage** 选项时, 抑止信息级消息的输出。

另外, 如果指定错误号, 就能抑止指定错误号的消息输出。也能用连字符 (-) 指定要抑止的错误号范围。如果将警告和错误级消息号指定为错误号, 就假设由 **change_message** 更改为信息级, 抑止消息的输出。

在省略本选项时解释为 **nomessage**。

例

nomessage=4, 200-203, 1300

抑止 L0004、L0200 ~ L0203 和 L1300 的消息输出。

没被参照的定义符号的通知

MSg_unused

连接程序 < 输出 > [选项项目 :] [消息输出的指定] [没被参照的定义符号的通知]

格 式

MSg_unused

说 明

如果指定本选项, 就通过消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 注

- 在输入文件为 **absolute** 格式时, 本选项指定无效。
- 为了输出消息, 需要同时指定 **message** 选项。
- 在编译时, 对于 **inline** 展开的函数可能有消息输出。此时, 可通过 **static** 声明对函数定义, 来抑止消息的输出。
- 在以下某种情况下, 因为不能正确解析参照关系, 所以消息输出的信息不正确。
 - 在汇编时不指定 **goptimize** 选项, 并且在同一个文件中有向同一个段转移的情况 (只在单片机为 H8 系列时)
 - 参照同一个文件中的常数符号
 - 在编译时优化有效, 并且在调用直接从属函数时
 - 在编译时外部变量存取优化有效 (只在单片机为 SuperH 系列时)
 - 在源文件中记述 **#pragma tbr** 时直接指定偏移量的情况 (只在单片机为 SH2A/SH2A-FPU 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

连接程序 < 输出 > [选项项目 :] [段内数据的紧凑分配]

格 式 Data_stuff

说 明 在连接时，紧凑分配段内的数据。常数区、初始化数据区和未初始化数据区为本选项功能对象的段。

如果指定本选项，就在缩小由编译单位的段的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调整进行连接。通过本选项的指定，能缩小由调整产生的冗长的空区域，减小全数据段的容量。

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

< 编译后的数据段容量（SuperH 编译程序的输出例子） >

tp1.obj : 4+1+1 = 6 字节

tp2.obj : 1+3[*]+4+1 = 9 字节

< 连接后的 tp1.obj 和 tp2.obj 的数据段容量 >

1) 没有 data_stuff 指定

按照各段的调整连接目标文件（常规处理）。

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，在缩小由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，所以具体情况可能与上述例子不同。

备 注 如果在连接被指定了 SuperH 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。

在下述的任一情况下，本选项指定无效。

- 指定 form=library,object 时
- 输入绝对文件时
- 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。

数据记录的字节数指定

BYte_count

连接程序 < 输出 > [数据记录长度 :]

格 式 BYte_count=< 数值 >

说 明 本选项指定在生成 Intel-Hex 格式文件时的数据记录字节数的最大值。能对字节数指定 1 字节的 16 进制数（01 ~ FF），在不记述本选项时，以 FF 的字节最大值生成 Intel-Hex 文件。

例 byte_count=10

备 注 在生成的文件格式不是 Intel-Hex 格式（form=hex）时，本选项无效。

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	对话框菜单	指定内容
1 列表文件	LISt [= < 文件名 >]	连接程序 < 列表 > [连接列表的输出]	指定列表文件输出
2 列表内容	SHow [= <sub>[,...]] <sub> : { SYmbol Reference SEction Xreference }	连接程序 < 列表 > [列表内容 :]	符号信息 参照次数 段信息 对照表信息

列表文件

<i>LISt</i>	
连接程序 < 列表 >[连接列表的输出]	
格 式	LISt [= < 文件名 >]
说 明	指定列表文件的输出和列表文件名。 当不指定列表文件名时，建立的列表文件的文件名为输出（或者开始输出）文件同名的文件名，其扩展名在指定 form=library 或者 extract 时为 “ lbp ”，否则为 “ map ”。

列表内容

SHow

连接程序 < 列表 > [列表内容 :]

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	输出模块内的符号名一览表。
		reference	不能指定。
		section	输出模块内的段一览表。
		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	输出对照表信息。

备 注 在指定 form={ object | relocate } 时, show=reference 无效。
 在指定 form=library 时, show=xreference 无效。
 关于对照表信息的输出, 有以下限制:

- 当输出文件为 relocatable 格式且使用 data_stuff 选项时, 不输出对照表信息。
- 当输入文件为 absolute 格式时, 不输出参照侧的地址信息
- 在汇编时没有指定 optimize 选项的情况下, 不输出有关转移到同一文件内的信息 (只在单片机为 H8 系列时)。
- 不输出有关参照同一文件内的常数符号的信息。
- 在编译时优化有效的情况下, 不输出调用直接从属函数时的信息。
- 在 map 优化有效的情况下, 除了基准符号以外, 不输出变量的参照信息 (只在单片机为 SuperH 系列时)。
- 如果在源文件中记述 #pragma tbr 时直接指定偏移量, 就不输出该函数的信息 (只在单片机为 SH2A/SH2A-FPU 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

项目	命令行格式	对话框菜单	指定内容
1 优化	<u>OPTimize</u> [= <sub>[,...]] <sub>: { STring_unify SYmbol_delete Variable_access Register SAME_code SHort_format Function_call Branch SPeed SAFe } NOOPTimize	连接程序 < 优化 > [优化方法 :] [优化设定 :] [设定 :]	进行优化 统一常数 / 字符串 删除未参照的符号 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 优化转移指令 执行速度优先的优化 安全的优化 不进行优化
2 共用码长度	SAMESize = < 长度 > (省略时: <u>sames=1e</u>)	连接程序 < 优化 > [统一长度 :]	指定作为共用码统一对象的最小长度
3 配置信息	PROfile = < 文件名 >	连接程序 < 优化 > [配置文件的信息 :]	指定配置信息文件 (进行动态优化)
4 高速缓存的容量	CACHesize = <sub> <sub>: Size = < 容量 > Align = < 行容量 > (省略时: <u>ca=s=8,a=20</u>)	连接程序 < 优化 > [高速缓存的容量 :]	指定高速缓存的容量 指定高速缓存块的容量
5 优化的部分抑止	SYmbol_forbid = < 符号名 >[,...] SAMECode_forbid = < 函数名 >[,...] Variable_forbid = < 符号名 >[,...] FUnction_forbid = < 函数名 >[,...] SSection_forbid =<sub> [,...] <sub>: [< 文件名 > < 模块名 >] (< 段名 >[,...]) Absolute_forbid = < 地址 > [+ < 容量 >] [,...]	连接程序 < 优化 > [优化方法 :] [优化的部分抑止]	未参照符号删除的抑止符号 共用码统一的抑止符号 有效地利用短绝对寻址方式的抑止符号 有效地利用间接寻址方式的抑止符号 优化抑止段 优化抑止的地址范围

优化

OPTimize**NOOPTimize**

连接程序 < 优化 > [优化方法 :] [优化设定] [设定 :]

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。
 当指定 **optimize** 选项时，对编译或者汇编时指定 **goptimize** 选项的文件进行优化。
 当指定 **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 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	进行全部优化。	○	×	○	○
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++ 程序中的 const 限定型变量 • 字符串数据的初始值 / 字面常量	○	×	○	×
symbol_delete	删除 1 次也没有被参照的变量 / 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。必须在编译和汇编时指定 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，在单片机种类为 H8SX 时，也使用下述区域： H8SXN : 0x100 ~ 0x1FF H8SXM、H8SXA、H8SXX : 0x200 ~ 0x3FF 必须在编译和汇编时指定 cpu 选项。	×	×	○	○
branch	基于程序的分配信息，优化转移指令长度。如果执行其他优化 项目，就与此选项的指定无关，必定执行此选项。	○	×	○	○
speed	执行不会引起目标速度下降的优化项目。 与 optimize=string_unify、symbol_delete、variable_access、 register、short_format、branch 相同。	○	×	○	○
safe	执行不受变量属性或者函数属性限制的优化项目。 与 optimize=string_unify、register、short_format、branch 相同。	○	×	○	○

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

- 备 注
- 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`) 就无效。
 - 只在单片机种类为 H8SX 时，`optimize=short_format` 的指定才有效。
 - 在单片机种类为 SH-2A/SH2A-FPU 时，因 `optimize=register` 的功能，可能会增加代码大小。

共用码长度

SAMESize

连接程序 < 优化 > [统一长度 :]

格 式 `SAMESize = < 长度 >`

说 明 通过共用码统一优化 (`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 注 在没有指定 `optimize=same_code` 时，本选项无效。

配置信息

PROfile

连接程序 < 优化 > [配置文件的信息 :]

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优化的优先顺序。	×	×	○	○
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 SuperH 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 SuperH 的 C/C++ 程序 SHA: 用于 SuperH 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，而进行输入文件单位的移动。

备 注 在没有指定 optimize 时，本选项无效。

高速缓存的容量

CAchesize

连接程序 < 优化 > [高速缓存的容量 :]

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

说 明 指定高速缓存的容量和高速缓存块的容量。
 当指定 profile 选项时，用于转移指令的优化（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 注 在没有指定 profile 时，本选项无效。

优化的部分抑止

SYmbol_forbid
SAMECode_forbid
Variable_forbid
FUnction_forbid
SSection_forbid
Absolute_forbid

连接程序 < 优化 > [优化方法 :] [优化的部分抑止]

格 式 SYmbol_forbid = < 符号名 > [, ...]
 SAMECode_forbid = < 函数名 > [, ...]
 Variable_forbid = < 符号名 > [, ...]
 FUnction_forbid = < 函数名 > [, ...]
 SSection_forbid = < sub > [, ...]
 < sub > : [< 文件名 > | < 模块名 >] (< 段名 > [, ...])
 Absolute_forbid = < 地址 > [+ < 容量 >] [, ...]

说 明 抑止特定符号、段和地址范围的优化。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，用双引号将含有参数串的程序中的定义名括起来指定。但是，在参数为 void 时，用 “函数名 ()” 指定。各选项的含义如表 4.10 所示。

表 4.10 优化的部分抑止的选项一览表

选项	参数	含 义
symbol_forbid	函数名 变量名	抑止未参照符号的删除优化。
samecode_forbid	函数名	抑止共用码的统一优化。
variable_forbid	变量名	抑止短绝对寻址方式的有效利用优化。
function_forbid	函数名	抑止间接寻址方式的有效利用优化。
section_forbid	段名 文件名 模块名	抑止特定段的优化。如果在指定段名的同时指定输入文件名或者库模块名，除了对整个段外，还能对特定文件限定优化抑止对象。
absolute_forbid	地址 [+ 容量]	抑止地址 + 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)。
 section_forbid=(p1) ; 抑止对 p1 段的全部优化。
 section_forbid=a.obj(p1,p2) ; 抑止对 a.obj 内的 p1,p2 段的全部优化。

备 注 在不使用优化的连接处理时，本选项无效。
 在对有路径记述的输入文件进行优化抑止时，必须用 section_forbid 选项给文件名记述路径。

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 >]	连接程序 < 段 > [设定项目 :] [段]	指定段的起始地址
2 符号地址文件	FSymbol = < 段名 >[,...]	连接程序 < 段 > [设定项目 :] [符号地址文件]	输出外部定义符号地址的定义文件

段地址

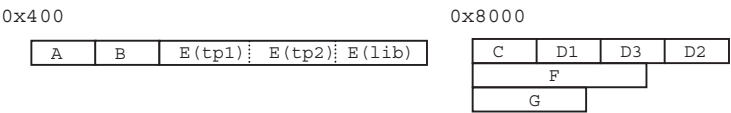
START	连接程序 < 段 >[设定项目 :][段]

格 式 START = <sub>[,...]
 <sub> : [(]< 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。
 也能用通配符 "*" 指定段名。按照输入顺序展开用通配符指定的段。
 能用冒号 ":" 分开指定段，给相同地址分配多个段（段覆盖分配）。
 在对相同地址指定分配的段之间，按照指定顺序进行分配。
 另外，通过用圆括号 "(" 括起来，能改变覆盖分配的对象段。
 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，从地址 0 开始分配。
 将没有用 start 选项指定的段分配到最后分配地址的后面。

例 按以下顺序举例表示输入目标时的段分配。
 （括号内是持有目标的段）
 tp1.obj(A,D1,E) -> tp2.obj(B,D3,F) -> tp3.obj(C,D2,E,G)

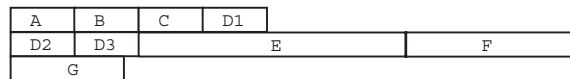
(1) -start=A,B,E/400,C,D*:F/G/8000



- 用 ":" 分开的 C, F, G 段分配在相同地址。
- 用通配符记述的段（在此是以 D 开始的段）按输入顺序分配。
- 指定的同名段（在此是 E 段）从输入的目标按顺序分配。
- 由库输入的同名段（在此是 E 段）分配在输入目标的后面。

(2) -start=A,B,C,D1,D2,D3,E,F:G/400

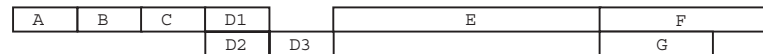
0x400



- 以紧接在 ":" 后的段（本例是 A,D2,G）为头，分别分配在相同地址。

(3) -start=A,B,C,(D1:D2,D3),E,(F:G)/400

0x400



- 在用 "()" 将相同地址的分配括起来时，以紧接在 "()" 前的段（本例是 C,E）的最后一个段为头，进行 "()" 内的相同地址的分配。
- 紧接在 "()" 后的段（在此是 E）分配在 "()" 内的最后段的下一个。

备 注

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

括号 "()" 不能嵌套记述。

在括号 "()" 内至少需要记述 1 个冒号 ":"。在不记述冒号 ":" 时不能使用括号 "()"。

在记述括号 "()" 时，不能在 "()" 外记述冒号 ":"。

在使用括号 "()" 并记述本选项时，连接程序的优化功能无效。

符号地址文件

FSymbol

连接程序 <段>[设定项目 :][符号地址文件]

格 式 `FSymbol = <段名>[, ...]`

说 明 以汇编程序的控制指令格式将指定的段内外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 注 在指定 `form={object | relocate | library}` 或者 `strip` 时，本选项无效。

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	对话框菜单	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>	连接程序 <验证> [地址一致性的检查 :]	指定 cpu 信息文件 指定可分配的段地址范围
2 物理空间的重叠检查	PS_check = <sub>[:<sub>...] <sub>:<LS>,<LS>[,...] <LS>:<开始地址> -<终端地址>	连接程序 <验证> [物理空间的重叠检查 :]	指定物理空间的重叠地址范围

*地址一致性的检查***CPu**

连接程序 <验证>[地址一致性的检查 :]

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，在不同属性之间不进行目标转移。
备 注	在指定 form={object relocate library} 或者 strip 时，本选项无效。 在单片机种类不为 SHDSP、SH2DSP、SH3DSP、SH4ALDSP 时，存储器种类的 xrom、xram、yrom、yram 指定无效。

物理空间的重叠检查

PS_check

连接程序 <验证>[物理空间的重叠检查 :]

格 式	PS_check = <sub>[:<sub>…] < sub > :<LS>,<LS>[,…] < LS> : < 起始地址> - < 结束地址>
说 明	本选项检测尽管地址值没有重叠而实际分配在存储器中出现重叠的目标。 通过使用本选项，对于 SH3 或者 SH4，能检测逻辑地址不重叠而实际分配在存储器中出现重叠的目标。 如果用本选项检测到重叠，就出错并结束连接处理。 必须给选项记述存储器中重叠的地址范围（格式中的 <LS>）。 在要检查多个物理存储器时，可用 ":" 分开记述。
例	在 SH4 的 MMU 无效时，4G 字节的地址空间映像到 512M 字节（29 位）的外部存储空间（映像时忽视 4G 字节地址的高 3 位）。 例如，能用以下记述检测将用户模式能使用的 U0 区（0x00000000 ~ 0x7fffffff）映像到外部存储空间（512M）时的重叠目标。 -PS_check=00000000-1fffffffff,20000000-3fffffffff,40000000-5fffffffff,60000000-7fffffffff 通过本选项的记述，表示在实际存储空间中地址 00000000、20000000、40000000、60000000 全部被分配在相同位置。
备 注	本选项只对 SuperH 系列的单片机有效。 在输出格式（form 选项）为 object、relocate、library 时或者在进行输入 absolute 文件的处理时，本选项无效。 有关单片机的地址空间的规格，请参照各单片机的硬件手册。

4.2.7 其他选项

表 4.13 其他种类选项一览表

项目	命令行格式	对话框菜单	指定内容
1 结束码	S9	连接程序 < 其他 > [其他选项 :] [将 S9 记录输出到末端]	总是输出 S9 记录
2 堆栈信息文件	STACK	连接程序 < 其他 > [其他选项 :] [堆栈信息文件 (sni) 的输出]	输出堆栈使用量的信息文件
3 调试信息的压缩	COmpress NOCompress	连接程序 < 其他 > [其他选项 :] [调试信息的压缩]	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]	连接程序 < 其他 > [其他选项 :] [削减装入输入文件时的存储器使用量]	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 >[,...]) } }	连接程序 < 其他 > [用户指定选项 :]	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 >[,...]) }	连接程序 < 其他 > [用户指定选项 :]	删除符号名和模块名
7 模块的置换	REPlace = <sub>[,...] <sub> : < 文件 > [(< 模块 >[,...])]	连接程序 < 其他 > [用户指定选项 :]	置换库文件中的同名模块
8 模块的抽出	EXTract = < 模块 >[,...]	连接程序 < 其他 > [用户指定选项 :]	抽出库文件中指定的模块
9 调试信息的删除	STRip	连接程序 < 其他 > [用户指定选项 :]	删除绝对文件和库文件的调试信息
10 消息级	CHange_message = <sub>[,...] <sub>: { Information Warning Error } [=< 错误号 > [< 错误号 >[,...]]	连接程序 < 其他 > [用户指定选项 :]	更改消息级
11 局部符号名的隐含指定	Hide	连接程序 < 其他 > [用户指定选项 :]	删除局部符号名的信息

结束码

S9

连接程序 < 其他 >[其他选项 :] [将 S9 记录输出到末端]

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 注 在没有指定 form=stype 时，本选项无效。

堆栈信息文件

STACK

连接程序 < 其他 >[其他选项 :] [堆栈信息文件 (sni) 的输出]

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

调试信息的压缩

Compress**N**O**C**ompress

连接程序 < 其他 >[其他选项 :][调试信息的压缩]]

格 式 C**o**mpress
 NO**C**ompress

说 明 指定是否压缩调试信息。
 当指定 **compress** 选项时，压缩调试信息。
 当指定 **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；如果指定 **nocompress** 选项，就会缩短连接时间。
 在省略本选项时解释为 **nocompress**。

备 注 在指定 **form**={**object** | **relocate** | **library** | **hexadecimal** | **stype** | **binary**} 或者 **strip** 选项时，本选项无效。

存储器使用量的削减指定

MEMory

连接程序 < 其他 >[其他选项 :][削减文件装入时的存储器使用量]]

格 式 M**E**M**o**ry = [High | Low]

说 明 指定连接时的存储器使用量。
 当指定 **memory=high** 选项时，进行通常处理。
 当指定 **memory=low** 选项时，将连接时所需的信息进行小单位的装入，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，处理速度慢于 **memory=high** 选项的指定。

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而使运行变慢，就请指定 **memory=low** 选项试一试。

备 注 如果指定下述选项，本选项的指定无效。
 同时指定 **optimize**、**compress**、**delete**、**rename**、**map**、**stack**、**replace**、**list** 和 **show**[={**reference** | **xreference**}]
 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。

备 注 在指定 `extract` 或者 `strip` 时, 本选项无效。

备 注 在指定 extract 或者 strip 时, 本选项无效。

模块的置换

REPlace

连接程序 < 其他 >[用户指定选项 :]

- 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与 **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(md11) ; 将库文件 lib1.lib 内的模块 md11 和模块 md11
 进行置换。
- 备 注 在指定 form={object | relocate | absolute | hexadecimal | stype |
 binary}、extract 或者 strip 时, 本选项无效。

模块的抽出

EXTract

连接程序 < 其他 >[用户指定选项 :]

- 格 式 EXTract = < 模块名 >[, ...]
- 说 明 抽出库模块。
 从 **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 注 在指定 form={absolute | hexadecimal | stype | binary} 或者 strip 时, 本
 选项无效。

调试信息的删除

STRip

连接程序 < 其他 > [用户指定选项 :]

格 式	STRip
说 明	删除绝对文件和库文件的调试信息。 当指定 strip 选项时，输入文件和输出文件一一对应。
例	<pre>input=file1.abs file2.abs file3.abs strip</pre> 删除 file1.abs、file2.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 注	在指定 form={object relocate hexadecimal stype binary} 时，本选项无效。

消息级

CChange_message

连接程序 < 其他 > [用户指定选项 :]

格 式	CChange_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、警告和错误级的消息级。 能指定在消息输出时是继续还是中断执行。
例	<pre>change_message=warning=2310</pre> 将 L2310 更改为警告级，并且在输出 L2310 时还继续处理。 <pre>change_message=error</pre> 将全部的信息、警告消息更改为错误级。 只要输出一个消息，就中断处理。

局部符号名的隐含指定

Hide

连接程序 <其他>[用户指定选项 :]

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。完全不影响生成文件的运行。
 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：

- 源文件：指定 **static** 型限定词的变量名和函数名等
- 源文件：**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 源文件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 注 本选项只在输出文件格式为 **absolute**、**relocate**、**library** 时才有效。
 如果输入编译和汇编时指定 **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。
 在进行外部变量存取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 >	连接程序 < 子命令文件 > [子命令文件的指定]	通过子命令文件指定选项

子命令文件

SUBcommand	连接程序 < 子命令文件 > [子命令文件的指定]
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 >[, …]] [△ &] [;< 注释 >] 也能用空格字符代替 “=”，分开指定选项和子选项。 在 input 选项的情况下，能指定空格字符来分隔子选项。 在子命令文件中，1 行指定 1 个选项。 不能将子选项记述在 1 行时，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: <code>optlnk file1.obj -sub=test.sub file4.obj</code> 指定子命令: <code>input file2.obj file3.obj ; 这是注释。</code> <code>library lib1.lib, & ; 指定继续行。</code> <code>lib2.lib</code> 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 <code>file1.obj</code>、<code>file2.obj</code>、<code>file3.obj</code>、<code>file4.obj</code>。

4.2.9 单片机选项

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 [SBR 值:]	指定 8 位绝对区域的起始地址

8 位绝对区域地址值的指定

SBr		CPU [SBR 值:]
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，就抑止 abs8 区域的优化。	
备 注	本选项只在单片机种类为 H8SX 时才有效。 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	L <code>O</code> go NO <code>L</code> Ogo	—	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

版权

<i>L</i> <code>O</code> go <i>NO</i> <code>L</code> Ogo		无（no <code>l</code> ogo 总是有效）
格 式	<u>L</u> <code>O</code> go NO <code>L</code> Ogo	
说 明	指定是否输出版权。 当指定 <code>l</code> ogo 选项时，输出版权的显示。 当指定 no <code>l</code> ogo 选项时，抑止版权显示的输出。 在省略本选项时解释为 <code>l</code> ogo。	

继续处理

END

无

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)`
 `orm=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

无

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 命令行格式

启动标准库生成工具的命令行的格式如下：

```
lbgsh [ △ < 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、子选项遵循编译程序选项。与编译程序选项的不同点如下所示。有关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，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。

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>[,...] <sub>:{ <u>ALL</u> RUNTIME CTYPE MATH MATHF STDARG STDIO STDLIB STRING IOS NEW COMPLEX CPPSTRING }	标准库 < 标准库 > [项目 :]	指定生成对象的文件 全部库函数 运行库 ctype.h+ 运行库 math.h+ 运行库 mathf.h+ 运行库 stdarg.h+ 运行库 stdio.h+ 运行库 stdlib.h+ 运行库 string.h+ 运行库 ios+ 运行库 new+ 运行库 complex+ 运行库 string+ 运行库
2 输出文件	OUTPut = < 文件名 >	标准库 < 目标 > [目标输出目录 :]	指定输出库文件名
3 简易输入 / 输出函数	NOFloat	标准库 < 目标 > [简易输入 / 输出函数]	生成简易输入 / 输出函数
4 可重入库	REent	标准库 < 目标 > [生成可重入库]	生成可重入库

头文件

Head		标准库 < 标准库 >[项目 :]	
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文件。
 有关各头文件和库函数的关系，请参照 “10.4 C/C++ 库”。运行库（runtime）总是为生成对象文件。
 在省略本选项时解释为 head=all。

例 lbgsh -output=sh2.lib -head=mathf -cpu=sh2
 通过 -cpu=sh2 编译 mathf.h 中定义的库函数和运行库，输出库文件 sh2.lib。

输出文件

OUTPut		标准库 < 目标 >[目标输出目录 :]	
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	lbgsh -output=sh2.lib -optimize=1 -speed -goptimize -cpu=sh2 通过 -optimize=1 -speed -goptimize -cpu=sh2 编译用于全部标准库的源文件， 输出库文件 sh2.lib。		

简易输入 / 输出函数

NOFloat

标准库 < 目标 > [简易输入 / 输出函数]

格 式 NOFloat

说 明 不支持浮点转换（%f、%e、%E、%g、%G），生成简易输入 / 输出函数。
当输入 / 输出不需要进行浮点转换的文件时，能减小 ROM 容量。
< 对象函数 >
fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf

备 注 如果在输入 / 输出函数中指定浮点型，就不保证在连接由本选项建立的库时的运行。

可重入库

REent

标准库 < 目标 > [生成可重入库]

格 式 REent

说 明 生成可重入库。但是，rand 和 srand 函数不可重入。

备 注 当连接可重入库时，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，或者在编译时通过 define 选项定义 _REENTRANT。

5.2.2 不可指定的选项

不能对标准库生成工具指定的选项如表 5.2 所示。如果指定就被忽视。

表 5.2 不可指定的选项一览表

项目	选项	编译程序的解释	内容
1 include 文件的文件夹	include	无	—
2 文件间 inline 展开的文件夹指定	file_inline_path	无	—
3 宏名的定义	define	无	—
4 信息级消息	message nomessage	nomessage	不输出
5 预处理器的展开	preprocessor	无	—
6 预处理器展开时的输出限制	noline	无	—
7 目标格式	code	code = machinecode	输出机器语言程序
8 调试信息	debug nodebug	nodebug	不输出
9 目标文件的输出指定	objectfile	无	—
10 模板示例的生成功能	template	无	不使用模板功能
11 列表文件	listfile nolistfile	nolistfile	不输出
12 列表内容和格式	show	无	—
13 文件间的 inline 展开	file_inline	无	—
14 注释的嵌套	comment	无	不使用注释的嵌套
15 MAC 寄存器的保证	macsave	macsave = 1	保证 MACH 和 MACL 寄存器的内容
16 消息级	change_message	无	—
17 C/C++ 语言的选择	lang	无	遵循文件扩展名
18 版权的输出抑止	logo nologo	nologo	抑止版权的输出
19 字符串内的字符码	euc sjis latin1	无	不使用字符码
20 目标码内的日语汉字转换	outcode	无	不使用字符码
21 TBR 相对函数的调用指定	tbr	无	—

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`、`division`、`endian`、`fpu`、`round`、`denormalize`、`pic`、`double=float`、`rtti`、`pack` 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（包含指定）含 `#pragma global_register` 声明的头文件。当通过综合开发环境指定时，必须通过 < 其他 >[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。

使用编辑功能，可对无法输出到栈信息文件的汇编程序（由汇编程序进行汇编）的堆栈使用量进行追加・修正以及对符号进行堆栈值的设定，并且还能算出整个系统的堆栈使用量。

作为调用信息文件 (*.cal)，可保存和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

环境变量	说明
1 path	指定执行文件的保存文件夹。 指定格式: PC 版 C> path= < 执行文件的路径名 >[< 已存在的路径名 >...] UNIX 版 C shell %set path =(< 执行文件的路径名 > \$path) Bourne shell %PATH=:< 执行文件的路径名 >[< 已存在的路径名 >...] %export PATH
2 SHC_LIB	指定编译程序的装入模块的保存文件夹。在用 PC 版的命令提示符进行编译时或者在 UNIX 版的情况下，必须指定此环境变量。 指定格式: PC 版 C> set SHC_LIB= < 执行文件的路径名 > UNIX 版 C shell %setenv SHC_LIB < 执行文件的路径名 > Bourne shell %SHC_LIB = < 执行文件的路径名 > %export SHC_LIB
3 SHCPU	通过环境变量指定由编译程序 / 汇编程序的 cpu 选项指定的单片机种类。 <CPU> SH1 SH2 SH2E SH2A SH2AFPU SH2DSP SH3 SH3DSP SH4 SH4A SH4ALDSP 在 SHCPU 环境变量的单片机指定和 cpu 选项的单片机指定不同时，输出警告消息，并优先 cpu 选项的指定。 如果在编译时指定 SHDSP，就解释为指定了 SH2DSP。 指定格式: PC 版 C> set SHCPU= <CPU> UNIX 版 C shell %setenv SHCPU <CPU> Bourne shell %SHCPU=<CPU> %export SHCPU

环境变量	说明
4 SHC_INC*	指定编译程序的 include 文件的保存文件夹。 系统 include 文件的检索顺序为 include 选项指定的文件夹、SHC_INC 指定的文件夹、系统文件夹（SHC_LIB）。 用户 include 文件的检索顺序为当前文件夹、include 选项指定的文件夹、SHC_INC 指定的文件夹、系统文件夹（SHC_LIB）。 当没有指定此环境变量时，对 UNIX 版假设 SHC_LIB，PC 版无默认值。 指定格式： PC 版 C> set SHC_INC = <include 路径名 > [;< include 路径名 > ;...] UNIX 版 C shell %setenv SHC_INC <include 路径名 > [:< include 路径名 > :...] Bourne shell %SHC_INC = <include 路径名 > [:< include 路径名 > :...] %export SHC_INC
5 SHC_TMP	指定编译程序建立临时文件的文件夹。在用 PC 版的命令提示符进行编译时，必须指定此环境变量。在 UNIX 版的情况下没有指定此环境变量时，只要指定了环境变量 TMPDIR，就将临时文件建立在 TMPDIR 指定的文件夹；如果没有指定 SHC_TMP 和 TMPDIR，就将临时文件建立在 /usr/tmp。 指定格式： PC 版 C> set SHC_TMP= <临时文件的路径名 > UNIX 版 C shell %setenv SHC_TMP <临时文件的路径名 > Bourne shell %SHC_TMP = <临时文件的路径名 > %export SHC_TMP
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文件夹。如果没有指定此环境变量，就在当前文件夹下建立临时文件。 指定格式： PC 版 C> set HLNK_TMP= <临时文件的路径名 > UNIX 版 C shell %setenv HLNK_TMP <临时文件的路径名 > Bourne shell %HLNK_TMP = <临时文件的路径名 > %export HLNK_TMP

环境变量	说明
8 HLNK_DIR*	指定优化连接编辑程序的输入文件的保存文件夹。 由 input 选项和 library 选项指定的文件检索顺序为当前文件夹、指定的文件夹。 但是，对于用通配符指定的文件，只在当前文件夹中检索。 指定格式： PC 版 C> set HLNK_DIR= < 输入文件的路径名 >[< 输入文件的路径名 > ;...] UNIX 版 C shell %setenv HLNK_DIR < 输入文件的路径名 >[< 输入文件的路径名 > :...] Bourne shell %HLNK_DIR = < 输入文件的路径名 > [< 输入文件的路径名 > :...] %export HLNK_DIR

【注】 * 当指定多个文件夹时，PC 版必须用 “;”（分号）或者 “,”（逗号）分开；UNIX 版必须用 “:”（冒号）分开。

7.2 预定义宏

编译程序按照选项的指定和版本进行如下的预定义宏。

表 7.2 预定义宏

	选项	预定义宏
1	cpu=sh1	#define _SH1
	cpu=sh2	#define _SH2
	cpu=sh2e	#define _SH2E
	cpu=sh2a	#define _SH2A
	cpu=sh2afpu	#define _SH2AFPU
	cpu=sh2dsp	#define _SH2DSP
	cpu=sh3	#define _SH3
	cpu=sh3dsp	#define _SH3DSP
	cpu=sh4	#define _SH4
	cpu=sh4a	#define _SH4A
	cpu=sh4aldsp	#define _SH4ALDSP
2	pic=1	#define _PIC
3	endian=big	#define _BIG
4	endian=little	#define _LIT
5	double=float	#define _FLT、#define __FLT__
6	fpu=single	#define _FPS
7	fpu=double	#define _FPD
8	denormalize=on	#define _DON
9	round=nearest	#define _RON
10	dspc	#define _DSPC
11	fixed_const	#define _FXD
12	—	#define __HITACHI_VERSION__ *1
13	—	#define __HITACHI__ *2
14	—	#define _SH *2
15	—	#define __RENESAS_VERSION__ *1
16	—	#define __RENESAS__ *2

【注】*1 __HITACHI_VERSION__ 和 __RENESAS_VERSION__ 的值参照如下：

- 源程序中：__HITACHI_VERSION__==aabb

aa: version 部分

bb: revision 部分

- 编译程序中的定义例子

```
#define __HITACHI_VERSION__ 0x0701 // V.7.1.00 时
```

```
#define __HITACHI_VERSION__ 0x0900 // V.9.00.00 时
```

```
#define __RENESAS_VERSION__ 0x0900 // V.9.00.00 时
```

*2 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。综合开发环境中使用的标准文件扩展名如表 8.1 所示。

表 8.1 综合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*	用于 C 程序的列表文件
5	lis、lpp*	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	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	Motorola S 格式
18	hex	Intel（扩展）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
26	bls	外部符号分配信息文件

【注】 * 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 tpdir 文件夹下的文件扩展名如表 8.2 所示。

表 8.2 tpdir 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	子选项 *1	选项省略时
1	源列表信息	源程序列表 *1	show=source show=nosource	不输出
		include 文件展开后的源程序列表 *2	show=include show=noinclude	不输出
		宏展开后的源程序列表 *2	show=expansion show=noexpansion	不输出
2	目标信息	目标程序的机器语言和汇编码	show=object show=noobject	输出
3	统计信息	错误的总数、源程序的行数、段大小、符号数	show=statistics show=nostatistics	输出
4	命令指定的信息	显示命令指定的文件名和选项	—	输出

【注】 *1 全部选项在指定 listfile 选项时有效。

*2 如果同时指定 show=object，就将源程序列表输出到目标信息中。

*3 在指定 show=source 时，include 文件和宏展开后的源程序列表有效。

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=noinclude,noexpansion 时）和输出预处理后的源程序格式（在指定 show=include,expansion 时）。输出格式分别如图 8.1、图 8.2 所示，图 8.2 中的阴影部分表示不同点。

***** SOURCE LISTING *****						
FILE NAME: m0260.c						
Seq	File	Line	0	1	2	3
1	m0260.c	1	#include	"header.h"		
4	m0260.c	2				
5	m0260.c	3	int	sum2(void)		
6	m0260.c	4	{	int j;		
7	m0260.c	5				
8	m0260.c	6	#ifdef	SMALL		
9	m0260.c	7	j=SML_INT;			
10	m0260.c	8	#else			
11	m0260.c	9	j=LRG_INT;			
12	m0260.c	10	#endif			
13	m0260.c	11				
14	m0260.c	12	return j; /* continue	123456789012345678901234567		
			+2345678901234567890 */			
15	m0260.c	13	}			
(1)	(2)	(3)	(7)			

图 8.1 show=noinclude,noexpansion 的源列表信息

***** SOURCE LISTING *****						
FILE NAME: m0260.c						
Seq	File	Line	0	1	2	3
1	m0260.c	1	#include	"header.h"		
2	header.h	1	#define	SML_INT	1	
3	header.h	2	#define	LRG_INT	100	(4)
4	m0260.c	2				
5	m0260.c	3	int	sum2(void)		
6	m0260.c	4	{	int j;		
7	m0260.c	5				
8	m0260.c	6	#ifdef	SMALL		
9	m0260.c	7	X	j=SML_INT;		
10	m0260.c	8	#else			
11	m0260.c	9	(5)	j=100;		
12	m0260.c	10	#endif			
13	m0260.c	11	(6)			
14	m0260.c	12	return j; /* continue	123456789012345678901234567		
(1)	(2)	(3)	(7)	+2345678901234567890 */		
15	m0260.c	13	}			
			(7)			

图 8.2 show=include,expansion 的源列表信息

- (1) 列表的行号
- (2) 源程序文件名或者 include 文件名
- (3) 源程序或者 include 文件中的行号
- (4) 在指定 show=include 时，进行 include 文件展开的源行
- (5) 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
- (6) 在指定 show=expansion 时，由 #define 语句进行宏替换的源行
- (7) 因源程序的行长于编译列表的行而跨行显示的源行

8.2.3 目标信息

目标信息的输出例子如图 8.3 所示。

***** OBJECT LISTING *****					
FILE NAME: m0251.c					
SCT	OFFSET	CODE	C LABEL	INSTRUCTION OPERAND	COMMENT
(1)	(2)	(3)		(4)	(5)
	m0251.c	1	extern int multipli(int);		
	m0251.c	2			
	m0251.c	3	int multipli(int x)		
P	00000000		_multipli:		; function: multipli
					; frame size=16 (6)
	00000000	4F22	STS.L	PR,R15	
	00000002	7FF4	ADD	#-12,R15	
	00000004	1F42	MOV.L	R4,@(8,R15)	
	m0251.c	4	{		
	m0251.c	5	int i;		
	m0251.c	6	int j;		
	m0251.c	7			
	m0251.c	8	j=1;		
	00000006	E201	MOV	#1,R2	
	00000008	2F22	MOV.L	R2,@R15	
	m0251.c	9	for(i=1; i<=x; i++){		
	0000000A	E301	MOV	#1,R3	
	0000000C	1F31	MOV.L	R3,@(4,R15)	
	0000000E	A009	BRA	L213	
	00000010	0009	NOP		
	00000012		L214:		
	m0251.c	10	j*=i;		
	00000012	50F1	MOV.L	@(4,R15),R0	
	00000014	61F2	MOV	@R15,R1	
	00000016	D30A	MOV.L	L216+2,R3	; __muli
	00000018	430B	JSR	@R3	
	.		.		
	.		.		

图 8.3 show=source,object 的目标信息

- (1) 各段的段名 (P、C、D、B、C\$INIT、C\$VTBL)
- (2) 从各段起始地址开始的偏移量
- (3) 各段的偏移地址的内容
- (4) 对应机器语言的汇编码
- (5) 对应程序的注释
- (6) 栈帧的大小 (字节数)

8.2.4 统计信息

统计信息的输出例子如图 8.4 所示。

```

***** STATISTICS INFORMATION *****

***** ERROR INFORMATION ***** (1)

NUMBER OF ERRORS:          0
NUMBER OF WARNINGS:        0
NUMBER OF INFORMATIONS:    0

***** SOURCE LINE INFORMATION ***** (2)

COMPILED SOURCE LINE:      13

***** SECTION SIZE INFORMATION ***** (3)

PROGRAM    SECTION(P):      0x000044 Byte(s)
CONSTANT   SECTION(C):      0x000000 Byte(s)
DATA       SECTION(D):      0x000000 Byte(s)
BSS        SECTION(B):      0x000000 Byte(s)

TOTAL PROGRAM SECTION: 00000044 Byte(s)
TOTAL CONSTANT SECTION: 00000000 Byte(s)
TOTAL DATA   SECTION: 00000000 Byte(s)
TOTAL BSS     SECTION: 00000000 Byte(s)

TOTAL PROGRAM SIZE: 0x000044 Byte(s)

***** LABEL INFORMATION ***** (4)

NUMBER OF EXTERNAL REFERENCE SYMBOLS: 1
NUMBER OF EXTERNAL DEFINITION SYMBOLS: 1
NUMBER OF INTERNAL/EXTERNAL SYMBOLS: 6

```

图 8.4 统计信息

(1) 各级消息的总数

(2) 源文件的编译行数

(3) 各段大小的合计

(4) 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

【注】 当没有指定 message 选项时，不输出各级消息 (1) 的 NUMBER OF INFORMATIONS。如果指定 noobject 选项，就在发生错误级或者致命级错误时不输出段大小的信息 (3) 和标号信息 (4)。另外，当指定 code=asmcode 时，段大小的信息 (3) 用 0 和 1 表示该段是否存在。

8.2.5 命令指定的信息

显示由启动编译程序时的命令指定的文件名和选项。命令指定的信息输出例子如图 8.5 所示。

```

*** COMMAND PARAMETER ***
-listfile test.c

```

图 8.5 命令指定的信息

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	对照表信息	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 8.6 所示。

```
PROGRAM NAME = "SAMPLE" (7)
1 1 .HEADING ""SAMPLE""
2 2 POINT .ASSIGNA 16
3 3 Parm1 .REG (R0)
4 4 Parm2 .REG (R1)
5 5 WORK1 .REG (R2)
6 6 WORK2 .REG (R3)
7 7 WORK3 .REG (R4)
8 8 WORK4 .REG (R5)
?
20 00000000 9 I1 FIX_MUL :
21 00000000 2107 10 I1 DIV0S Parm1,Parm2
22 00000002 0229 11 I1 MOVT WORK1
23 00000004 4011 12 I1 CMP/PZ Parm1
24 00000006 8900 13 I1 BT MUL01
25 00000008 600B 14 I1 NEG Parm1,Parm1
(1) (2) (3) (4) (5) (6)
?
231 ***** BEGIN-POOL *****
232 00000180 00018000 DATA FOR SOURCE-LINE 17
233 00000184 00024000 DATA FOR SOURCE-LINE 18
234 00000188 00030000 DATA FOR SOURCE-LINE 19
235 0000018C 00050000 DATA FOR SOURCE-LINE 20
236 ***** END-POOL *****
237 35 .END
****TOTAL ERRORS 0
****TOTAL WARNINGS 0
(9)
```

(8)

图 8.6 源列表

- (1) 是行号（10 进制）。
- (2) 是位置计数器的值（16 进制）。
- (3) 是目标码（16 进制），当是 .RES、.SRES、.SRESC、.SRESZ、.FRES 中的某一个操作时，以字节单位表示确保的区域容量。
- (4) 是源行号（10 进制）。
- (5) 展开区分
表示文件包含、带条件汇编功能和宏功能展开的区分。
In 文件包含（n 表示包含的嵌套层）
C 带条件汇编的成立、重复展开、带条件的重复展开
M 宏展开
- (6) 是源语句。
- (7) 是由 .HEADING 设定的头文件。
- (8) 是文字池。
- (9) 是错误和警告的总数，错误消息输出到出现错误的源语句的下一行。

8.3.3 对照表

输出对照表信息。对照表信息的输出例子如图 8.7 所示。

*** CROSS REFERENCE LIST									
NAME	SECTION	ATTR	VALUE	SEQUENCE					
FIX_DIV	SAMPLE		00000088	91*	223				
FIX_MUL	SAMPLE		00000000	19*	218				
MUL01	SAMPLE		0000000A	23	25*				
MUL02	SAMPLE		00000010	26	28*				
MUL03	SAMPLE		00000082	87	89*				
Parm1		REG		3*	20	22	24	24	
					28	29	29	31	32
					32	35	36	36	38
					40	45	49	55	57
					59	61	63	65	67
					69	71	73	75	77
					79	81	83	85	88
					88	93	94	99	101
Parm2		REG		4*	20	25	27	27	
					28	31	33	33	35
					38	41	43	44	46
					48	54	56	58	60
					62	64	66	68	70
(1)	(2)	(3)	(4)	(5)					

图 8.7 对照表列表

- (1) 是符号名。
- (2) 是包含符号的段名（段名的最初 8 个字符）。
- (3) 是符号的属性。
 - EXPT 外部定义符号
 - IMPT 外部参照符号
 - SCT 段名
 - REG 由 .REG 汇编程序控制指令定义的符号
 - FREG 由 .FREG 汇编程序控制指令定义的符号
 - ASGN 由 .ASSIGN 汇编程序控制指令定义的符号
 - EQU 由 .EQU 汇编程序控制指令定义的符号
 - MDEF 双重定义符号
 - UDEF 未定义符号
 - 不显示 其他的符号
- (4) 是符号的值。
- (5) 是定义或参照符号的列表行号（10 进制），行号后的星号（*）表示定义行。

8.3.4 段信息列表

输出段信息。段信息的输出例子如图 8.8 所示。

*** SECTION DATA LIST			
SECTION	ATTRIBUTE	SIZE	START
SAMPLE	REL-CODE	000000190	
(1)	(2)	(3)	(4)

图 8.8 段信息

- (1) 是段名。
- (2) 段的种类：
 - REL 相对地址段
 - ABS 绝对地址段
 - CODE 代码段
 - DATA 数据段
 - STACK 堆栈段
 - DUMMY 虚设段
- (3) 是段的大小（16 进制，以字节为单位）
- (4) 是绝对地址段的起始地址

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	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	show=symbol	不输出
		在指定 show=reference 时，还显示各符号的参照次数和是否进行优化	show=reference	不输出
5	符号删除优化信息	显示优化删除的符号	show=symbol	不输出
6	变量存取优化对象的符号信息	显示在 8 位 /16 位绝对寻址方式的参照次数	show=reference	不输出
7	函数存取优化对象的符号信息	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】 * 全部选项只在指定 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 8.9 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)
-list
-show
```

} (1)

图 8.9 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 8.10 所示。

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

图 8.10 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 8.11 所示。

```
*** Mapping List ***
```

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

图 8.11 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) 显示段的调整数。

8.4.5 符号信息

当指定 `show=symbol` 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` 时，还输出各符号的参照次数和是否进行优化。符号信息的输出例子如图 8.12 所示。

*** Symbol List ***					
SECTION=(1)					
FILE=(2)					
	<u>START</u>	<u>END</u>	<u>SIZE</u>		
	(3)	(4)	(5)		
<u>SYMBOL</u>	<u>ADDR</u>	<u>SIZE</u>	<u>INFO</u>	<u>COUNTS</u>	<u>OPT</u>
(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 8.12 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) 符号种类显示如下：

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）被删除的符号长度和种类。符号删除优化信息的输出例子如图 8.13 所示。

```
*** Delete Symbols ***

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

图 8.13 符号删除信息的输出例子（连接列表）

- (1) 显示被删除的符号名。
- (2) 显示被删除的符号长度。
- (3) 被删除的符号种类显示如下：

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 变量存取优化对象的符号信息

不输出面向 SuperH RISC engine 单片机的本信息（图 8.14）。

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

SYMBOL                               SIZE   COUNTS  OPTIMIZE

*** Variable Accessible with Abs16 ***

SYMBOL                               SIZE   COUNTS  OPTIMIZE
```

图 8.14 变量存取优化对象符号信息的输出例子（连接列表）

8.4.8 函数存取优化对象的符号信息

不输出面向 SuperH RISC engine 单片机的本信息（图 8.15）。

```
*** Function Call ***

SYMBOL                               COUNTS  OPTIMIZE
```

图 8.15 函数存取优化对象符号信息的输出例子（连接列表）

8.4.9 对照表信息

输出符号的参照信息（对照表信息）。对照表信息的输出例子如图 8.16 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 8.16 对照表信息的输出例子（连接列表）

- (1) 是 Unit 号，以目标为单位的识别号。
- (2) 是目标名，为连接时的输入指定顺序。
- (3) 是符号名，按段的升序输出。
- (4) 是符号的分配地址。
在指定 form=rel 时，为从段起始地址开始的相对值。
- (5) 表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	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
		当指定 show=symbol 时，还显示模块内的符号名一览表	show=symbol	不输出
		当指定 show=section 时，还显示各模块内的段名和符号名一览表	show=section	不输出

【注】 * 全部选项只在指定 list 选项时才有效。

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 8.17 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
output  test.lib
-list
-show
```

(2)

(1)

图 8.17 选项信息的输出例子（库列表）

- (1) 按照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。

8.5.3 错误信息

输出错误信息。错误信息的输出例子如图 8.18 所示。

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

图 8.18 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 8.19 所示。

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

图 8.19 库信息的输出例子（库列表）

- (1) 显示库名。
- (2) 显示单片机名。
- (3) 显示 Endian 种类。
- (4) 显示库文件的属性是系统库还是用户库。
- (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。

当指定 show=symbol 时，输出模块内符号名一览表；当指定 show=section 时，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 8.20 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 8.20 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，是分配存储器的最小单位，具有以下性质：

- 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
 - 定义在开始执行程序时有无初始值。有初始值的数据和没有初始值的数据不能混在同一个段内。只要有一个初始值，没有初始值的区域就被初始化为 0。
- 写操作
 - 表示是否可以在执行程序时进行写操作。
- 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		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	4byte*2	保存机器语言	
2 常数区	C*1	data	相对 方式	有 不可	4byte	保存 const 型的数据	
3 初始化的数据区	D*1	data	相对 方式	有 可	4byte	保存有初始值的数据	
4 未初始化的数据区	B*1	data	相对 方式	无 可	4byte	保存没有初始值的数据	
5 X 存储器常数区	\$XC	data	相对 方式	有 不可	4byte	将 const 型的数据保存到 X 存储器	
6 Y 存储器常数区	\$YC	data	相对 方式	有 不可	4byte	将 const 型的数据保存到 Y 存储器	
7 X 存储器初始化数据区	\$XD	data	相对 方式	有 可	4byte	将有初始值的数据保存到 X 存储器	
8 Y 存储器初始化数据区	\$YD	data	相对 方式	有 可	4byte	将有初始值的数据保存到 Y 存储器	

名称	段		方式 种类	初始值		调整数	内容
	名称	属性		写操作			
9	X 存储器未初始化数据区	\$XB	data	相对方式	无 可	4byte	将没有初始值的数据保存到 X 存储器
10	Y 存储器未初始化数据区	\$YB	data	相对方式	无 可	4byte	将没有初始值的数据保存到 Y 存储器
11	GBR 段	\$G0	data	相对方式	有 可	4byte	保存 #pragma gbr_base 指定的有初始值的数据 在没有初始值时, 保存 0
12	GBR 段	\$G1	data	相对形式	有 可	4byte	保存 #pragma gbr_base1 指定的有初始值的数据 在没有初始值时, 保存 0
13	C++ 初始处理 / 后处理的数据区	C\$INIT	data	相对方式	有 不可	4byte	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
14	C++ 虚拟函数表区	C\$VTBL	data	相对方式	有 不可	4byte	保存在类声明中有虚拟函数时调用虚拟函数的数据
15	堆栈区	S	stack	相对方式	无 可	4byte	执行程序时所需的区域 请参照 “9.2.1(2) 动态区的分配”
16	堆区	—	—	相对方式	无 可	—	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”
17	TBR 表的区域	\$TBR	data	相对方式	有 不可	4byte	保存以 TBR 相对调用函数的数据
18	绝对地址变量区	\$ADDRESS \$<section> <address>*3	data	绝对方式	有 / 无 可 / 不可 *4	—	保存 #pragma address 指定的变量

【注】 *1 能通过 section 选项或者扩展名 #pragma section 转换段名。

*2 在指定 align16 选项时为 16byte, 在指定 align32 选项时为 32byte。

*3 <section> 是 C、D、B 的段名, <address> 为绝对地址值。

*4 初始值和写操作取决于 <section> 的属性。

例 1：用程序例子表示 C 程序和编译程序生成段的对应。

<pre>int a=1; char b; const int c=0; void main(){ ... }</pre>	程序区(main()){...})	P
	常数区(c)	C
	初始化的数据区(a)	D
	未初始化的数据区(b)	B
C程序	编译程序生成的区域和保存的数据	段名

例 2：用程序例子表示 C++ 程序和编译程序生成段的对应。

<pre>class A{ int m; A(int p); ~A(); }; A a(1); int b; extern const char c='a'; int d=1; void f(){...}</pre>	程序区(f()){...})	P
	常数区(c)	C
	初始化的数据区(d)	D
	未初始化的数据区(a,b)	B
	初始处理/后处理数据区 (&A::A、&A::~~A)	C\$INIT
C++程序	编译程序生成的区域和保存的数据	段名

9.1.3 汇编程序的段

在汇编程序中，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“11.4 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=<调整数>
                绝对寻址方式段: LOCATE=<地址值>
```

例：表示汇编程序的段的声明例子。

```
.CPU      SH2
.OUTPUT   DBG
SIZE:     .EQU      8

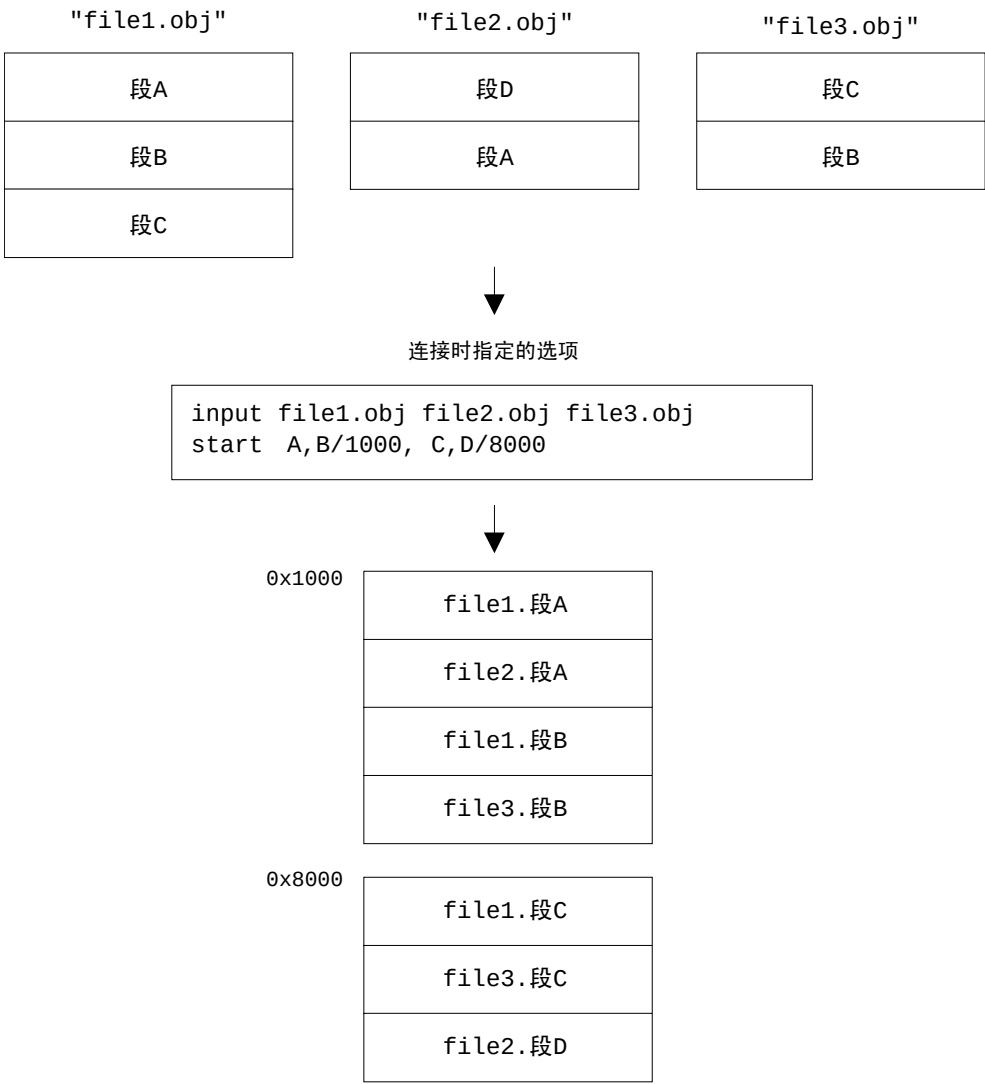
        .SECTION   A, CODE, ALIGN=4                ; (1)
START:
        MOV.L      LITERAL, R0
        MOV.L      LITERAL+4, R1
        MOV.L      #SIZE, R2
LOOP:
        CMP/PL     R2
        BF         EXIT
        MOV.B      @R0+, R3
        MOV.B      R3, @R1
        ADD        #-1, R2
        ADD        #1, R1
        BRA        LOOP
        NOP
EXIT:
        SLEEP
        NOP
LITERAL:
        .DATA.L    CONST
        .DATA.L    DATA
;
        .SECTION   B, DATA, LOCATE=H'00002000      ; (2)
CONST:
        .DATA.B    H'01, H'02, H'03, H'04, H'05, H'06, H'07, H'08
;
        .SECTION   C, STACK, ALIGN=4                ; (3)
DATA:
        .RES.B     8
        .END
```

- (1) 声明段名 A、调整数 4 和相对寻址方式的 code 段。
- (2) 声明段名 B、分配地址 H'2000 和绝对寻址方式的 data 段。
- (3) 声明段名 C、调整数 4 和相对寻址方式的 stack 段。

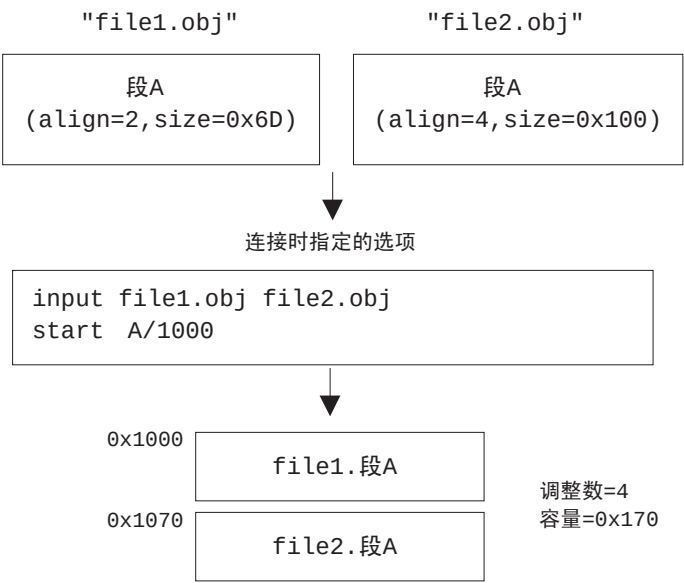
9.1.4 段的连接

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

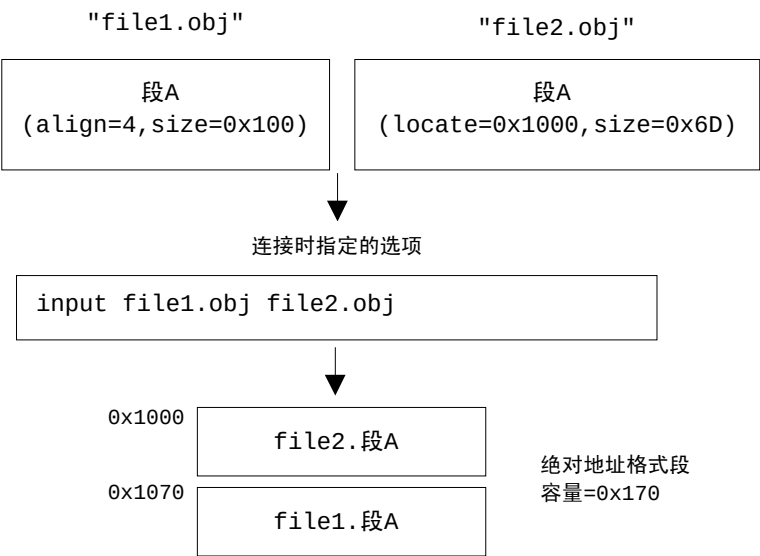
(1) 按照文件的输入顺序连续分配不同文件的同名段。



(2) 在调整后连接不同调整数的同名段。段的调整数以大的为准。

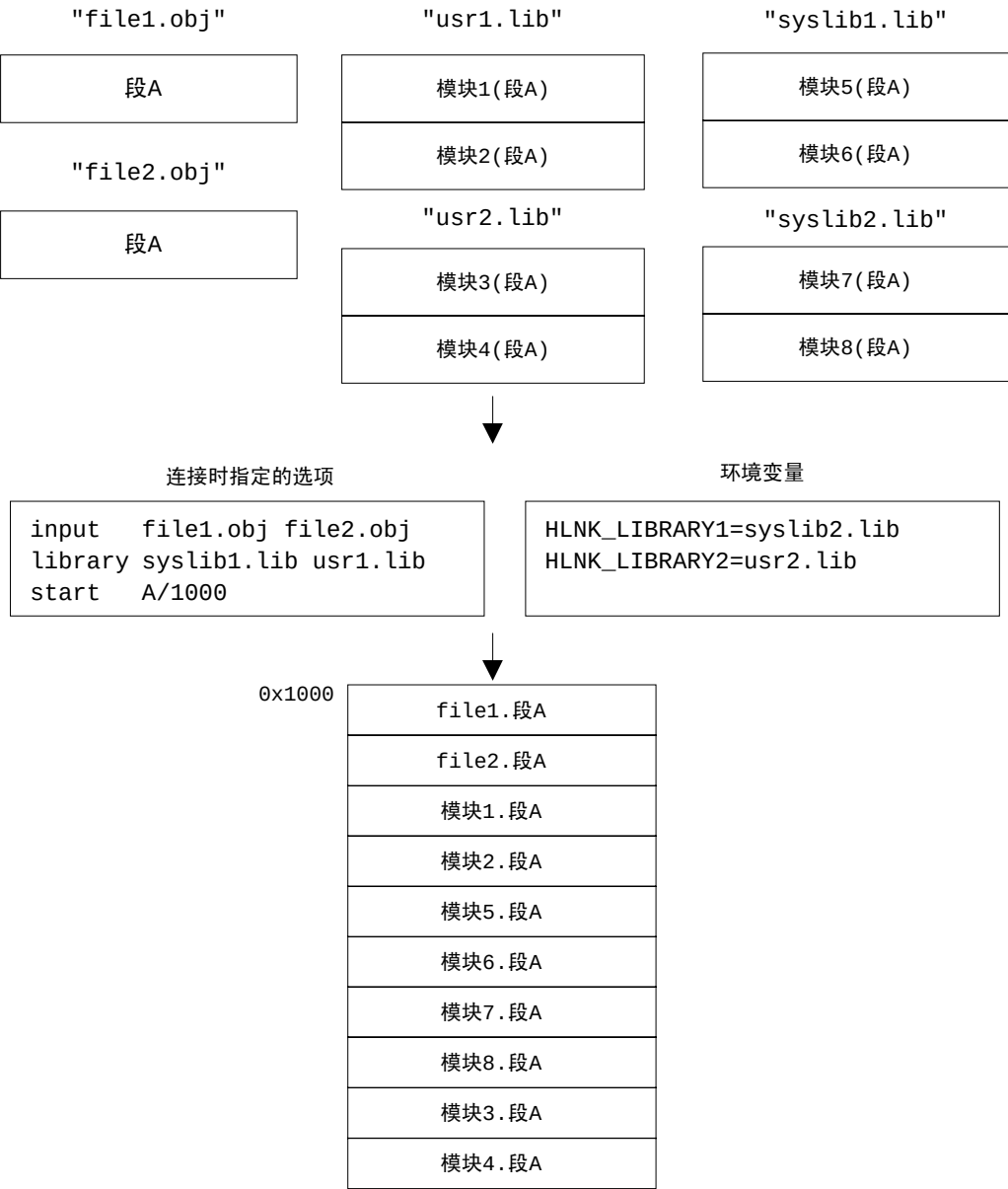


(3) 在同名段含有绝对寻址方式和相对寻址方式时，将相对寻址方式的目标连接到绝对寻址方式的目标后面。即使在指定可再定位文件的输出（指定 form=relocate），该段也变为绝对寻址方式段。



(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



9.2 初始设定程序的建立

本章说明将程序嵌入到采用 SuperH RISC engine 单片机的系统的方法。

要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 的存储区。
- 程序执行环境的设定
程序执行环境的设定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，当使用输入 / 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（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 C 库函数（exit、atexit、abort 函数）进行程序结束处理时，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和设定程序的实例。

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和结束处理函数的建立例子。

9.2.1 存储区的分配

为了将本编译程序输出的目标程序嵌入系统，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。

C/C++ 程序使用的存储区有静态分配区（如同与 C/C++ 程序中的函数对应的机器语言以及由外部数据定义或者静态数据成员声明的数据区）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、初始化的数据区、未初始化的数据区、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大小和 C/C++ 程序使用的库函数大小的总和。

因为在连接目标程序后，将包含库的各段大小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 (1)	START (2)	END (3)	SIZE (4)	ALIGN (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、汇编单位的段大小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和汇编列表中的段信息例子分别如图 9.2 和图 9.3 所示。

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

PROGRAM	SECTION(P)	:0x00004A Byte(s)
CONSTANT	SECTION(C)	:0x000018 Byte(s)
DATA	SECTION(D)	:0x000004 Byte(s)
BSS	SECTION(B)	:0x000004 Byte(s)
TOTAL PROGRAM	SECTION:	0000004A Byte(s)
TOTAL CONSTANT	SECTION:	00000018 Byte(s)
TOTAL DATA	SECTION:	00000004 Byte(s)
TOTAL BSS	SECTION:	00000004 Byte(s)
TOTAL PROGRAM SIZE: 0x00006A 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，文件单位的段大小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 语言规格规定的库函数和面向内部函数的 C++ 类库以外，包含 C/C++ 程序执行时所必需的进行算术运算的例程（运行库）。因此，必须注意：即使在 C/C++ 程序中不使用库函数，也可能需要标准库。

因为作为外部参照符号，C/C++ 程序使用的运行库被输出到本编译程序生成的汇编程序（指定 code=asmcode），所以能通过参照其符号名得知 C/C++ 程序使用的运行库名。具体例子如下所示：

- C/C++ 程序例子

```
f(int a, int b)
{
    a /= b;
    return a;
}
```

- 在 C 编译时生成的汇编程序例子

```
.IMPORT    _ _divls          ; (运行库的外部参照声明)
.EXPORT    _f
.SECTION   P, CODE, ALIGN=4

_f:
                                ;function: f
                                ;frame size=4

    STS.L    PR, @-R15
    MOV      R5, R0
    MOV.L    L218, R3          ; _ _divls
    JSR      @R3
    MOV      R4, R1
    LDS.L    @R15+, PR
    RTS
    NOP

L218:
    .DATA.L   _ _divls
    .END
```

(c) ROM 和 RAM 的分配

在对程序进行 ROM 化时，根据段是否有初始值、能否进行写操作来决定是分配到 ROM 还是分配到 RAM。

在对 C/C++ 程序的各段进行 ROM 化时，进行如下的分配（ROM 和 RAM）：

- 程序区 (段 P) →ROM
- 常数区 (段 C、\$G0、\$G1*3) →ROM
- 未初始化的数据区 (段 B、\$G0、\$G1*3) →RAM
- 初始化的数据区 (段 D、\$G0、\$G1*3) →ROM、RAM(参照 (d))
- 初始处理 / 后处理的数据区 *1 (段 C\$INIT) →ROM
- 虚拟函数表区 *2 (段 C\$VTBL) →ROM

【注】 *1 在 C++ 程序中有全局类目标时，由编译程序生成。
*2 在 C++ 程序中有虚拟函数声明时，由编译程序生成。
*3 \$G0 和 \$G1 只能分配到这些区域中的某 1 个区域。

(d) 初始化数据区的分配

如同初始化的数据区，在连接时将有初始值并能在程序执行时更改值的段保存到 ROM，而在程序开始执行时将其复制到 RAM。因此，需要用优化连接编辑程序的 rom 选项双重确保 ROM 和 RAM 中的区域。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)” 中说明将值从 ROM 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。

静态区的分配例子如图 9.4 所示。

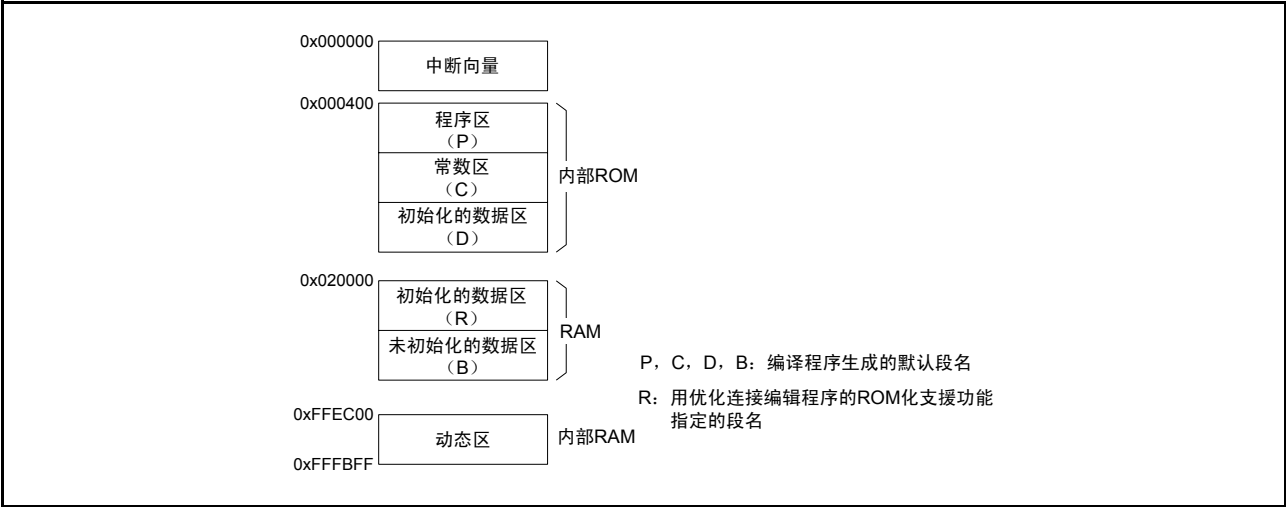


图 9.4 静态区的分配例子

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 D 相同大小的段 R 确保到输出装入模块，在参照分配在段 D 的符号时进行再定位，使其变为段 R 中的地址。段 D 和段 R 分别为 ROM 和 RAM 中的初始化数据的段名。

[2] 将段 P、C、D 分配到从内部 ROM 地址 0x400 开始的连续区域，并且将段 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 C/C++ 程序和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”。

对于用 V6 版本为止的汇编程序汇编的程序，栈解析工具无法计算其堆栈使用量。请参考以下的 C/C++ 程序的堆栈使用量计算方法，计算汇编程序的堆栈使用量，并且加上用栈解析工具计算的堆栈使用量。

• C/C++ 程序的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。

◆ 各函数使用的堆栈区

能从编译程序输出的目标列表中的 frame size 得知各函数使用的堆栈区。

目标列表和堆栈分配的具体例子如下所示，说明该堆栈使用量的计算方法。

例：表示以下 C 程序的目标列表和堆栈使用量的计算方法，C++ 程序也同样。

```
extern int h(char, int *, double );
int h(char a, register int *b, double c)
{
    char *d;

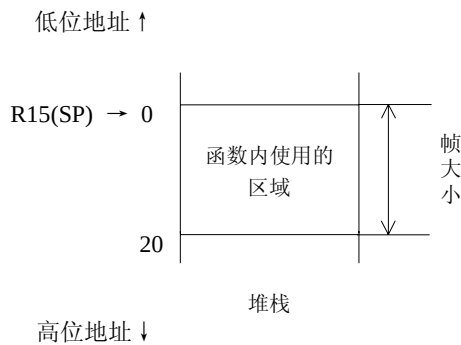
    d= &a;
    h(*d,b,c);
    {
        register int i;

        i= *d;
        return i;
    }
}
```

***** OBJECT LISTING *****

FILE NAME: m0251.c

SCT	OFFSET	CODE	C LABEL	INSTRUCTION	OPERAND	COMMENT
P						
	00000000		_h:			;function: h
						;frame size=20
	00000000	2FE6		MOV.L	R14,@-R15	
	00000002	4F22		STS.L	PR,@-R15	
				:		



函数使用的堆栈区容量与帧大小的值相同，因此在上述例子中函数 h 使用的堆栈区容量为目标列表中 COMMENT 项的 frame size 值（20 字节）。

有关分配到堆栈中的参数区的参数，请参照“9.3.2(4) 有关设定和参照参数、返回值的规则”。

◆ 堆栈使用量的计算方法

从函数调用关系计算使用的堆栈区容量。

例：函数调用关系和各函数的堆栈使用量的例子如图 9.5 所示。

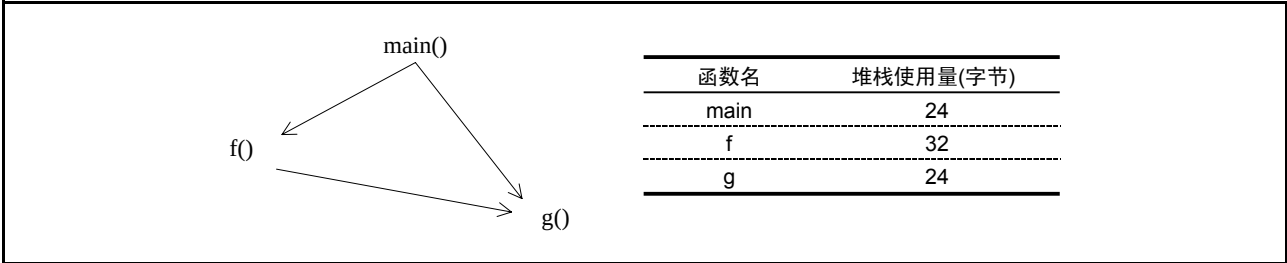


图 9.5 函数调用关系和堆栈使用量的例子

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。

表 9.2 堆栈容量的计算例子

调用路径	堆栈容量的合计 (字节)
main(24)→f(32)→g(24)	80
main(24)→g(24)	48

需要通过此方法，对调用级最深的函数计算堆栈区容量，并且至少分配该最大值（此时为 80 个字节）的堆栈区。

【注】 在 C/C++ 程序中进行递归调用时，必须在计算递归调用次数的最大值后，将该函数的堆栈区容量乘上递归调用次数。

(c) 堆区

堆区中使用的存储区容量是 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，在每次调用时，存储器管理库函数将 4 个字节用作管理区，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(4) C/C++ 库函数的初始设定（`_INITLIB`）”，作为堆区确保的区域容量（`HEAPSIZE`）计算如下：

$$\text{HEAPSIZE} = \text{\_sbrk\_size} \times n \quad (n \geq 1)$$

(由存储器管理库分配的区域容量) + 管理区容量 ≤ `HEAPSIZE`

在内部处理中，输入 / 输出库函数使用存储器管理库函数。输入 / 输出中分配的区域容量为：
516 字节 × (同时打开的文件数的最大值)

【注】 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` 或者 `deletefree` 函数释放的区域，但是根据重复分配，即使空区域的总容量很充足，也被分割到空区域小的区域，所以有可能发生不能确保所需新的大容量区域的情况。为了避免此情况，必须按以下的注意事项使用堆区：

- 1. 必须尽量在程序执行开始后立即确保大容量的区域。
- 2. 必须尽量固定释放后重新利用的数据区容量。

- 动态区的分配方法

动态区分配在 RAM 中。

通过将堆栈区的最高位地址（作为栈指针 SP）设定到向量表，决定堆栈区的分配地址。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的中断运行和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 不同，需要中断处理程序。

通过低级接口例程（sbrk）的初始设定决定堆区的分配地址。

请分别参照“9.2.2(1) 向量表的设定（VEC_TBL）”和“9.2.2(6) 低级接口例程”。

9.2.2 执行环境的设定

本节说明有关程序执行时所需环境的设定处理。但是，程序的执行环境因各用户系统而不同，所以需要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。

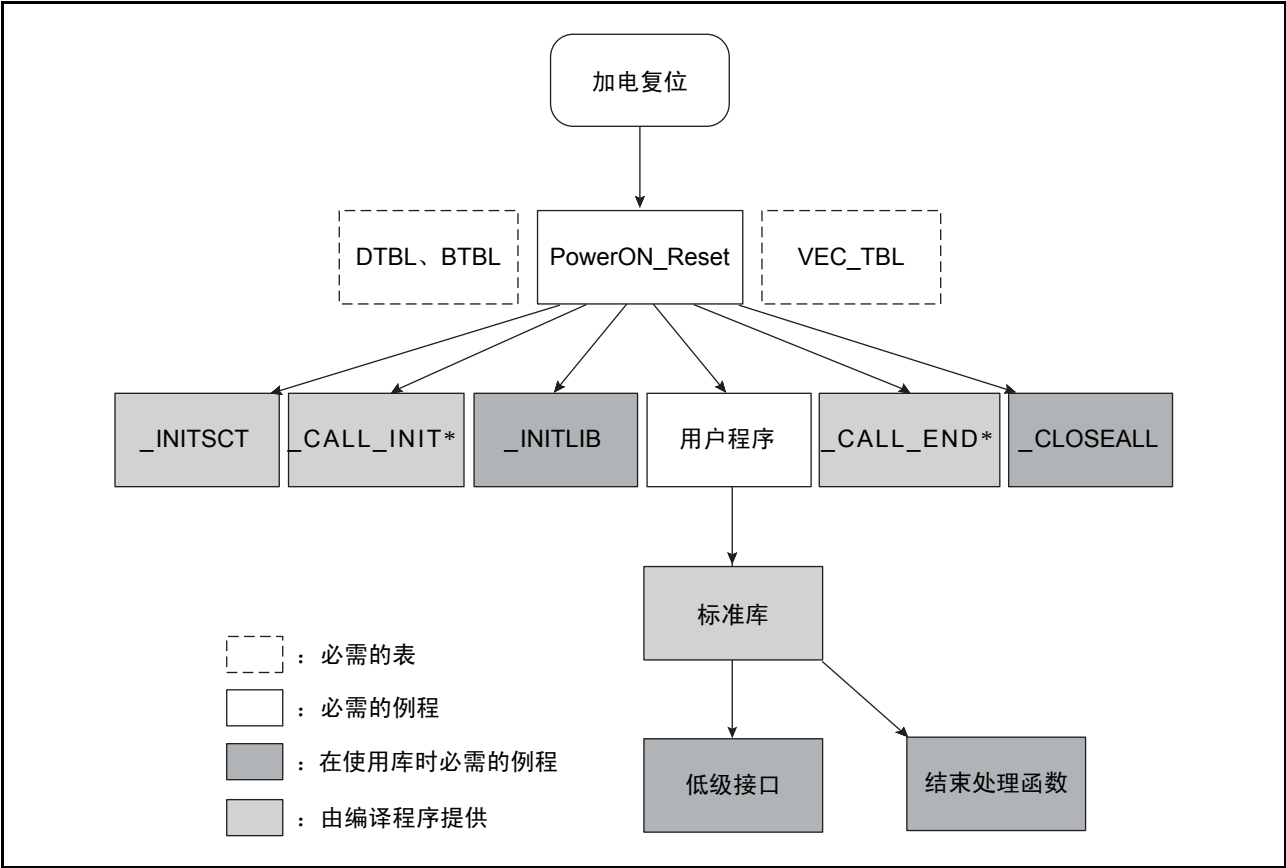


图 9.6 程序构成的例子

【注】 * 在有全局类目标声明时所需的处理

各构成例程的内容如下：

- 向量表的设定（VEC_TBL）
为了通过加电复位启动寄存器的初始设定程序（PowerON_Reset）并给栈指针（SP）设定值，进行向量表的设定。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的中断运行和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 不同，不要中断处理程序。
- 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（未初始化的数据区）进行初始化，并且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）
在库函数中，对需要初始设定的内容进行初始设定。尤其是进行标准输入 / 输出的准备。
- 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）时所需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。

*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，需要建立 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地址 0，并且为了设定栈指针（SP），给地址 4 设定堆栈区的最高位地址。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的中断运行和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 不同，不要中断处理程序。

另外，在用户系统使用中断处理时，也必须在本例程中进行中断向量的设定。编程的例子如下所示：

例 1: 用于 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 的向量表例子

```
#pragma interrupt (IRQ0)

extern void PowerON_Reset_PC(void);
extern void PowerON_Reset_SP(void);
extern void Manual_Reset_PC(void);
extern void Manual_Reset_SP(void);

extern void IRQ0(void);

#pragma section VECTTBL /* 根据 #pragma section 声明将 RESET_Vectors 输出 */
                        /* 到 CVECTTBL 段 */
                        /* 指定: 在连接时通过 start 选项将 CVECTTBL 段分配到 */
                        /* 地址 0x0 */
void (*const RESET_Vectors[])(void)={
    (void*) PowerON_Reset_PC,
    (void*) PowerON_Reset_SP,
    (void*) Manual_Reset_PC,
```

```

        (void*) Manual_Reset_SP
    };

#pragma section VECT2      /* 根据 #pragma section 声明将 vec_table2 输出      */
                           /* 到 CTECT2 段                                */
                           /* 指定: 在连接时通过 start 地址将 CTECT2 段分配到      */
                           /* 指定的地址                                    */

void (*const vec_table2[])(void)={IRQ0};

```

例 2: 在程序中使用存储体 0 时的中断处理程序的例子 (SH7708)

```

////////////////////////////////////
;                               env.inc                               ;
////////////////////////////////////

EXPEVT:
        .EQU      H'FFFFFFD4

INTEVT:
        .EQU      H'FFFFFFD8

////////////////////////////////////
;                               vect.inc                               ;
////////////////////////////////////

SR_Init:
        .EQU      B'0000000000000000000000000000000011110000

;<<VECTOR DATA START (POWER ON RESET)>>
        ;H'000 Power On Reset
        .GLOBAL   PowerON_Reset
;<<VECTOR DATA END (POWER ON RESET)>>
;<<VECTOR DATA START (MANUAL RESET)>>
        ;H'020 Manual Reset
        .GLOBAL   Manual_Reset
;<<VECTOR DATA END (MANUAL RESET)>>
        ;H'040 TLB miss/invalid (load)
        .GLOBAL   INT_TLBMiss_Load
        ;H'060 TLB miss/invalid (store)
        .GLOBAL   INT_TLBMiss_Store
        ;H'080 Initial page write
        .GLOBAL   INT_TLBInitial_Page
        ;H'0A0 TLB protect (load)
        .GLOBAL   INT_TLBProtect_Load
        ;H'0C0 TLB protect (store)
        .GLOBAL   INT_TLBProtect_Store
        ;H'0E0 Address error (load)

```

```
.GLOBAL    INT_Address_load
;H'100 Address error (store)
.GLOBAL    INT_Address_store
;H'120 Reserved
.GLOBAL    INT_Reserved1
;H'140 Reserved
.GLOBAL    INT_Reserved2
;H'160 TRAPA
.GLOBAL    INT_TRAPA
;H'180 Illegal code
.GLOBAL    INT_Illegal_code
;H'1A0 Illegal slot
.GLOBAL    INT_Illegal_slot
;H'1C0 NMI
.GLOBAL    INT_NMI
;H'1E0 User breakpoint trap
.GLOBAL    INT_User_Break
;H'200 External hardware interrupt
.GLOBAL    INT_Extern_0000
;H'220 External hardware interrupt
.GLOBAL    INT_Extern_0001
;H'240 External hardware interrupt
.GLOBAL    _INT_Extern_0010
;H'260 External hardware interrupt
.GLOBAL    _INT_Extern_0011
;H'280 External hardware interrupt
.GLOBAL    _INT_Extern_0100
;H'2A0 External hardware interrupt
.GLOBAL    _INT_Extern_0101
;H'2C0 External hardware interrupt
.GLOBAL    _INT_Extern_0110
;H'2E0 External hardware interrupt
.GLOBAL    _INT_Extern_0111
;H'300 External hardware interrupt
.GLOBAL    _INT_Extern_1000
;H'320 External hardware interrupt
.GLOBAL    _INT_Extern_1001
;H'340 External hardware interrupt
.GLOBAL    _INT_Extern_1010
;H'360 External hardware interrupt
.GLOBAL    _INT_Extern_1011
;H'380 External hardware interrupt
.GLOBAL    _INT_Extern_1100
;H'3A0 External hardware interrupt
.GLOBAL    _INT_Extern_1101
;H'3C0 External hardware interrupt
```

Rev.1.00 2007.06.20
RCJ10J0045-0100

Page 170 of 853

Rev.1.00 2007.06.20
RCJ10J0045-0100

Page 171 of 853

```

MOV.L    @R0,R0
JMP      @R0
NOP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                     exceptional interrupt                                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .SECTION    INTHandler, CODE
        .EXPORT     INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L    #EXPEVT,R0                ; set event address
        MOV.L    @R0,R1                    ; set exception code
        MOV.L    #_INT_Vectors,R0         ; set vector table address
        ADD      #-(H'40),R1               ; exception code - H'40
        SHLR2    R1
        SHLR     R1
        MOV.L    @(R0,R1),R3               ; set interrupt function addr
;
        MOV.L    #_INT_MASK,R0            ; interrupt mask table addr
        SHLR2    R1
        MOV.B    @(R0,R1),R1               ; interrupt mask
        EXTU.B   R1,R1
;
        STC      SR,R0                    ; save SR
        MOV.L    #(RBBLCclr&IMASKclr),R2  ; RB,BL,mask clear data
;
        AND      R2,R0                    ; clear mask data
        OR       R1,R0                    ; set interrupt mask
        LDC      R0,SSR                   ; set current status
;
        LDC.L    R3,SPC
        MOV.L    #__int_term,R0           ; set interrupt terminate
        LDS      R0,PR
;
        RTE
        NOP
;
        .POOL
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                     Interrupt terminate                                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

```

        .ALIGN      4
__int_term:
        MOV.L      #MDRBBLset,R0          ; set MD,BL,RB
        LDC.L      R0,SR
;
        POP_EXP_BASE_REG
;
        RTE                          ; return
        NOP
;
        .POOL
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                TLB miss interrupt                                ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .ORG      H'300
_TLBmissHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L      #EXPEVT,R0              ; set event address
        MOV.L      @R0,R1                  ; set exception code
        MOV.L      #_INT_Vectors,R0       ; set vector table address
        ADD        #-(H'40),R1             ; exception code - H'40
        SHLR2      R1
        SHLR       R1
        MOV.L      @(R0,R1),R3             ; set interrupt function addr
;
        MOV.L      #_INT_MASK,R0          ; interrupt mask table addr
        SHLR2      R1
        MOV.B      @(R0,R1),R1             ; interrupt mask
        EXTU.B     R1,R1
;
        STC        SR,R0                  ; save SR
        MOV.L      #(RBBLClr&IMASKClr),R2
;                                ; RB,BL,mask clear data
        AND        R2,R0                  ; clear mask data
        OR         R1,R0                  ; set interrupt mask
        LDC        R0,SSR                 ; set current status
;
        LDC.L      R3,SPC
        MOV.L      #__int_term,R0         ; set interrupt terminate
        LDS        R0,PR
;
        RTE
        NOP

```



```

;
        .POOL
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                     IRQ                                     ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .ORG      H'500
_IRQHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L     #INTEVT,R0           ; set event address
        MOV.L     @R0,R1              ; set exception code
        MOV.L     #_INT_Vectors,R0    ; set vector table address
        ADD       #-(H'40),R1         ; exception code - H'40
        SHLR2     R1
        SHLR      R1
        MOV.L     @(R0,R1),R3         ; set interrupt function addr
;
        MOV.L     #_INT_MASK,R0       ; interrupt mask table addr
        SHLR2     R1
        MOV.B     @(R0,R1),R1         ; interrupt mask
        EXTU.B    R1,R1
;
        STC       SR,R0              ; save SR
        MOV.L     #(RBBLCclr&IMASKclr),R2
                                           ; RB,BL,mask clear data
        AND       R2,R0              ; clear mask data
        OR        R1,R0              ; set interrupt mask
        LDC       R0,SSR              ; set current status
;
        LDC.L     R3,SPC
        MOV.L     #__int_term,R0      ; set interrupt terminate
        LDS       R0,PR
;
        RTE
        NOP
;
        .POOL
        .END

```

【注】 不能连接指定 #pragma interrupt 的函数。

(2) 初始设定 (PowerON_Reset)

使用库函数时，在本函数中调用库的初始设定 “_INITLIB” 和文件关闭处理 “_CLOSEALL”。

“PowerON_Reset” 的编程例子如下所示。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的中断运行和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 不同，需要中断处理程序。

例：

```
#include <_h_c_lib.h>
#include <machine.h>

#pragma entry PowerON_Reset_PC
#pragma stacksize 0x100

#define SR_Init      0x000000F0 /* 指定 cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp 时 */
                               /* 的初始值为 0x400000F0 */
#define FPSCR_Init   0x00040001 /* 只限指定 cpu=sh2afpu|sh4|sh4a 时 */
                               /* 指定 fpu=double 时的初始值为 0x000C0001 */
#define INT_OFFSET   0x10

extern unsigned int INT_Vectors;
extern void PowerON_Reset_PC();
extern void main();
#ifdef _ _cplusplus
extern "C" {
#endif
extern void _INIT_IOLIB();
extern void _INIT_OTHERLIB();
extern void _CLOSEALL();
#ifdef _ _cplusplus
}
#endif

void PowerON_Reset_PC() {
    set_vbr((void *)(INT_Vectors - INT_OFFSET));
    set_fpscr(FPSCR_Init); /* 只能在指定 cpu=sh2afpu|sh4|sh4a 时设定 */
    _INIT_SCT();
    _INIT_IOLIB();
    _INIT_OTHERLIB();
#ifdef _ _cplusplus
    _CALL_INIT();
#endif
    set_cr(SR_Init);
    main();
#ifdef _ _cplusplus
    _CALL_END();
#endif
    _CLOSEALL();
    sleep();
}
```

(3) 段初始化表 (DTBL、BTBL)

在段的初始化例程 (_INITSCT) 中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。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 声明未初始化的数据区, 用 C\$DSEC 声明初始化的数据区。

编程的例子如下:

例:

```
#pragma section $DSEC          // 将段名设定为 C$DSEC
static const struct {
    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")};

#pragma section $BSEC          // 将段名设定为 C$BSEC
static const struct {
    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")};
```

(4) 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, 请参考以下方针:

- 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 函数或者 strtok 函数时, 需要进行标准输入 / 输出以外的初始设定 (_INIT_OTHERLIB)。

库初始设定的程序例子如下所示, FILE 型数据如图 9.7 所示。

```
#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。
                                          // (省略时: 1024)
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;                // 禁止存取打开失败时的文件
    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;                // 禁止存取打开失败时的文件
    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;                // 禁止存取打开失败时的文件
    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。

```

/* 用C语言进行文件型数据的声明 */

struct _iobuf{
    unsigned char *_bufptr; /* 指向缓冲器的指针 */
    long _bufcnt; /* 缓冲器计数 */
    unsigned char *_bufbase; /* 指向缓冲器的基址指针 */
    long _buflen; /* 缓冲器长 */
    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

(5) 文件的关闭（_CLOSEALL）

通常给文件输出时，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，就有可能无法将输出给文件的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
}

```

(6) 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	sbrk__X	确保 X 存储区
8	sbrk__Y	确保 Y 存储区
9	error_addr*	取得 errno 地址
10	wait_sem*	确保信标
11	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，并且必须在 “9.2.2(4) C/C++ 库函数的初始设定 (_INITLIB)” 的 “_INIT_LOWLEVEL” 函数中进行。

在说明以下的低级输入 / 输出的基本方法后，说明各接口例程的详细内容。

【注】函数名 open、close、read、write、lseek、sbrk、sbrk__X、sbrk__Y、error_addr、wait_sem、signal_sem 是低级接口例程的已保留的标识符，不能在用户程序中使用。

(a) 输入 / 输出的基本方法

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 例程对指定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及其运行和使用注意事项。
用以下格式表示各例程的接口。另外，低级接口例程必须进行函数原型声明，而且在 C++ 程序中声明时必须附加 “extern "C"”。

凡例：

(例程名)			简单说明
			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 进行对应第 1 参数（文件名）的文件操作准备。
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）。每次读写时需用 open 例程返回的文件序号参照文件的种类。
第 2 参数的 mode 指定文件打开时的处理，此数据的各位的含义如下所示：

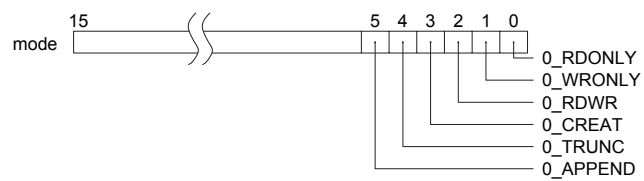


表 9.4 open 例程 mode 位的说明

mode 位	说明
O_RDONLY (bit0)	此位为 1 时，用只读模式打开将文件
O_WRONLY (bit1)	此位为 1 时，用只写模式打开文件
O_RDWR (bit2)	此位为 1 时，用读写模式打开文件
O_CREAT (bit3)	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 (bit4)	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 (bit5)	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

当 mode 指定的文件处理和实际文件的特性出现矛盾时，必须进行错误处理。如果能正常打开文件，就返回可被以后的 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 正常： 正常打开文件的文件序号
 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，传递从 open 例程取得的文件序号。 为了能重新使用，必须释放 open 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的缓冲，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	将数据从第 1 参数 (fileno) 指定的文件读到第 2 参数 (buf) 指向的区域。用第 3 参数 (count) 指定读数据的字节数。 如果文件结束，就只能读取少于或者等于 count 指定的字节数。 文件的读写位置只向前移动读取的字节数。 在能正常读取时，返回实际读取的字节数；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 将数据从第 2 参数 (buf) 指向的区域写到第 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, 只能写少于或者等于 count 指定的字节数。
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。
在能正常写时, 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内的位置。
必须根据第 3 参数 (base), 用以下的方法计算并设定文件内的新位置。
(1) 在 base 为 0 时 设定到从文件起始位置开始的 offset 字节的位置。
(2) 在 base 为 1 时 设定到当前位置加上 offset 字节的位置。
(3) 在 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或者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。
在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

<i>char *sbrk (size_t size)</i>		
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。 在能正常分配时，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

X 存储区的分配

<i>char __X *sbrk__X (size_t size)</i>		
说 明	作为参数，传递分配 X 存储区的容量。 在连续调用 sbrk__X 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。 在能正常分配时，返回分配区的起始地址；否则，返回 “(char __X *)-1”。	
返回值	正常：	分配区的起始地址
	异常：	(char __X *)-1
参 数	size	分配的数据长度

Y 存储区的分配

char _ _Y *sbrk_ _Y (size_t size)

说 明	作为参数，传递分配 Y 存储区的容量。 在连续调用 sbrk_ _Y 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。 在能正常分配时，返回分配区的起始地址；否则，返回 “(char _ _Y *)-1”。	
返回值	正常：	分配区的起始地址
	异常：	(char _ _Y *)-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

<i>int wait_sem (int semnum)</i>		
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：	1
	异常：	0
参 数	semnum	信标 ID

信标的释放

<i>int signal_sem (int semnum)</i>		
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：	1
	异常：	0
参 数	semnum	信标 ID

(c) 低级接口例程的建立例子

```

/*****
/*
/*-----
/*
/*      SuperH RISC engine 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 */
#define HEAPSIZ 1024

/*****
/*
/*      参照函数的声明:
/*      由仿真调试程序将字符输入 / 输出到控制台的汇编程序的参照
/*****
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];        /* 文件打开时的模式设定位置 */

union HEAP_TYPE{
    long dummy;           /* 虚设 4 字节的边界 */
    char heap[HEAPSIZ];   /* 声明由 sbrk 管理的区域 */
};

static union HEAP_TYPE heap_area;
static _X union HEAP_TYPE heap_area _X;
static _Y union HEAP_TYPE heap_area _Y;

static char *brk=(char*)&heap_area; /* sbrk 分配的区域结束地址 */
static _X char *brk_ _X=(char _X*)&heap_area _X;
/* sbrk_ _X 分配的区域结束地址 */
static _Y char *brk_ _Y=(char _Y*)&heap_area _Y;
/* sbrk_ _Y 分配的区域结束地址 */

```

```

/*****
/*          open: 打开文件          */
/*          返回值: 文件序号   (成功)      */
/*          -1       (失败)      */
*****/
int open(char *name,          /* 文件名   */
          int mode)          /* 文件模式 */
{
    /* 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)      */
*****/
int 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)    */
*****/
int read(int fileno,          /* 文件序号          */
        char *buf,          /* 传送目标的缓冲区地址 */
        unsigned int count) /* 读的字符数          */
{
    unsigned int i;

    /* 根据文件序号检查模式, 1个1个字符输入并保存到缓冲区 */

    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)    */
*****/
int write(int fileno,          /* 文件序号 */
        char *buf,          /* 传送源的缓冲区地址 */
        unsigned int count) /* 写的字符数 */
{
    unsigned int i;
    char c;

    /* 根据文件序号检查模式, 1个1个字符输出 */

    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)          */
*****/
long lseek(int fileno,          /* 文件序号          */
           long offset,          /* 读写位置          */
           int base)          /* 偏移的起点          */
{
    return -1;
}

/*****
/*          sbrk: 存储区的分配          */
/*          返回值: 分配的区域起始地址 (成功)          */
/*          -1          (失败)          */
*****/
char *sbrk(size_t size)          /* 分配的区域容量          */
{
    char *p;

    /* 检查空区域 */

    if (brk+size>heap_area.heap+HEAPSIZE) {
        return (char *)-1;
    }

    p=brk;          /* 分配区          */
    brk+=size;          /* 更新结束地址          */
    return p;
}

/*****
/*          sbrk_ _X:_ _X 存储区的分配          */
/*          返回值: 分配的区域起始地址 (成功)          */
/*          -1          (失败)          */
*****/
char _ _X *sbrk_ _X(size_t size)          /* 分配的区域容量          */
{
    _ _X char *p;

    /* 检查空区域 */

    if (brk_ _X+size>heap_area_ _X.heap+HEAPSIZE) {
        return (char _ _X *)-1;
    }

    p=brk_ _X;          /* 分配区          */
    brk_ _X+=size;          /* 更新结束地址          */
    return p;
}

```

```

/*****
/*          sbrk_ _Y:_ _Y 存储区的分配          */
/*          返回值: 分配的区域起始地址   (成功)          */
/*          -1          (失败)          */
*****/
char _ _Y *sbrk_ _Y(size_t size)      /* 分配的区域容量 */
{
    _ _Y char *p;

    /* 检查空区域 */

    if (brk_ _Y+size>heap_area_ _Y.heap+HEAPSIZE) {
        return (char _ _Y *)-1;
    }

    p=brk_ _Y;          /* 分配区          */
    brk_ _Y+=size;      /* 更新结束地址 */
    return p;
}

;
;          lowlvl.src          ;
;
;          SuperH RISC engine 系列 仿真调试程序 接口例程          ;
;          - 输入 / 输出 1 个字符 -          ;
;
;          .EXPORT    _charput          ;
;          .EXPORT    _charget          ;
SIM_IO:
;          .EQU      H'0000          ; 指定 TRAP_ADDRESS
;
;          .SECTION   P, CODE, ALIGN=4

```

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;          _charput: 输出 1 个字符
;          C 程序接口: charput(char)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

_charput:
    MOV.L    O_PAR,R0        ; 设定缓冲区地址
    MOV.B    R4,@R0          ; 将参数设定到缓冲区
    MOV.L    #O_PAR,R1       ; 设定参数块地址
    MOV.L    #H'01220000,R0   ; 设定功能码 (PUTC)
    MOV.W    #SIM_IO,R2      ; 设定系统调用地址
    JSR      @R2
    NOP
    RTS
    NOP

    .ALIGN    4

O_PAR:
    .DATA.L   OUT_BUF        ; 参数块区

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;          charget: 输入 1 个字符
;          C 程序接口: char charget(void)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .ALIGN    4

_charget:
    MOV.L    #I_PAR,R1       ; 设定参数块地址
    MOV.L    #H'01210000,R0   ; 设定功能码 (GETC)
    MOV.W    #SIM_IO,R2      ; 设定系统调用地址
    JSR      @R2
    NOP
    MOV.L    I_PAR,R0         ; 设定缓冲区地址
    MOV.B    @R0,R0           ; 将输入数据设定为返回值
    RTS
    NOP

    .ALIGN    4

I_PAR:
    .DATA.L   IN_BUF         ; 参数块区

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;          定义输入 / 输出的缓冲区
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    .SECTION   B, DATA, ALIGN=4

OUT_BUF:
    .RES.L    1              ; 输出缓冲区

IN_BUF:
    .RES.L    1              ; 输入缓冲区

    .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）的库时需要本例程。

如果 `wait_sem` 函数或者 `signal_sem` 函数返回 `NG`，就给 `errno` 设定以下内容后从库函数返回。

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
	EIOBRESM	_iob 的信标资源的释放失败

如果在确保信标后发生更高优先级的中断，再次进行信标的确保，就会发生死锁，所以不能发生如共享资源处理用的中断嵌套。

```
#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 MALLOC_SEM_ _X  4      /* malloc_ _X 的信标 No.  */
#define MALLOC_SEM_ _Y  5      /* malloc_ _Y 的信标 No.  */
#define SEMSIZE          6
#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);
}

```

(7) 结束处理例程

(a) 建立结束处理注册和执行 (atexit) 例程的例子

说明进行结束处理注册的库 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（在此为注册的函数地址）。

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) 建立程序结束 (exit) 例程的例子

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程序结束处理。

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，通过将结束码设定为外部变量并且在即将调用 main 函数前返回到由 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” 的 “main” 函数调用。

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) 建立异常结束 (abort) 例程的例子

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，还需要在以下的情况下调用 abort 函数。

- 异常处理无法正常运行
- 调用纯虚拟函数
- dynamic_cast 失败
- typeid 失败
- 类数组的 delete 时无法取得信息
- 在注册类目标的析构函数调用信息时出现矛盾

以下的程序例子在将信息输出到标准输出设备后关闭文件，然后进入无限循环等待复位。

例：

```
#include <stdio.h>

#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef __cplusplus
extern "C"
#endif
void abort(void)
{
    printf("program is abort !!\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

说明有关在连接 C/C++ 程序和汇编程序时应该注意的事项。

- 外部名的相互参照方法
- 函数调用的规则

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 inline 的函数名（C++ 程序）
- 非 inline 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 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）	C/C++ 程序（参照侧）
.EXPORT _a, _b	extern int a,b;
.SECTION D, DATA, ALIGN=4	
_a: .DATA.L 1	void f()
_b: .DATA.L 1	{
.END	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）	汇编程序（参照侧）
int a;	.IMPORT _a
	.SECTION P, CODE, ALIGN=4
	MOV.L A_a, R1
	MOV.L @R1, R0
	ADD #1, R0
	RTS
	MOV.L R0, @R1
	.ALIGN 4
	A_a: .DATA.L _a
	.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（被调用侧）

```
extern "C"
void sub ( )
{
    :
}
```

汇编程序（调用侧）

```
.IMPORT _sub
.SECTION P, CODE, ALIGN=4
:

STS.L    PR, @-R15
MOV.L    R1, @(1, R15)
MOV      R3, R12
MOV.L    A_sub, R0
JSR      @R0
NOP
LDS.L    @R15+, PR
:
A_sub:   .DATA.L    _sub
.END
```

9.3.2 函数调用的规则

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的以下 4 个规则：

- 1. 有关栈指针的规则
- 2. 有关分配和释放堆栈帧的规则
- 3. 有关寄存器的规则
- 4. 有关设定和参照参数、返回值的规则

(1) 有关栈指针的规则

不能将有效数据保存到低于栈指针指向的地址（地址 0 的方向）的堆栈区，保存到低于栈指针指向地址的数据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），栈指针指向调用源函数使用的堆栈最低位地址。调用源函数必须分配和设定高于此栈指针指向的区域地址的数据

在函数返回时，释放由被调用函数确保的区域，然后用通常的 RTS 指令返回到调用源函数。调用源函数释放高于此区域的地址的区域（返回值地址和参数区）。

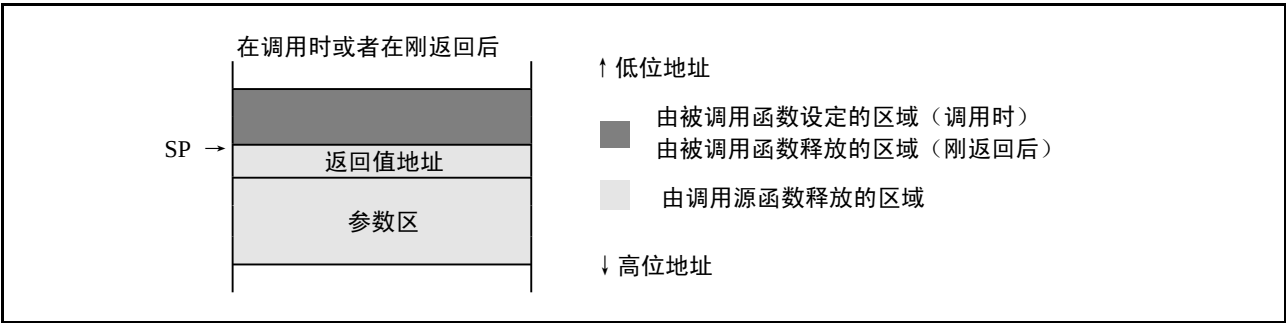


图 9.8 有关分配和释放堆栈帧的规则

(3) 有关寄存器的规则

在函数调用前后，有保证和不保证值的寄存器。寄存器的保证规则如表 9.5 所示。

表 9.5 函数调用前后的寄存器保证规则

项目	对象寄存器	编程时的注意点
1 不保证的寄存器	R0 ~ R7, FR0 ~ FR11*1 DR0 ~ DR10*2 FPUL*1,*2、FPSCR*1,*2 A0*3、A0G*3、A1*3、 A1G*3、M0*3、M1*3、 X0*3、X1*3、 Y0*3、Y1*3、DSR*3、 MOD*3、RS*3、RE*3	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，在被调用函数侧不进行保存而能使用。但是，在指定 fpscr=safe 时，FPSCR 为被保证的寄存器。
2 保证的寄存器	R8 ~ R15 MACH、MACL、PR FR12 ~ FR15*1 DR12 ~ DR14*2	保存函数内使用的对象寄存器的值，返回时恢复。但是，在指定 macsave=0 时，MACH 和 MACL 为不被保证的寄存器。另外，在指定 gbr=auto 时，GBR 为被保证的寄存器。

【注】 *1 是 SH-2E、SH2A-FPU、SH-4、SH-4A 的单精度浮点寄存器。
*2 是 SH2A-FPU、SH-4、SH-4A 的双精度浮点寄存器。
*3 是 SH2-DSP、SH3-DSP、SH4AL-DSP 的 DSP 寄存器。

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=4

_sub: MOV.L    R14, @-R15      ; 保存函数内使用的寄存器
      MOV.L    R13, @-R15
      ADD      #-8, R15

      ; 函数的处理
      ; （因为 R0 ~ R7 为不被保证的寄存器，
      ; 所以在函数内不进行保存而能使用）

      ADD      #8, R15
      MOV.L    @R15+, R13    ; 恢复被保存的寄存器
      RTS
      MOV.L    @R15+, R14
      .END
```

C/C++ 程序（调用侧）

```
#ifdef _cplusplus
extern "C"
#endif
void sub();

void f()
{
    sub();
}
```

(b) 从汇编程序调用 C++ 程序的子例程时

C/C++ 程序（被调用侧）

```
void sub()  
{  
:  
}
```

汇编程序（调用侧）

```
.IMPORT _sub  
.SECTION P, CODE, ALIGN=4  
:  
  
STS.L PR, @-R15  
MOV.L R1, @(4, R15)  
MOV R3, R12  
MOV.L A_sub, R0  
JSR @R0  
NOP  
LDS.L @R15+, PR  
:  
A_sub: .DATA.L _sub  
.END
```

用.IMPORT声明（C）给被调用函数名前附加了下划线"_"的外部名
用.IMPORT声明（C++）编译程序从函数声明/定义生成的外部名*

在函数调用时，保存PR寄存器（返回地址保存寄存器）
如果寄存器R0~R7中存在有效值，就将其保存到空闲寄存器（R8~R14）或者堆栈
函数“sub”的调用
恢复PR寄存器
函数“sub”的地址数据

【注】* 在进行 C++ 编译时，从函数名或者静态数据成员生成的外部名按照一定的规则进行转换。当要知道编译程序生的外部名时，必须参照编译程序生成的外部名（用 code=asmcode 或者 listfile 选项）。另外，如果用“extern “C””对 C++ 函数进行函数定义，外部名的生成规则就与 C 函数相同，但是该函数不能多重定义。

(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。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寄存器或者堆栈的参数分配区后调用函数。因为在调用侧的函数中不参照返回后的参数分配区，所以即使在被调用侧的函数中更改参数值，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：

◆ 声明型的参数的型转换

将函数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

没有函数原型声明的型的参数按照以下规则进行型转换：

- 将 (signed)char 型、unsigned char 型、(signed)short 型、unsigned short 型的参数转换为 (signed)int 型。
- 将 float 型的参数转换为 double 型。
- 不转换上述以外的型。

◆ 返回值的型转换

将返回值转换为该函数的返回型。

例：

```
(1)
long f( );
long f( )
{
    float x;
    return x;    ← 按函数原型声明将返回值转换为long型。
}

(2)
void p(int,... );
void f( )
{
    char c;
    p(1.0, c);
}
```

→ 因为c不对应参数的型声明，所以被转换为int型。

→ 因为1.0对应参数的int型，所以被转换为int型。

(b) 参数分配区

有将参数分配到寄存器和不分配到寄存器而分配到参数区的情况。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 R4。

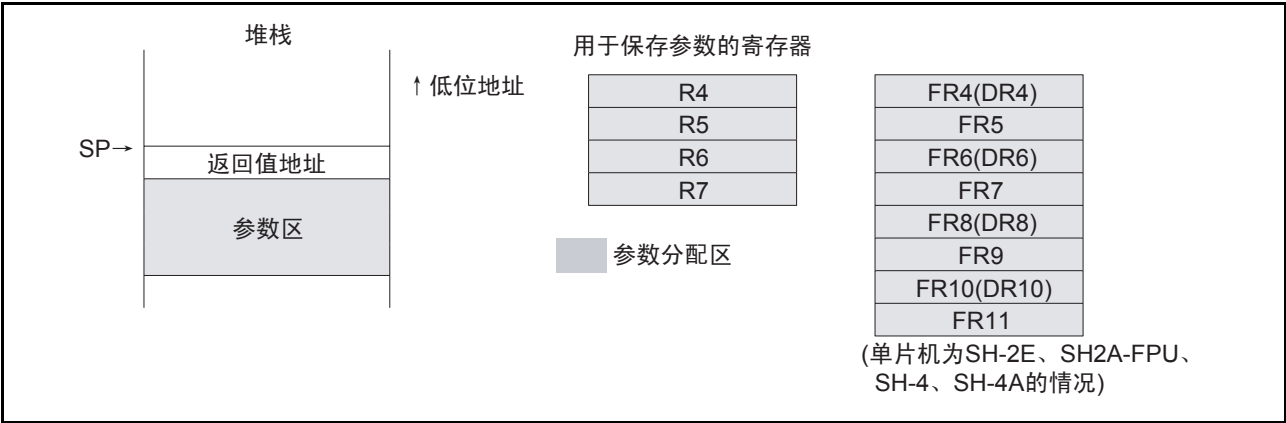


图 9.9 参数分配区

表 9.6 参数分配区的一般规则

分配规则		
用寄存器传递的参数		为堆栈传递的参数
用于保存参数的寄存器	对象型	
R4 ~ R7	char、unsigned char、bool、 shor、unsigned short、 int、unsigned int、 long、unsigned long、 float（单片机不为 SH-2E、SH2A-FPU、SH-4、SH-4A 时）、指针、指向数据成员的指针、参照	(1) 参数的型不为寄存器传递对象型的参数 (2) 用函数原型声明对具有可变个数参数的函数进行了声明的参数 *3 (3) 其他参数全部分配到 R4 ~ R7 时 (4) 其他参数全部分配到 FR4(DR4) ~ FR11(DR10) 时
FR4 ~ FR11*1	SH-2E 时 • 参数为 float 型 • 参数为 double 型并指定 double=float SH2A-FPU、SH-4、SH-4A 时 • 参数型为 float 型并无 fpu=double 选项的指定 • 参数型为 double 型并指定 fpu=single	(5) long long、unsigned long long 型的参数 (6) __fixed 型、long __fixed 型、__accum 型、long __accum 型的参数
DR4 ~ DR10*2	SH2A-FPU、SH-4、SH-4A 时 • 参数型为 double 型并不指定 fpu=single • 参数型为 float 型并指定 fpu=double	

【注】 *1 是 SH-2E、SH2A-FPU、SH-4、SH-4A 的单精度浮点寄存器。
*2 是 SH2A-FPU、SH-4、SH-4A 的双精度浮点寄存器。
*3 在用函数原型声明对具有可变个数参数的函数进行了声明时，将声明中没有对应的型的参数及其前一个的参数传递到堆栈。

例：
int f2(int,int,int,int,...);
:
f2(a,b,c,x,y,z); → x、y、z 传递到堆栈。

(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。

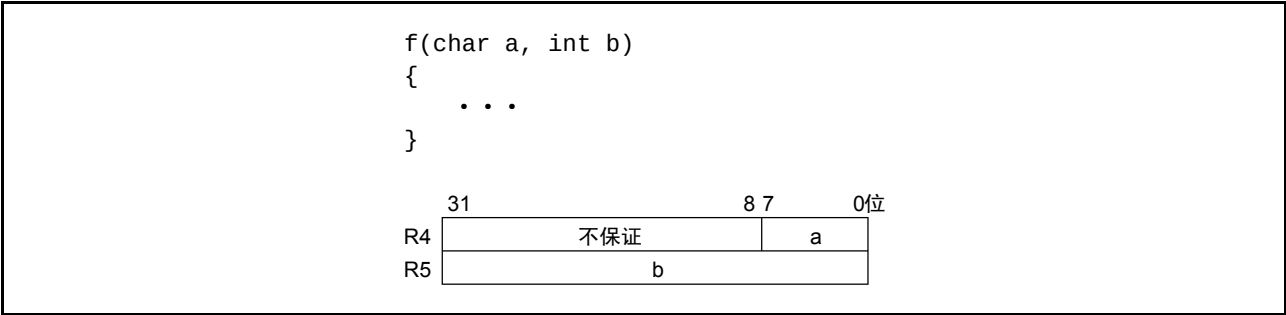


图 9.10 用于参数保存的寄存器的分配例子

- 分配到堆栈的参数区
按照源程序中声明的顺序从低位地址开始分配到堆栈的参数区。

【注】 有关结构体型、联合体型和 class 型的参数的注意事项

在设定结构体型、联合体型和 class 型的参数时，与该型原来的调整无关而分配为 4 个字节边界，而且该区域使用 4 的倍数字节的区域。这是因为 SuperH RISC engine 单片机的栈指针以 4 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。
当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
1 (signed)char、 unsigned char、 (signed)short、 unsigned short、(signed)int、 unsigned int、long、 unsigned long、float、 指针、bool、 参照、 指向数据成员的指针	R0 : 32 位 不保证 (signed)char 和 unsigned char 的高位 3 字节、(signed)short 和 unsigned short 的高位 2 字节的内容，但是在指定 rtnext 选项时，(signed)char 和 (signed)short 型进行符号扩展，unsigned char 和 unsigned short 型进行零扩展。 FR0 : 32 位 (1) SH-2E 时 • 返回值为 float 型 • 返回值为 double 型并指定 double=float (2) SH2A-FPU、SH-4、SH-4A 时 • 返回值为 float 型并无 fpu=double 的指定 • 返回值为浮点型并指定 fpu=single
2 double、long double、 结构体、联合体、 class 型和指向函数的指针	返回值的设定区（存储器） DR0:64 位 SH2A-FPU、SH-4、SH-4A 时 • 返回值为 double 型并无 fpu=single 的指定 • 返回值为浮点型并指定 fpu=double
3 (signed)long long、 unsigned long long	返回值的设定区（存储器）
4 __fixed、long __fixed、 __accum、 long __accum	返回值的设定区（存储器）

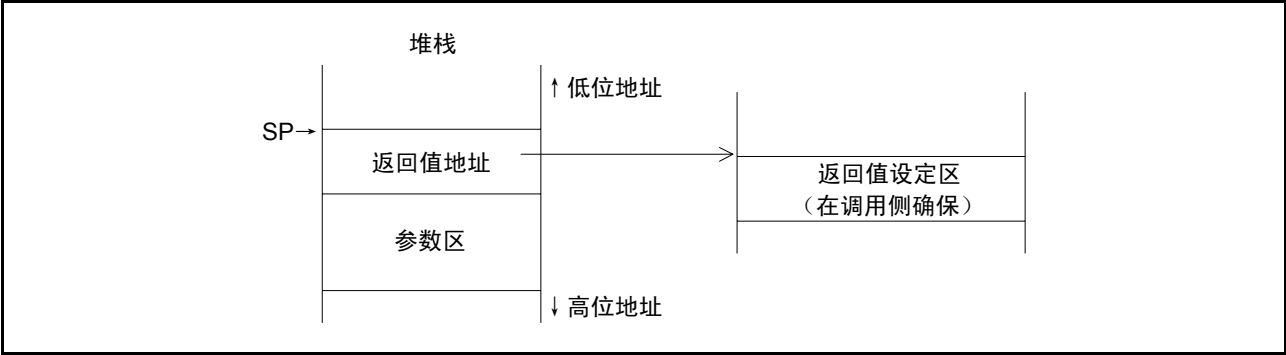


图 9.11 将返回值设定到存储器时返回值的设定区

例 5: 函数的返回型超过 4 个字节或者为类时，在参数区的前面设定返回值地址。另外，在类的大小不为 4 的倍数字节时，会产生空区域。

```
struct s{char x,y,z;}a;
double f(struct s);
:
f(a);
```

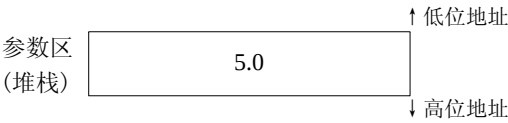


例 6: 在单片机为 SH-2E 时，float 型的参数分配到 FPU 寄存器。

```
int f(char,float,short,float,double);
:
f(1,2.0,3,4.0,5.0);
```

R4	不保证	1
R5	不保证	3
R6		
R7		

FR4	2.0
FR5	4.0
FR6	
FR7	
FR8	
FR9	
FR10	
FR11	



例 7: 在单片机为 SH2A-FPU、SH-4、SH-4A 时，float/double 型的参数分配到 FPU 寄存器。

```
int f(char,float,double,float,short);
:
f(1,2.0,4.0,5.0,3);
```

R4	不保证	1
R5	不保证	3
R6		
R7		

FR4(DR4)	2.0
FR5	5.0
FR6(DR6)	4.0
FR7	
FR8(DR8)	
FR9	
FR10(DR10)	
FR11	

9.3.4 寄存器和堆栈区的使用方法

说明编译程序的寄存器和堆栈区的用法。
函数内的寄存器和堆栈区全部由编译程序操作，所以不需要用户特别注意该区域的用法。
寄存器和堆栈区的用法如图 9.12 所示。

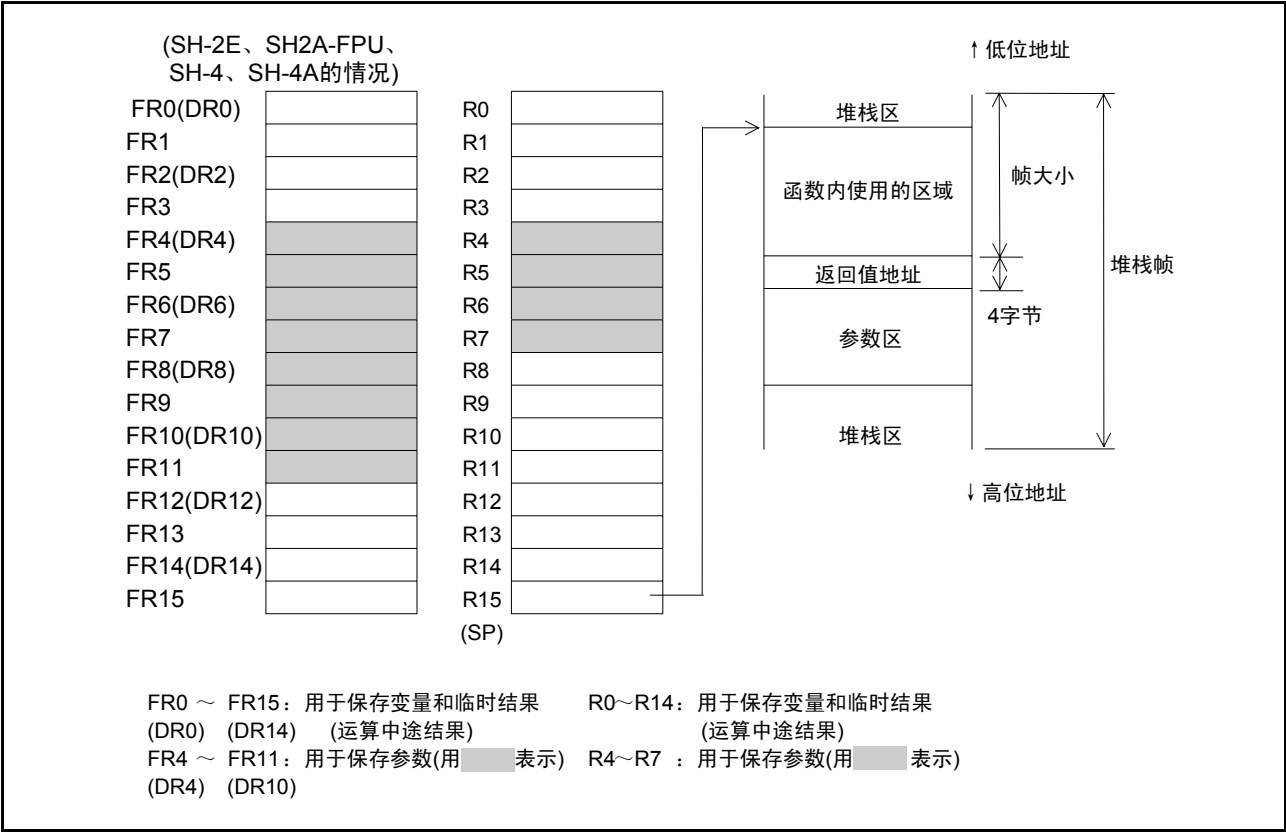


图 9.12 寄存器和堆栈区的用法

9.4 程序建立时的注意事项

对于编译程序，本节说明编程时的注意事项以及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) 带 float 型参数的函数

带 float 型参数的函数必须进行函数原型声明，或者将 float 型更改为 double 型。如果接受和传递没有函数原型声明的 float 型参数，就不保证数据的值。

例

```
void f(float);
void g()
{
    float a;
    ...
    f(a);
}
void f(float x)
{...}
```

函数“f”是带 float 型的参数。必须进行函数原型声明。

(2) C/C++ 语言规格没有规定评价顺序的表达式

对于 C/C++ 语言规格没有规定评价顺序的表达式，如果编制的程序因评价顺序而产生不同结果，就不保证其运行。

例：

`a[i]=a[++i];` 根据是先评价还是后评价赋值表达式的右边，左边的值发生变化。

`sub(++i, i);` 根据是先评价还是后评价函数的第 1 个参数，第 2 参数的值发生变化。

(3) 溢出运算和被零除

即使有上溢运算或者浮点被零除，也不输出错误信息，但是如果在一个常数或者常数之间的运算中发生上溢运算，就在编译时输出错误信息。

例：

```
void main()
{
    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，输出对上溢的编译错误信息 */`

```

    ia=999999999999;      /* (W) 检测常数的上溢          */
    fa=3.5e+40f;          /* (W) 检测浮点运算的上溢          */

    /* 对于执行时的上出，不输出错误信息 */

    ib=ib+32767;          /* 忽视运算结果的上溢          */
    fb=fb+3.4e+38f;       /* 忽视浮点运算结果的上溢          */
}

```

(4) 对 const 型变量的写操作

必须注意：即使声明了 const 型的变量，如果通过型转换将其转换为非 const 型的后赋值，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对 const 型变量的写操作进行检查。

例

```

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。            */

```

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                                    */

```

(5) 有关数学函数库的精度

必须注意：当 $x \approx 1$ 时， $\text{acos}(x)$ 和 $\text{asin}(x)$ 函数的误差变大。

误差范围如下：

$\text{acos}(1.0 - \epsilon)$ 中的绝对误差	双精度 2^{-39} ($\epsilon = 2^{-33}$)
	单精度 2^{-21} ($\epsilon = 2^{-19}$)
$\text{asin}(1.0 - \epsilon)$ 中的绝对误差	双精度 2^{-39} ($\epsilon = 2^{-28}$)
	单精度 2^{-21} ($\epsilon = 2^{-16}$)

9.4.2 用 C++ 编译程序对 C 程序进行编译时的注意事项

(1) 函数原型声明

在使用函数前需要进行函数原型声明，此时还必须声明形式参数的型。

```
extern void func1();
void g()
{
    func1(1); // 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 cvalue1;      // 错误
const cvalue2 = 1; // 内部连接
```

```
const cvalue1=0;
// 赋初始值
extern const cvalue2 = 1;
// 和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，就不能给指向其他目标型的指针（指向函数的指针和指向成员的指针除外）赋值。

```
void func(void *ptrv, int *ptri)
{
    ptri = ptrv; //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) 有关单片机选择的注意事项

(a) 必须统一编译和汇编时指定的单片机种类

必须统一编译和汇编时用 `cpu` 选项指定的单片机种类。如果将不同的单片机种类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单片机相同的单片机种类。

在汇编由编译程序生成的汇编程序时，必须用 `cpu` 选项指定和编译时相同的单片机种类。

(c) 在连接时，必须按照单片机种类指定标准库。

必须指定对应单片机种类的库，否则不保证运行。

(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。

- `endian = big | little` (SH-2、SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP)
- `pic = 0 | 1` (SH-1 除外)
- `fpu = single | double` (SH2A-FPU、SH-4、SH-4A)
- `fpscr = safe | aggressive` (SH2A-FPU、SH-4、SH-4A)
- `round = zero | nearest` (SH2A-FPU、SH-4、SH-4A)
- `denormalize = on | off` (SH-4、SH-4A)
- `double = float` (SH2A-FPU、SH-4、SH-4A 除外)
- `exception | noexception`
- `rtti = on | off`
- `pack = 1 | 4`
- `rtnext | nortnext`
- `macsave`
- `gbr = auto | user`
- `bit_order = left | right`
- `auto_enum`

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性定义项目，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 日语汉字码或者 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	不支持语言规定外的字符和转义序列。
6 包含超过 1 个字符的字符常数或者超过 1 个字符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，如果指定的字符超过 1 个字符，就输出警告错误。
7 用于将多字节字符转换为宽字符的 locale 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	整数型的表示方法及其值	如表 10.5 所示。
2	将短于整数值长度的带符号整数型或者无符号整数型转换为相同长度的带符号整数型时的值（用转换后的型无法表示结果值时）	整数值的低位 4 个字节（转换后的型长度为 4 个字节时）、低位 2 个字节（转换后的型长度为 2 个字节时）或者低位 1 个字节（转换后的型长度为 1 个字节时）为转换后的值。
3	带符号整数的逐位运算结果	为带符号的值。
4	整数除法中的余数符号	和被除数符号相同。
5	负数的带符号标量型的右移结果	保持符号位。

表 10.5 整数型及其值的范围

	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	-2147483648 ~ 2147483647	4 个字节
7	unsigned int	0 ~ 4294967295	4 个字节
8	long	-2147483648 ~ 2147483647	4 个字节
9	unsigned long	0 ~ 4294967295	4 个字节
10	long long	-9223372036854775808 ~ 9223372036854775807	8 个字节
11	unsigned long long	0 ~ 18446744073709551615	8 个字节

(5) 浮点

表 10.6 浮点的规格

项 目		编译程序的规格
1	浮点型的表示方法及其值	浮点型有 float 型、double 型和 long double 型。
2	将整数转换为不能正确表示实际值的浮点型时的舍去方法	double 型。在“10.1.3 浮点型的规格”中
3	将浮点型转换为较短的浮点型时的舍去或者舍入方法	说明浮点型的内部表示、转换规格以及运算规格等性质。浮点型能表示的值的极限值如表 10.7 所示。

表 10.7 浮点型的极限值

项 目		极限值	
		10 进制数表示 *1	16 进制数表示
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 } 型的最大值 long double }	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffff
4	double } 型的正的 long double } 最小值	4.9406564584124655e-324 (4.9406564584124654e-324)	0000000000000001

【注】 *1 10 进制数表示的极限值是不为 0 或者不为无穷大的极限值。() 内表示理论值。

*2 当指定 double=float 时, double 型的值和 float 型相同。

当指定 fpu=single 时, double、long double 型的值和 float 型相同。

当指定 fpu=double 时, float 型的值和 double 型相同。

(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数型 (size_t)	unsigned long 型
2	从指针型转换到整数型 (指针型的长度 ≥ 整数型的长度)	为指针型低位字节的值。
3	从指针型转换到整数型 (指针型的长度 < 整数型的长度)	进行符号扩展。
4	从整数型转换到指针型 (整数型的长度 ≥ 指针型的长度)	为整数型低位字节的值。
5	从整数型转换到指针型 (整数型的长度 < 指针型的长度)	进行符号扩展。
6	为了保持相同数组中成员指针之间的差所需要的整数型 (ptrdiff_t)	int 型

(7) 寄存器

表 10.9 寄存器的规格

项 目		编译程序的规格	
1	能分配到寄存器的变量的最大个数	7 个	char、unsigned char、bool、short、unsigned short、int、unsigned int、long、unsigned long、指针
		4 个	float* ²
		2 个	double* ³
2	能分配到寄存器的变量型	char、unsigned char、bool、short、unsigned short、int、unsigned int、long、unsigned long、float* ² 、double* ³ 、指针	

【注】 *1 对变量进行寄存器分配时，不受是否已进行 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。

*2 只限单片机为 SH-2E、SH2A-FPU、SH-4 或者 SH-4A 的情况。

*3 只限单片机为 SH2A-FPU、SH-4 或者 SH-4A 的情况。

(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	由不同型成员存取的联合体型的成员参照	能参照，但不保证值。
2	类和结构体成员的调整	在类和结构体成员中，调整数的最大值为该类和结构体的调整数。分配方法的详细规格请参照“10.1.2(2) 结构体 / 联合体 (C 语言) 和 class 型 (C++ 语言)”。
3	int 型的位域的符号	signed int 型
4	int 型的长度中的位域分配顺序	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	在位域中允许的类型说明符	char、unsigned char、bool、short、unsigned short、int、unsigned int、long、unsigned long、enum、long long、unsigned long long
7	表示枚举型值的整数型	int 型。但是，在指定枚举型长度选项时，请参照“2.2.5 其他选项”。

【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。

(9) 型限定词

表 10.11 型限定词的规格

项 目	编译程序的规格
1 volatile 型数据的存取种类	没规定。

(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。

以下举例说明限定基本型的型个数的计算方法：

例：

(i) int a; a 为 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 在条件编译的常数表达式中，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 如果找不到文件，就按照环境变量 SHC_INC 指定的文件夹、系统文件夹（SHC_LIB）的顺序检索各文件夹。*1
3 是否支持用双重引用符括起来的 include 文件	支持。从当前文件夹读取 include 文件。如果在当前文件夹中不存在该文件，就按本表的项 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 的操作	请参照“10.3.1 #pragma”。
6 __DATE__、__TIME__ 的值	设定编译开始时的主机定时器的值。

10.1.2 数据的内部表示

本节阐述型名和数据的内部表示的对应。数据的内部表示由以下项目构成：

- 数据长度
数据占有区域的大小。
- 数据的调整数
是有关分配数据的地址限制，有分配到任意地址的 1 个字节调整数、分配到偶数字节的 2 个字节调整数和分配到 4 的倍数字节的 4 个字节调整数。
- 值的范围
表示标量型（C 语言）或者基本型（C++ 语言）的值的范围。
- 数据的分配例子
表示为结构体 / 联合体（C 语言）或者 class 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示如表 10.15 所示：

表 10.15 标量型和基本型的内部表示

型名	长度 (byte)	调整数 (byte)	有 / 无 符号	值的范围	
				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	4	4	有	$-2^{31}(-2147483648)$	$2^{31}-1(2147483647)$
7 unsigned int	4	4	无	0	$2^{32}-1(4294967295)$
8 long	4	4	有	$-2^{31}(-2147483648)$	$2^{31}-1(2147483647)$
9 unsigned long	4	4	无	0	$2^{32}-1(4294967295)$
10 long long	8	4	有	-2^{63} (-9223372036854775808)	$2^{63}-1$ (9223372036854775807)
11 unsigned long long	8	4	无	0	$2^{64}-1$ (18446744073709551615)
12 enum* ¹	4	4	有	$-2^{31}(-2147483648)$	$2^{31}-1(2147483647)$
13 float	4* ⁴	4	有	$-\infty$	$+\infty$
14 double long double	8* ² ,* ⁴	4	有	$-\infty$	$+\infty$
15 指针	4	4	无	0	$2^{32}-1(4294967295)$
16 bool* ³	4	4	有	—	—
17 参照 * ³	4	4	无	0	$2^{32}-1(4294967295)$
18 指向数据成员的指针 * ³	4	4	有	0	$2^{32}-1(4294967295)$
19 指向函数成员的指针 * ³ * ⁵	12	4	—	—	—

【注】 *1 当指定 auto_enum 选项时，枚举型的长度可变。

*2 当指定 double=float 时，double 型的长度为 4 个字节。

*3 只在 C++ 编译时有效。

- *4 当指定 `cpu=sh2afpu | sh4 | sh4a` 并且 `fpu=single` 时，`double`、`long double` 型作为 4 个字节（`float` 型）处理；
当指定 `cpu=sh2afpu | sh4 | sh4a` 并且 `fpu=double` 时，`float` 型作为 8 个字节（`double` 型）处理。
- *5 用以下的数据结构表示指向函数成员和虚拟函数成员的指针：

```
class PMF{
public:
    long d;           // 目标的偏移值
    long i;           // 当对象成员函数是虚拟函数时，为虚拟函数表中的 index

    union{
        void (*f)(); // 当对象成员函数是非虚拟函数时，为函数地址
        long offset; // 当对象成员函数是虚拟函数时，为虚拟函数表的目标中的偏移
    };
};
```

(2) 结构体 / 联合体（C 语言）和 `class` 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 `class` 型的内部表示。
结构体 / 联合体 和 `class` 型的内部表示如表 10.16 所示：

表 10.16 结构体 / 联合体 和 `class` 型的内部表示

型名	调整数（byte）	长度（byte）	数据的分配例子		
1 数组型	数组元素的调整数	数组元素的个数 × 元素长度	<code>char a[10];</code>	调整数 长度	1byte 10byte
2 结构体型	结构体成员的调整数中的最大值	成员长度的和 参照“(a) 结构体数据的分配方法”	<code>struct { char a,b; };</code>	调整数 长度	1byte 2byte
3 联合体型	联合体成员的调整数中的最大值	最大成员长度 参照“(b) 联合体数据的分配方法”	<code>union { char a,b; };</code>	调整数 长度	1byte 1byte
4 <code>class</code> 型	1) 有虚拟函数时： 总是 4 2) 上述以外： 数据成员的调整数中的最大值	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”	<code>class B: public A{ virtual void f(); };</code> <code>class A{ char a; };</code>	调整数 长度 边界调整数 长度	4byte 8byte 1byte 1byte

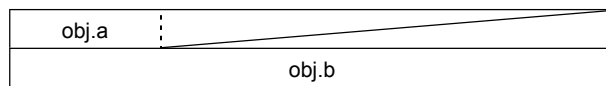
在下例中没有标明长度的  表示 4 个字节； 表示空区域。

(a) 结构体数据的分配方法

- 在分配结构体型的各成员时，为了符合该成员型名的边界调整数，有可能在和前一个成员之间产生 1 个字节的空区域。

例：

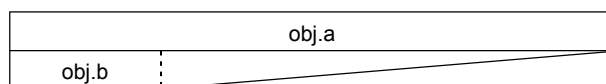
```
struct {
    char a;
    int b;
} obj;
```



- 在结构体具有 4 个字节的调整数并且最后的成员以 1、2 或者 3 字节结束时，作为结构体型的区域，还包含其后的字节。

例：

```
struct {
    int a;
    char b;
} obj;
```

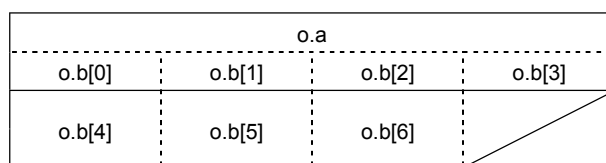


(b) 联合体数据的分配方法

- 在联合体具有 4 个字节的调整数并且最大成员长度不是 4 的倍数时，作为联合体型的区域，还含有成为 4 的倍数的剩余字节。

例：

```
union {
    int a;
    char b[7];
} o;
```

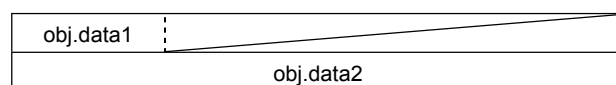


(c) 类数据的分配方法

- 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：

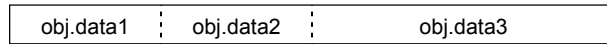
```
class A{
    char data1;
    int data2;
public:
    A();
    char getData1(){return data1;}
}obj;
```



- 在是从调整为 1 的基类派生出的类并且其派生类的第 1 个成员为 1 个字节的数据时，分配为没有空区域的数据成员。

例：

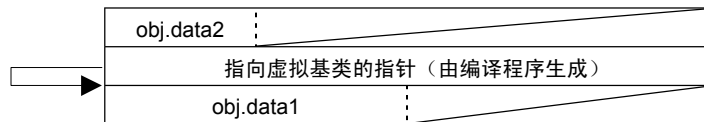
```
class A{
    char data1;
};
class B:public A{
    char data2;
    short data3;
}obj;
```



- 当类中有虚拟基类时，分配指向虚拟基类的指针。

例：

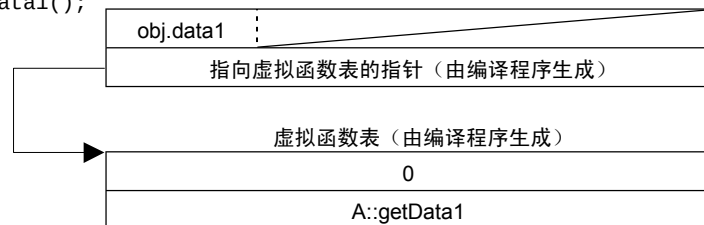
```
class A{
    short data1;
};
class B: virtual protected A{
    char data2;
}obj;
```



- 当类中有虚拟函数时，编译程序生成虚拟函数表，分配指向虚拟函数表的指针。

例：

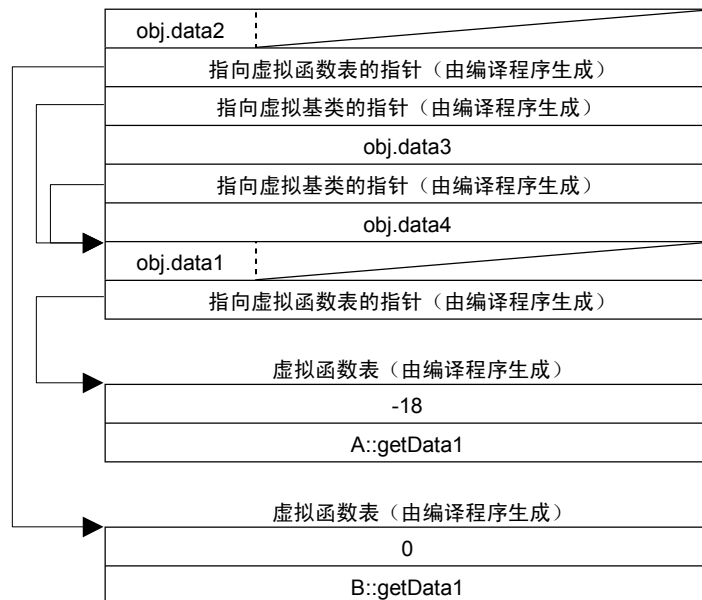
```
class A{
    char data1;
public:
    virtual char getData1();
}obj;
```



- 有虚拟基类、基类和虚拟函数的类的例子如下所示：

例：

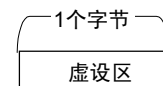
```
class A{
    char data1 ;
    virtual char getData1();
};
class B:virtual public A{
    char data2;
    char getData2();
    char getData1();
};
class C:virtual protected A{
    int data3;
};
class D:virtual public A,public B,public C{
public:
    int data4;
    char getData1();
}obj;
```



- 在空类的情况下，分配 1 个字节的虚设区。

例：

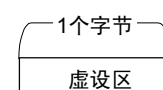
```
class A{
    void fun();
}obj;
```



- 即使在以空类为基类的空类时，虚设区也为 1 个字节。

例：

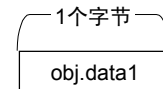
```
class A{
    void fun();
};
class B: A{
    void sub();
}obj;
```



- 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。

(a) 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 在位域中允许的型说明符	(signed)char、unsigned char、bool* ¹ 、 (signed)short、unsigned short、enum、 (signed)int、unsigned int、 (signed)long、unsigned long、 (signed)long long、unsigned long long
2 扩展为声明的型时的符号处理	无符号（指定 unsigned 的型）→ 零扩展* ³ 有符号（没有指定 unsigned 的型）→ 符号扩展* ⁴

【注】 *1 只能在 C++ 程序中指定 bool。

*2 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。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，能表示的值只为 0 和 -1。表示 0 和 1 时，必须用无符号（unsigned）声明。

*3 零扩展：扩展时给高位写 0。

*4 符号扩展：扩展时，将位域数据的最高位解释为符号，给扩展位全部置符号位。

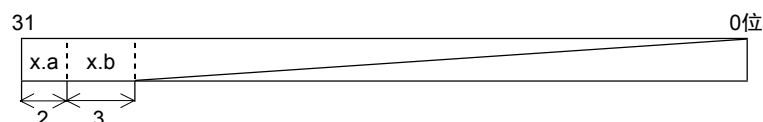
(b) 位域的分配方法

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：

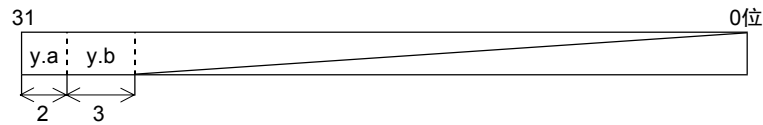
```
struct b1 {
    int a:2;
    int b:3;
} x;
```



- 连续有相同长度的型说明符时，尽可能分配到相同区域。

例：

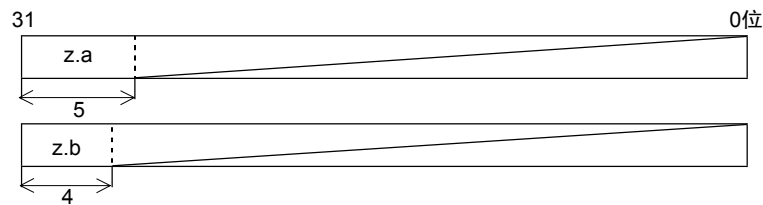
```
struct b1 {
    long    a:2;
    unsigned int b:3;
} y;
```



- 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：

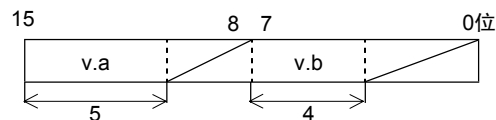
```
struct b1 {
    int    a:5;
    char   b:4;
} z;
```



- 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，也分配到下一个区域，而剩余的区域为未使用区域。

例：

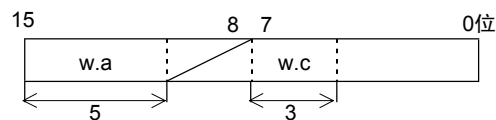
```
struct b2 {
    char a:5;
    char b:4;
} v;
```



- 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;
```



【备注】 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“10.3.1 `#pragma`”的 `#pragma bit_order`。

(4) Little Endian 的存储器分配

Little Endian 的存储器内的数据排列如下：

(a) 1 个字节的数据 ((signed)char、unsigned char 型)

无论是 Big Endian 还是 Little Endian，1 个字节数据中的位排列顺序相同。

(b) 2 个字节的数据 ((signed)short、unsigned short 型)

2 个字节数据中的高位字节和低位字节的排列顺序相反。

例：当地址 0x100 中有 2 个字节的数据 0x1234 时

Big Endian:	地址 0x100: 0x12	Little Endian:	地址 0x100: 0x34
	地址 0x101: 0x34		地址 0x101: 0x12

(c) 4 个字节的数据 ((signed)int、unsigned int、(signed)long、unsigned long、float、bool 型)

对于 Big Endian 和 Little Endian，4 个字节数据中的字节排列顺序相反。

例：当地址 0x100 中有 4 个字节的数据 0x12345678 时

Big Endian:	地址 0x100: 0x12	Little Endian:	地址 0x100: 0x78
	地址 0x101: 0x34		地址 0x101: 0x56
	地址 0x102: 0x56		地址 0x102: 0x34
	地址 0x103: 0x78		地址 0x103: 0x12

(d) 8 个字节数据 ((signed)long long、unsigned long long、double 型)

对于 Big Endian 和 Little Endian，8 个字节数据中的字节排列顺序相反。

例：当地址 0x100 中有 8 个字节的数据 0x0123456789abcdef 时

Big Endian:	地址 0x100: 0x01	Little Endian:	地址 0x100: 0xef
	地址 0x101: 0x23		地址 0x101: 0xcd
	地址 0x102: 0x45		地址 0x102: 0xab
	地址 0x103: 0x67		地址 0x103: 0x89
	地址 0x104: 0x89		地址 0x104: 0x67
	地址 0x105: 0xab		地址 0x105: 0x45
	地址 0x106: 0xcd		地址 0x106: 0x23
	地址 0x107: 0xef		地址 0x107: 0x01

(e) 结构体 / 联合体 和 class 型数据

结构体 / 联合体 和 class 型数据的各成员的分配与 Big Endian 相同。但是，按照各数据的长度规则，各成员的字节排列顺序相反。

例：当地址 0x100 中有

```
struct {
    short a;
    int b;
}z = {0x1234, 0x56789abc};
```

时，

Big Endian:	地址 0x100: 0x12	Little Endian:	地址 0x100: 0x34
	地址 0x101: 0x34		地址 0x101: 0x12
	地址 0x102: 空区域		地址 0x102: 空区域
	地址 0x103: 空区域		地址 0x103: 空区域
	地址 0x104: 0x56		地址 0x104: 0xbc
	地址 0x105: 0x78		地址 0x105: 0x9a
	地址 0x106: 0x9a		地址 0x106: 0x78
	地址 0x107: 0xbc		地址 0x107: 0x56

(f) 位域

位域的区域分配也与 Big Endian 相同。但是，按照各数据的长度规则，各区域的字节排列顺序相反。

例：当地址 0x100 中有

```
struct {
    long a:16;
    unsigned int b:15;
    short c:5;
}y={1,1,1};
```

时，

Big Endian:	地址 0x100: 0x00	Little Endian:	地址 0x100: 0x01
	地址 0x101: 0x01		地址 0x101: 0x00
	地址 0x102: 0x00		地址 0x102: 0x02
	地址 0x103: 0x02		地址 0x103: 0x00
	地址 0x104: 0x08		地址 0x104: 0x00
	地址 0x105: 0x00		地址 0x105: 0x08
	地址 0x106: 空区域		地址 0x106: 空区域
	地址 0x107: 空区域		地址 0x107: 空区域

10.1.3 浮点型的规格

(1) 浮点型的内部表示

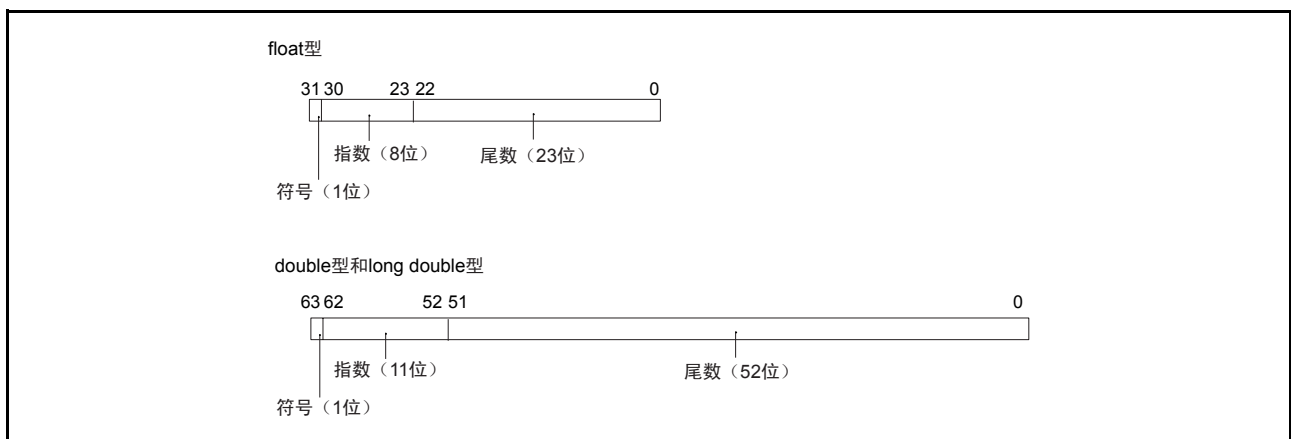
编译程序处理的浮点型的内部表示符合 IEEE 的格式，在此阐述 IEEE 格式的浮点型内部表示的概要。

(a) 内部表示的格式

float 型用 IEEE 的单精度格式（32 位）表示；double 型和 long double 型用 IEEE 的双精度格式（64 位）表示。

(b) 浮点数据格式

float 型、double 型和 long double 型的浮点数据格式如图 10.1 所示：



- 【注】 当指定 `double=float` 时，double 型的内部表示和 float 型相同。
 当指定 `cpu=sh2afpu | sh4 | sh4a` 并且 `fpu=single` 时，double 型、long double 型的内部表示和 float 型相同。
 当指定 `cpu=sh2afpu | sh4 | sh4a` 并且 `fpu=double` 时，float 型的内部表示和 double 型相同。

图 10.1 浮点数据格式

内部表示的各构成要素的含义如下所示：

- (i) 符号
表示浮点型的符号。0 表示正，1 表示负。
- (ii) 指数
用 2 的乘方表示浮点型的指数。
- (iii) 尾数
对应浮点型的有效数字的数据。

(c) 表示的值的种类

浮点型除一般的实数值以外，还能表示无穷大等的值。浮点型表示值的种类如下所示：

- (i) 正规化数
当指数不为 0 或者指数的全部位都不为 1 时，表示一般的实数值。
- (ii) 非正规化数
当指数为 0 并且尾数不为 0 时，表示绝对值较小的实数值。
- (iii) 零
当指数和尾数都为 0 时，表示 0.0 的值。
- (iv) 无穷大
当指数的全部位为 1 并且尾数为 0 时，表示无穷大。

(v) 非数值

当指数的全部位为 1 并且尾数不为 0 时，表示得到“0.0/0.0”、“ ∞/∞ ”、“ $\infty-\infty$ ”等不对应数值的运算结果。

决定浮点型表示值的条件如表 10.18 所示。

表 10.18 浮点型表示值的种类

尾数	0	指数	
		不为 0 并且全部位不为 1	全部位为 1
0	0	正规化数	无穷大
0 以外	非正规化数		非数值

【注】非正规化数表示绝对值小的浮点型（正规化数不能表示的范围），但是有效位数比正规化数少。因此，当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。

当指定 `cpu=sh4 | sh4a` 并且 `denormalize=off` 时，非正规化数就作为 0 处理。

当指定 `cpu=sh4 | sh4a` 并且 `denormalize=on` 时，非正规化数就作为非正规化数处理。

(2) float 型

float 型的内部表示由 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，实际的指数为减去 127 的值，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，后面部分解释为有小数点的尾数。

正规化数的表示值为：

$$(-1)^{< \text{符号} >} \times 2^{< \text{指数} > - 127} \times (1 + < \text{尾数} > \times 2^{-23})$$

例：

```

31 30      23 22      0
1 10000000 11000000000000000000000

```

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 × 2¹ = -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 ~ $2^{23}-1$ 的值，实际尾数假设 2^{23} 的位为 0，后面部分解释为有小数点的尾数。

非正规化数的表示值为：

$$(-1)^{< \text{符号} >} \times 2^{-126} \times (< \text{尾数} > \times 2^{-23})$$

例：

```

31 30      23 22      0
0 00000000 11000000000000000000000

```

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。有关因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 $+\infty$ 、 $-\infty$ 。

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】当单片机为 SH-2E、SH2A-FPU、SH-4 或者 SH-4A 时，尾数最高位为 0 的非数值称为 qNaN；尾数最高位为 1 的非数值称为 sNaN。

对于其他尾数域和符号域的值没有规定。

(3) double 型和 long double 型

double 型和 long double 型的内部表示由 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。

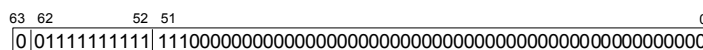
指数为 $1 \sim 2046$ ($2^{11}-2$) 的值, 实际指数为减去 1023 的值, 其范围为 $-1022 \sim 1023$ 。

尾数为 $0 \sim 2^{52}-1$ 的值，实际尾数假设 2^{52} 的位为 1，后面部分解释为有小数点的尾数。

正规化数的表示值为:

$$(-1)^{\langle \text{符号} \rangle} \times 2^{\langle \text{指数} \rangle - 1023} \times (1 + \langle \text{尾数} \rangle \times 2^{-52})$$

例：



符号: +

指数: $1111111111_{(2)} - 1023 = 0$

尾数: $1.111_{(2)} = 1.875$

(2) 表示二进制数。

值 : $1.875 \times 2^0 = 1.875$

(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 $1 \sim 2^{52}-1$ 的值，实际尾数假设 2^{52} 的位为 0，后面部分解释为有小数点的尾数。

非正规化数的表示值为:

$$(-1)^{\langle \text{符号} \rangle} \times 2^{-1022} \times (\langle \text{尾数} \rangle \times 2^{-52})$$

10

值 : 0.875×2^{-1022}

[2-4] 在其它单片机时，为非正规化数。

[3] 如果将符号相反的无穷大相加、符号相同的无穷大相减、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] 如果在将浮点型转换为整数时发生上溢，就不保证结果值。

【注】编译时进行常数表达式的运算。如果此时检测到上溢、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] 操作数的单方或者双方含有非数值时，其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] 操作数的单方或者双方为非数值时，其比较运算或者等同运算的结果对于“!=”总是真，而对于其他总是伪。

(b) 10 进制表示和内部表示之间的转换

本项说明有关源程序中的浮点常数和内部表示之间的转换规格、或者由 C 库函数进行的 ASCII 字符串浮点型的 10 进制表示和内部表示之间的转换规格：

(i) 从 10 进制表示转换为内部表示时，首先将 10 进制表示转换为 10 进制表示的标准形。

10 进制表示的标准形为“ $\pm M \times 10^{\pm N}$ ”，M、N 的范围如下：

[1] float 型的标准形

$$0 \leq M \leq 10^9-1$$

$$0 \leq N \leq 99$$

[2] double 型和 long double 型的标准形

$$0 \leq M \leq 10^{17}-1$$

$$0 \leq N \leq 999$$

如果不能转换为标准形的 10 进制表示，就出现上溢或者下溢。另外，10 进制表示含有多于标准形的有效数字时，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，原来 10 进制表示的 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`。

从内部表示转换为 10 进制表示时，先转换为 10 进制表示的标准形，然后根据指定格式转换为 ASCII 字符串。

(ii) 10 进制表示的标准形和内部表示之间的转换

当指数太大或者太小时，10 进制表示的标准形和内部表示之间的转换无法正确进行。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型，能正确进行“(a)(i) 结果值的舍入方法”所示的舍入。在此范围内不发生上溢或者下溢。

float 型： $0 \leq M \leq 10^9-1$ 、 $0 \leq N \leq 13$

double 型和 long double 型： $0 \leq M \leq 10^{17}-1$ 、 $0 \leq N \leq 27$

[2] 误差的极限值

转换超出 [1] 所示范围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，在转换时就有可能出现上溢或者下溢。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，通过优先级和用“右”或“左”表现的结合性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	适用的表达式
1	++ -- (后缀) () [] -> .	左	后缀表达式
2	++ -- (前缀) ! ~ + - * & sizeof	右	单项表达式
3	(型名)	右	数据型转换式
4	* / %	左	乘除法表达式
5	+ -	左	加减法表达式
6	<< >>	左	位单位的移位表达式
7	< <= > >=	左	关系表达式
8	== !=	左	相等表达式
9	&	左	位单位的 AND 表达式
10	^	左	位单位的 XOR 表达式
11		左	位单位的 OR 表达式
12	&&	左	逻辑 AND 表达式
13		左	逻辑 OR 表达式
14	?:	左	条件表达式
15	= += -= *= /= %= <<= >>= &= = ^=	右	赋值表达式
16	,	左	逗点表达式

10.2 DSP-C 规格

表示 DSP-C 语言编译程序的规格。DSP-C 只在指定 dspc 选项时有效。

10.2.1 定点型

表 10.20 定点型的内部表示

型	长度 (存储器内 的长度)	调整数 (byte)	值的范围		后缀
			最小值	最大值	
<code>__fixed</code>	16bit(16bit)	2	-1.0	$1.0 \cdot 2^{-15}$ (0.999969482421875)	r
<code>long __fixed</code>	32bit(32bit)	4	-1.0	$1.0 \cdot 2^{-31}$ (0.9999999995343387126922607421875)	R
<code>__accum</code>	24bit(32bit)* ¹	4	-256.0	$256.0 \cdot 2^{-15}$ (255.999969482421875)	a
<code>long __accum</code>	40bit(64bit)* ¹	4	-256.0	$256.0 \cdot 2^{-31}$ (255.9999999995343387126922607421875)	A

【注】*1 保存到存储器时向右靠，起始部分进行符号扩展。

例：

(i) `(__accum)128.5a` → 00 40 40 00

(ii) `(long __accum)(-256.0A)` → FF FF FF 80 00 00 00 00

10.2.2 型限定词

(1) 存储器型限定词

使用以下型限定词，明确地指定数据保存的 X/Y 存储器：

`__X`：将数据保存到 X 存储器

`__Y`：将数据保存到 Y 存储器

存储器型限定词和段的对应如表 10.21 所示。

表 10.21 存储器型限定词的规格

名称	段	内容
1 常数区	<code>\$XC</code>	const 型的数据 (保存到 X 存储器)
	<code>\$YC</code>	const 型的数据 (保存到 Y 存储器)
2 初始化的数据区	<code>\$XD</code>	有初期值的数据 (保存到 X 存储器)
	<code>\$YD</code>	有初期值的数据 (保存到 Y 存储器)
3 未初始化的数据区	<code>\$XB</code>	无初期值的数据 (保存到 X 存储器)
	<code>\$YB</code>	无初期值的数据 (保存到 Y 存储器)

【注】*1 不能给同一变量指定 2 个存储器型限定词。否则就输出错误。

*2 在通过指定 `#pragma section` 进行段转换时，带存储器型限定词的变量不为段转换对象。

*3 不能用存储器型限定词更改函数本身的存储器保存目标。

*4 如果给没有指定 `static` 的局部变量指定存储器型限定词，就输出警告，并且存储器型限定词的指定无效。但是，对于指向带存储器型限定词的数据的指针，其指定有效。

使用 `__X/__Y` 型限定词时的存储器的保存例子如下所示：

- `__X int a;` // 保存到 X 存储器
- `int __X b;` // 保存到 X 存储器
- `__Y int *c;` // 指向 Y 存储器内 int 型的指针（存储器不定）
- `int __Y *d;` // 指向 Y 存储器内 int 型的指针（存储器不定）
- `int *__Y e;` // 指向 int 型的指针（保存到 Y 存储器）
- `__X int *__Y f;` // 指向 X 存储器 int 型的指针（保存到 Y 存储器）

(2) 饱和型限定词

用于指定饱和运算的型限定词如下：

`__sat`

`__sat` 型限定词只能用于 `__fixed` 型和 `long __fixed` 型数据。否则就发生错误。

如果在表达式中至少指定了 1 个 `__sat`，就进行饱和运算。

例：

```
__fixed a;
__sat __fixed b;
__fixed c;

a=-0.75r;
b=-0.75r;
c=a+b;          // c=-1.0r
```

(3) 循环型限定词

用于指定模寻址的型限定词如下：

`__circ`

只能给指定了存储器型限定词（`__X/__Y`）的 `__fixed` 型的 1 维数组或者指针指定模寻址。否则，就发生错误。

模寻址以内部函数 `set_circ_x()/set_circ_y()` 和 `clr_circ()` 之间的 1 维数组或者指针为对象。有关内部函数的规格请参照“10.3.3 内部函数”。

如果同时对多个数组指定模寻址，或者参照了在上述内部函数之外指定 `__circ` 的数组和指针，就不保证运行。

如果指定负方向的模寻址，就不保证运行。

在连接时，需要将模寻址的数据配置到高 16 位地址相同的存储器。

不能直接参照数组的内容。

【注】 以下任何情况下都不保证运行（也可能输出警告）。

- 指定了 `optimize=0` 或者 `optimize=debug_only`
- 给非局部变量指定了 `__circ` 指针
- 给 `__circ` 指针指定了 `volatile`
- 只更新而没参照 `__circ` 指针
- 在内部函数 `set_circ_x()/set_circ_y()` 和 `clr_circ()` 之间调用了函数

10.2.3 常数

能通过给数值附加表 10.20 所示的后缀，明确地表示定点常数。但是，如果数值包含整数，就分别将附加后缀 r、R 的常数作为 `__accum` 型、`long __accum` 型的常数处理。

在省略常数后缀时，作为 `double` 型常数处理；在指定 `fixed_const` 选项时，作为定点常数处理。

没有饱和型的定点常数，但通过明确的型转换，可作为饱和型处理。

例：

```
__fixed a;
__fixed b;

a=-0.75r;
b=a + (__sat __fixed)(-0.75r);    // b=-1.0r。
```

因为一元负运算符不是定点常数的一部分，所以作为 `__fixed` 型，`-1.0r` 不是有效的值。`-1.0r` 需要记述为 `(-0.5r-0.5r)`（本手册为了简单化，有时记述为 `-1.0r`，表示 `__fixed` 型的 `-1.0`）。

如果小数超过各型能表示的精度或者除符号以外的整数超过 `__accum` 型、`long __accum` 型的最大整数值 255，就输出警告。舍去超出精度的小数部分，而将整数溢出部分的最低位作为符号处理，舍去其他溢出的部分。

10.2.4 型转换

有关型转换的规则如表 10.22 所示。

表 10.22 型转换的规则

转换		规格
<code>__fixed</code>	<code>→ long __fixed</code>	低 16 位初始化为零。
<code>__accum</code>	<code>→ long __accum</code>	值不变。
<code>long __fixed</code>	<code>→ __fixed</code>	舍去低 16 位。
<code>long __accum</code>	<code>→ __accum</code>	小数的精度降低。
<code>__fixed</code>	<code>→ __accum</code>	对高 8 位进行符号扩展。
<code>long __fixed</code>	<code>→ long __accum</code>	值不变。
<code>__fixed</code>	<code>→ long __accum</code>	对高 8 位进行符号扩展，低 16 位初始化为零。 值不变。
<code>long __fixed</code>	<code>→ __accum</code>	对高 8 位进行符号扩展，舍去低 16 位。 小数的精度降低。
<code>__accum</code>	<code>→ __fixed</code>	舍去高 8 位，第 9 位作为符号位。
<code>long __accum</code>	<code>→ long __fixed</code>	整数为 0 时，值不变。
<code>__accum</code>	<code>→ long __fixed</code>	舍去高 8 位，第 9 位作为符号位。 整数为 0 时，值不变。
<code>long __accum</code>	<code>→ __fixed</code>	舍去高 8 位和低 16 位，第 9 位作为符号位。 整数为 0 时，值不变。 小数的精度降低。
<code>__fixed</code>	<code>→ 带符号的整数型</code>	在 <code>-1.0r</code> 、 <code>-1.0R</code> 时为 <code>-1</code> ，否则为 <code>0</code> 。
<code>long __fixed</code>	<code>→ 带符号的整数型</code>	
<code>__accum</code>	<code>→ 带符号的整数型</code>	舍去小数。
<code>long __accum</code>	<code>→ 带符号的整数型</code>	转换后的值为 <code>-256 ~ 255</code> 之间的整数。

转换	规格
<code>__fixed</code> → 无符号的整数型	在 -1.0r、-1.0R 时为转换后的型的最大值，否则为 0。
<code>long __fixed</code> → 无符号的整数型	
<code>__accum</code> → 无符号的整数型	舍去小数。
<code>long __accum</code> → 无符号的整数型	在正值时，转换后的值为 0 ~ 255 的整数。 在负值时，为 (转换前的值 +1+ 转换后的型的最大值)。
带符号的整数型 → <code>__fixed</code>	转换前的最高位为转换后的最高位，其他的位全部为 0。
带符号的整数型 → <code>long __fixed</code>	
带符号的整数型 → <code>__accum</code>	整数为值的低 9 位，小数为 0。
带符号的整数型 → <code>long __accum</code>	
无符号的整数型 → <code>__fixed</code>	转换后的全部位为 0。
无符号的整数型 → <code>long __fixed</code>	
无符号的整数型 → <code>__accum</code>	整数为值的低 9 位，小数为 0。
无符号的整数型 → <code>long __accum</code>	
定点 → 浮点	能用转换后的型表示的值和原来的值相同； 不能表示时，向最近的值舍入。
浮点 → 定点	小数相同于定点 → 浮点的转换。 整数相同于浮点 → 整数的转换。 当整数在定点能表示的范围内时，保存原来的值。否则，就将溢出部分的最低位作为符号位处理。 另外，即使对转换后的型指定了饱和处理，也不进行饱和处理。

10.2.5 算术转换

当参加运算的 2 个操作数的型不同时，在调整为图 10.2 中上层的类型后进行运算。

如果记述了图 10.2 中没有上下关系的型（整型 ⇔ 定点型、`__accum` ⇔ `long __fixed`）之间的运算，就会出错；如果需要运算，就必须通过明确的型转换来进行型的调整。

但是，当结果值在保证的范围内，也可在考虑到效率等方面而不根据上述的转换规则进行运算。

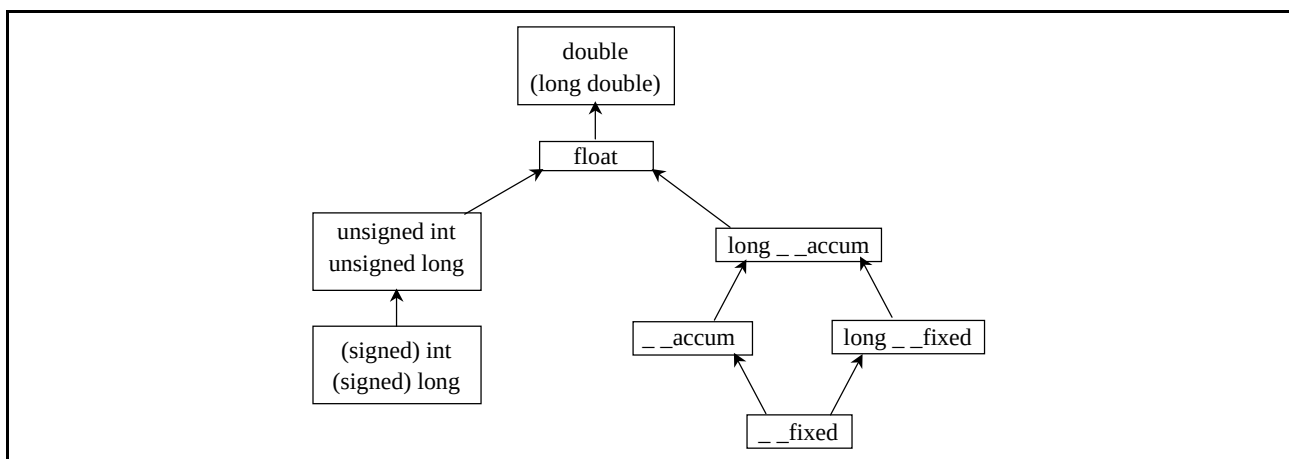


图 10.2 算术转换的规则

10.2.6 指针转换

(a) `__circ` 型限定词

如果将指定 `__circ` 的指针转换为没有指定 `__circ` 的型，就输出警告。不能使用模寻址。

如果将没有指定 `__circ` 的指针转换为指定 `__circ` 的型，就不输出警告，但是不进行模寻址。

(b) 存储器型限定词 (`__X/__Y`)

如果转换为其他存储器型限定词，就会出错。

10.2.7 运算符

不能对定点型指定以下的运算符，否则就会出错。

- 1 的补码运算符 (`~`)
- 逐位的 AND 运算符 (`&`、`&=`)
- 逐位的 OR 运算符 (`|`、`|=`)
- 逐位的 XOR 运算符 (`^`、`^=`)
- 移位运算符 (`<<`、`>>`、`<<=`、`>>=`)
- 余数运算符 (`%`、`%=`)

`sizeof` 运算符的值如表 10.23 所示。

表 10.23 `sizeof` 运算符的值

	型	值
1	<code>__fixed</code>	2
2	<code>long __fixed</code>	4
3	<code>__accum</code>	4
4	<code>long __accum</code>	8

10.2.8 库

(a) `fixed.h`

用 `<fixed.h>` 定义定点型的极限值。详细内容请参照“10.4.1 标准 C 库”。

(b) `stdio.h`

用于定点的转换说明符如表 10.24 所示。

表 10.24 定点转换说明符

	转换说明符	含义
1	<code>%r</code>	<code>__fixed</code> 值
2	<code>%lr</code>	<code>long __fixed</code> 值
3	<code>%a</code>	<code>__accum</code> 值
4	<code>%la</code>	<code>long __accum</code> 值
5	<code>%P</code>	<code>__circ</code> 指针值

定点的转换以 `%f` 转换（浮点转换）为基准。`%P` 进行和 `%p`（指针转换）相同的转换。

(c) `stdlib.h`

用于定点的函数如表 10.25 所示。详细内容请参照“10.4.1 标准 C 库”。

表 10.25 函数一览表

种类	函数名
1 字符串数值转换函数	<code>long __fixed atolfixed(const char * nptr);</code>
	<code>long __accum atolaccum(const char * nptr);</code>
	<code>long __fixed strtolfixed(const char * nptr, char ** endptr);</code>
	<code>long __accum strtolaccum(const char * nptr, char ** endptr);</code>
2 存储区管理函数	<code>void __X *calloc__X(size_t nelem, size_t elsize);</code>
	<code>void free__X(void __X *ptr);</code>
	<code>void __X *malloc__X(size_t size);</code>
	<code>void __X *realloc__X(void __X *ptr, size_t size);</code>
	<code>void __Y *calloc__Y(size_t nelem, size_t elsize);</code>
	<code>void free__Y(void __Y *ptr);</code>
	<code>void __Y *malloc__Y(size_t size);</code>
	<code>void __Y *realloc__Y(void __Y *ptr, size_t size);</code>

【注】 *1 需要准备用于 X/Y 存储器的低级接口例程。

`char __X *sbrk__X(int size);`

`char __Y *sbrk__Y(int size);`

详细内容请参照“9.2.2(6) 低级接口例程”。

(d) `string.h`

用于定点的函数如表 10.26 所示。详细内容请参照“10.4.1 标准 C 库”。

表 10.26 函数一览表

种类	函数名
1 存储区内容复制函数	<code>void __X *memcpy__X__X(void __X * s1, const void __X * s2, size_t n);</code>
	<code>void __X *memcpy__X__Y(void __X * s1, const void __Y * s2, size_t n);</code>
	<code>void __Y *memcpy__Y__X(void __Y * s1, const void __X * s2, size_t n);</code>
	<code>void __Y *memcpy__Y__Y(void __Y * s1, const void __Y * s2, size_t n);</code>

(e) DSP 库

当指定 `dspc` 选项时，能给 `short` 型数组和指针的参数指定 `__fixed` 型数组和指针。详细内容请参照“10.4.5 DSP 库”。

10.3 扩展功能

编译程序支持以下的扩展功能：

- #pragma
- 段地址运算符
- 内部函数

10.3.1 #pragma

#pragma 一览如下所示：

表 10.27 存储器分配的扩展功能

	#pragma	功能
1	#pragma section	指定段的转换
2	#pragma abs16 #pragma abs20 #pragma abs28 #pragma abs32	指定地址区
3	#pragma stacksize	建立堆栈段

表 10.28 函数的扩展功能

	#pragma	功能
1	#pragma interrupt	建立中断函数
2	#pragma inline	指定函数的 inline 展开
3	#pragma inline_asm	汇编描述函数的 inline 展开
4	#pragma regsave #pragma noregsave #pragma noregalloc	控制寄存器的保存 / 恢复代码的输出
5	#pragma entry	建立入口函数
6	#pragma ifunc	抑止浮点寄存器的保存 / 恢复
7	#pragma tbr	指定 TBR 寄存器相对寻址的函数调用

表 10.29 其他扩展功能

	#pragma	功能
1	#pragma global_register	分配全局变量的寄存器
2	#pragma gbr_base #pragma gbr_base1	指定 GBR 基址的变量
3	#pragma bit_order	指定位域的排列顺序
4	#pragma pack #pragma unpack	指定结构体、联合体、类的调整数
5	#pragma address	给变量指定绝对地址

上述扩展功能中有可指定数据成员和函数成员的功能，指定方法是 (类名 :: 成员名)。可指定的成员种类请参照各功能的记述方法。

(1) 有关存储器分配的扩展功能

段的转换指定

#pragma section

格 式 `#pragma 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 10.30 所示。

表 10.30 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	<code>#pragma section <xx></code>	P*1	P<xx>
2	常数区		C*1	C<xx>
3	初始化的数据区		D*1	D<xx>
4	未初始化的数据区		B*1	B<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 注 必须在函数定义以外的位置声明 `#pragma section`。
1 个文件中能声明的段名最多为 2045 个。
如果和存储器型限定词 (`_X/_Y`) 同时指定，`#pragma section` 的指定就无效。
在使用 C++ 语言时，必须对编译单位的最初声明使用 `#pragma section` 的指定。

```

例
    #pragma section _A
    extern int N;
    #pragma section

    int N;

```

```

/* 变量 N 分配在 B_A 段 */
-----
extern int N;

#pragma section _A
int N;
#pragma section

/* 变量 N 分配在 B 段 ( 默认值 ) */
-----

```

在指定静态类成员变量的段时，需要对类的成员声明和实际的状态定义进行 `#pragma section` 的指定。

例

```

class A
{
    private:
        // 没有初始值
        #pragma section DATA
        static int data_;
        #pragma section

        // 有初始值
        #pragma section TABLE
        static int table_[2];
        #pragma section
};

// 没有初始值
#pragma section DATA
int A::data_;
#pragma section

// 有初始值
#pragma section TABLE
int A::table_[2]={0, 1};
#pragma section

```

地址区的指定

#pragma abs16
 #pragma abs20
 #pragma abs28
 #pragma abs32

格 式 #pragma abs16 [()< 识别符 >[, ...]]
 #pragma abs20 [()< 识别符 >[, ...]]
 #pragma abs28 [()< 识别符 >[, ...]]
 #pragma abs32 [()< 识别符 >[, ...]]

说 明 将 #pragma abs16|abs20|abs28|abs32 声明的变量或者函数分配到下表所示的存储空间进行处理。能通过此项操作减小程序大小。
 能给识别符指定变量、全局函数、静态数据成员和函数成员。

表 10.31 地址范围

#pragma 扩展名	地址范围	
	低位	高位
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F ^{*1}
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

【注】*1 注意 0x07FFFF7F 时的情况。

备 注 #pragma abs16|abs20|abs28|abs32 不能指定自动目标或者非静态数据成员。
 必须将 #pragma abs16|abs20|abs28|abs32 声明的变量和函数分配到指定的地址范围内。
 不能给相同的识别符指定多个地址范围。
 如果和 abs16|abs20|abs28|abs32 选项同时指定, #pragma abs16|abs20|abs28|abs32 就有效。
 如果和 #pragma gbr_base|gbr_base1 选项同时指定, #pragma abs16|abs20|abs28|abs32 就无效。

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, ALIGN=4`

`.RES.B     100`

备 注 长度的 < 常数 > 必须指定 4 的倍数。
只能在文件中指定一次 `#pragma stacksize`。

(2) 有关函数的扩展功能

中断函数的建立

#pragma interrupt

格 式 `#pragma interrupt [( )<函数名>[( 中断规格 )][, ...[ ]]`

说 明 用 `#pragma interrupt` 声明中断函数。
能给函数名指定全局函数和静态函数成员。
中断规格一览如表 10.32 所示。

表 10.32 中断规格一览表

项 目	格 式	选 项	指 定 内 容
1 堆栈转换的指定	<code>sp=</code>	<code>{<变量> </code> <code>&<变量> </code> <code><常数> </code> <code><变量>+<常数> </code> <code>&<变量>+<常数></code> <code>}</code>	用变量或者常数指定 新堆栈的地址 <code><变量></code> : 变量 (指针型) <code>&<变量></code> : 变量 (目标型) 的地址 <code><常数></code> : 常数值
2 陷阱指令返回的指定	<code>tn=</code>	<code><常数></code>	用 TRAPA 指令指定结束常数值 (陷阱向量号)
3 寄存器组的指定	<code>resbank</code>	无	抑止以下寄存器的保存代码的输出: R0 ~ R14、GBR、MACH、MACL、PR 当没有指定 <code>tn</code> 时, 在 RTE 指令前生成 RESBANK 指令
寄存器组的切换和 RTS 指令返回的指定	<code>sr_rts</code>	无	用 RTS 指令结束 只对函数内使用的寄存器生成保存代码 在函数的最后将 SR 寄存器的 RB 位和 BL 位置位
中断处理函数的指定	<code>bank</code>	无	存在 <code>sr_jsr()</code> 内部函数时, 生成 SSR/SPC 寄存器的保存代码 抑止 R0 ~ R7 寄存器的保存代码的输出 对其它函数内使用的寄存器生成保存代码
RTS 指令返回的指定	<code>rts</code>	无	用 RTS 指令结束 抑止 R0 ~ R7 寄存器的保存代码的输出 对其它函数内使用的寄存器生成保存代码

使用 `#pragma interrupt` 声明的函数在函数处理前后保证全部寄存器 (在函数入口 / 出口处, 保存和恢复函数内使用的全部寄存器), 通常用 RTE 指令返回。如果指定陷阱指令返回, 就用 TRAPA 指令返回。如果没有指定中断规格, 就作为单纯的中断函数处理。另外, 能重复进行堆栈转换的指定和陷阱指令返回的指定。

例

```
extern int STK[100];
#pragma interrupt(f(sp=STK+400, tn=10),A::g)
class A{
public:
    static void g();
};
```

说明

(a) 堆栈转换的指定

将 **STK+400** 设定为中断函数 **f** 使用的堆栈指针。

(b) 陷阱指令返回的指定

在中断函数结束时，通过 **TRAPA #10** 开始陷阱异常的处理。陷阱异常处理开始时的 **SP** 如图 10.3 所示。必须在陷阱程序中使用 **RTE** 指令恢复 **PC**（程序计数器）和 **SR**（状态寄存器），然后从中断函数返回。

(c) C++ 程序中能指定的函数成员是静态函数成员。例如，将类 **A** 的静态函数成员 **g** 指定为中断函数。必须注意：不能指定非静态函数成员。

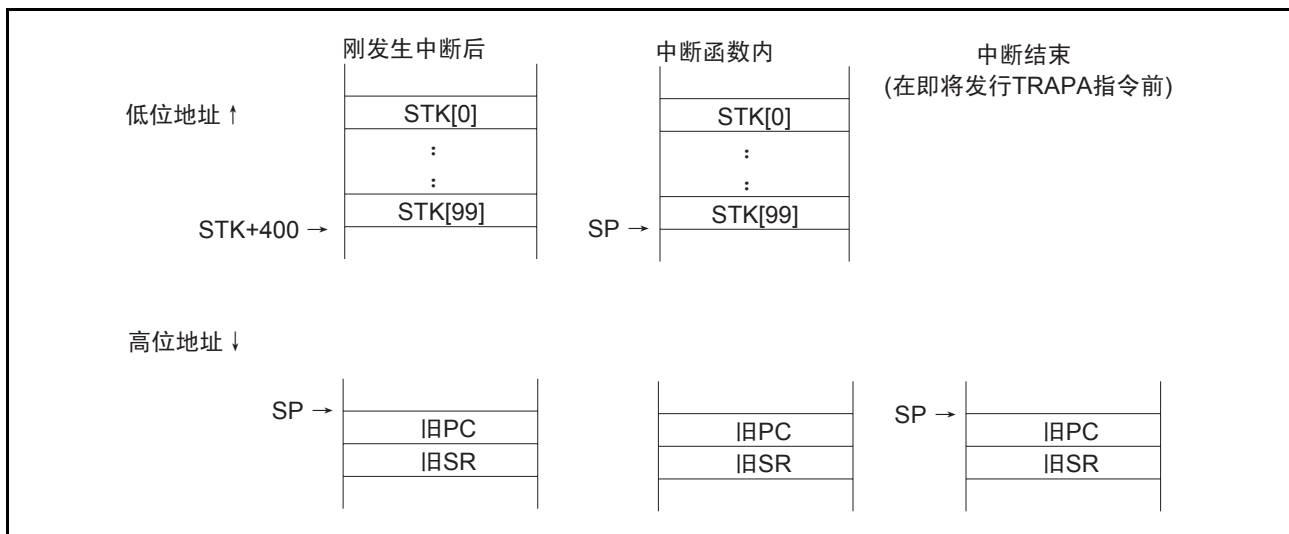


图 10.3 中断函数的堆栈使用例子

通过指定 **sr_rts/bank** 以及使用内部函数 **sr_jsr()**，能建立多重中断函数。

例

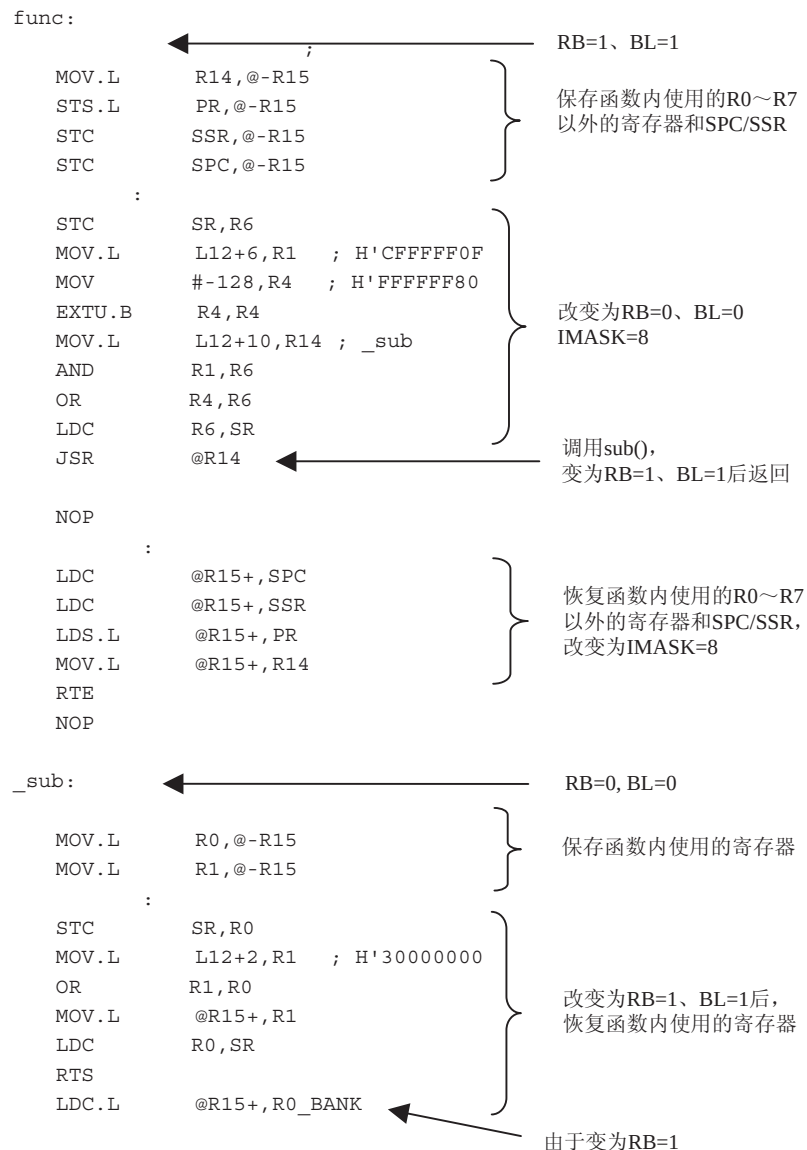
```
#include <machine.h>

// 声明处理函数
#pragma interrupt (func(bank))
void func();

// 声明中断处理函数
#pragma interrupt (sub(sr_rts))
void sub();
void func() {
    :
    sr_jsr(sub, 8); // RB=0、BL=0
                  // 中断级设定为8
                  // 调用sub()
    :
}

void sub() {
    :
}

—
```

通过使用 `rts/bank` 的指定，能建立有效利用寄存器组的高效率的中断函数。

例

```
#include <machine.h>

// 声明处理函数
#pragma interrupt (func(bank))
void func();

// 声明中断处理函数
#pragma interrupt (sub(rts))
void sub();
void func() {
    :
    sub();
    :
}

void sub() {
    :
}
```

```

_func:
    STS.L    PR, @-R15      ; RB=1, BL=1
    :
    MOV.L    L12, R14; _sub
    JSR      @R14
    NOP
    :
    LDS.L    @R15+, PR
    RTE
    NOP

_sub:
    MOV.L    R14, @-R15
    MOV.L    R13, @-R15
    :
    MOV.L    @R15+, R13
    RTS
    MOV.L    @R15+, R14

```

保存函数内使用的R0~R7以外的寄存器

恢复函数内使用的R0~R7以外的寄存器

保存函数内使用的R0~R7以外的寄存器

恢复函数内使用的R0~R7以外的寄存器

备 注

只在指定 `cpu=sh2a|sh2afpu` 时，`resbank` 的指定才有效。

必须在 `resbank` 指定的函数发生中断前，允许寄存器组的使用。

如果指定 `resbank` 和 `tn`，就不生成寄存器保存码和 `RESBANK` 指令。此时，需要在陷阱程序中生成 `RESBANK` 指令。

当从 `resbank` 指定的函数恢复时，用 `#pragma global_register` 指定的变量即使在中断处理中被改写，也恢复为中断前的值。

因为 SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的中断运行和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 不同，所以需要中断处理程序。另外，对于 SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP，如果给 `#pragma interrupt` 和 `#pragma noregsave` 指定相同的函数，就只保存和恢复该函数内使用的函数调用前要保证的寄存器。

如果指定 `repeat` 选项，就保存和恢复 RS/RE 寄存器。

如果指定 `dspe` 选项，就保存和恢复 DSP 寄存器（X0、X1、Y0、Y1、M0、M1、A0、A0G、A1、A1G）、DSR 寄存器和 MOD 寄存器。

能对中断函数的定义指定的函数是全局函数（C/C++ 程序）和静态函数成员（C++ 程序）。

另外，函数返回值的型只是 `void` 型，不能指定 `return` 语句的返回值，否则就输出错误。

例

```

#pragma interrupt (f1(sp=100), f2)
void f1(){...}.....(a)
int f2(){...}.....(b)

```

说明

(a) 是正确的声明。

(b) 是错误的的声明，因为函数返回值的型不是 `void` 型，所以输出错误信息。

只在指定 `cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp` 时，`sr_rts/bank/rts` 的指定才有效。`sr_rts/bank/rts` 和其他中断规格的同时指定的有效组合如下：

	#pragma interrupt					#pragma noregsave
	sp	tn	sr_rts	bank	rts	
sr_rts	错误	错误	错误	错误	错误	错误
bank	有效	错误	错误	错误	错误	有效
rts	错误	错误	错误	错误	错误	错误

在以下任一情况下，就出错：

- 不从内部函数 `sr_jsr()` 调用指定 `sr_rts` 的函数时
- 调用指定 `bank` 的函数时
- 从即不指定 `bank` 也不指定 `rts` 的函数调用指定 `rts` 的函数时

不能在程序中调用声明为中断函数的函数，否则就输出错误。但是，如果在没有中断函数声明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。

例 1

```

有中断函数声明时，
#pragma interrupt (f1)
void f1(){...}
int f2(){f1();}.....(a)

```

说明

因为将函数 `f1` 声明为中断函数，所以不能在程序中进行调用，对 (a) 输出错误信息。

例 2

```

没有声明中断函数时，
int f1();
int f2(){f1();}.....(b)

```

说明

因为没有将函数 `f1` 声明为中断函数，所以作为非中断函数 `int f1();` 生成目标。如果通过其他编译单位，将函数 `f1` 声明为中断函数，就不保证执行时的运行。

函数的 inline 展开

#pragma inline

格 式 `#pragma inline [( ) < 函数名 > [, ... [ ) ]]`

说 明 用 `#pragma inline` 声明要 `inline` 展开的函数。
 能给函数名指定全局函数和静态函数成员。
 在调用的位置，对用 `#pragma inline` 指定函数名的函数和指定函数说明符 `inline`（C++ 语言）的函数进行 `inline` 展开。

例 源程序

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
main()
{
    x=func(10,20);
}
```

展开信息

```
int x;
main()
{
    int func_result;
    {
        int a_1=10, b_1=20;
        func_result=(a_1+b_1)/2;
    }
    x=func_result;
}
```

备 注 在以下情况下不进行 `inline` 展开：

- 函数定义在 `#pragma inline` 指定前。
- 有可变参数的函数。
- 在函数内参照参数地址。
- 通过展开对象函数的地址进行调用。

 必须在函数定义前指定 `#pragma inline`。

 对于 `#pragma inline` 指定的函数，也生成外部定义。如果在各程序文件中记述了 `inline` 函数的实体，就必须在函数的声明时指定 `static`。如果指定 `static`，就不生成外部定义。`inline`（C++ 语言）指定的函数不生成外部定义。

汇编记述函数的 inline 展开

#pragma inline_asm

格 式 `#pragma inline_asm [( ) < 函数名 > [(size= 数值) ] [, ... ( ) ]]`

说 明 对 `#pragma inline_asm` 声明的汇编记述函数进行 `inline` 展开。
 只能给函数名指定全局函数，而不能指定函数成员。
 因为和通常的函数调用一样，将汇编程序嵌入式 `inline` 函数的参数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 `inline` 函数的返回值设定到 `R0`；在 SH-2E、SH2A-FPU、SH-4、SH-4A 时，将单精度浮点型的返回值设定到 `FR0`；在 SH2A-FPU、SH-4、SH-4A 时，将双精度浮点型的返回值设定到 `DR0`。设定返回值的寄存器因和选项的组合而不同。详细内容请参照表 9.7。
 能通过 `(size= 数值)` 来指定汇编程序嵌入式 `inline` 函数的大小。

例

源程序

```
#pragma inline_asm(sub)
static int sub(int a)
{
    ROTLR4
    MOVR4,R0
}
int x;
main()
{
    x=0x55555555;
    x=sub(x);
}
```

输出结果 (一部)

```

:
_main                                ;function main
                                     ;frame size = 4
MOV.L    R14,@-R15
MOV.L    L220+2,R14                ;x
MOV.L    L220+6,R3                 ;H'55555555
MOV.L    R3,@R14
MOV      R3,R4
BRA      L219
NOP
L220:
      .RES.W    1
      .DATA.L   x
      .DATA.L   H'55555555
L219:
      ROTL      R4
```

```

MOV      R4,R0
.ALIGN   4
MOV.L    R0,@R14
RTS
MOV.L    @R15+,R14
.SECTION B,DATA,ALIGN=4
_x:                                     ;static: x
.RES.L    1
.END

```

备 注

必须在函数定义前指定 `#pragma inline_asm`。

对于用 `#pragma inline_asm` 指定的函数，也生成外部定义。

在汇编程序嵌入式 `inline` 函数内使用标号时，只能使用局部标号。

在汇编程序嵌入式 `inline` 函数内在函数的出口 / 入口，使用要保证的寄存器（参照表 9.5）时，需要在汇编程序嵌入式 `inline` 函数的开头和最后对这些寄存器进行保存和恢复。另外，在使用 FR12 ~ FR15（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时）、DR12 ~ DR14（单片机为 SH2A-FPU、SH-4、SH-4A 时）的寄存器时，也需要在汇编程序嵌入式 `inline` 函数的开头和最后对这些寄存器进行保存和恢复。

不能在汇编程序嵌入式 `inline` 函数的最后记述 RTS。

在使用本功能时，必须指定 `code=asmcode` 进行编译。

用（`size= 数值`）指定的数值不能小于实际目标长度的值，否则就不保证运行。另外，如果数值为浮点型或者为小于等于 0 的值，就会出错。

在本函数内使用由 `#pragma global_register` 功能指定的寄存器时，也需要在汇编程序嵌入式 `inline` 函数的开头和最后对这些寄存器进行保存和恢复。

能指定的函数只有全局函数，而不能指定函数成员。

不能使用会生成文字池的记述（`MOV.L #100000,R0` 等）。

寄存器的保存/恢复代码的控制功能

#pragma regsave
#pragma noregsave
#pragma noregalloc

格 式 **#pragma regsave** [()<函数名>[, ...[]]
 #pragma noregsave [()<函数名>[, ...[]]
 #pragma noregalloc [()<函数名>[, ...[]]

说 明 能给函数名指定全局函数和函数成员。
 用 **#pragma regsave** 指定的函数在函数的出口 / 入口，保存和恢复要保证的寄存器（参照表 9.5）。另外，跨越函数调用，生成不分配 R8-R14（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时还有 FR12-FR15）的目标。
 用 **#pragma noregsave** 指定的函数在函数的出口 / 入口，保存和恢复要保证的寄存器。用 **#pragma noregalloc** 指定的函数在函数的出口 / 入口，不保存也不恢复要保证的寄存器。另外，跨越函数调用，生成不分配 R8-R14（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时还有 FR12-FR15）的目标。
 能对同一函数重复指定 **#pragma regsave** 和 **#pragma noregalloc**。此时，在函数的出口 / 入口，保存和恢复全部要保证的寄存器 R8-R14（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时还有 FR12-FR15）。同时，跨越函数调用，生成不分配寄存器 R8-R14（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时还有 FR12-FR15）的目标。
 能在以下条件下使用 **#pragma noregsave** 指定的函数：
 - 不从其他函数调用，而作为最初启动的函数使用。
 - 从指定 **#pragma regsave** 的函数调用。
 - 从指定 **#pragma regsave** 的函数，通过指定 **#pragma noregalloc** 的函数调用。

例 **#pragma noregsave(f,A::j)**
 #pragma noregalloc(g)
 #pragma regsave(h)
 class A{
 public:
 static void j();
 };
 void f();
 void g();
 void h();
 void h() {
 g();
 f(); /* 从 #pragma regsave 函数 (h) 直接调用 #pragma noregsave 函数 (f) */
 }
 void g() {
 f(); /* 从 #pragma regsave 函数 (h)，通过 #pragma noregsave 函数 (f,A::j) 的 */
 /* #pragma noregalloc 函数 (g) 调用 */
 A::j();
 }
 void f(){}

备 注 必须注意：如果不通过上述方法调用指定 **#pragma noregsave** 的函数，就不能保证结果。

入口函数的建立

#pragma entry

格 式 `#pragma entry [( ) <函数名> [(sp=<常数>)] [ ) ]]`

说 明 将 <函数名> 指定的函数作为入口函数处理。
入口函数不建立任何寄存器的保存和恢复代码。
在单片机为 SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 时，如果指定 `sp=<常数>` 或者进行 `#pragma stacksize` 声明，就在函数开头输出堆栈指针的初始设定代码。

例 1

```
#pragma entry INIT(sp=0x10000)
void INIT() {
    :
}
```

< 代码展开例 >

```
.SECTION P, CODE
_INIT:
    MOV.L    L1, R15
    :
L1:      .DATA.L  H'00010000
    :
```

例 2

```
#pragma stacksize 100
#pragma entry INIT
void INIT() {
    :
}
```

< 代码展开例 >

```
.SECTION S, STACK
.RES.B    100
.SECTION P, CODE
_INIT:
    MOV.L    L1, R15
    :
L1:      .DATA.L  STARTOF S + SIZEOF S
    :
```

备 注 必须在函数声明前指定 `#pragma entry`。
在一个装入模块中最多只能指定 1 个入口函数。
必须将 <常数> 指定为 4 的倍数。
在指定 `cpu=sh1|sh2|sh2e|sh2a|sh2afpu|sh2dsp` 时，`sp=<常数>` 的指定无效。

浮点寄存器的保存/恢复的抑止

#pragma ifunc

格 式 `#pragma ifunc [( ) < 函数名 > ( )]`

说 明 在用 < 函数名 > 指定的函数中，抑止浮点寄存器的保存和恢复。

备 注 必须在函数声明前指定 `#pragma ifunc`。
 只在指定 `cpu=sh2e|sh2afpu|sh4|sh4a` 时，`#pragma ifunc` 的指定才有效。
 如果在指定 `#pragma ifunc` 的函数内使用浮点，就会出错。

例

```
float f;
#pragma ifunc(func)
void func(void) {
    f=0.0f;    /* 错误 */
}
```

TBR 寄存器相对寻址

#pragma tbr

格 式 `#pragma tbr [( ) < 函数名 > [( {sn=< 段名 > | ov=< 偏移值 > } ) ] [, ... ( ) ]]`

说 明 用 `#pragma tbr` 声明通过 TBR 寄存器相对寻址的函数调用。
 规格的详细内容如下：

(a) `#pragma tbr` < 函数名 >

在调用 < 函数名 > 的函数时，通过 TBR 寄存器相对寻址进行函数调用。
 另外，如果有 < 函数名 > 的定义，就将 `func` 的地址输出到 \$TBR 段。

例

```
#pragma tbr func
void func() {}
#pragma section AA
void func2() {
    func();
}
```

< 代码展开例 >

```
_func:
    RTS/N

_func2:
    STS.L   PR, @-R15
    JSR/N   @@($_func - (START OF $TBR), TBR)
    LDS.L   @R15+, PR
    RTS/N
```

```

        .SECTION $TBR,DATA,ALIGN=4
$_func:
        .DATA.L _func

```

(b) #pragma tbr <函数名>(sn=<段名>)

在调用 <函数名> 的函数时，通过 TBR 寄存器相对寻址进行函数调用。

另外，如果有 <函数名> 的定义，就将 func 的地址输出到 \$TBR<段名> 选项。

例

```

#pragma tbr func(sn=_A)
void func() {}
#pragma section AA
void func2() {
    func();
}

```

<代码展开例>

```

_func:
    RTS/N

_func2:
    STS.L    PR,@-R15
    JSR/N    @@($_func-(START OF $TBR_A),TBR)
    LDS.L    @R15+,PR
    RTS/N

```

```

        .SECTION $TBR_A,DATA,ALIGN=4
$_func:
        .DATA.L _func

```

(c) #pragma tbr <函数名>(ov=<偏移值>)

在调用 <函数名> 的函数时，通过 TBR 寄存器相对寻址进行函数调用。

disp 值为 <偏移值>，必须将 <偏移值> 指定为 0 ~ 1020 范围内的 4 的倍数。

编译程序不建立 TBR 地址表。

例

```

#pragma tbr func(ov=32)
void func() {}
#pragma section AA
void func2() {
    func();
}

```

<代码展开例>

```

_func:
    RTS/N

```

```
_func2:
    STS.L   PR,@-R15
    JSR/N   @@(32,TBR)
    LDS.L   @R15+,PR
    RTS/N
```

备 注 只在指定 `cpu=sh2a|sh2afpu` 时，`#pragma tbr` 才有效。
 如果和 `tbr` 选项同时指定，`#pragma tbr` 选项就有效。
 在指定 `pic=1` 时，`#pragma tbr` 的指定无效。
 如果给相同函数同时指定 `sn` 和 `ov`，最初指定的选项有效。
 必须在函数调用前，通过内部函数 `set_tbr()` 将该段的起始地址设定到 `TBR`。
 各段能指定 `#pragma tbr` 的函数最多为 255 个。

(3) 其他扩展功能

全局变量的寄存器分配

#pragma global_register

格 式 `#pragma global_register [( < 变量名 > = < 寄存器名 > [, ... ] )]`

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。
能给变量名指定全局变量和静态数据成员。

例

```
#pragma global_register(a=R8,A::b=R9)
class A{
    public:
        static int b;
};
int a;
void g()
{
    a=A::b;
}
```

备 注 在全局变量中，能用于单纯型或者指针型的变量。不能指定 (unsigned)long long 型的变量。另外，只在指定 `double=float` 时，才能指定 `double` 型的变量（单片机不为 SH2A-FPU、SH-4、SH-4A 时）。能指定的寄存器为 R8-R14、FR12-FR15（单片机为 SH-2E、SH2A-FPU、SH-4、SH-4A 时）、DR12-DR14（单片机为 SH2A-FPU、SH-4、SH-4A 时）。

不能设定初始值，也不能参照地址。

不能保证从连接目标参照指定的变量。

能指定静态数据成员，但不能指定非静态数据成员。

能给 FR12-FR15 设定的变量型：

在 SH-2E 情况下

- float 型变量
- double 型变量（指定 `double=float`）

在 SH2A-FPU、SH-4 或者 SH-4A 的情况下

- float 型变量（没有指定 `fpu=double`）
- double 型变量（指定 `fpu=single`）

能给 DR12 ~ DR14 设定的变量型：

在 SH2A-FPU、SH-4、SH-4A 的情况下

- float 型变量（指定 `fpu=double`）
- double 型变量（没有指定 `fpu=single`）

GBR 基址的变量指定

#pragma gbr_base**#pragma gbr_base1**

格 式	<pre>#pragma gbr_base [() 变量名 [, ...] [)] #pragma gbr_base1 [() 变量名 [, ...] [)]</pre>
说 明	指定通过 GBR 寄存器和偏移量存取变量。 能给变量名指定变量和静态数据成员。 将 #pragma gbr_base 指定的变量分配到 \$G0 段。 将 #pragma gbr_base1 指定的变量分配到 \$G1 段。 #pragma gbr_base 指定变量在从 GBR 寄存器指向的地址偏移 0-127 个字节的位置。 对于不能用 #pragma gbr_base 存取的范围（GBR 寄存器指向地址的偏移量超过 128 个字节）的变量，可指定 #pragma gbr_base1。在 char 型、unsigned char 型的情况下，从 GBR 寄存器指向的地址最大偏移 255 个字节；在 short 型、unsigned short 型的情况下，最大偏移 510 个字节；在 int 型、unsigned int 型、long 型、unsigned long 型、float 型、double 型的情况下，最大偏移 1020 个字节。 编译程序以上述的指定为基础，对变量的参照和设定，通过 GBR 相对寻址生成最适合的目标程序。另外，对段 \$G0 内的 char 型、unsigned char 型数据，通过 GBR 间接寻址生成最适合的位指令。
备 注	如果连接后的段 \$G0 的总长度超过 128 个字节，就不保证运行。 另外，如果在段 \$G1 内有超过上述 #pragma gbr_base1 规约中各型名所示的偏移量的数据，就不保证运行。 在连接时，必须将段 \$G1 分配在段 \$G0 的 128 个字节后。 如果使用本功能，就必须在开始执行程序时将段 \$G0 的起始地址设定到 GBR 寄存器。 能指定静态数据成员，但不能指定非静态数据成员。 在指定 gbr=auto 时，#pragma gbr_base、#pragma gbr_base1 的指定无效。

位域的排列顺序的指定

#pragma bit_order

格 式 `#pragma bit_order [{left|right}]`

说 明 指定位域的排列顺序的转换。
 在指定 **left** 时，从高位开始分别分配成员；在指定 **right** 时，从低位开始分别分配成员。
 默认值设定是 **left**。
 如果省略 **left|right**，以后就以选项为基准。

例

```
#pragma bit_order left
typedef struct{
    unsigned char a:2;
    unsigned char b:3;
}x;
```

⇒

7	6	5	4	3	2	1	0
x.a		x.b					

□: 空区域

```
#pragma bit_order right
typedef struct{
    unsigned char a:2;
    unsigned char b:3;
}y;
```

⇒

7	6	5	4	3	2	1	0
			y.b		y.a		

```
// 不同长度的成员时
#pragma bit_order right
typedef struct{
    unsigned short a:2;
    unsigned char b:3;
}z;
```

⇒

						3	2	0
z.a								
7					4	3	0	
				z.b				

```
// 超过型的长度时
#pragma bit_order right
typedef struct{
    unsigned char a:5;
    unsigned char b:4;
}v;
```

⇒

7			5	4				0
		v.a						
7					4	3	0	
		v.b						

备 注 只要不转换排列顺序，指定的位域排列顺序就有效。
 位域的详细内容请参照“10.1.2(3) 位域”。

结构体、联合体、类的调整数的指定

#pragma pack

#pragma unpack

格 式 #pragma pack {1|4}
 #pragma unpack

说 明 指定源程序中指定位置后的结构体、联合体、类成员的调整数。
 在没有指定本扩展名时，按照 pack 选项的指定。另外，对于在 #pragma unpack 指定位置后声明的结构体、联合体、类成员的调整数，也按照 pack 选项的指定。#pragma pack 扩展名和调整数的关系如表 10.33 所示。

表 10.33 #pragma pack 扩展名和成员的调整数

扩展名 / 成员的型	#pragma pack 1	#pragma pack 4	#pragma unpack 或者没有指定
(unsigned)char	1	1	1
(unsigned)short、__fixed	1	2	按照 pack 选项
(unsigned)int、(unsigned)long、 (unsigned)long long、long __fixed、 __accum、long __accum、 浮点型、指针型	1	4	按照 pack 选项
调整数为 1 的结构体、联合体、类	1	1	1
调整数为 2 的结构体、联合体、类	1	2	按照 pack 选项
调整数为 4 的结构体、联合体、类	1	4	按照 pack 选项

```

例      #pragma pack 1
        struct S1 {
            char a;      /* offset:0 */
            int b;       /* offset:1 */
            char c;      /* offset:5 */
        } ST1;
        #pragma pack 4
        struct S2 {
            char a;      /* offset:0 */
                        /* gap:3 byte */
            int b;       /* offset:4 */
                        /* gap:0 byte */
            char c;      /* offset:8 */
                        /* gap:3 byte */
        } ST2;

```

备 注 不能用指针存取指定 pack=1 或者 #pragma pack 1 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），否则就输出警告。

例

```
#pragma pack 1
struct st {
    char x;
    int y;
} ST;
int *p=&ST.y; /* ST.y 的地址有可能为奇数 */

void func(void) {
    ST.y=1;      /* 能正确存取          */
    *p=1;        /* 可能不能正确存取          */
}
```

也能通过 `pack` 选项指定结构体、联合体、类成员的调整数。如果指定选项和 `#pragma`，就优先 `#pragma` 的指定。

绝对地址的指定

#pragma address

格 式 `#pragma address [( ) < 变量名 > = < 绝对地址 > [, ... [ ) ]]`

说 明 将指定的变量分配到指定地址。此时编译程序按各指定的变量设定段，在连接时分配到指定的绝对地址。在对连续地址指定变量时，就将这些变量分配到同一个段。

例

(1) 标量变量

```
#pragma address A=0x100
int A;
void func() {
    A=0;
}
```

< 代码展开例 >

```
_func:
    MOV     #1, R2
    SHLL8   R2
    MOV     #0, R4
    RTS
    MOV.L   R4, @R2

    .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_A:
    .RES.L  1
```

(2) 结构体

```
#pragma address ST=0x100
struct {
    int a;
    int b;
} ST;
void func() {
    ST.b=0;
}
```

< 代码展开例 >

```
_func:
    MOV     #1, R2
    SHLL8   R2
    MOV     #0, R4
    RTS
    MOV.L   R4, @(4, R2)
```

```

        .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
__ST:
        .RES.L    2

```

(3) 将多个变量分配到连续区域

```

#pragma address A=0x100,B=0x104
int A,B;
void func() {
    A=0;
    B=0;
}

```

< 代码展开例 >

```

__func:
    MOV     #1, R2
    SHLL8   R2
    MOV     #0, R4
    MOV.L   R4, @R2
    RTS
    MOV.L   R4, @(4, R2)

```

```

        .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
__A:
        .RES.L    1
__B:
        .RES.L    1

```

(4) 将多个变量分配到非连续区域

```

#pragma address A=0x100,B=0x108
int A,B;
void func() {
    A=0;
    B=0;
}

```

< 代码展开例 >

```

__func:
    MOV     #1, R2
    SHLL8   R2
    MOV     #0, R4
    MOV.L   R4, @R2
    RTS
    MOV.L   R4, @(8, R2)

```

```

        .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
__A:

```

```

        .RES.L 1

        .SECTION $ADDRESS$B108, DATA, LOCATE=H'108
_B:
        .RES.L 1

```

(5) 将不同属性的变量分配到连续区域

```

#pragma address A=0x100,B=0x104
int A;
const int B=0;
void func() {
    A=0;
}

```

< 代码展开例 >

```

_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    RTS
    MOV.L   R4,@R2

        .SECTION $ADDRESS$B100, DATA, LOCATE=H'100
_A:
        .RES.L 1

        .SECTION $ADDRESS$C104, DATA, LOCATE=H'104
_B:
        .DATA.L H'00000000

```

备 注

必须在变量声明前指定 `#pragma address`。

如果指定结构体 / 联合体成员或者变量以外的符号名，就出错。

如果给调整数为 2 的变量或结构体指定奇数地址，或者不给调整数为 4 的变量或结构体指定 4 的倍数地址，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`，就出错。

```

#pragma section
#pragma abs16/abs20/abs28/abs32
#pragma gbr_base/gbr_base1
#pragma global_register

```

10.3.2 段地址运算符

段地址运算符

__sectop

__second

__secsize

```
格 式      _ _sectop("<段名>")
           _ _secend("<段名>")
           _ _secksize("<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。
- 生成由 `__secsz` 指定的 <段名> 的长度。

```

例
<_ _sectop, _ _secend>
#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
#pragma entry INIT
void main(void);
void INIT(void)
{
    _INITSCT();
    main();
    sleep();
}

<_ _seclsize>
unsigned int size = _ _seclsize("NAME");
↓
_size:
        .DATA.L      SIZEOF NAME

```

10.3.3 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：

- 状态寄存器的设定和参照
- 向量基址寄存器的设定和参照
- 利用全局基址寄存器的 I/O 功能
- 不和 C/C++ 语言使用的寄存器资源竞争的系统指令
- 利用浮点单元的多媒体指令、控制寄存器的设定和参照

与一般函数一样，用函数调用格式记述内部函数。

内部函数一览如表 10.34 所示。

表 10.34 内部函数一览表

项目	规格	功能
1	void set_cr(int cr)	设定 SR
2	int get_cr(void)	参照 SR
3	void set_imask(int mask)	设定中断屏蔽
4	int get_imask(void)	参照中断屏蔽
5	void set_vbr(void *base)	设定 VBR
6	void *get_vbr(void)	参照 VBR
7	void set_gbr(void *base)	设定 GBR
8	void *get_gbr(void)	参照 GBR
9	unsigned char gbr_read_byte(int offset)	参照 GBR 基址的字节
10	unsigned short gbr_read_word(int offset)	参照 GBR 基址的字
11	unsigned long gbr_read_long(int offset)	参照 GBR 基址的长字
12	void gbr_write_byte(int offset, unsigned char data)	设定 GBR 基址的字节
13	void gbr_write_word(int offset, unsigned short data)	设定 GBR 基址的字
14	void gbr_write_long(int offset, unsigned long data)	设定 GBR 基址的长字
15	void gbr_and_byte(int offset, unsigned char mask)	GBR 基址的字节 AND
16	void gbr_or_byte(int offset, unsigned char mask)	GBR 基址的字节 OR
17	void gbr_xor_byte(int offset, unsigned char mask)	GBR 基址的字节 XOR
18	int gbr_tst_byte(int offset, unsigned char mask)	GBR 基址的字节 TEST
19	void sleep (void)	SLEEP 指令
20	int tas (char *addr)	TAS 指令
21	int trapa (int trap_no)	TRAPA 指令
22	int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)	OS 系统调用
23	void prefetch (void *p)	PREF 指令
24	void trace(long v)	TRACE 指令
25	void ldtlb(void)	LDTLB 指令
26	void nop(void)	NOP 指令
27	long dmuls_h(long data1, long data2)	带符号 64 位乘法的高 32 位
28	unsigned long dmuls_l(long data1, long data2)	带符号 64 位乘法的低 32 位
29	unsigned long dmulu_h(unsigned long data1, unsigned long data2)	无符号 64 位乘法的高 32 位

项目	规格	功能	
30	64 位的乘法	unsigned long dmulu_l(unsigned long data1, unsigned long data2)	无符号 64 位乘法的低 32b 位
31	数据的高位和 低位的交换	unsigned short swapb(unsigned short data)	SWAP.B 指令
32		unsigned long swapw(unsigned long data)	SWAP.W 指令
33		unsigned long end_cnvl(unsigned long data)	4 字节数据高位和低位的交换
34	乘加运算	int macw(short *ptr1, short *ptr2, unsigned int count)	MAC.W 指令
		int macwl(short *ptr1, short *ptr2, unsigned int count, unsigned int mask)	
35		int macl(int *ptr1, int *ptr2, unsigned int count)	MAC.L 指令
		int macll (int *ptr1, int *ptr2, unsigned int count, unsigned int mask)	
36	浮点单元	void set_fpscr(int cr)	设定 FPSCR
37		int get_fpscr(void)	参照 FPSCR
38	单精度浮点 向量运算	float fipr(float vect1[4], float vect2[4])	FIPR 指令
39		void ftrv(float vec1[4], float vec2[4])	FTRV 指令
40		void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	在对 4 维向量进行 4×4 行列转换后, 加上 4 维向量
41		void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	在对 4 维向量进行 4×4 行列转换后, 减去 4 维向量
42		void add4(float vec1[4], float vec2[4], float vec3[4])	4 维向量的和
43		void sub4(float vec1[4], float vec2[4], float vec3[4])	4 维向量的差
44		void mtrx4mul(float mat1[4][4], float mat2[4][4])	4×4 行列的乘法
45		void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	4×4 行列的乘加
46		void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	4×4 行列乘减
47	扩展寄存器的存取	void ld_ext(float mat[4][4])	装入到扩展寄存器
48		void st_ext(float mat[4][4])	保存扩展寄存器
49	DSP 指令	long __fixed pabs_lf(long __fixed data)	绝对值
		long __accum pabs_la(long __accum data)	
50		__fixed pdmsb_lf(long __fixed data)	MSB 检测
		__fixed pdmsb_la(long __accum data)	
51		long __fixed psha_lf(long __fixed data, int count)	算术移位
		long __accum psha_la(long __accum data, int count)	
52		__accum rndtoa(long __accum data)	舍入运算
		__fixed rndtof(long __fixed data)	
53		long __fixed long_as_lfixed(long data)	位模式的复制
		long lfixed_as_long(long __fixed data)	
54		void set_circ_x(__X __circ __fixed array[], size_t size)	设定模寻址
		void set_circ_y(__Y __circ __fixed array[], size_t size)	
55		void clr_circ(void)	解除模寻址
56		void set_cs(unsigned int mode)	设定 CS 位 (DSR 寄存器)
57	正弦和余弦	void fsca(long angle, float *sinv, float *cosv)	正弦和余弦的计算

项目	规格	功能
58 平方根的倒数	float fsrra(float data)	平方根的倒数
59 指令高速缓存的无效化	void icbi(void *p)	指令高速缓存块的无效化
60 高速缓存块的操作	void ocbi(void *p) void ocbp(void *p) void ocbwb(void *p)	高速缓存块的无效化 高速缓存块的清除 高速缓存块的回写
61 指令高速缓存的预取	void prefi(void *p)	指令高速缓存块的预取
62 系统同步	void synco(void)	数据操作的同步
63 T 位的参照和设定	int movt(void)	参照 T 位
64	void clrt(void)	清除 T 位
65	void sett(void)	T 位的置位
66 抽出连接寄存器的中间部分	unsigned long xtrct(unsigned long data1, unsigned long data2)	从连接的 64 位内容中抽出中间的 32 位
67	long addc(long data1, long data2)	2 个数据和 T 位相加, 进位反映到 T 位
68 带进位的加法	int ovf_addc(long data1, long data2)	2 个数据和 T 位相加, 参照进位
69	long addv(long data1, long data2)	2 个数据相加, 进位反映到 T 位
70	int ovf_addv(long data1, long data2)	2 个数据相加, 参照进位
71	long subc(long data1, long data2)	data1 减去 data2 和 T 位, 借位反映到 T 位
72 带借位的减法	int unf_subc(long data1, long data2)	data1 减去 data2 和 T 位, 参照借位
73	long subv(long data1, long data2)	data1 减去 data2, 借位反映到 T 位
74	int unf_subv(long data1, long data2)	data1 减去 data2, 参照借位
75 符号取反	long negc(long data)	0 减去 data 和 T 位, 借位反映到 T 位
76	unsigned long div1(unsigned long data1, unsigned long data2)	对 data1/data2 执行单步除法, 结果反映到 T 位
77 1 位除法	int div0s(long data1, long data2)	对 data1/data2 的带符号除法进行初始设定, 参照 T 位
78	void div0u(void)	无符号除法的初始设定
79	unsigned long rotl(unsigned long data)	将数据左循环 1 位, 从操作数移出的位反映到 T 位
80	unsigned long rotr(unsigned long data)	将数据右循环 1 位, 移出操作数的反映到 T 位
81 循环	unsigned long rotcl(unsigned long data)	将数据带 T 位左循环 1 位, 从操作数移出的位反映到 T 位
82	unsigned long rotrc(unsigned long data)	将数据带 T 位右循环 1 位, 从操作数移出的位反映到 T 位
83	unsigned long shll(unsigned long data)	将数据左移 1 位, 从操作数移出的位反映到 T 位
84 移位	unsigned long shlr(unsigned long data)	将数据逻辑右移 1 位, 从操作数移出的位反映到 T 位
85	long shar(long data)	将数据算术右移 1 位, 从操作数的移出位反映到 T 位

项目	规格	功能
86	long clipsb(long data)	1 个字节数据的带符号饱和运算
87	long clipsw(long data)	2 个字节数据的带符号饱和运算
88	unsigned long clipub(unsigned long data)	1 个字节数据的无符号饱和运算
89	unsigned long clipuw(unsigned long data)	2 个字节数据的无符号饱和运算
90	void set_tbr(void *data)	将 data 设定到 TBR
91	void *get_tbr(void)	参照 TBR 的值
92	void sr_jsr(void *func, int imask) ;	将 SR.RB 和 SR.BL 清 0，给 SR.10-13 设定 imask，调用 func 函数

在使用内部函数时，必须包含 <machine.h>、<umachine.h> 或者 <smachine.h>。

对应 SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP 的执行模式，将 <machine.h> 内容分为表 10.35 的内容。

表 10.35 内部函数的 include 文件

include 文件	内容
1 <machine.h>	全部内部函数
2 <smachine.h>	只能在特权模式下使用的内部函数
3 <umachine.h>	项 2 除外的内部函数

SR 的设定

void set_cr(int cr)

说 明 给状态寄存器设定 cr（32 位）。

头文件 <machine.h> 或者 <smachine.h>

参 数 cr 设定值

例

```
#include <machine.h>
void main(void)
{
    set_cr(0x60000000); /* Supervisor, RBank=1, BL=0, Imask=0 */
}
```

SR 的参照

int get_cr(void)

说 明 参照状态寄存器的值。

头文件 <machine.h> 或者 <smachine.h>

返回值 状态寄存器的值

例

```
#include <machine.h>
void main(void)
{
    set_cr(get_cr()|0x1000000); /* set BL bit */
}
```

中断屏蔽的设置

void set_imask(int mask)

说 明 给中断屏蔽（4 位）设定 mask（4 位）。

头文件 <machine.h> 或者 <smachine.h>

参 数 mask 设定值（4 位）

例

```
#include <machine.h>
void main(void)
{
    set_imask(15);
}
```

中断屏蔽的参照

int get_imask(void)

说 明 参照中断屏蔽（4 位）。

头文件 <machine.h> 或者 <smachine.h>

返回值 中断屏蔽的值

例

```
#include <machine.h>
void main(void)
{
    int mask;
    mask=get_imask();
}
```

VBR 的设置

void set_vbr(void *base)

说 明 给向量基址寄存器（VBR）设定 base（32 位）。

头文件 <machine.h> 或者 <smachine.h>

参 数 base 设定值

例

```
#include <machine.h>
#define VBR 0x0000FC00
void main(void)
{
    set_vbr((void *)VBR);
}
```

VBR 的参照

void *get_vbr(void)

说 明 参照向量基址寄存器（VBR）。

头文件 <machine.h> 或者 <smachine.h>

返回值 向量基址寄存器的值

例

```
#include <machine.h>
void main(void)
{
    void *vbr;
    vbr=get_vbr();
}
```

GBR 的设定

void set_gbr(void *base)

说 明	给全局基址寄存器（GBR）设定 base（32 位）。
头文件	<machine.h> 或者 <umachine.h>
参 数	base 设定值
例	<pre>#include <machine.h> #define IOBASE 0x05fffec0 void main(void) { set_gbr((void *)IOBASE); }</pre>
备 注	在改变 GBR 设定时必须注意：因为 GBR 是控制寄存器，所以本编译程序不能保证每个函数的内容。 如果指定 gbr=auto，就不能使用本函数。

GBR 的参照

void *get_gbr(void)

说 明	参照全局基址寄存器（GBR）的值。
头文件	<machine.h> 或者 <umachine.h>
返回值	全局基址寄存器的值
例	<pre>#include <machine.h> void main(void) { void *gbr; gbr=get_gbr(); }</pre>
备 注	如果指定 gbr=auto，就不能使用本函数。

GBR 基址的字节参照

unsigned char gbr_read_byte(int offset)

说 明 参照 GBR 相对 offset 的字节数据（8 位）。

头文件 <machine.h> 或者 <umachine.h>

返回值 字节数据（8 位）的参照值

参 数 offset 偏移地址

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    if(gbr_read_byte(BDATA)!=0)
        :
}
```

备 注 offset 必须是常数。
offset 能指定的范围为 +255 个字节。
如果指定 gbr=auto，就不能使用本函数。

GBR 基址的字参照

unsigned short gbr_read_word(int offset)

说 明 参照 GBR 相对 offset 的字数据（16 位）。

头文件 <machine.h> 或者 <umachine.h>

返回值 字数据（16 位）的参照值

参 数 offset 偏移地址

例

```
#include <machine.h>
#define WDATA 0
void main(void)
{
    if(gbr_read_word(WDATA)!=0)
        :
}
```

备 注 offset 必须是常数。
offset 能指定的范围为 +510 个字节。
如果指定 gbr=auto，就不能使用本函数。

GBR 基址长的字参照***unsigned long gbr_read_long(int offset)***

说 明 参照 GBR 相对 offset 的长字数据（32 位）。

头文件 <machine.h> 或者 <umachine.h>

返回值 长字数据（32 位）的参照值

参 数 offset 偏移地址

例

```
#include <machine.h>
#define LDATA 0
void main(void)
{
    if(gbr_read_long(LDATA)!=0)
        :
}
```

备 注 offset 必须是常数。
offset 能指定的范围为 +1020 个字节。
如果指定 gbr=auto，就不能使用本函数。

GBR 基址的字节设定***void gbr_write_byte(int offset, unsigned char data)***

说 明 给 GBR 相对 offset 设定 data（8 位）。

头文件 <machine.h> 或者 <umachine.h>

参 数 offset 偏移地址
data 设定值（8 位）

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_write_byte(BDATA,0);
}
```

备 注 offset 必须是常数。
offset 能指定的范围为 +255 个字节。
如果指定 gbr=auto，就不能使用本函数。

GBR 基址的字设定

`void gbr_write_word(int offset, unsigned short data)`

说 明 给 GBR 相对 offset 设定 data（16 位）。

头文件 <machine.h> 或者 <umachine.h>

参 数 offset 偏移地址
 data 设定值（16 位）

例 `#include <machine.h>`
 `#define WDATA 0`
 `void main(void)`
 `{`
 `gbr_write_word(WDATA, 0);`
 `}`

备 注 offset 必须是常数。
 offset 能指定的范围为 +510 个字节。
 如果指定 gbr=auto，就不能使用本函数。

GBR 基址长的字设定

`void gbr_write_long(int offset, unsigned long data)`

说 明 给 GBR 相对 offset 设定 data（32 位）。

头文件 <machine.h> 或者 <umachine.h>

参 数 offset 偏移地址
 data 设定值（32 位）

例 `#include <machine.h>`
 `#define LDATA 0`
 `void main(void)`
 `{`
 `gbr_write_long(LDATA, 0);`
 `}`

备 注 offset 必须是常数。
 offset 能指定的范围为 +1020 个字节。
 如果指定 gbr=auto，就不能使用本函数。

GBR 基址的字节 AND

`void gbr_and_byte(int offset, unsigned char mask)`

说 明 求 GBR 相对 `offset` 的字节数据和 `mask` 的 AND，结果设定到 `offset`。

头文件 `<machine.h>` 或者 `<umachine.h>`

参 数 `offset` 偏移地址
 `mask` 数据（8 位）

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_and_byte(BDATA, 0x01);
}
```

备 注 `mask` 必须是常数。
 `offset` 能指定的范围为 +255 个字节。
 `mask` 能指定的范围为 0 ~ +255。
 如果指定 `gbr=auto`，就不能使用本函数。

GBR 基址的字节 OR

`void gbr_or_byte(int offset, unsigned char mask)`

说 明 求 GBR 相对 `offset` 的字节数据和 `mask` 的 OR，结果设定到 `offset`。

头文件 `<machine.h>` 或者 `<umachine.h>`

参 数 `offset` 偏移地址
 `mask` 数据（8 位）

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_or_byte(BDATA, 0x01);
}
```

备 注 `mask` 必须是常数。
 `offset` 能指定的范围为 +255 个字节。
 `mask` 能指定的范围为 0 ~ +255。
 如果指定 `gbr=auto`，就不能使用本函数。

GBR 基址的字节 XOR***void gbr_xor_byte(int offset, unsigned char mask)***

说 明 求 GBR 相对 **offset** 的字节数据和 **mask** 的 XOR，结果设定到 **offset**。

头文件 <machine.h> 或者 <umachine.h>

参 数 **offset** 偏移地址
 mask 数据（8 位）

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_xor_byte(BDATA, 0x01);
}
```

备 注 **mask** 必须是常数。
offset 能指定的范围为 +255 个字节。
mask 能指定的范围为 0 ~ +255。
 如果指定 **gbr=auto**，就不能使用本函数。

GBR 基址的字节 TEST***int gbr_tst_byte(int offset, unsigned char mask)***

说 明 求 GBR 相对 **offset** 的字节数据和 **mask** 的 AND，判断结果是否为 0，结果设定到 **T** 位。

头文件 <machine.h> 或者 <umachine.h>

返回值 **T** 位的反向值

参 数 **offset** 偏移地址
 mask 数据（8 位）

例

```
#include <machine.h>
#define BDATA 0
int a;
void main(void)
{
    if(gbr_tst_byte(BDATA, 0))
        a = 0;
}
```

备 注 **mask** 必须是常数。
offset 能指定的范围为 +255 个字节。
mask 能指定的范围为 0 ~ +255。
 如果指定 **gbr=auto**，就不能使用本函数。

GBR 内部函数的使用例

```

#include <machine.h>
#define CDATE1 0
#define CDATE2 1
#define CDATE3 2
#define SDATE1 4
#define IDATA1 8
#define IDATA2 12

struct {
    char    cdata1;          /* offset 0          */
    char    cdata2;          /* offset 1          */
    char    cdata3;          /* offset 2          */
    short   sdata1;          /* offset 4          */
    int     idata1;          /* offset 8          */
    int     idata2;          /* offset 12         */
} table;
void f();

void f()
{
    set_gbr(&table);          /* 将 table 的起始地址设定到 GBR          */
    :                          /*                                          */
    gbr_write_byte(CDATE2,10); /* 将 10 设定到 table.cdata2              */
    gbr_write_long(IDATA2,100); /* 将 100 设定到 table.idata2             */
    :
    if(gbr_read_byte(CDATE2)!=10) /* 参照 table.cdata2 的值                */
        gbr_and_byte(CDATE2,10); /* 求 table.cdata2 的值和 10 的 AND,      */
                                /* 结果设定到 table.cdata2                */
    gbr_or_byte(CDATE2,0x0F); /* 求 table.cdata2 的值和 0x0f 的 OR      */
    :                          /* 结果设定到 table.cdata2,              */
    sleep();                  /* 展开为 SLEEP 指令                      */
}

```

GBR 内部函数有效的使用方法：

- 如果将频繁存取的目标分配到存储器，就将该目标的起始地址设定到 GBR。
 - 尽量将频繁使用逻辑运算的字节数据声明到结构体的前 128 个字节。
- 从而能减少结构体存取所需的起始地址装入指令、逻辑运算所需的存储器装入 / 存储指令。

SLEEP 指令

void sleep(void)

说 明 展开为低功耗状态指令 SLEEP。

头文件 <machine.h> 或者 <smachine.h>

例

```
#include <machine.h>
void main(void)
{
    sleep();
}
```

TAS 指令

int tas (char *addr)

说 明 展开为 TAS.B @Rn。

头文件 <machine.h> 或者 <umachine.h>

返回值 TAS 指令执行结果的 T 位值

参 数 addr 用 TAS 指令指定的地址

例

```
#include <machine.h>
char a;
void main(void)
{
    tas(&a);
}
```

TRAPA 指令***int trapa (int trap_no)***

说 明	展开为 TRAPA #trap_no。
头文件	<machine.h> 或者 <umachine.h>
参 数	trap_no 陷阱号
例	<pre>#include <machine.h> void main(void) { trapa(0); }</pre>
备 注	trap_no 必须是 0 ~ 255 的常数。

OS 系统调用***int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)***

说 明	发行 HI7000 和各种 OS 的系统调用。当执行 trapa_svc 时，将 code 设定到 R0，para1 ~ para4 设定到 R4 ~ R7 后，执行 TRAPA #trap_no 指令。
头文件	<machine.h> 或者 <umachine.h>
参 数	trap_no 陷阱号 code 功能码 para1 ~ para4 参数（0 ~ 4 个，可变） 型 type1 ~ type4 为标量型或者指针型。
例	<pre>#include <machine.h> #define SIG_SEM 0xffc8 void main(void) { trapa_svc(63, SIG_SEM, 0x05); }</pre>
备 注	trap_no 必须是 0 ~ 255 的常数。

PREF 指令***void prefetch (void *p)***

- 说 明 将 p 指向的区域（16 个字节，但是区域为 (int)p&0xffffffff0）的 16 个字节（在 cpu=sh4|sh4a|sh4aldsp 时为从 (int)p&0xffffffe0 开始的 32 字节）的数据读到高速缓存。
- 头文件 <machine.h> 或者 <umachine.h>
- 参 数 p 要预取的地址
- 例
- ```
#include <machine.h>
char a[1200];
void main(void)
{
 char *pa = a;
 prefetch(pa);
}
```
- 备 注 只在指定 cpu=sh2a|sh2afpu|sh3|sh3dsp|sh4|sh4a|sh4aldsp 时才能使用。  
不影响程序运行。

**TRACE 指令*****void trace (long v)***

- 说 明 支持部分仿真器所持有的软件地址功能。
- 头文件 <machine.h> 或者 <umachine.h>
- 参 数 v 要指定的变量
- 例
- ```
#include <machine.h>
void main(void)
{
    long v;
    trace(v);
}
```
- 备 注 如果指定 cpu=sh1，就不能使用。
有关软件地址功能的详细内容，请参照使用的仿真器的用户手册。
只能在连接仿真器进行调试的阶段时，使用本函数。在未连接仿真器的状态下，不能使用本函数。

LDTLB 指令***void ldtlb (void)***

说 明 展开为 LDTLB 指令。

头文件 <machine.h> 或者 <smachine.h>

例

```
#include <machine.h>
void main(void)
{
    ldtlb();
}
```

备 注 只在指定 cpu=sh3 | sh3dsp | sh4 | sh4a | sh4aldsp 时才能使用。

NOP 指令***void nop (void)***

说 明 展开为 NOP 指令。

头文件 <machine.h> 或者 <umachine.h>

例

```
#include <machine.h>
void main(void)
{
    int a;
    if (a) {
        nop();
    }
}
```

64 位乘法

long dmuls_h (long data1, long data2)

说 明	进行带符号 32 位 × 带符号 32 位 → 带符号 64 位的乘法运算，参照结果的高 32 位。
头文件	<machine.h> 或者 <umachine.h>
返回值	乘法运算结果的高 32 位
例	<pre>#include <machine.h> extern long data1, data2; extern long result; void main(void) { result = dmuls_h(data1, data2); }</pre>
备 注	如果指定 cpu=sh1，就不能使用本函数。

64 位乘法

unsigned long dmuls_l (long data1, long data2)

说 明	进行带符号 32 位 × 带符号 32 位 → 带符号 64 位的乘法运算，参照结果的低 32 位。
头文件	<machine.h> 或者 <umachine.h>
返回值	乘法运算结果的低 32 位
例	<pre>#include <machine.h> extern long data1, data2; extern unsigned long result; void main(void) { result = dmuls_l(data1, data2); }</pre>
备 注	如果指定 cpu=sh1，就不能使用本函数。

64 位乘法

unsigned long dmulu_h (unsigned long data1, unsigned long data2)

说 明 进行无符号 32 位 × 无符号 32 位 → 无符号 64 位的乘法运算，参照结果的高 32 位。

头文件 <machine.h> 或者 <umachine.h>

返回值 乘法运算结果的高 32 位

例

```
#include <machine.h>
extern unsigned long data1, data2;
extern unsigned long result;
void main(void)
{
    result = dmulu_h(data1, data2);
}
```

备 注 如果指定 cpu=sh1，就不能使用本函数。

64 位乘法

unsigned long dmulu_l (unsigned long data1, unsigned long data2)

说 明 进行无符号 32 位 × 无符号 32 位 → 无符号 64 位的乘法运算，参照结果的低 32 位。

头文件 <machine.h> 或者 <umachine.h>

返回值 乘法运算结果的低 32 位

例

```
#include <machine.h>
extern unsigned long data1, data2;
extern unsigned long result;
void main(void)
{
    result = dmulu_l(data1, data2);
}
```

备 注 如果指定 cpu=sh1，就不能使用本函数。

SWAP.B 指令***unsigned short swapb(unsigned short data)***

说 明 将 2 个字节数据的高位和低位的 1 个字节进行交换。

头文件 <machine.h> 或者 <umachine.h>

返回值 2 个字节数据的高位和低位的 1 个字节的交换结果

```
例
#include <machine.h>
extern unsigned short data;
extern unsigned short result;
void main(void)
{
    result = swapb(data);
    /* 例如, 当 data=0x1234 时, result=0x3412。 */
}
```

SWAP.W 指令***unsigned long swapw(unsigned long data)***

说 明 将 4 个字节数据的高位和低位 2 个字节进行交换。

头文件 <machine.h> 或者 <umachine.h>

返回值 4 个字节数据的高位和低位的 2 个字节的交换结果

```
例
#include <machine.h>
extern unsigned long data;
extern unsigned long result;
void main(void)
{
    result = swapw(data);
    /* 例如, 当 data=0x12345678 时, result=0x56781234。 */
}
```

4 个字节数据的排列更改

unsigned long end_cnv(unsigned long data)

说 明 将 4 个字节数据按字节进行高低位倒序排列。

头文件 <machine.h> 或者 <umachine.h>

返回值 4 个字节数据的排列结果

```
例
#include <machine.h>
extern unsigned long data;
extern unsigned long result;
void main(void)
{
    result = end_cnv(data);
    /* 例如, 当 data=0x12345678 时, result=0x78563412。 */
}
```

MAC.W 指令

int macw(short *ptr1, short *ptr2, unsigned int count)***int macwl(short *ptr1, short *ptr2, unsigned int count, unsigned int mask)***

说 明 对 2 个数据表的内容进行乘加运算。

头文件 <machine.h> 或者 <umachine.h>

返回值 运算结果

参 数	ptr1	要进行乘加运算的数据的起始地址
	ptr2	要进行乘加运算的数据的起始地址
	count	执行乘加运算的次数
	mask	对应环形缓冲器的地址屏蔽

```
例
#include <machine.h>
short tbl1[]={a1,a2,a3,a4};
short tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
    result1=macw(tbl1,tbl2,3);    /* 求 a1*b1+a2*b2+a3*b3 */
    result2=macwl(tbl1,tbl2,4,0xffffffffb);
                                /* 求 a1*b1+a2*b2+a3*b1+a4*b2 */
}
```

备 注 因为不检查参数, 所以必须注意以下几点:

- 必须对 ptr1 和 ptr2 指向的表进行 2 个字节的调整。
- 必须对 macwl 的 ptr2 指向的表进行环形缓冲器屏蔽 ×2 长度的调整。

MAC.L 指令

int macl(int *ptr1, int *ptr2, unsigned int count)
int macll(int *ptr1, int *ptr2, unsigned int count, unsigned int mask)

说 明 对 2 个数据表的内容进行乘加运算。

头文件 <machine.h> 或者 <umachine.h>

返回值 运算结果

参 数 ptr1 要进行乘加运算的数据的起始地址
 ptr2 要进行乘加运算的数据的起始地址
 count 执行乘加运算的次数
 mask 对应环形缓冲器的地址屏蔽

例

```
#include <machine.h>
int tbl1[]={a1,a2,a3,a4};
int tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
    result1=macl(tbl1,tbl2,3); /* 求 a1*b1+a2*b2+a3*b3    */
    result2=macll(tbl1,tbl2,4,0xffffffff7);
                                /* 求 a1*b1+a2*b2+a3*b1+a4*b2    */
}
```

备 注 如果指定 cpu=sh1, 就不能使用本函数。
 因为不检查参数, 所以必须注意以下几点:
 • 必须对 ptr1 和 ptr2 指向的表进行 4 个字节的调整。
 • 必须对 macll 的 ptr2 指向的表进行环形缓冲器屏蔽 ×2 长度的调整。

FPSCR 的设置

void set_fpscr(int cr)

说 明 给浮点状态控制寄存器 (FPSCR) 设定 cr (32 位)。

头文件 <machine.h> 或者 <umachine.h>

参 数 cr 设定值 (32 位)

例

```
#include <machine.h>
void main(void)
{
    set_fpscr(0);
}
```

备 注 只在指定 cpu=sh2e|sh2afpu|sh4|sh4a 时才能使用。

FPSCR 的参照***int get_fpscr(void)***

说 明	参照浮点状态控制寄存器（FPSCR）。
头文件	<machine.h> 或者 <umachine.h>
返回值	FPSCR 的值
例	<pre>#include <machine.h> int cr; void main(void) { cr=get_fpscr(); }</pre>
备 注	只在指定 cpu=sh2e sh2afpu sh4 sh4a 时才能使用。

FIPR 指令***float fipr(float vect1[4], float vect2[4])***

说 明	求 2 个向量的内积。
头文件	<machine.h> 或者 <umachine.h>
返回值	运算结果
参 数	vect1 vect2 向量 向量
例	<pre>#include <machine.h> extern float data1[4],data2[4]; float result; void main(void) { result=fipr(data1,data2); }</pre>
备 注	只在指定 cpu=sh4 sh4a 时才能使用。

FTRV 指令**void ftrv(float vec1[4], float vec2[4])**

说 明 通过 tbl (4×4 行列) 转换 vec1 (向量), 结果保存到 vec2 (向量)。
tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 <umachine.h>

参 数 vec1 向量
 vec2 向量

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4], data2[4];
void main(void)
{
    ld_ext(tbl);
    ftrv(data1, data2);
    /* i=0, 1, 2, 3
       data2[i]=data1[0]*tbl[0][i]+data1[1]*tbl[1][i]
               +data1[2]*tbl[2][i]+data1[3]*tbl[3][i]
       为 data2 的结果。 */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。
在更改浮点状态控制寄存器 (FPSCR) 的浮点寄存器组位 (FR) 后, ld_ext() 函数和 st_ext() 函数存取扩展寄存器。在中断函数内使用 ld_ext() 函数或者 st_ext() 函数时, 必须在向量运算内部函数的前后更改中断屏蔽。举例如下:

例:

```
extern float mat1[4][4];
extern float vec1[4], vec2[4];
#pragma interrupt (intfunc)
void intfunc(){
    :
    ld_ext();
    :
}
void normfunc(){
    :
    int maskdata=get_imask();
    set_imask(15);
    ld_ext(mat1);
    ftrv(vec1, vec2);
    set_imask(maskdata);
    :
}
```

在 4 维向量的 4×4 行列转换后，求与 4 维向量的和

void ftrvadd(float vec1[4], float vec2[4], float vec3[4])

说 明 通过 tbl（4×4 行列）转换 vec1（向量），转换结果加上 vec2（向量），结果保存到 vec3（向量）。
tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 machine.h>

参 数

vec1	向量
vec2	向量
vec3	向量

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    ld_ext(tbl);
    ftrvadd(data1,data2,data3); /* data3=data1×tbl+data2 */
    /* i=0,1,2,3
        data3[i]=data1[0]*tbl[0][i]
                +data1[1]*tbl[1][i]
                +data1[2]*tbl[2][i]
                +data1[3]*tbl[3][i]
                +data2[i]
        为 data3 的结果。 */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。

在 4 维向量的 4×4 行列转换后，求与 4 维向量的差

void ftrvsub(float vec1[4], float vec2[4], float vec3[4])

说 明 通过 tbl (4×4 行列) 转换 vec1 (向量)，转换结果减去 vec2 (向量)，结果保存到 vec3 (向量)。

tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 machine.h>

参 数 vec1 向量
 vec2 向量
 vec3 向量

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    ld_ext(tbl);
    ftrvsub(data1,data2,data3); /* data3=data1×tbl-data2 */
    /* i=0,1,2,3
        data3[i]=data1[0]*tbl[0][i]
                +data1[1]*tbl[1][i]
                +data1[2]*tbl[2][i]
                +data1[3]*tbl[3][i]
                -data2[i]
        为 data3 的结果。 */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。

4 维向量的和

void add4(float vec1[4], float vec2[4], float vec3[4])

说 明 **vec1**（向量）加上 **vec2**（向量），结果保存到 **vec3**（向量）。

头文件 <machine.h> 或者 <umachine.h>

参 数 **vec1** 向量
 vec2 向量
 vec3 向量

例 `#include <machine.h>`
 `extern float data1[4];`
 `extern float data2[4];`
 `extern float data3[4];`
 `void main(void)`
 `{`
 `add4(data1, data2, data3);      /* data3=data1+data2 */`
 `}`

备 注 只在指定 `cpu=sh2afpu|sh4|sh4a` 时才能使用。

4 维向量的差

void sub4(float vec1[4], float vec2[4], float vec3[4])

说 明 **vec1**（向量）减去 **vec2**（向量），结果保存到 **vec3**（向量）。

头文件 <machine.h> 或者 <umachine.h>

参 数 **vec1** 向量
 vec2 向量
 vec3 向量

例 `#include <machine.h>`
 `extern float data1[4];`
 `extern float data2[4];`
 `extern float data3[4];`
 `void main(void)`
 `{`
 `sub4(data1, data2, data3);      /* data3=data1-data2 */`
 `}`

备 注 只在指定 `cpu=sh2afpu|sh4|sh4a` 时才能使用。

4×4 行列的乘法

void mtrx4mul(float mat1[4], float mat2[4])

说 明 通过 tbl (4×4 行列) 转换 mat1 (4×4 行列), 结果保存到 mat2。
tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 <umachine.h>

参 数 mat1 4×4 行列
 mat2 4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4mul(tbl1,tbl2);    /* tbl2=tbl1×tbl */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。
本函数用于行列运算, 不可交换。
例:

```
extern float matA[][];
extern float matB[][];
int judge(){
    float data1[4][4], data2[4][4];
    set_imask(15);
    ld_ext(matA);
    mtrx4mul(matB,data1);    /* data1=matB×matA */
    ld_ext(matB);
    mtrx4mul(matA,data2);    /* data2=matA×matB */
    /* 此时, data1[][] 和 data2[][] 的各元素必须一致。 */
}
```

4×4 行列的乘加

void mtrx4muladd(float mat1[4], float mat2[4], float mat3[4])

说 明 通过 tbl (4×4 行列) 转换 mat1 (4×4 行列), 转换结果加上 mat2 (4×4 行列), 结果保存到 mat3。
tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 <umachine.h>

参 数

mat1	4×4 行列
mat2	4×4 行列
mat3	4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4muladd(tbl1, tbl2, tbl3); /* tbl3=tbl1×tbl+tbl2 */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。
本函数用于行列运算, 不可交换。

4×4 行列的乘减

void mtrx4mulsub(float mat1[4], float mat2[4], float mat3[4])

说 明 通过 tbl (4×4 行列) 转换 mat1 (4×4 行列), 转换结果减去 mat2 (4×4 行列), 结果保存到 mat3
tbl 是通过内部函数 ld_ext() 装入的行列。

头文件 <machine.h> 或者 <umachine.h>

参 数

mat1	4×4 行列
mat2	4×4 行列
mat3	4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4mulsub(tbl1, tbl2, tbl3); /* tbl3=tbl1×tbl-tbl2 */
}
```

备 注 只在指定 cpu=sh4|sh4a 时才能使用。
本函数用于行列运算, 不可交换。

装入到扩展寄存器

void ld_ext(float mat[4][4])

说 明 将 **mat**（4×4 行列）装入到扩展寄存器。

头文件 <machine.h> 或者 <umachine.h>

参 数 **mat** 4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
void main(void)
{
    ld_ext(tbl);
}
```

备 注 只在指定 **cpu=sh4|sh4a** 时才能使用。
在更改浮点状态控制寄存器（FPSCR）的浮点寄存器组位（FR）后，本函数存取扩展寄存器。在中断函数内使用本函数时，必须在向量运算内部函数的前后更改中断屏蔽。

保存扩展寄存器

void st_ext(float mat[4][4])

说 明 将扩展寄存器的内容保存到 **mat**（4×4 行列）。

头文件 <machine.h> 或者 <umachine.h>

参 数 **mat** 4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
void main(void)
{
    st_ext(tbl);
}
```

备 注 只在指定 **cpu=sh4|sh4a** 时才能使用。
在更改浮点状态控制寄存器（FPSCR）的浮点寄存器组位（FR），本函数存取扩展寄存器。在中断函数内使用本函数时，必须在向量运算内部函数的前后更改中断屏蔽。

绝对值

long __fixed pabs_lf(long __fixed data)***long __accum pabs_la(long __accum data)***

说 明 取绝对值。

头文件 <machine.h> 或者 <umachine.h>

返回值 运算结果

参 数 data 要取绝对值的数据

例

```
#include <machine.h>
long __fixed result;
long __fixed ptr;
void main(void)
{
    result=pabs_lf(ptr);
}
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用。
在返回值的型 (pabs_lf() 为 long __fixed 型, pabs_la() 为 long __accum 型) 的值不能表示 data 绝对值的结果时, 不能保证运行。

MSB 的检测

__fixed pdmsb_lf(long __fixed data)***__fixed pdmsb_la(long __accum data)***

说 明 检测 MSB (求使数据正规化的位移量)。

头文件 <machine.h> 或者 <umachine.h>

返回值 运算结果

参 数 data 要检测 MSB 的数据

例

```
#include <machine.h>
__fixed result;
long __fixed ptr;
void main(void)
{
    result=pdmsb_lf(ptr);
}
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用。

算术移位

`long __fixed psha_lf(long __fixed data, int count)`
`long __accum psha_la(long __accum data, int count)`

说 明 进行算术移位。

头文件 <machine.h> 或者 <umachine.h>

返回值 运算结果

参 数 **data** 要进行算术移位的数据
 count 算术移位的位数

例

```
#include <machine.h>
long __fixed result;
long __fixed ptr;
int count;
void main(void)
{
    result=psha_lf(ptr,count);
}
```

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 时才能使用。
 `count` 能指定的范围为 `-32 ~ +32`。如果指定正值，就左移 `count` 位；如果指定负值，就右移 `count` 绝对值的位；如果指定值超出范围，就不保证运行。

位模式的复制

`long __fixed long_as_lfixed(long data)`
`long lfixed_as_long(long __fixed data)`

说 明 复制位模式（通用寄存器 DSP 寄存器之间复制）。

头文件 <machine.h> 或者 <umachine.h>

返回值 复制结果

参 数 **data** 要复制的数据

例

```
#include <machine.h>
long __fixed result;
long ptr;
void main(void)
{
    result=long_as_lfixed(ptr);
}
```

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 时才能使用。

舍入运算

`__accum rndtoa(long __accum data)`
`__fixed rndtof(long __fixed data)`

说 明 进行舍入运算。

头文件 `<machine.h>` 或者 `<umachine.h>`

返回值 运算结果

参 数 `data` 要进行舍入运算的数据

例

```
#include <machine.h>
__accum result;
long __accum ptr;
void main(void)
{
    result=rndtoa(ptr);
}
```

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 时才能使用。

模寻址的设定

`void set_circ_x(__X__circ __fixed array[ ], size_t size)`
`void set_circ_y(__Y__circ __fixed array[ ], size_t size)`

说 明 设定模寻址。

头文件 `<machine.h>` 或者 `<smachine.h>`

参 数 `array[ ]` 要进行模寻址的数据
 `size` 数据长度

例

```
#include <machine.h>
__circ __X __fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
__Y __fixed output[8];

void main(void)
{
    int i;
    set_circ_x(input, sizeof(input)); /* 设定模寻址 */
    for (i = 0; i < 8; i++) {
        output[i] = input[i];
    }
    clr_circ(); /* 解除模寻址 */
}
```

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 时才能使用。

模寻址的解除

void clr_circ()

说 明 解除模寻址。
将从 SR 右侧开始的第 10、11 位初始化为零。

头文件 <machine.h> 或者 <smachine.h>

```
例
#include <machine.h>
_ _circ _ _X _ _fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
_ _Y _ _fixed output[8];
void main(void)
{
    int i;
    set_circ_x(input, sizeof(input));    /* 设定模寻址 */
    for (i = 0; i < 8; i++) {
        output[i] = input[i];
    }
    clr_circ();                          /* 解除模寻址 */
}
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用。

CS 位的设定

void set_cs(unsigned int mode)

说 明 设定 CS 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 mode 要设定的模式 (0 ~ 5)

要设定的值	模式
0	进位 / 借位模式
1	负值模式
2	零值模式
3	溢出模式
4	带符号的大于模式
5	带符号的大于等于模式

```
例
#include <machine.h>
#define MODE 1
void main(void)
{
    set_cs(MODE);
}
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用。

正弦和余弦的计算

void fsca(long angle, float *sinv, float *cosv)

说 明	从 <code>angle</code> 指定的角度开始计算正弦和余弦的近似值，结果设定到 <code>sinv</code> 和 <code>cosv</code> 指向的区域	
头文件	<code><machine.h></code> 或者 <code><umachine.h></code>	
参 数	<code>angle</code>	要计算正弦和余弦的角度 (在 32 位区域中以小数点在 2^{16} 位的右侧的定点数据表示时，必须将位信息指定为 <code>angle</code>)
	<code>sinv</code>	保存运算结果的地址 (正弦)
	<code>cosv</code>	保存运算结果的地址 (余弦)
例	<pre>#include <machine.h> long angle = (45<<16)/360; /* 45 度 */ float *sinv; float *cosv; void main(void) { fsca(angle, sinv, cosv); }</pre>	
备 注	只在指定 <code>cpu=sh4a</code> 时才能使用。	

平方根的倒数

float fsrra(float data)

说 明	计算平方根倒数的近似值。	
头文件	<code><machine.h></code> 或者 <code><umachine.h></code>	
参 数	<code>data</code>	要计算平方根倒数的值
返回值	运算结果	
例	<pre>#include <machine.h> float data; float result; void main(void) { result=fsrra(data); }</pre>	
备 注	只在指定 <code>cpu=sh4a</code> 时才能使用。	

指令高速缓存的无效化

void icbi(void *p)

说 明	使指令高速缓存无效。
头文件	<machine.h> 或者 <umachine.h>
参 数	p 变量或者函数的地址
例	<pre>#include <machine.h> extern int *p; void main(void) { icbi(p); }</pre>
备 注	只在指定 cpu=sh4a sh4aldsp 时才能使用。

高速缓存块的操作

void ocbi(void *p)**void ocbp(void *p)****void ocbwb(void *p)**

说 明	进行高速缓存块的操作。 ocbi: 高速缓存块的无效化 ocbp: 高速缓存块的清除 ocbwb: 高速缓存块的回写
头文件	<machine.h> 或者 <umachine.h>
参 数	p 变量或者函数的地址
例	<pre>#include <machine.h> extern int *p; void main(void) { ocbi(p); }</pre>
备 注	只在指定 cpu=sh4 sh4a sh4aldsp 时才能使用。

指令高速缓存的预取

void prefetch(void *p)

说 明 将以 32 个字节边界开始的 32 个字节的指令块读到指令高速缓存。

头文件 <machine.h> 或者 <umachine.h>

参 数 p 要预取的地址

例

```
#include <machine.h>
void *pa;
void main(void)
{
    prefetch(pa);
}
```

备 注 只在指定 cpu=sh4a|sh4aldsp 时才能使用。

系统同步

void synco(void)

说 明 展开为 SYNC0 指令。
SYNC0 指令用于数据操作的同步。当执行 SYNC0 指令时，在指令 SYNC0 前的数据操作结束后，开始 SYNC0 指令后的指令。

头文件 <machine.h> 或者 <umachine.h>

例

```
#include <machine.h>
void main(void)
{
    synco();
}
```

备 注 只在指定 cpu=sh4a|sh4aldsp 时才能使用。

T 位的参照

int movt (void)

说 明	参照 SR 寄存器 T 位的值。
头文件	<machine.h> 或者 <umachine.h>
返回值	T 位的值
例	<pre>#include <machine.h> extern int sr_t; void main(void) { sr_t = movt(); }</pre>

T 位的清除

void clrt (void)

说 明	清除 SR 寄存器的 T 位。
头文件	<machine.h> 或者 <umachine.h>
例	<pre>#include <machine.h> void main(void) { clrt(); }</pre>

*T 位的置位****void sett (void)***

说 明 将 SR 寄存器的 T 位置位。

头文件 <machine.h> 或者 <umachine.h>

例

```
#include <machine.h>
void main(void)
{
    sett();
}
```

*抽出连接寄存器的中间部分****unsigned long xtrct (unsigned long data1, unsigned long data2)***

说 明 从连接 data1 和 data2 的 64 位内容中抽出中间 32 位。

头文件 <machine.h> 或者 <umachine.h>

参数

data1	数据的高 32 位
data2	数据的低 32 位

返回值 (data1 的低 16 位):(data2 的高 16 位)

例

```
#include <machine.h>
extern unsigned long result,data1,data2;
void main(void)
{
    result = xtrct(data1,data2);
}
```

带进位的加法

long addc (long data1, long data2)

说 明 2 个数据和 T 位相加，进位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 加法数据 1
 data2 加法数据 2

返回值 加法结果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    result = addc(data1,data2);
}
```

备 注 因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。

```
if (a) {...}                              /* 将比较结果设定到 T 位    */
result[1] = addc(data1[1], data2[1]);   /* 加上比较结果               */
result[0] = addc(data1[0], data2[0]);   /* 反映上式的运算结果        */
```

带进位的加法

int ovf_addc (long data1, long data2)

说 明 2 个数据和 T 位相加，参照进位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 加法数据 1
 data2 加法数据 2

返回值 进位

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    if (ovf_addc(data1,data2)) {
        result = 0;
    }
}
```

备 注 因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为要这些运算结果反映到 T 位，所以本函数不能正常运行。

```
if (a) {...}                      /* 将比较结果设定到 T 位      */
if (ovf_addc(data1,data2)) {      /* 加上比较结果              */
```

带进位的加法

long addv (long data1, long data2)

说 明 2 个数据相加，进位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 加法数据 1
 data2 加法数据 2

返回值 加法结果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    result = addv(data1,data2);
}
```

带进位的加法

int ovf_addv (long data1, long data2)

说 明 2 个数据相加，参照进位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 加法数据 1
 data2 加法数据 2

返回值 进位

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    if (ovf_addv(data1,data2)) {
        result = 0;
    }
}
```

带借位的减法

long subc (long data1, long data2)

说 明 data1 减去 data2 和 T 位，借位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 减法数据 1
 data2 减法数据 2

返回值 减法结果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    result = subc(data1,data2);
}
```

备 注 因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。

```
if (a) {...}                              /* 将比较结果设定到 T 位      */
result[0] = subc(data1[0], data2[0]);   /* 减去比较结果              */
result[1] = subc(data1[1], data2[1]);   /* 反映上式的运算结果        */
```

带借位的减法

int unf_subc (long data1, long data2)

说 明 data1 减去 data2 和 T 位，参照借位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 减法数据 1
 data2 减法数据 2

返回值 借位

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    if (unf_subc(data1,data2)) {
        result = 0;
    }
}
```

备 注 因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。

```
if (a) {...}                              /* 将比较结果设定到 T 位      */
if (unf_subc(data1,data2)) {            /* 减去比较结果              */
```

带借位的减法

long subv (long data1, long data2)

说 明 data1 减去 data2，借位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 减法数据 1
 data2 减法数据 2

返回值 减法结果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
    result = subv(data1,data2);
}
```


带借位的减法

int unf_subv (long data1, long data2)

说 明	data1 减去 data2，参照借位。		
头文件	<machine.h> 或者 <umachine.h>		
参 数	data1	减法数据 1	
	data2	减法数据 2	
返回值	借位		
例	<pre>#include <machine.h> extern long result,data1,data2; void main() { if (unf_subv(data1,data2)) { result = 0; } }</pre>		

符号取反

long negc (long data)

说 明	0 减去 data 和 T 位，借位反映到 T 位。		
头文件	<machine.h> 或者 <umachine.h>		
参 数	data	数据	
返回值	符号取反的结果		
例	<pre>#include <machine.h> extern long result,data; void main() { result = negc(data); }</pre>		
备 注	因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。 <pre>if (a) {...} result[0] = negc(data[0]); /* 减去比较结果 */ result[1] = negc(data[1]); /* 反映上式的运算结果 */</pre>		

1 位除法

unsigned long div1 (unsigned long data1, unsigned long data2)

说 明 执行 data1/data2 的单步除法，结果反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 被除数
 data2 除数

返回值 被除数的更新值

例

```
#include <machine.h>
extern unsigned long data1,data2;
void main(void)
{
    div0u();
    data1 = div1(data1,data2);
}
```

备 注 虽然能重复使用本函数进行除法运算，但是在重复中不能改写 M、Q、T 位（即使记述比较或者移位等指令，也不改写 T 位）。必须在本函数之前使用 div0s() 或者 div0u() 进行 M、Q、T 位的初始化。

1 位除法

int div0s (long data1, long data2)

说 明 进行 data1/data2 的带符号除法的初始设定，参照 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data1 被除数
 data2 除数

返回值 T 位的值

例

```
#include <machine.h>
extern long data1,data2;
void main(void)
{
    (void)div0s(data1,data2);
    data1 = div1(data1,data2);
}
```

1 位除法

void div0u (void)

说 明 进行无符号除法的初始设定。

头文件 <machine.h> 或者 <umachine.h>

例

```
#include <machine.h>
extern unsigned long data1,data2;
void main(void)
{
    div0u();
    data1 = div1(data1,data2);
}
```

循环

unsigned long rotl (unsigned long data)

说 明 将数据左循环 1 位。
从操作数移出的位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 数据

返回值 左循环 1 位的结果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
    result = rotl(data);
}
```

循环

unsigned long rotr (unsigned long data)

说 明	将数据右循环 1 位。 从操作数移出的位反映到 T 位。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	右循环 1 位的结果
例	<pre>#include <machine.h> extern unsigned long result,data; void main() { result = rotr(data); }</pre>

循环

unsigned long rotcl (unsigned long data)

说 明	将数据带 T 位左循环 1 位。 从操作数移出的位反映到 T 位。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	左循环 1 位的结果
例	<pre>#include <machine.h> extern unsigned long result,data; void main() { result = rotcl(data); }</pre>
备 注	因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。 <pre>if (a) {...} /* 将比较结果到设定 T 位 */ result[1] = rotcl(data[1]); /* 循环比较结果 */</pre>

循环

unsigned long rotr (unsigned long data)

说 明 将数据带 T 位右循环 1 位。
从操作数移出的位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 数据

返回值 右循环 1 位的结果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
    result = rotr(data);
}
```

备 注 因为本内部函数参照 T 位的内容，所以必须在此之前记述正确的 T 位的值。如果在此之前记述比较或者移位等指令，因为这些运算结果反映到 T 位，所以本函数不能正常运行。

```
if (a) {...} /* 将比较结果设定到 T 位 */
result[1] = rotr(data[1]); /* 循环比较结果 */
```

左移

unsigned long shll (unsigned long data)

说 明 将数据左移 1 位。
从操作数移出的位反映到 T 位。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 数据

返回值 左移 1 位的结果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
    result = shll(data);
}
```

右移

unsigned long shlr (unsigned long data)

说 明	将数据逻辑右移 1 位。 从操作数移出的位反映到 T 位。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	右移 1 位的结果
例	<pre>#include <machine.h> extern unsigned long result,data; void main() { result = shlr(data); }</pre>

右移

long shar (long data)

说 明	将数据算术右移 1 位。 从操作数移出的位反映到 T 位。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	右移 1 位的结果
例	<pre>#include <machine.h> extern long result,data; void main() { result = shar(data); }</pre>

饱和运算

long clipsb (long data)

说 明	如果数据在 -128 ~ 127 范围内，就设定该值；如果不在范围内，就设定上限值或者下限值。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	-128 (data < -128) data (-128 <= data <= 127) 127 (127 < data)
例	<pre>#include <machine.h> extern long result,data; void main() { result = clipsb(data); }</pre>
备 注	本函数只在指定 cpu=sh2a sh2afpu 时才有效。

饱和运算

long clipsw (long data)

说 明	如果数据在 -32768 ~ 32767 范围内，就设定该值；如果不在此范围内，就设定上限值或者下限值。
头文件	<machine.h> 或者 <umachine.h>
参 数	data 数据
返回值	-32768 (data < -32768) data (-32768 <= data <= 32767) 32767 (32767 < data)
例	<pre>#include <machine.h> extern long result,data; void main() { result = clipsw(data); }</pre>
备 注	本函数只在指定 cpu=sh2a sh2afpu 时才有效。

饱和运算

unsigned long clipub (unsigned long data)

说 明 如果数据在 0 ~ 255 范围内，就设定该值；如果不在此范围内，就设定上限值。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 数据

返回值 data (data <= 255)
255 (255 < data)

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
    result = clipub(data);
}
```

备 注 本函数只在指定 cpu=sh2a|sh2afpu 时才有效。

饱和运算

unsigned long clipuw (unsigned long data)

说 明 如果数据在 0 ~ 65535 范围内，就设定该值；如果不在此范围内，就设定上限值。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 数据

返回值 data (data <= 65535)
65535 (65535 < data)

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
    result = clipuw(data);
}
```

备 注 本函数只在指定 cpu=sh2a|sh2afpu 时才有效。

TBR 的设置

void set_tbr (void *data)

说 明 给 TBR 设定 data。

头文件 <machine.h> 或者 <umachine.h>

参 数 data 设定值

例

```
#include <machine.h>
void *data;
void main(void)
{
    set_tbr(data);
}
```

备 注 本函数只在指定 cpu=sh2a | sh2afpu 时才有效。

TBR 的参照

void *get_tbr (void)

说 明 参照 TBR 的值。

头文件 <machine.h> 或者 <umachine.h>

返回值 TBR 的值

例

```
#include <machine.h>
void *result;
void main(void)
{
    result = get_tbr();
}
```

备 注 本函数只在指定 cpu=sh2a | sh2afpu 时才有效。

多重中断

void *sr_jsr (void *func, int imask)

说 明 清除 SR 寄存器的 RB 位和 BL 位，将中断屏蔽设定为 imask，然后调用 func 函数。
imask 的可指定范围是 0 ~ 15，但是如果指定 0，就不进行中断屏蔽的设定，而只清除 RB 位和 BL 位。

头文件 <machine.h> 或者 <smachine.h>

返回值 无

例

```
#include <machine.h>
#pragma interrupt func1(bank)
extern void func2(void);
void func1(void)
{
    sr_jsr(func2, 15);
}

_func1:
    MOV.L    R14,@-R15
    STS.L    PR,@-R15
    STC      SSR,@-R15
    STC      SPC,@-R15
    STC      SR,R4
    MOV.L    L11,R1      ; H'FFFFFF0F
    MOV      #-16,R5     ; H'FFFFFFF0
    AND      R1,R4      ; 清除 RB 和 BL 位
    EXTU.B   R5,R5
    MOV.L    L11+4,R14   ; _func2
    OR       R5,R4      ; 将中断屏蔽设定为 15
    LDC      R4,SR
    JSR      @R14
    NOP
    LDC      @R15+,SPC
    LDC      @R15+,SSR
    LDS.L    @R15+,PR
    MOV.L    @R15+,R14
    RTE
    NOP
```

备 注 只在指定 cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp 时才能使用。
只能对 func 指定参数、或者持有返回值的函数、或者该函数的指针。
在使用 sr_jsr() 时，如果用 #pragma global_register 指定 R8 ~ R14 的全部寄存器，就出错。
如果不在指定中断规格 bank 的函数内使用 sr_jsr()，就出错。
如果将设定了 0 的变量指定为 imask，中断屏蔽就被设定为 0。

10.4 C/C++ 库

10.4.1 标准 C 库

(1) 库的概要

说明 C/C++ 语言中能标准利用的 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这些库所需的宏名。

库的种类和对应的标准 include 文件如表 10.36 所示。

表 10.36 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	进行字符操作和检查的库。	<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。	<stdio.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，为了提高程序建立作业的效率，除以上标准 include 文件以外，还有如表 10.37 所示的只由宏名定义构成的标准 include 文件。

表 10.37 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>	定义有关浮点型内部表示的各种极限值。
3	<limits.h>	定义有关编译程序内部处理的各种极限值。
4	<errno.h>	在库函数中发生错误时设定给 errno 的值。
5	<fixed.h>	定义有关定点数的内部表示的各种极限值。

(b) 库的说明格式

按标准 include 文件的分类说明库的各函数。这些分类由标准 include 文件中定义的宏名和函数声明的说明格式（参照图 10.4）以及各函数的说明格式（参照图 10.5）构成。

标准 include 文件的说明格式如图 10.4、函数的说明格式如图 10.5 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。
- 在有实现性定义规格时，或者在本 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 10.4 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，由各编译程序定义的对错误种类的值被设定到 <code>errno</code> *。
备 考	补充说明或者使用注意事项。
实现性定义规格	本编译程序的处理方法。

图 10.5 函数的说明格式

【注】 * `errno` 是保存库函数执行期间发生错误的错误种类的变量，详细内容请参照“10.4.1(2) <stddef.h>”。

(c) 库函数说明所使用的术语

(i) 流输入 / 输出

在数据输入 / 输出时，如果按字符调用输入 / 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，将缓冲区内的一次性的输入 / 输出。

站在程序的角度来看，按字符调用输入 / 输出函数的方法比较简单。

通过库函数自动管理缓冲区，在程序中不需在意缓冲区的状态，从而能高效率地进行以字符为单位的输入 / 输出。

这种为了实现高效率的数据输入 / 输出而不需在意详细手段、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 / 输出所需的缓冲区和相关信息存储在一个结构体中，此结构体在标准 include 文件 `<stdio.h>` 中被定义为 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 指针指定给其他流输入 / 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，在连接时被连接。

宏由 `#define` 语句定义在与该函数相关的标准 include 文件中。

有关宏，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。
- 如果将有副作用的表达式（赋值表达式、递增、递减）指定为宏的参数，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` 递增 2 次，而且结果值也和最初的 `a` 的绝对值不同。

(iv) EOF

EOF 是 `getc` 函数、`getchar` 函数、`fgetc` 函数等函数（从文件输入数据的函数）在文件结束（End Of File）时的返回值，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，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，库函数中的字符串参数也必须遵循此规定。将表示此字符串结束的字符 `"\0"` 称为空字符。

(vii) 返回码

一些库函数用返回值通知被指定的处理是否成功等结果。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”）；输出时，通过输出换行字符（“\n”），结束当前行的输出。文本文件是用于保存各种处理的标准文本而进行文件输入 / 输出的文件。对于文本文件，由库函数输入 / 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。由库函数输入 / 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

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标准文件。

(x) 浮点型

浮点型是用近似实数表示的数。在 C/C++ 语言的源程序中用 10 进制数表示浮点型，但是在计算机内部通常用 2 进制数表示浮点型。在 2 进制数的情况下，浮点型的表示如下：

$$2^n \times m \quad (n: \text{整数、} m: 2 \text{ 进制小数})$$

在此， n 称为浮点型的指数， m 称为尾数。为了用固定的数据长度表示浮点型，通常固定 n 和 m 的位数。以下说明有关浮点型的术语：

- 基数

是指示用几进制表示浮点型的整数值。通常，基数为 2。

- 舍入

在将精度高于浮点型的运算中途结果保存为浮点型时进行舍入。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 $2^n \times m$ 的格式表示浮点型时，同一数值可能有不同的表示。

例：

$$2^5 \times 1.0_{(2)} \quad (_{(2)} \text{ 表示 2 进制数})$$

$$2^6 \times 0.1_{(2)}$$

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 的表示，称它为正规化浮点型，将浮点型转换为如此表示的操作称为正规化。

- 保护位

在保存浮点型运算的中途结果时，为了进行舍入，通常需要准备比实际浮点型多 1 位的数据。但是，这种方法在位数减少时无法得到正确的结果，所以使用再增加 1 位的方法来保存运算中途结果，此位称为保护位。

(xi) 文件存取模式

指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 10.38 所示。

表 10.38 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) 实现性定义

定义因编译程序而不同。

(xiii) 错误指示符和文件结束指示符

按每个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，只用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，能通过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，保持文件中表示当前读写位置的数据，该数据称为位置指示符。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，就不保证运行。

库并不在任何时候都检测错误。在不使用以下说明所示的格式调用库函数，不保证运行。

(2) <stddef.h>

定义标准 include 文件中共用的宏名。

以下全部为实现性定义：

种类	定义名	说明
型（宏）	ptrdiff_t	是二个指针相减结果的型。
	size_t	是 sizeof 运算符的运算结果的型。
常数（宏）	NULL	是什么都没有指向的指针值。 是能用相等运算符（==）与 0 比较的结果为真的值。
变量（宏）	errno	如果在库函数的处理中发生错误，就将定义在各库中的错误代码设定到此 errno。 在调用库函数前将 errno 置 0，在结束库函数处理后，通过检查设定在 errno 的码，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。

实现性定义规格

	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

(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 NDEBUG 的宏名（#define NDEBUG）。

【注】 如果对 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	<pre>#include <assert.h> int expression; assert (expression);</pre>
备 注	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。 诊断信息中含有参数的程序文本、源文件名和源行号。
实现性定义规格	对于 assert (expression)，当 expression 为伪时输出信息： ASSERTION FAILED: Δ 表达式 Δ FILE Δ < 文件名 >, line Δ < 行号 >

(4) <ctype.h>

判断字符的种类以及转换字符。

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	判断是否为空格字符除外的打印字符。
	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，如果输入的参数值不在能用 unsigned char 型表示的范围内并且不为 EOF 时，就不保证该函数的运行。字符种类一览表如表 10.39 所示。

表 10.39 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	能在显示器上显示的含空格(' ')的字符。 对应 ASCII 码的 0x20 ~ 0x7E。
6 控制字符	打印字符除外的字符。
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。

实现性定义的规格

	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 型表示的字符集。判断的结果为真的字符如表 10.40 所示。

表 10.40 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	<pre>#include <ctype.h> int c, ret; ret=isalnum(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=isalpha(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=iscntrl(c);</pre>		

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	:	0 以外
	字符 c 不为 10 进制数字时	:	0
参 数	c	要判断的字符	
例	<pre>#include <ctype.h> int c, ret; ret=isdigit(c);</pre>		

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	:	0 以外
	字符 c 不为非空格字符的打印字符时	:	0
参 数	c	要判断的字符	
例	<pre>#include <ctype.h> int c, ret; ret=isgraph(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=islower(c);</pre>		

打印字符的判断

int isprint(int c)

说 明	判断字符是否为含空格（' '）字符的打印字符		
头文件	<ctype.h>		
返回值	字符 c 为含空格字符的打印字符时	:	0 以外
	字符 c 不为含空格字符的打印字符时	:	0
参 数	c	要判断的字符	
例	<pre>#include <ctype.h> int c, ret; ret=isprint(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=ispunct(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=isspace(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=isupper(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=isxdigit(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=tolower(c);</pre>		

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	<pre>#include <ctype.h> int c, ret; ret=toupper(c);</pre>		

(5) <float.h>

定义有关浮点型的内部表示的各种极限值。

以下全部为实现性定义：

种类	定义名	定义值	说明
常数	FLT_RADIX	2	指数表示的基数。
(宏)	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性定义。
	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	表示是否对浮点值进行正规化处理。本宏定义的含义如下： - 正规化：1 - 不正规化：0
	FLT_MAX	3.4028235677973364e+38F	作为浮点值，float 型能表示的最大值。
	DBL_MAX	1.7976931348623158e+308	作为浮点值，double 型能表示的最大值。
	LDBL_MAX	1.7976931348623158e+308	作为浮点值，long double 型能表示的最大值。
	FLT_MAX_EXP	127	作为浮点值，float 型能表示的基数的乘方的最大值。
	DBL_MAX_EXP	1023	作为浮点值，double 型能表示的基数的乘方的最大值。
	LDBL_MAX_EXP	1023	作为浮点值，long double 型能表示的基数的乘方的最大值。
	FLT_MAX_10_EXP	38	作为浮点值，float 型能表示的 10 的乘方的最大值。
	DBL_MAX_10_EXP	308	作为浮点值，double 型能表示的 10 的乘方的最大值。
	LDBL_MAX_10_EXP	308	作为浮点值，long double 型能表示的 10 的乘方的最大值。
	FLT_MIN	1.175494351e-38F	作为浮点值，float 型能表示的正值的最小值。
	DBL_MIN	2.2250738585072014e-308	作为浮点值，double 型能表示的正值的最小值。
	LDBL_MIN	2.2250738585072014e-308	作为浮点值，long double 型能表示的正值的最小值。
	FLT_MIN_EXP	-149	作为正值，float 型能表示的浮点值的基数的乘方的最小值。
	DBL_MIN_EXP	-1074	作为正值，double 型能表示的浮点值的基数的乘方的最小值。
	LDBL_MIN_EXP	-1074	作为正值，long double 型能表示的浮点值的基数的乘方的最小值。
	FLT_MIN_10_EXP	-44	作为正值，float 型能表示的浮点值的 10 的乘方的最小值。
	DBL_MIN_10_EXP	-323	作为正值，double 型能表示的浮点值的 10 的乘方的最小值。
	LDBL_MIN_10_EXP	-323	作为正值，long double 型能表示的浮点值的 10 的乘方的最小值。
	FLT_DIG	6	float 型浮点值的 10 进制精度的最大位数。

种类	定义名	定义值	说明
常数 (宏)	DBL_DIG	15	double 型浮点值的 10 进制精度的最大位数。
	LDBL_DIG	15	long double 型浮点值的 10 进制精度的最大位数。
	FLT_MANT_DIG	24	按基数表示 float 型的浮点值时的尾数的最大位数。
	DBL_MANT_DIG	53	按基数表示 double 型的浮点值时的尾数的最大位数。
	LDBL_MANT_DIG	53	按基数表示 long double 型的浮点值时的尾数的最大位数。
	FLT_EXP_DIG	8	按基数表示 float 型的浮点值时的指数的最大位数。
	DBL_EXP_DIG	11	按基数表示 double 型的浮点值时的指数的最大位数。
	LDBL_EXP_DIG	11	按基数表示 long double 型的浮点值时的指数的最大位数。
	FLT_POS_EPS	5.9604648328104311e-8F	表示 float 型的 $1.0+x \neq 1.0$ 的最小浮点值 x 。
	DBL_POS_EPS	1.1102230246251567e-16	表示 double 型的 $1.0+x \neq 1.0$ 的最小浮点值 x 。
	LDBL_POS_EPS	1.1102230246251567e-16	表示 long double 型的 $1.0+x \neq 1.0$ 的最小浮点值 x 。
	FLT_NEG_EPS	2.9802324164052156e-8F	表示 float 型的 $1.0-x \neq 1.0$ 的最小浮点值 x 。
	DBL_NEG_EPS	5.5511151231257834e-17	表示 double 型的 $1.0-x \neq 1.0$ 的最小浮点值 x 。
	LDBL_NEG_EPS	5.5511151231257834e-17	表示 long double 型的 $1.0-x \neq 1.0$ 的最小浮点值 x 。
	FLT_POS_EPS_EXP	-23	表示 float 型的 $1.0+(\text{基数})^n \neq 1.0$ 的最小整数 n 。
	DBL_POS_EPS_EXP	-52	表示 double 型的 $1.0+(\text{基数})^n \neq 1.0$ 的最小整数 n 。
	LDBL_POS_EPS_EXP	-52	表示 long double 型的 $1.0+(\text{基数})^n \neq 1.0$ 的最小整数 n 。
	FLT_NEG_EPS_EXP	-24	表示 float 型的 $1.0-(\text{基数})^n \neq 1.0$ 的最小整数 n 。
	DBL_NEG_EPS_EXP	-53	表示 double 型的 $1.0-(\text{基数})^n \neq 1.0$ 的最小整数 n 。
	LDBL_NEG_EPS_EXP	-53	表示 long double 型的 $1.0-(\text{基数})^n \neq 1.0$ 的最小整数 n 。

(6) <limits.h>

定义有关整数型数据的内部表示的各种极限值。

以下全部为实现性定义：

种类	定义名	定义值	说明
常数	CHAR_BIT	8	表示 char 型由几位构成。
(宏)	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	2147483647	作为值，int 型变量具有的最大值。
	INT_MIN	-2147483647-1	作为值，int 型变量具有的最小值。
	UINT_MAX	4294967295U	作为值，unsigned int 型变量具有的最大值。
	LONG_MAX	2147483647L	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。
	LLONG_MAX	9223372036854775807LL	作为值，long long 型变量具有的最大值。
	LLONG_MIN	-9223372036854775807LL-1LL	作为值，long long 型变量具有的最小值。
	ULLONG_MAX	18446744073709551615ULL	作为值，unsigned long long 型变量具有的最大值。

(7) <errno.h>

定义在库函数中发生错误时给 `errno` 设定的值。

以下全部为实现性定义：

种类	定义名	说明
变量（宏）	<code>errno</code>	<code>int</code> 型变量。在库函数中发生错误时，设定错误号。
常数（宏）	<code>ERANGE</code>	请参照“12.3 C 标准库函数的错误信息”。
	<code>EDOM</code>	
	<code>ESTRN</code>	
	<code>PTRERR</code>	
	<code>ECBASE</code>	
	<code>ETLN</code>	
	<code>EEXP</code>	
	<code>EEXPN</code>	
	<code>EFLOATO</code>	
	<code>EFLOATU</code>	
	<code>EDBLO</code>	
	<code>EDBLU</code>	
	<code>ELDBLO</code>	
	<code>ELDBLU</code>	
	<code>NOTOPN</code>	
	<code>EBADF</code>	
	<code>ECSPEC</code>	
	<code>EFIXEDO</code>	
	<code>EFIXEDU</code>	
	<code>EACCUMO</code>	
	<code>EACCUMU</code>	
	<code>ELFIXEDO</code>	
	<code>ELFIXEDU</code>	
	<code>ELACCUMO</code>	
	<code>ELACCUMU</code>	
	<code>EMALRESM</code>	
	<code>EMALFRSM</code>	
	<code>ETOKRESM</code>	
	<code>ETOKFRSM</code>	
	<code>EIOBRESM</code>	
	<code>EIOBFRSM</code>	

(8) <fixed.h>

定义有关定点数的内部表示的各种极限值。

以下全部为实现性定义

种类	定义名	定义值	说明
常数 (宏)	FIXED_BIT	16	表示 <code>__fixed</code> 型由几位构成。
	FIXED_MIN	(-0.5r-0.5r)	作为值, <code>__fixed</code> 型变量具有的最小值。
	FIXED_MAX	0.999969482421875r	作为值, <code>__fixed</code> 型的变量具有的最大值。
	FIXED_EPSILON	0.000030517578125r	在大于能用 <code>__fixed</code> 型表示的 0.0r 值中, 最小值和 0.0r 的差。
	LFIXED_BIT	32	表示 <code>long __fixed</code> 型由几位构成。
	LFIXED_MIN	(-0.5R-0.5R)	作为值, <code>long __fixed</code> 型的变量具有的最小值。
	LFIXED_MAX	0.9999999953433871269226074 21875R	作为值, <code>long __fixed</code> 型的变量具有的最大值。
	LFIXED_EPSILON	0.0000000046566128730773925 78125R	在大于能用 <code>long __fixed</code> 型表示的 0.0R 的值中, 最小值和 0.0R 的差。
	ACCUM_BIT	24	表示 <code>__accum</code> 型由几位构成。
	ACCUM_MIN	(-128.0a-128.0a)	作为值, <code>__accum</code> 型的变量具有的最小值。
	ACCUM_MAX	255.999969482421875a	作为值, <code>__accum</code> 型的变量具有的最大值。
	ACCUM_EPSILON	0.000030517578125a	在大于能用 <code>__accum</code> 型表示的 0.0a 的值中, 最小值和 0.0a 的差。
	LACCUM_BIT	40	表示 <code>long __accum</code> 型由几位构成。
	LACCUM_MIN	(-128.0A-128.0A)	作为值, <code>long __accum</code> 型的变量具有的最小值。
	LACCUM_MAX	255.99999999534338712692260 7421875A	作为值, <code>long __accum</code> 型的变量具有的最大值。
	LACCUM_EPSILON	0.0000000046566128730773925 78125A	在大于能用 <code>long __accum</code> 型表示的 0.0A 的值, 最小值和 0.0A 的差。

(9) <math.h>

计算各种数值。

以下常数（宏）全部为实现性定义：

种类	定义名	说明
常数（宏）	EDOM	当输入给函数的参数值超出函数中定义的值的范围时，设定给 errno 的值。
	ERANGE	当函数计算结果不能表示为 double 型的值或者发生上溢 / 下溢时，设定给 errno 的值。
	HUGE_VAL	当函数的计算结果发生下溢时，作为函数的返回值。
函数	acos	计算浮点值的反余弦。
	asin	计算浮点值的反正弦。
	atan	计算浮点值的反正切。
	atan2	计算浮点值和浮点值相除后的结果的反正切。
	cos	计算浮点值的弧度值的余弦。
	sin	计算浮点值的弧度值的正弦。
	tan	计算浮点值的弧度值的正切。
	cosh	计算浮点值的双曲余弦。
	sinh	计算浮点值的双曲正弦。
	tanh	计算浮点值的双曲正切。
	exp	计算浮点值的指数函数。
	frexp	将浮点值分为 [0.5, 1.0) 的值和 2 的乘方的积。
	ldexp	计算浮点值和 2 的乘方的积。
	log	计算浮点值的自然对数。
	log10	计算以 10 为底的浮点值的对数。
	modf	将浮点值分为整数和小数。
	pow	计算浮点值的乘方。
	sqrt	计算浮点值的正平方根。
	ceil	求上舍入浮点值的小数部分后的整数值。
	fabs	计算浮点值的绝对值。
	floor	求舍去浮点值的小数部分后的整数值。
	fmod	计算浮点值和浮点值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

当输入给函数的参数值超出函数中定义的值的范围时，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性定义。

(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，当计算结果发生上溢时，将和正确计算时相同符号的 `HUGE_VAL` 值作为返回值返回。当计算结果发生下溢时，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)
3      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. 是否发生范围错误取决于编译程序规定的浮点型内部表示格式。例如：在采用能将无穷大作为值表示的内部表示格式时，能不发生范围错误而执行 `<math.h>` 的库函数。
3. 在指定 `cpu=sh2afpu | sh4 | sh4a` 并且使用 `fabs` 函数和 `sqrt` 函数时，或者在指定 `cpu=sh2e` 和 `double=float` 并且使用 `fabs` 函数时，即使发生错误，也不设定 `errno`。

实现性定义规格

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 浮点型的规格”。
2	在数学函数中发生下溢错误时，是否将宏“ERANGE”的值设定到“errno”	不设定。
3	当 <code>fmod</code> 函数中第 2 个实际参数的值为 0 时，是否发生范围错误	发生范围错误。

反余弦

double acos(double d)

说 明	计算浮点值的反余弦。
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 要计算反余弦的浮点值
例	<pre>#include <math.h> double d, ret; ret=acos(d);</pre>
错误条件	当 d 的值超出 $[-1.0, 1.0]$ 的范围时, 发生定义域错误。
备 注	acos 函数返回值的范围为 $[0, \pi]$ 。

反正弦

double asin(double d)

说 明	计算浮点值的反正弦。
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 要计算反正弦的浮点值
例	<pre>#include <math.h> double d, ret; ret=asin(d);</pre>
错误条件	当 d 的值超出 $[-1.0, 1.0]$ 的范围时, 发生定义域错误。
备 注	asin 函数返回值的范围为 $[-\pi/2, \pi/2]$ 。

反正切

`double atan(double d)`

说 明	计算浮点值的反正切。
头文件	<code><math.h></code>
返回值	d 的反正切值
参 数	d 要计算反正切的浮点值
例	<pre>#include <math.h> double d, ret; ret=atan(d);</pre>
备 注	atan 函数返回值的范围为 $(-\pi/2, \pi/2)$ 。

除法运算后的反正切

`double atan2(double y, double x)`

说 明 计算浮点值和浮点值相除后的结果的反正切。

头文件 `<math.h>`

返回值 正常: y 除 x 后的反正切值
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, 发生定义域错误。

备 注 `atan2` 函数返回值的范围为 $(-\pi, \pi]$ 。`atan2` 函数表示的含义如图 10.6 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 $\pm\pi/2$ 。

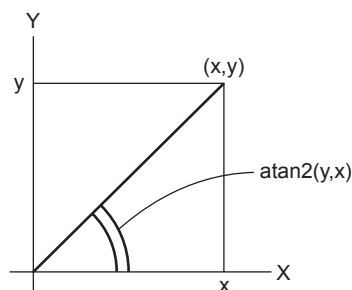


图 10.6 `atan2` 函数的含义

余弦

double cos(double d)

说 明	计算浮点值的弧度值的余弦。	
头文件	<math.h>	
返回值	d 的余弦值	
参 数	d	要计算余弦的弧度值
例	<pre>#include <math.h> double d, ret; ret=cos(d);</pre>	

正弦

double sin(double d)

说 明	计算浮点值的弧度值的正弦。	
头文件	<math.h>	
返回值	d 的正弦值	
参 数	d	要计算正弦的弧度值
例	<pre>#include <math.h> double d, ret; ret=sin(d);</pre>	

正切

double tan(double d)

说 明	计算浮点值的弧度值的正切。	
头文件	<math.h>	
返回值	d 的正切值	
参 数	d	要计算正切的弧度值
例	<pre>#include <math.h> double d, ret; ret=tan(d);</pre>	

双曲余弦

double cosh(double d)

说 明	计算浮点值的双曲余弦。	
头文件	<math.h>	
返回值	d 的双曲余弦值	
参 数	d	要计算双曲余弦的浮点值
例	<pre>#include <math.h> double d, ret; ret=cosh(d);</pre>	

双曲正弦

`double sinh(double d)`

说 明	计算浮点值的双曲正弦。	
头文件	<code><math.h></code>	
返回值	d 的双曲正弦值	
参 数	d	要计算双曲正弦的浮点值
例	<pre>#include <math.h> double d, ret; ret=sinh(d);</pre>	

双曲正切

`double tanh(double d)`

说 明	计算浮点值的双曲正切。	
头文件	<code><math.h></code>	
返回值	d 的双曲正切值	
参 数	d	要计算双曲正切的浮点值
例	<pre>#include <math.h> double d, ret; ret=tanh(d);</pre>	

指数函数

double exp(double d)

说 明	计算浮点值的指数函数。	
头文件	<math.h>	
返回值	d 的指数函数值	
参 数	d	要计算指数函数的浮点值
例	<pre>#include <math.h> double d, ret; ret=exp(d);</pre>	

将浮点值分为尾数和指数

double frexp(double value, int *exp)

说 明	将浮点值分为 $[0.5, 1.0)$ 的值和 2 的乘方的积。	
头文件	<math.h>	
返回值	value 为 0.0 时	: 0.0
	value 不为 0.0 时	: $ret * 2^{exp}$ 的指向区域的值 = value 定义的 ret 值
参 数	value	要分为 $[0.5, 1.0)$ 的值和 2 的乘方的积的浮点值
	exp	指向保存 2 的乘方值的存储区的指针
例	<pre>#include <math.h> double ret, value; int *exp; ret=frexp(value, exp);</pre>	
备 注	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 exp 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，exp 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值

double ldexp(double e, int f)

说 明	计算浮点值和 2 的乘方的积。	
头文件	<math.h>	
返回值	$e \times 2^f$ 运算结果的值	
参 数	e	要计算 2 的乘方值的浮点值
	f	2 的乘方值
例	<pre>#include <math.h> double ret, e; int f; ret=ldexp(e, f);</pre>	

自然对数

double log(double d)

说 明	计算浮点值的自然对数。	
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	要计算自然对数的浮点值
例	<pre>#include <math.h> double d, ret; ret=log(d);</pre>	
错误条件	当 d 的值是负值时, 发生定义域错误。 当 d 的值是 0.0 时, 发生范围错误。	

常用对数

`double log10(double d)`

说 明	计算以 10 为底的浮点值的对数。	
头文件	<code><math.h></code>	
返回值	正常: <code>d</code> 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	<code>d</code>	要计算以 10 为底的对数的浮点值
例	<pre>#include <math.h> double d, ret; ret=log10(d);</pre>	
错误条件	当 <code>d</code> 的值是负值时, 发生定义域错误。 当 <code>d</code> 的值是 0.0 时, 发生范围错误。	

将浮点值分为整数和小数

`double modf(double a, double *b)`

说 明	将浮点值分为整数和小数。	
头文件	<code><math.h></code>	
返回值	<code>a</code> 的小数	
参 数	<code>a</code>	要分为整数和小数的浮点值
	<code>b</code>	指向保存整数的存储区的指针
例	<pre>#include <math.h> double a, *b, ret; ret=modf(a,b);</pre>	

乘方

double pow(double x, double y)

说 明 计算浮点值的乘方。

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 的值小于等于 0.0 、或者 x 的值为负并且 y 的值不为整数时, 发生定义域错误。

平方根

double sqrt(double d)

说 明 计算浮点值的正平方根。

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 要计算正平方根的浮点值

例

```
#include <math.h>
double d, ret;
ret=sqrt(d);
```

错误条件 当 d 的值是负值时, 发生定义域错误。

上舍入

double ceil(double d)

说 明 求上舍入浮点值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值

例

```
#include <math.h>
double d, ret;
ret=ceil(d);
```

备 注 ceil 函数将大于等于 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 计算浮点值的绝对值。

头文件 <math.h>

返回值 d 的绝对值

参 数 d 要计算绝对值的浮点值

例

```
#include <math.h>
double d, ret;
ret=fabs(d);
```

舍去

double floor(double d)

说 明 求舍去浮点值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值

例

```
#include <math.h>
double d, ret;
ret=floor(d);
```

备 注 **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 计算浮点值和浮点值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 : **x** 除 **y** 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 注 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。

在无法表示 **x/y** 的商时，不保证结果值。

(10) <mathf.h>

进行各种数值计算。

在 <mathf.h> 中声明了 ANSI 规格规定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）全部为实现性定义：

种类	定义名	说明
常数（宏）	EDOM	当输入给函数的参数值超出函数中定义的值的范围时，设定给 errno 的值。
	ERANGE	当函数计算结果不能表示为 float 型的值或者发生下溢 / 下溢时，设定给 errno 的值。
	HUGE_VALF	当函数的计算结果发生下溢时，为函数的返回值。
函数	acosf	计算浮点值的反余弦。
	asinf	计算浮点值的反正弦。
	atanf	计算浮点值的反正切。
	atan2f	计算浮点值和浮点值相除后的结果的反正切。
	cosf	计算浮点值的弧度值的余弦。
	sinf	计算浮点值的弧度值的正弦。
	tanf	计算浮点值的弧度值的正切。
	coshf	计算浮点值的双曲余弦。
	sinhf	计算浮点值的双曲正弦。
	tanhf	计算浮点值的双曲正切。
	expf	计算浮点值的指数函数。
	frexpf	将浮点值分为 [0.5, 1.0) 的值和 2 的乘方的积。
	ldexpf	计算浮点值和 2 的乘方的积。
	logf	计算浮点值的自然对数。
	log10f	计算以 10 为底的浮点值的对数。
	modff	将浮点值分为整数和小数。
	powf	计算浮点值的乘方。
	sqrtf	计算浮点值的正平方根。
	ceilf	求上舍入浮点值的小数部分后的整数值。
	fabsf	计算浮点值的绝对值。
	floorf	求舍去浮点值的小数部分后的整数值。
	fmodf	计算浮点值和浮点值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

当输入给函数的参数值超出函数中定义的值的范围时，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性定义。

(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，当计算结果发生上溢时，将和正确计算时相同符号的 `HUGE_VALF` 值作为返回值返回。当计算结果发生下溢时，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)
3      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. 是否发生范围错误取决于编译程序的浮点型内部表示格式。例如：在采用能将无穷大作为值表示的内部表示格式时，能不发生范围错误而执行 `< mathf.h >` 的库函数。
3. 在指定 `cpu=sh2afpu | sh4 | sh4a` 并且使用 `fbsf` 函数和 `sqrtrf` 函数时，或者在指定 `cpu=sh2e` 并且使用 `fbsf` 函数时，即使发生错误，也不设定 `errno`。

实现性定义规格

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 浮点型的规格”。
2	在数学函数中发生下溢错误时，是否将宏“ERANGE”的值设定到“errno”	不设定。
3	当 <code>fmod</code> 函数中第 2 个实际参数的值为 0 时，是否发生范围错误	发生范围错误。

反余弦

float acosf(float f)

说 明	计算浮点值的反余弦。
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 要计算反余弦的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=acosf(f);</pre>
错误条件	当 f 的值超出 $[-1.0, 1.0]$ 的范围时, 发生定义域错误。
备 注	acosf 函数返回值的范围为 $[0, \pi]$ 。

反正弦

float asinf(float f)

说 明	计算浮点值的反正弦。
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 要计算反正弦的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=asinf(f);</pre>
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 注	asinf 函数返回值的范围为 $[-\pi/2, \pi/2]$ 。

反正切

float atanf(float f)

说 明	计算浮点值的反正切。
头文件	<mathf.h>
返回值	F 的反正切值
参 数	f 要计算反正切的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=atanf(f);</pre>
备 注	atanf 函数返回值的范围为 $(-\pi/2, \pi/2)$ 。

除法运算后的反正切

float atan2f(float y, float x)

说 明 计算浮点值和浮点值相除后的结果的反正切。

头文件 <mathf.h>

返回值 正常: y 除 x 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, 发生定义域错误。

备 注 atan2f 函数返回值的范围为 $(-\pi, \pi]$ 。 atan2f 函数表示的含义如图 10.7 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 $\pm\pi/2$ 。

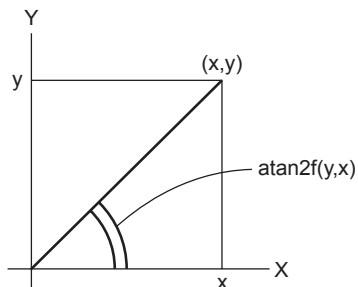


图 10.7 atan2f 函数的含义

余弦

float cosf(float f)

说 明	计算浮点值的弧度值的余弦。	
头文件	<mathf.h>	
返回值	f 的余弦值	
参 数	f	要计算余弦的弧度值
例	<pre>#include <mathf.h> float f, ret; ret=cosf(f);</pre>	

正弦

float sinf(float f)

说 明	计算浮点值的弧度值的正弦	
头文件	<mathf.h>	
返回值	f 的正弦值	
参 数	f	要计算正弦的弧度值
例	<pre>#include <mathf.h> float f, ret; ret=sinf(f);</pre>	

正切

float tanf(float f)

说 明	计算浮点值的弧度值的正切。	
头文件	<mathf.h>	
返回值	f 的正切值	
参 数	f	要计算正切的弧度值
例	<pre>#include <mathf.h> float f, ret; ret=tanf(f);</pre>	

双曲余弦

float coshf(float f)

说 明	计算浮点值的双曲余弦。	
头文件	<mathf.h>	
返回值	f 的双曲余弦值	
参 数	f	要计算双曲余弦的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=coshf(f);</pre>	

双曲正弦

float sinh(float f)

说 明	计算浮点值的双曲正弦。	
头文件	<mathf.h>	
返回值	f 的双曲正弦值	
参 数	f	要计算双曲正弦的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=sinhf(f);</pre>	

双曲正切

float tanhf(float f)

说 明	计算浮点值的双曲正切。	
头文件	<mathf.h>	
返回值	f 的双曲正切值	
参 数	f	要计算双曲正切的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=tanhf(f);</pre>	

指数函数

float expf(float f)

说 明	计算浮点值的指数函数。
头文件	<mathf.h>
返回值	f 的指数函数值
参 数	f 要计算指数函数的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=expf(f);</pre>

将浮点值分为尾数和指数

float frexpf(float value, int *exp)

说 明	将浮点型分为 [0.5, 1.0) 的值和 2 的乘方的积。
头文件	<mathf.h>
返回值	value 为 0.0 时 : 0.0 value 不为 0.0 时 : ret*2^{exp} 的指向的区域的值 = 用 value 定义的 ret 值
参 数	value 要分解为 [0.5,1.0) 的值和 2 的乘方积的浮点值 exp 指向保存 2 的乘方值的存储区的指针
例	<pre>#include <mathf.h> float ret, value; int *exp; ret=frexpf(value,exp);</pre>
备 注	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 exp 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, exp 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值

float ldexpf(float e, int f)

说 明	计算浮点值和 2 的乘方的积。	
头文件	<mathf.h>	
返回值	$e \cdot 2^f$ 运算结果的值	
参 数	e	要计算 2 的乘方值的浮点值
	f	2 的乘方值
例	<pre>#include <mathf.h> float ret, e; int f; ret=ldexpf(e, f);</pre>	

自然对数

float logf(float f)

说 明	计算浮点值的自然对数。	
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	要计算自然对数的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=logf(f);</pre>	
错误条件	当 f 的值是负值时, 发生定义域错误。 当 f 的值为 0.0, 发生范围错误。	

常用对数

float log10f(float f)

说 明	计算以 10 为底的浮点值的对数。
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 要计算以 10 为底的对数的浮点值
例	<pre>#include <mathf.h> float f, ret; ret=log10f(f);</pre>
错误条件	当 f 的值是负值时, 发生定义域错误。 当 f 的值为 0.0 时, 发生范围错误。

将浮点值分为整数和小数

float modff(float a, float *b)

说 明	将浮点值分为整数和小数。
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值 b 指向保存整数的存储区的指针
例	<pre>#include <mathf.h> float a, *b, ret; ret=modff(a,b);</pre>

乘方

float powf(float x, float y)

说 明 计算浮点值的乘方。

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 的值小于等于 0.0、或者 x 的值为负值并且 y 的值不为整数时, 发生定义域错误。

平方根

float sqrtf(float f)

说 明 计算浮点值的正平方根。

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 要计算正平方根的浮点值

例

```
#include <mathf.h>
float f, ret;
ret=sqrtf(f);
```

错误条件 当 f 的值是负值时, 发生定义域错误。

上舍入

float ceilf(float f)

说 明 求上舍入浮点值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值

例

```
#include <mathf.h>
float f, ret;
ret=ceilf(f);
```

备 注 **ceilf** 函数将大于等于 **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 计算浮点值的绝对值。

头文件 <mathf.h>

返回值 **f** 的绝对值

参 数 **f** 要计算绝对值的浮点值

例

```
#include <mathf.h>
float f, ret;
ret=fabsf(f);
```

舍去

float floorf(float f)

说 明 求舍去浮点值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值

例

```
#include <mathf.h>
float f, ret;
ret=floorf(f);
```

备 注 **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 计算浮点值和浮点值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: **x** 除 **y** 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 注 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。

在无法表示 **x/y** 的商时，不保证结果值。

(11) <setjmp.h>

支持函数之间的控制移动。

以下的宏为实现性定义。

种类	定义名	说明
型（宏）	jmp_buf	是对应存储区的型名，用于预先保存可进行函数间控制移动的信息。
函数	setjmp	将当前执行中的函数的执行环境保存到指定（由 jmp_buf 定义）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。

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 sub();
5      void main()
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，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。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）。

其结果是从第 9 行继续执行。

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 注 longjmp 函数使用由 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，有可能丢失一部分当前执行环境（表达式的评价中途结果等），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 注 longjmp 函数从第 1 个参数 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 函数的第 2 个参数 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。

(12) <stdarg.h>

对于具有可变个数的参数的函数，能参照该参数。

以下全部为实现性定义。

种类	定义名	说明
型（宏）	va_list	是由 va_start、va_arg、va_end 宏共用的变量型，用于参照可变个数的参数。
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，能对当前参照中的参数的下一个参数进行参照。
	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，给第 1 个参数指定要输出的数据个数，执行函数 prlist，输出后面的参数（第 1 个参数指定的个数）。

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 宏给第 2 个参数指定参数的型名（此时为 int 型）。

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 注 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，可对当前参照中的参数的下一个参数进行参照。

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 存取的参数型

例

```
#include <stdarg.h>
va_list ap;
int ret;
ret=va_arg(ap,int);
```

备 注 将由 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，其结果是依次将可变个数的参数作为本宏的返回值返回。
必须给第 2 个参数 type 指定参照的型。
ap 必须是由 va_start 初始化的 ap。
对于 type 的型，如果在指定函数的参数时指定了因型转换而改变长度的型（如 char 型、unsigned char 型、short 型、unsigned short 型、float 型），就不能正确地参照参数。如果指定这样的型，就不保证运行。

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 注 ap 必须是由 `va_star` 初始化的 `ap`。
另外，如果在函数 `return` 前不调用 `va_end` 宏，就不保证该函数的运行。

(13) <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）全部为实现性定义。

种类	定义名	说明
常数（宏）	FILE	是结构体的型，用于预先保存流输入 / 输出处理所需的缓冲区指针、错误指示符和结束指示符等各种控制信息。
	_IOFBF	作为缓冲区的使用方法，指示输入 / 输出处理使用全部缓冲区。
	_IOLBF	作为缓冲区的使用方法，指示输入 / 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，指示输入 / 输出处理不使用缓冲区。
	BUFSIZ	是输入 / 输出处理所需的缓冲区的大小。
	EOF	表示文件的结束（End Of File），即从文件没有更多的输入内容。
	L_tmpnam*	是保存临时文件名（由 tmpnam 函数生成）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*	是本实现性定义保证能同时打开的文件数。
	TMP_MAX*	是由 tmpnam 函数生成的固有文件名的个数的最大值。
	stderr	是指向标准错误文件的文件指针。
	stdin	是指向标准输入文件的文件指针。
	stdout	是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	在用户程序中定义和设定流输入 / 输出的缓冲区。
	setvbuf	在用户程序中定义和设定流输入 / 输出的缓冲区。
	fprintf	按格式将数据输出到流输入 / 输出文件。
	fscanf	按格式转换从流输入 / 输出文件输入的数据。
	printf	将按格式转换的数据输出到标准输出文件（stdout）。
	scanf	按格式转换从标准输入文件（stdin）输入的数据。
	sprintf	将按格式转换的数据输出到指定的区域。
	sscanf	按格式转换从指定存储区输入的数据。
	vfprintf	按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	按格式将可变个数的参数列表输出到标准输出文件。
	vsprintf	按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。

种类	定义名	说明
函数	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。

【注】* 在本实现性定义中，没被定义。

实现性定义规格

项目	编译程序的规格
1 是否需要表示输入文本最后一行结束的换行字符	没规定。取决于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 “^” 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。取决于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 10.41 所示。

表 10.41 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 / 输出文件进行一系列输入 / 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 函数的返回值返回，然后输出错误信息，结束程序。

在 fopen 函数正常结束时，返回指向被打开文件的信息保存数据（FILE 型）的指针，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 注 **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** 函数就将其输出到文件后关闭。
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 注 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 注 **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，从该文件的最后位置开始写处理。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，还可在表示文件存取模式的字符串后面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 注 **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，也进行以下处理）。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有效。
freopen 函数通常返回和 **fp** 相同的值，但在发生错误时，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 注 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 注 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、或者缓冲区满、或者有请求输入时，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** 设定不正确的值时或者在不接受缓冲区的使用方法等请求时，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 按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, ...` 按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 注 `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，将转换后的内容输出到文件指针 `fp` 指向的流输入 / 输出文件。
`fprintf` 函数通常返回转换输出的数据个数，但在发生错误时，返回负值。
格式规格如下：

【格式概要】

格式字符串由 2 种字符串构成：

• 通常字符

输出以下转换规格字符串以外的字符。

• 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：

$$\% \text{ [标志...]} \left\{ \begin{array}{l} \text{[*]} \\ \text{[字段宽度]} \end{array} \right\} \left[\begin{array}{l} \text{[-]} \\ \left\{ \begin{array}{l} \text{[*]} \\ \text{[精度]} \end{array} \right\} \end{array} \right] \text{ [指定参数长度] 转换字符}$$

对于此转换规格，如果没有实际的输出参数，就不保证其运行。另外，如果实际输出参数的个数多于转换规格，就全部忽视多余的参数。

【转换规格的说明】

(a) 标志

指定对带符号等的输出数据的修正。能指定的标志种类和含义如表 10.42 所示。

表 10.42 标志的种类和含义

种类	含义
1 -	当转换后的数据的字符数小于指定的字段宽度时，在字段内将该数据向左靠紧输出。
2 +	在转换为带符号的数据时，根据该数据的符号，给转换后的数据的前头附加正号或者负号。
3 空白	在转换为带符号的数据时，如果转换后的数据的前头没有附加符号，就给该数据的前头附加空格。 和“+”一起使用时，忽略本标志。
4 #	根据表 10.44 说明的转换种类，对转换后的数据进行修正： 1. 进行 c、d、i、s、u 转换时 忽略本标志。 2. 进行 o 转换时 在转换后的数据的前头附加 0。 3. 进行 x（或者 X）转换时 在转换后的数据的前头附加 0x（或者 0X）。 4. 进行 e、E、f、g、G 转换时 即使在转换后的数据中没有小数，也输出小数点。另外，在进行 g 或者 G 转换时，删除附加在转换后的数据后面的 0。

(b) 字段宽度

用任意的 10 进制数指定转换后的数据的输出字符数。

当转换后的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，就在数据的后面附加空格）。

如果转换后的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 10.44 说明的转换种类，指定转换后的数据的精度。

以句点 (.) 后连续 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出现矛盾时，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：

- 进行 d、i、o、u、x、X 转换时
表示转换后数据的最小位数。
- 进行 e、E、f 转换时
表示转换后数据的小数的位数。
- 进行 g、G 转换时
表示转换后数据的最大有效位数。
- 进行 s 转换时
表示打印的最大字符数。

(d) 参数的长度指定

进行 d、i、o、u、x、X、e、E、f、g、G 转换时（参照表 10.44）

指定要转换的数据长度（short 型、long 型、long long 型、long double 型）。在其他转换时，忽略本指定。长度指定的种类及其含义如表 10.43 所示。

表 10.43 参数长度指定的种类及其含义

	种类	含义
1	h	在进行 d、i、o、u、x、X 转换中，指定要转换的数据为 short 型或者 unsigned short 型。
2	l	在进行 d、i、o、u、x、X 转换中，指定要转换的数据为 long 型、unsigned long 型或者 double 型。
3	L	在进行 e、E、f、g、G 转换中，指定要转换的数据为 long double 型。
4	ll	在进行 d、i、o、u、x、X 转换中，指定要转换的数据为 long long 型或者 unsigned long long 型。在进行 n 转换中，指定要转换的数据为指向 long long 型的指针型。

(e) 转换字符

指定将要转换的数据转换为何种格式。

当要转换的数据为结构体、数组型或者指向这些型的指针时，除了对字符数组进行 s 转换和对指针进行 p 转换以外，不保证其运行。

转换字符和转换方式如表 10.44 所示。如果将此表中未记述的英文字符作为转换字符指定，就不保证其运行。另外，在指定除此以外的字符时，运行因编译程序而不同。

表 10.44 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 转换的规格相同。	int 型	精度的指定表示至少输出的字符数。如果转换后的字符数小于精度值，就在字符串的前面附加 0。在省略精度时，假设为 1。另外，对于值为 0 的数据，如果指定的精度为 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。有小数时，在小数点前至少输出 1 位数字。在省略精度时，假设为 6。另外，如果指定的精度为 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后的位数。对于转换后的字符，在小数点前输出 1 位数字，而在小数点后输出和精度相等位数的数字。在省略精度时，假设为 6。另外，如果指定的精度为 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	

	转换 字符	转换种类	转换方式	转换对象的 数据型	有关精度的注意事项
10	g	g 转换（或者 G 转换）	根据转换值和指定的有效位数的 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 数据的指数小于 -4 或者大于指定 的有效位数的精度，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后的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。
13	s	s 转换	对于由 char 型的指针型数据指向 的字符串，输出空字符（表示字 符串结束）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 不含有空格、水平制表符、换行 字符）。	指向 char 型的指 针型	精度的指定表示要输出的字符数。 如果省略精度，就对由数据指向的 字符串，输出空字符之前的字符 （但是，不输出空字符。另外，在 转换字符串中不含有空格、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	%	不进行数据转 换。	输出 %。	无	

(f) 用 “*” 指定字段宽度或者精度

能用 “*” 指定字段宽度或者精度的值。此时，将对应转换规格的参数值用作字段宽度或者精度的值。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 按格式转换从流输入 / 输出文件输入的数据。

头文件 <stdio.h>

返回值 正常：输入转换成功的数据个数
异常：在进行输入数据的转换前输入数据结束时：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 注 `fscanf` 函数从文件指针 `fp` 指向的流输入 / 输出文件输入数据，并且按 `control` 指向的格式字符串对输入的数据进行转换和编辑，结果保存到 `ptr` 指向的存储区。
用于数据输入的格式规格如下所示：

【格式概要】

格式字符串由以下 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'），就跳过输入数据中的空格类字符。

• 通常字符

如果指定既不是以上的空格字符也不是 % 的字符，就输入 1 个字符数据。在此输入的字符必须和格式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，就忽略该指针。

【转换规格的说明】

- “*” 指定

抑止将输入的数据保存到参数指向的存储区。

- 字段宽度

用 10 进制数指定输入的数据的最大字符数。

- 转换后的数据长度

在进行 d、i、o、u、x、X、e、E、f 转换时（参照表 10.46），指定转换后的数据长度（short 型、long 型、long double 型）。在其他转换时，忽略本指定。长度指定的种类及其含义如表 10.45 所示。

表 10.45 转换后的数据长度的指定种类及其含义

种类		含义
1	h	在进行 d、i、o、u、x、X 转换中，指定转换后的数据为 short 型。
2	l	在进行 d、i、o、u、x、X 转换中，指定转换后的数据为 long 型；在进行 e、E、f 转换中，指定转换后的数据为 double 型。
3	L	在进行 e、E、f 转换中，指定转换后的数据为 long double 型。
4	ll	在进行 d、i、o、u、x、X 转换中，指定转换后的数据为 long long 型。

- 转换字符

按各转换字符指定的转换种类，转换输入的数据。但是，如果读到的字符为空格类字符或转换对象外的字符、或者超过指定的字段宽度，就结束处理。

表 10.46 转换字符和转换内容

转换 字符	转换种类	转换方式	对应参数的 型名
1 d	d 转换	将 10 进制数的字符串转换为整数型数据。	整数型
2 i	i 转换	将前头带符号的 10 进制数字字符串或者最后带 u (U) 或 l (L) 的 10 进制数字字符串转换为整数型数据。另外，将以 0x (或者 0X) 开头的字符串解释为 16 进制数，并转换为 int 型数据；将以 0 开头的字符串解释为 8 进制数，并转换为 int 型数据。	整数型
3 o	o 转换	将 8 进制数的字符串转换为整数型数据。	整数型
4 u	u 转换	将无符号的 10 进制数的字符串转换为整数型数据。	整数型
5 x	x 转换	将 16 进制数的字符串转换为整数型数据。	整数型
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（设定转换后的数据字符串需要能保存含空字符的容量）。	字符型
8 c	c 转换	输入 1 个字符。此时，即使输入的字符为空格类字符，也不跳。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，此时，用于保存数据（转换后）的存储区需要指定的容量。	char 型
9 e	e 转换	将表示浮点型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。输入格式是能用 strtod 函数表示的浮点型。	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。	整数型
16 [	[转换	指定 “[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 ^，就将输入数据中出现此字符集中没有的字符前的内容作为一个字符串输入。如果第一个字符为 “^”，就将出现此字符集中的字符（“^” 除外）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（设定转换后的数据字符串需要能保存含空字符的容量）。	文字型
17 %	不进行数据转换。	读 %。	无

如果转换字符不为表 10.46 中的英文字符时，就不保证其运行。另外，对于其他字符，其运行为实现性定义。

带格式的输

int printf(const char *control [, arg]...)

说 明 将按格式转换的数据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>
const char *control="%s";
int ret;
char buffer[]="Hello World\n";
ret=printf(control,buffer);
```

备 注 `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` 函数。

带格式的输入

int scanf(const char *control [, ptr]...)

说 明 按格式转换从标准输入文件（`stdin`）输入的数据。

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 注 `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，对输入的数据进行转换和编辑，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。如果在开始转换前标准输入文件结束，就返回 EOF。
格式规格的详细内容请参照 `fscanf` 函数。
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 将按格式转换的数据输出到指定的区域。

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[]="Hello World\n";
ret=sprintf(s,control,buffer);
```

备 注 `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。
 在转换输出的字符列的最后附加空字符。作为返回值输出的字符数中不包含此空字符。
 格式规格的详细内容请参照 `fprintf` 函数。

带格式的字符串输入

`int sscanf(const char *s, const char *control [, ptr ] ...)`

说 明 按格式转换从指定存储区输入的数据。

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 注 `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，对输入的数据进行转换和编辑，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，如果在开始转换前输入的数据结束，就返回 EOF。
 格式规格的详细内容请参照 `fscanf` 函数。

可变个数参数的文件输出

`int vfprintf(FILE *fp, const char *control, va_list arg)`

说 明 按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 注 `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` 函数返回转换输出的数据个数，但在发生输出错误时，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` 函数。
 必须通过 `va_start`（和随后的 `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 按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 注 **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 函数返回转换输出的数据个数，但在发生输出错误时，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** 函数。
 必须通过 **va_start**（和随后的 **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 按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>
char *s;
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=vsprintf(s,control,buffer);
        va_arg(ap,int);
        s += ret;
    }
}
```

备 注 `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。作为返回值输出的字符数中不包含此空字符。
格式规格详细内容请参照 `fprintf` 函数。
必须通过 `va_start`（和随后的 `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 注 **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 注 `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` 个字符或者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`，但在文件结束或者发生错误时，返回 `NULL`。

在文件结束时，`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 注 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，但在发生错误时，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 注 fputs 函数将 s 指向的字符串（空字符前的内容）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，但在发生错误时，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 注 **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。

1 个字符的输入

int getchar(void)

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 注 **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。

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	<pre>#include <stdio.h> char *ret, *s; ret=gets(s);</pre>
备 注	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 函数输入文件结束前的字符或者输入到换行字符为之的字符，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，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 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	<pre>#include <stdio.h> FILE *fp; int c, ret; ret=putc(c, fp);</pre>
错误条件	如果发生写错误，就对该文件设定错误指示符。
备 注	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 ，但在发生错误时，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（ <code>stdout</code> ）。
头文件	<code><stdio.h></code>
返回值	正常：输出的字符 异常：EOF
参 数	c 要输出的字符
例	<pre>#include <stdio.h> int c, ret; ret=putchar(c);</pre>
错误条件	如果发生写错误，就对该文件设定错误指示符。
备 注	<code>putchar</code> 函数将字符 <code>c</code> 输出到标准输出文件（ <code>stdout</code> ）。 <code>putchar</code> 宏通常返回输出的字符 <code>c</code> ，但在发生错误时，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（ <code>stdout</code> ）。
头文件	<code><stdio.h></code>
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	<pre>#include <stdio.h> const char *s; int ret; ret=puts(s);</pre>
备 注	<code>puts</code> 函数将 <code>s</code> 指向的字符串输出到标准输出文件（<code>stdout</code>）。此时，不输出表示字符串结束的字符，而输出换行字符。 <code>puts</code> 函数通常返回 0，但在发生错误时，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 注 `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`，但在发生错误时，返回 EOF。
对于 `ungetc` 函数，如果不执行 `fflush`、`fseek`、`rewind` 函数，就不保证调用多于 1 次时的运行。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 注 `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 / 输出文件将 `n` 个成员的数据输入到 `ptr` 指向的存储区。此时，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。
 当 `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	<code>ptr</code>	指向保存输出数据的存储区的指针
	<code>size</code>	1 个成员的字节数
	<code>n</code>	输出的成员个数
	<code>fp</code>	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 注 `fwrite` 函数将 `size` 指定的字节数作为 1 个成员，从 `ptr` 指向的存储区将 `n` 个成员的数据输出到文件指针 `fp` 指向的流输入 / 输出文件。此时，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 注 **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 10.47 所示。
fseek 函数通常返回 0，但在出现异常时，返回 0 以外的值。

表 10.47 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 指定的值不能小于 0。
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，就向文件的前方移动。
3	SEEK_END	从文件结束位置移动到 offset（字节）指定的位置。此时，offset 指定的值不能大于 0。

对于文本文件，偏移种类必须为 **SEEK_SET**，并且 **offset** 必须为 0 或者为 **ftell** 函数对该文件的返回值。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 注 **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。
对于二进制文件，**ftell** 函数返回从文件起始位置到当前位置的字节数；对于文本文件，返回位置指示符的位置，此位置是 **fseek** 函数能使用的实现性定义值。
在对文本文件 2 次使用 **ftell** 函数后，其返回值的差并不表示实际文件中的距离。

移动到文件的起始位置

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 注 **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	<pre>#include <stdio.h> FILE *fp; clearerr(fp);</pre>	
备 注	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	文件结束的判断	
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	<pre>#include <stdio.h> FILE *fp; int ret; ret=feof(fp);</pre>	
备 注	feof 函数判断文件指针 fp 指向的流输入 / 输出文件是否结束。 对指定的流输入 / 输出文件，feof 函数调查文件结束指示符，如果结束指示符被设定，就返回 0 以外的值，表示文件结束；否则就返回 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 注 **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。
对指定的流输入 / 输出文件，**ferror** 函数调查错误指示符，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；否则就返回 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 注 **perror** 函数将 **errno** 和 **s** 指向的错误信息相对应，并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，就以 **s** 指向的字符串、冒号、空格、实现性定义的错误信息、换行字符的格式输出信息。

(14) <stdlib.h>

定义 C 程序标准处理的函数。

以下宏是实现性定义。

种类	定义名	说明
型 (宏)	div_t	是 div 函数返回值的结构体型。
	ldiv_t	是 ldiv 函数返回值的结构体型。
	lldiv_t	是 lldiv 函数返回值的结构体型。
常数 (宏)	RAND_MAX	是 rand 函数生成的伪随机数的整数最大值。
函数	atof	将表示数的字符串转换为 double 型的浮点值。
	atoi	将表示 10 进制数的字符串转换为 int 型的整数值。
	atol	将表示 10 进制数的字符串转换为 long 型的整数值。
	atoll	将表示 10 进制数的字符串转换为 long long 型的整数值。
	atofixed	将表示数的字符串转换为 long __fixed 型的定点数值。
	atolaccum	将表示数的字符串转换为 long __accum 型的定点数值。
	strtod	将表示数的字符串转换为 double 型的浮点值。
	strtol	将表示数的字符串转换为 long 型的整数值。
	strtoul	将表示数的字符串转换为 unsigned long 型的整数值。
	strtoll	将表示数的字符串转换为 long long 型的整数值。
	strtoull	将表示数的字符串转换为 unsigned long long 型的整数值。
	strtolfixed	将表示数的字符串转换为 long __fixed 型的定点数值。
	strtolaccum	将表示数的字符串转换为 long __accum 型的定点数值。
	rand	生成 0 ~ RAND_MAX 范围的伪随机数。
	srand	设定由 rand 函数生成的伪随机数的初始值。
	calloc	分配存储区, 用 0 初始化全部被分配的存储区。
	free	释放已指定的存储区。
	malloc	分配存储区。
	realloc	将存储区的容量更改为指定的容量。
	calloc__X	分配 X 存储区, 用 0 初始化全部被分配的 X 存储区。
	free__X	释放已指定的 X 存储区。
	malloc__X	分配 X 存储区。
	realloc__X	将 X 存储区的容量更改为指定的容量。
	calloc__Y	分配 Y 存储区, 用 0 初始化全部被分配的 Y 存储区。
	free__Y	释放已指定的 Y 存储区。
	malloc__Y	分配 Y 存储区。
	realloc__Y	将 Y 存储区的容量更改为指定的容量。
	bsearch	进行 2 分检索。
	qsort	排序。
	abs	计算 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	计算 long 型整数的绝对值。
	ldiv	long 型整数除法运算的商和余数。
	llabs	计算 long long 型整数的绝对值。
	lldiv	long 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	将表示数的字符串转换为 double 型的浮点值。
头文件	<code><stdlib.h></code>
返回值	转换后的 double 型的浮点值
参 数	nptr 要转换的表示数的字符串的指针
例	<pre>#include <stdlib.h> const char *nptr; double ret; ret=atof(nptr);</pre>
错误条件	如果转换后的值发生上溢 / 下溢，就设定 errno 。
备 注	对不符合浮点型格式的最初字符前的字符串进行转换。 如果 atof 函数发生上溢等错误，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtod 函数。

将字符串转换为 int 型

int atoi(const char *nptr)

说 明	将表示 10 进制数的字符串转换为 int 型的整数值。
头文件	<code><stdlib.h></code>
返回值	转换后的 int 型的整数值
参 数	nptr 要转换的表示数的字符串的指针
例	<pre>#include <stdlib.h> const char *nptr; int ret; ret=atoi(nptr);</pre>
错误条件	如果转换后的值发生上溢，就设定 errno 。
备 注	对不符合 10 进制数格式的最初字符前的字符串进行转换。 如果 atoi 函数发生上溢等错误，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtol 函数。

将字符串转换为 long 型

long atol(const char *nptr)

说 明	将表示 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	转换后的 long 型的整数值
参 数	nptr 要转换的表示数的字符串的指针
例	<pre>#include <stdlib.h> const char *nptr; long ret; ret=atol(nptr);</pre>
错误条件	如果转换后的值发生上溢，就设定 errno 。
备 注	对不符合 10 进制数格式的最初字符前的字符串进行转换。 如果 atol 函数发生上溢等错误，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtoul 函数。

将字符串转换为 long long 型

long long atoll (const char *nptr)

说 明	将表示 10 进制数的字符串转换为 long long 型的整数值。
头文件	<stdlib.h>
返回值	转换后的 long long 型整数值
参 数	nptr 指向要转换的表示数的字符串的指针
例	<pre>#include <stdlib.h> const char *nptr; long long ret; ret=atoll(nptr);</pre>
错误条件	如果转换后的值发生上溢，就设定 errno 。
备 注	对不符合 10 进制数格式的最初字符前的字符进行转换。 如果 atoll 函数发生上溢，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtoll 函数。

*将字符串转换为 long __fixed 型****long __fixed atolfixed (const char *nptr)***

说 明 将表示数的字符串转换为 long __fixed 型的定点数值。

头文件 <stdlib.h>

返回值 转换后的 long __fixed 型定点数值

参 数 nptr 指向要转换的表示数字字符串的指针

例

```
#include <stdlib.h>
const char *nptr;
long __fixed ret;
ret=atolfixed(nptr);
```

错误条件 如果转换后的值发生上溢 / 下溢，就设定 errno。

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
对不符合定点格式的最初字符前的字符进行转换。
如果 atolfixed 函数发生上溢，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtolfixed 函数。

*将字符串转换为 long __accum 型****long __accum atolaccum (const char *nptr)***

说 明 将表示数的字符串转换为 long __accum 型的定点数值。

头文件 <stdlib.h>

返回值 转换后的 long __accum 型的定点数值

参 数 nptr 指向要转换的表示数字字符串的指针

例

```
#include <stdlib.h>
const char *nptr;
long __accum ret;
ret=atolaccum(nptr);
```

错误条件 如果转换后的值发生上溢 / 下溢，就设定 errno。

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
对不符合定点格式的最初字符前的字符进行转换。
如果 atolaccum 函数发生上溢，就不保证结果值。如果想得到发生错误时能保证的值，就必须使用 strtolaccum 函数。

将字符串转换为 double 型

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

说 明 将表示数的字符串转换为 double 型的浮点值。

头文件 <stdlib.h>

返回值 正常: nptr 指向的字符串是不以构成浮点型的字符开始时: 0
 nptr 指向的字符串是以构成浮点型的字符开始时: 转换后的 double 型的浮点值
 异常: 转换后的值发生上溢时: 具有和转换的字符串相同符号的 HUGE_VAL
 转换后的值发生下溢时: 0

参 数 nptr 指向要转换的表示数的字符串的指针
 endptr 指向存储区的指针, 该存储区保存最初不构成浮点值的字符的指针

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
double ret;
ret=strtod(nptr,endptr);
```

错误条件 如果转换后的值发生上溢 / 下溢, 就设定 errno。

备 注 strtod 函数根据 “10.1.3(4) 浮点运算的规格” 的规则, 将从最初的数字或者小数点到不构成浮点值的字符前的字符串转换为 double 型的浮点值。但是, 如果既不出现指数也不出现小数点, 就假设小数点在字符串的最后数字的后面。将指向最初不构成浮点型的字符的指针设定到 endptr 指向的区域。如果在读取数字前出现不构成浮点型的字符, 就设定 nptr 的值, 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 将表示数的字符串转换为 long 型的整数值。

头文件 <stdlib.h>

返回值 正常: nptr 指向的字符串是不以构成整数的字符开始时: 0
 nptr 指向的字符串是以构成整数的字符开始时: 转换后的 long 型的整数值
 异常: 转换后的值发生上溢时: LONG_MAX 或者 LONG_MIN (取决于要转换的字符串的符号)

参 数 nptr 指向要转换的表示数的字符串的指针
 endptr 指向存储区的指针, 该存储区保存最初不构成整数的字符的指针
 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 如果转换后的值发生上溢, 就设定 errno。

备 注 strtol 函数将从最初的数字到最初不构成整数的字符前的字符串转换为 long 型的整数值。
 将指向最初不构成整数的字符的指针设定到 endptr 指向的存储区。如果在读取最初的数字前出现不构成整数的字符, 就设定 nptr 的值, 但是在 endptr 为 NULL 时, 不进行此设定。
 当 base 为 0 时, 根据 “10.1.1(4) 整数” 的规则进行转换; 当 base 的值为 2 ~ 36 时, 表示转换时的基数。要转换的字符串中的 a (或者 A) ~ z (或者 Z) 的字符对应 10 ~ 35 的值。如果在要转换的字符串中存在大于等于 base 值的字符, 就结束转换处理。另外, 转换时忽略符号后的 0, 并且忽略 base 为 16 时的 0x (或者 0X)。

将字符串转换为 *unsigned long* 型***unsigned long strtoul (const char *nptr, char **endptr, int base)***

说 明 将表示数的字符串转换为 *unsigned long* 型的整数值。

头文件 <stdlib.h>

返回值 正常: *nptr* 指向的字符串是不以构成整数的字符开始时: 0
nptr 指向的字符串是以构成整数的字符开始时: 转换后的 *unsigned long* 型整数值
 异常: 转换后的值发生上溢时: *ULONG_MAX*

参 数 *nptr* 指向要转换的表示数的字符串的指针
endptr 指向存储区的指针, 该存储区保存最初不构成整数的字符的指针
base 转换基数 (0 或者 2 ~ 36)

例

```
#include <stdlib.h>
unsigned long ret;
const char *nptr;
char **endptr;
int base;
ret=strtoul(nptr,endptr,base);
```

错误条件 如果转换后的值发生上溢, 就设定 *errno*。

备 注 *strtoul* 函数将从最初的数字到最初不构成整数的字符前的字符串转换为 *unsigned long* 型的整数值。
 将指向最初不构成整数的字符的指针设定到 *endptr* 指向的存储区。如果在读取最初的数字前出现不构成整数的字符, 就设定 *nptr* 的值, 但是在 *endptr* 为 *NULL* 时, 不进行此设定。
 当 *base* 为 0 时, 根据 “10.1.1(4) 整数” 的规则进行转换; 当 *base* 的值为 2 ~ 36 时, 表示转换时的基数。要转换的字符串中的 a (或者 A) ~ z (或者 Z) 的字符对应 10 ~ 35 的值。如果在要转换的字符串中存在大于等于 *base* 值的字符, 就结束转换处理。另外, 转换时忽略符号后的 0, 并且忽略 *base* 为 16 时的 0x (或者 0X)。

将字符串转换为 long long 型

long long strtoll (const char *nptr, char **endptr, int base)

说 明 将表示数的字符串转换为 long long 型的整数值。

头文件 <stdlib.h>

返回值 正常: nptr 指向的字符串是不以构成整数的字符开始时: 0
 nptr 指向的字符串是以构成整数的字符开始时: 转换后的 long long 型的整数值

异常: 转换后的值发生上溢时: LLONG_MAX 或者 LLONG_MIN (取决于要转换的字符串的符号)

参 数 nptr 指向要转换的表示数的字符串的指针
 endptr 指向存储区的指针, 该存储区保存最初不构成整数的字符的指针
 base 转换基数 (0 或者 2 ~ 36)

例

```
#include <stdlib.h>
long long ret;
const char *nptr;
char **endptr;
int base;
ret=strtoll(nptr,endptr,base);
```

错误条件 如果转换后的值发生上溢, 就设定 errno。

备 注 strtoll 函数将从最初的数字到最初不构成整数的字符前的字符串转换为 long long 型的整数值。
 将指向最初不构成整数的字符的指针设定到 endptr 指向的存储区。如果在读取最初的数字前出现不构成整数的字符, 就设定 nptr 的值, 但是在 endptr 为 NULL 时, 不进行此设定。
 当 base 为 0 时, 根据 “10.1.1(4) 整数” 的规则进行转换; 当 base 的值为 2 ~ 36 时, 表示转换时的基数。要转换的字符串中的 a (或者 A) ~ z (或者 Z) 的字符对应 10 ~ 35 的值。如果在要转换的字符串中存在大于等于 base 值的字符, 就结束转换处理。另外, 转换时忽略符号后的 0, 并且忽略 base 为 16 时的 0x (或者 0X)。

将字符串转换为 *unsigned long long* 型***unsigned long long strtoull (const char *nptr, char **endptr, int base)***

说 明 将表示数的字符串转换为 *unsigned long long* 型的整数值。

头文件 <stdlib.h>

返回值 正常: *nptr* 指向的字符串是不以构成整数的字符开始时: 0
nptr 指向的字符串是以构成整数的字符开始时: 转换后的 *unsigned long long* 型整数值
 异常: 转换后的值发生上溢时: ULLONG_MAX

参 数 *nptr* 指向要转换的表示数的字符串的指针
endptr 指向存储区的指针, 该存储区保存最初不构成整数的字符的指针
base 转换基数 (0 或者 2 ~ 36)

例

```
#include <stdlib.h>
unsigned long long ret;
const char *nptr;
char **endptr;
int base;
ret=strtoull(nptr,endptr,base);
```

错误条件 如果转换后的值发生上溢, 就设定 *errno*。

备 注 *strtoull* 函数将从最初的数字到最初不构成整数的字符前的字符串转换为 *unsigned long long* 型的整数值。
 将指向最初不构成整数的字符的指针设定到 *endptr* 指向的存储区。如果在读取最初的数字前出现不构成整数的字符, 就设定 *nptr* 的值, 但是在 *endptr* 为 NULL 时, 不进行此设定。
 当 *base* 为 0 时, 根据 “10.1.1(4) 整数” 的规则进行转换; 当 *base* 的值为 2 ~ 36 时, 表示转换时的基数。要转换的字符串中的 a (或者 A) ~ z (或者 Z) 的字符对应 10 ~ 35 的值。如果在要转换的字符串中存在大于等于 *base* 值的字符, 就结束转换处理。另外, 转换时忽略符号后的 0, 并且忽略 *base* 为 16 时的 0x (或者 0X)。

将字符串转换为 `long _ _fixed` 型

`long _ _fixed strtolfixed (const char *nptr, char **endptr)`

说 明 将表示数的字符串转换为 `long _ _fixed` 型的定点数值。

头文件 `<stdlib.h>`

返回值 正常: `nptr` 指向的字符串是不以构成定点数开始时: 0
 `nptr` 指向的字符串是以构成定点数的字符开始时: 转换后的 `long _ _fixed` 型的定点数值
 异常: 转换后的值发生上溢时: `LFIXED_MAX` 或者 `LFIXED_MIN` (取决于要转换的字符串的符号)
 转换后的值发生下溢时: 0

参 数 `nptr` 指向要转换的表示数的字符串的指针
 `endptr` 指向存储区的指针, 该存储区保存最初不构成定点数值的字符的指针

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
long _ _fixed ret;
ret=strtolfixed(nptr,endptr);
```

错误条件 如果转换后的值发生上溢 / 下溢, 就设定 `errno`。

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 时才能使用本函数。
 `strtolfixed` 函数将从最初的数字或者小数点到最初不构成定点数值的字符前的字符串转换为 `long _ _fixed` 型的定点数值。但是, 如果既不出现指数也不出现小数点, 就假设小数点在字符串的最后数字的后面。将指向最初不构成定点数的字符的指针设定到 `Endptr` 指向的区域。如果在读取数字前出现不构成定点数的字符, 就设定 `nptr` 的值, 但是在 `endptr` 为 `NULL` 时, 不进行此设定。

将字符串转换为 long _ _accum 型

long _ _accum strtolaccum (const char *nptr, char **endptr)

说 明 将表示数的字符串转换为 long _ _accum 型的定点数值。

头文件 <stdlib.h>

返回值 正常: nptr 指向的字符串是不以构成定点数的字符开始时: 0
 nptr 指向的字符串是以构成定点数的字符开始时: 转换后的 long _ _accum 型的定点数值
 异常: 转换后的值发生上溢时: LACCUM_MAX 或者 LACCUM_MIN (取决于要转换的字符串的符号)
 转换后值发生下溢时: 0

参 数 nptr 指向要转换的表示数的字符串的指针
 endptr 指向存储区的指针, 该存储区保存最初不构成定点数值的字符的指针

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
long _ _accum ret;
ret=strtolaccum(nptr,endptr);
```

错误条件 如果转换后的值发生上溢 / 下溢, 就设定 errno。

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
 strtolaccum 函数将从最初的数字或者小数点到最初不构成定点数值的字符前的字符串转换为 long _ _accum 型的定点数值。但是, 如果既不出现指数也不出现小数点, 就假设小数点在字符串的最后数字的后面。将指向最初不构成定点数的字符的指针设定到 endptr 指向的区域。如果在读取数字前出现不构成定点数的字符, 就设定 nptr 的值, 但是在 endptr 为 NULL 时, 不进行此设定。

伪随机数的生成

int rand(void)

说 明 生成 0 ~ RAND_MAX 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 设定由 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 注 为了 rand 函数生成伪随机数串，srand 函数设定初始值。因此，当 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，就将生成伪随机数串的初始值设定为 1。

带初始化的存储区分配

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 注 分配 elsize 字节单位的 nelem 个存储区，并且用 0 初始化被分配存储区的全部的位。

存储区的释放

void free(void *ptr)

说 明 释放指定的存储区。

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 注 能重新分配使用释放后的 ptr 指向的存储区。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	<pre>#include <stdlib.h> size_t size; void *ret; ret=malloc(size);</pre>
备 注	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	<pre>#include <stdlib.h> size_t size; void *ptr, *ret; ret=realloc(ptr, size);</pre>
备 注	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 不是指向由 calloc、malloc、realloc 函数分配的存储区的指针或者 ptr 是指向已被 free、realloc 函数释放的存储区的指针时，不运行。

带初始化的 X 存储区分配

void _X *calloc_ _X (size_t nelem, size_t elsize)

说 明 分配 X 存储区，用 0 初始化全部被分配的 X 存储区。

头文件 <stdlib.h>

返回值 正常：被分配的 X 存储区的起始地址
异常：不能分配 X 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 _X *ret;
ret=calloc_ _X(nelem,elsize);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
分配 elsize 字节单位的 nelem 个 X 存储区，并且用 0 初始化被分配的 X 存储区的全部的位。

X 存储区的释放

void free_ _X (void _X *ptr)

说 明 释放指定的 X 存储区。

头文件 <stdlib.h>

参 数 ptr 要释放的 X 存储区的地址

例

```
#include <stdlib.h>
void _X *ptr;
free_ _X(ptr);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
能重新分配使用释放后的 ptr 指向的存储区。如果 ptr 为 NULL，就不进行任何操作。
对于不是由 calloc_ _X、malloc_ _X、realloc_ _X 函数分配的 X 存储区或者已由 free_ _X、realloc_ _X 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。

X 存储区的分配

void _X *malloc_ _X (size_t size)

说 明 分配 X 存储区。

头文件 <stdlib.h>

返回值 正常：被分配的 X 存储区的起始地址
异常：不能分配 X 存储区或者 size 为 0 时：NULL

参 数 size 要分配的 X 存储区的字节数

例

```
#include <stdlib.h>
size_t size;
void _ _X *ret;
ret=malloc_ _X(size);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
只分配 size 表示的字节长度的存储区。

X 存储区分配容量的变更

void _X *realloc_ _X (void _ _X *ptr, size_t size)

说 明 将 X 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 X 存储区的起始地址
异常：不能分配 X 存储区或者 size 为 0 时：NULL

参 数 ptr 要更改的 X 存储区的起始地址
size 更改后的 X 存储区的字节数

例

```
#include <stdlib.h>
size_t size;
void _ _X *ptr, *ret;
ret=realloc_ _X(ptr, size);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
将 ptr 指向的 X 存储区容量更改为 size 表示的字节长度的 X 存储区。如果新分配的 X 存储区容量小于更改前的存储区容量，新分配的 X 存储区的内容不变。
在 ptr 不是指向由 calloc_ _X、malloc_ _X、realloc_ _X 函数分配的 X 存储区的指针或者 ptr 是指向已被 free、realloc 函数释放的 X 存储区的指针时，不保证其运行。

带初始化的 Y 存储区的分配

void _Y *calloc_ _Y (size_t nelem, size_t elsize)

说 明 分配 Y 存储区，用 0 初始化全部被分配的 Y 存储区。

头文件 <stdlib.h>

返回值 正常：被分配的 Y 存储区的起始地址
异常：不能分配 Y 存储区或者某个参数为 0：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 _ _Y *ret;
ret=calloc_ _Y(nelem,elsize);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
分配 elsize 字节单位的 nelem 个 Y 存储区，并且用 0 初始化被分配的 Y 存储区的全部的位。

Y 存储区的释放

void free_ _Y (void _ _Y *ptr)

说 明 释放被指定的 Y 存储区。

头文件 <stdlib.h>

参 数 ptr 要释放的 Y 存储区的地址

例

```
#include <stdlib.h>
void _ _Y *ptr;
free_ _Y(ptr);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
能重新分配使用释放后的 ptr 指向的存储区。如果 ptr 为 NULL，就不进行任何操作。
对于不是由 calloc_ _Y、malloc_ _Y、realloc_ _Y 函数分配的 Y 存储区或者已由 free_ _Y、realloc_ _Y 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

Y 存储区的分配

void __Y *malloc__Y (size_t size)

说 明 分配 Y 存储区。

头文件 <stdlib.h>

返回值 正常：被分配的 Y 存储区的起始地址
异常：不能分配 Y 存储区或者 size 为 0 时：NULL

参 数 size 要分配的 Y 存储区的字节数

例

```
#include <stdlib.h>
size_t size;
void *ret;
ret=malloc__Y(size);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
只分配 size 表示的字节长度的 Y 存储区。

Y 存储区分配容量的变更

void __Y *realloc__Y (void __Y *ptr, size_t size)

说 明 将 Y 存储区的容量更改为指定的容量。

头文件 <stdlib.h>

返回值 正常：更改后的 Y 存储区的起始地址
异常：不能分配 Y 存储区或者 size 为 0 时：NULL

参 数 ptr 要更改的 Y 存储区的起始地址
size 更改后的 Y 存储区的字节数

例

```
#include <stdlib.h>
size_t size;
void *ptr, *ret;
ret=realloc__Y(ptr, size);
```

备 注 只在指定 cpu=sh2dsp|sh3dsp|sh4aldsp 和 dspc 时才能使用本函数。
将 ptr 指向的 Y 存储区的容量更改为 size 表示的字节长度容量的 Y 存储区。如果新的 Y 存储区的容量小于更改前的 Y 存储区的容量，新分配的 Y 存储区的容量不变。
在 ptr 不是指向由 calloc__Y、malloc__Y、realloc__Y 函数分配的 Y 存储区的指针或者 ptr 是指向已被 free__Y、realloc__Y 函数释放的 Y 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 注 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。比较的函数必须取得指向二个比较数据的指针 **p1**（第 1 个参数）和 **p2**（第 2 个参数），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 注 对 **base** 指向的表数据进行排序。数据的排列顺序用指向比较函数的指针指定。此函数必须取得指向二个比较数据的指针 **p1**（第 1 个参数）和 **p2**（第 2 个参数），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 **int** 型整数的绝对值。

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 注 在 **i** 绝对值不能表示为 **int** 型整数值时，不保证其运行。

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 numer 除 denom 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 注 在 j 绝对值的结果不能表示为 long 型的整数值时，不保证其运行。

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 numer 除 denom 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);
```

绝对值

long long llabs(long long j)

说 明 取得 long 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 long j;
long long ret;
    ret=llabs(j);
```

备 注 在 j 的绝对值的结果不能表示为 long long 型的整数值时，不保证其运行。

商和余数

`lldiv_t lldiv(long long numer, long long denom)`

说 明 计算 long long 型整数除法运算的商和余数。

头文件 <stdlib.h>

返回值 numer 除 denom 结果的商和余数

参 数 numer 被除数
 denom 除数

例

```
#include <stdlib.h>
long long numer, denom;
lldiv_t ret;
ret=lldiv(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	将指定容量的源存储区的内容复制到目标存储区。
	memcpy_ _X_ _X	将指定容量的源 X 存储区的内容复制到目标 X 存储区。
	memcpy_ _X_ _Y	将指定容量的源 Y 存储区的内容复制到目标 X 存储区。
	memcpy_ _Y_ _X	将指定容量的源 X 存储区的内容复制到目标 Y 存储区。
	memcpy_ _Y_ _Y	将指定容量的源 Y 存储区的内容复制到目标 Y 存储区。
	strcpy	将包含空字符的源字符串内容复制到目标存储区。
	strncpy	将指定字符数的源字符串复制到目标存储区。
	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，取得指定字符集中的字符最初出现前的字符数。
	strpbrk	在指定的字符串中，检索指定字符集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，取得指定字符集中的字符从头连续出现的字符数。
	strstr	在指定的字符串中，检索指定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	从指定的存储区的头开始设定指定字符数的指定字符。
	strerror	设定错误信息。
	strlen	计算字符串长度。
	memmove	将指定容量的源存储区的内容复制到目标存储区。即使复制源和复制目标的存储区重叠，也能正常复制。

实现性定义规格

项目		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 标准库函数的错误信息”。

使用本标准 include 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，就不保证运行。

例：

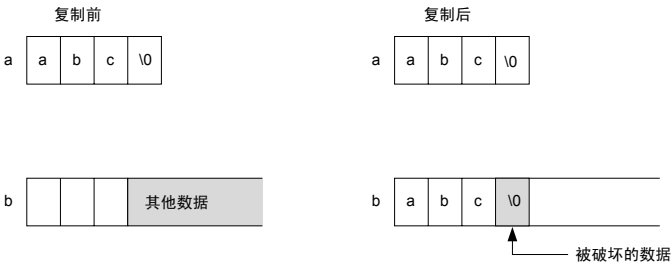
```
char a[]="abc";
char b[3];

:

:

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";

:

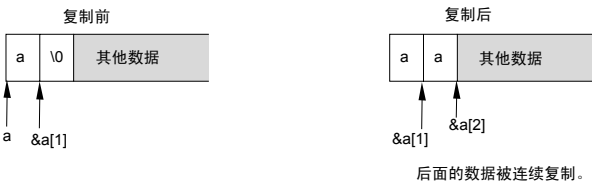
:

strcpy(&a[1], a);

:

:
```

此时，在读取复制源的字符串的空字符前，字符 'a' 覆盖空字符，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 将指定容量的源存储区的内容复制到目标存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标存储区的指针
 s2 指向源存储区的指针
 n 要复制的字符数

例 `#include <string.h>`
 `void *ret, *s1;`
 `const void *s2;`
 `size_t n;`
 `ret=memcpy(s1, s2, n);`

X 存储区的复制

void _X *memcpy_ _X_ _X(void _X *s1, const void _X *s2, size_t n)

说 明 将指定容量的源 X 存储区的内容复制到目标 X 存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标 X 存储区的指针
 s2 指向源 X 存储区的指针
 n 要复制的字符数

例 `#include <string.h>`
 `void _X *ret, *s1;`
 `const void _X *s2;`
 `size_t n;`
 `ret=memcpy_ _X_ _X(s1, s2, n);`

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 选项时才能使用本函数。

X 存储区的复制

`void __X *memcpy__X__Y(void __X *s1, const void __Y *s2, size_t n)`

说 明 将指定容量的源 Y 存储区的内容复制到目标 X 存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标 X 存储区的指针
 s2 指向源 Y 存储区的指针
 n 要复制的字符数

例 `#include <string.h>`
 `void __X *ret, *s1;`
 `const void __Y *s2;`
 `size_t n;`
 `ret=memcpy__X__Y(s1, s2, n);`

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 选项时才能使用本函数。

Y 存储区复制

`void __Y *memcpy__Y__X(void __Y *s1, const void __X *s2, size_t n)`

说 明 将指定容量的源 X 存储区的内容复制到目标 Y 存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标 Y 存储区的指针
 s2 指向源 X 存储区的指针
 n 要复制的字符数

例 `#include <string.h>`
 `void __Y *ret, *s1;`
 `const void __X *s2;`
 `size_t n;`
 `ret=memcpy__Y__X(s1, s2, n);`

备 注 只在 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 选项时才能使用本函数。

Y 存储区的复制

`void __Y *memcpy__Y__(void __Y *s1, const void __Y *s2, size_t n)`

说 明 将指定容量的源 Y 存储区的内容复制到目标 Y 存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标 Y 存储区的指针
 s2 指向源 Y 存储区的指针
 n 要复制的字符数

例 `#include <string.h>`
 `void __Y *ret, *s1;`
 `const void __Y *s2;`
 `size_t n;`
 `ret=memcpy__Y__(s1, s2, n);`

备 注 只在指定 `cpu=sh2dsp|sh3dsp|sh4aldsp` 和 `dspc` 选项时才能使用本函数。

字符串的复制

`char *strcpy(char *s1, const char *s2)`

说 明 将包含空字符的源字符串的内容复制到目标存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标存储区的指针
 s2 指向源字符串的指针

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 将指定字符数的源字符串复制到目标存储区。

头文件 <string.h>

返回值 s1 的值

参 数 s1 指向目标存储区的指针
 s2 指向源字符串的指针
 n 要复制的字符数

例 `#include <string.h>`
 `char *s1, *ret;`
 `const char *s2;`
 `size_t n;`
 `ret=strncpy(s1, s2, n);`

备 注 将 s2 指定的字符串的前 n 个字符（最多）复制到 s1 指定的存储区。当 s2 指定的字符串的字符数小于 n 个字符时，附加空字符，使其增加到 n 个字符；当 s2 指定的字符串的字符数大于 n 个字符时，复制到 s1 中的字符串的结束字符不为空字符。

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 指向被连接的字符串的指针
 s2 指向要连接的字符串的指针

例 `#include <string.h>`
 `char *s1, *ret;`
 `const char *s2;`
 `ret=strcat(s1, s2);`

备 注 将 s2 指定的字符串连接到 s1 指定的字符串的尾部。此时，s1 字符串结束的空字符被 s2 字符串的起始字符替换，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 指向被连接的字符串的指针
 s2 指向要连接的字符串的指针
 n 要连接的字符数

例 `#include <string.h>`
 `char *s1, *ret;`
 `const char *s2;`
 `size_t n;`
 `ret=strncat(s1, s2, n);`

备 注 将 s2 指定的字符串的前 n 个字符（最多）附加到 s1 指定的字符串的尾部。s1 字符串的结束空字符被 s2 字符串的起始字符替换。
 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 指定的存储区内容 > s2 指定的存储区内容时：正值
 s1 指定的存储区内容 == s2 指定的存储区内容时：0
 s1 指定的存储区内容 < s2 指定的存储区内容时：负值

参 数 s1 指向被比较的存储区的指针
 s2 指向要比较的存储区的指针
 n 要比较的存储区的字符数

例 `#include <string.h>`
 `const void *s1, *s2;`
 `size_t n;`
 `int ret;`
 `ret=memcmp(s1, s2, n);`

备 注 将 s1 指定的存储区和 s2 指定的存储区的前 n 个字符进行比较。此比较为实现性定义。

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` 指向被比较的字符串的指针
 `s2` 指向要比较的字符串的指针

例 `#include <string.h>`
 `const char *s1, *s2;`
 `int ret;`
 `ret=strcmp(s1,s2);`

备 注 将 `s1` 指定的字符串和 `s2` 指定的字符串的内容进行比较，将其结果设定为返回值。此比较为实现性定义。

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` 指向被比较的字符串的指针
 `s2` 指向要比较的字符串的指针
 `n` 要比较的字符数的最大值

例 `#include <string.h>`
 `const char *s1, *s2;`
 `size_t n;`
 `int ret;`
 `ret=strncmp(s1,s2,n);`

备 注 将 `s1` 指定的字符串和 `s2` 指定的字符串的前 `n` 个字符进行比较。此比较为实现性定义。

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 找到要检索的字符时 : 指向找到的字符的指针
 没有找到要检索的字符时 : NULL

参 数 s 指向要检索的存储区的指针
 c 要检索的字符
 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 注 在 s 指定的存储区的前 n 字符中，检索 c 指定的字符最初出现的位置，将指向该位置的指针作为返回值返回。

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 找到要检索的字符时 : 指向找到的字符的指针
 没有找到要检索的字符时 : NULL

参 数 s 指向要检索的字符串的指针
 c 要检索的字符

例

```
#include <string.h>
const char *s;
int c;
char *ret;
ret=strchr(s,c);
```

备 注 在 s 指定的字符串中，检索 c 指定的字符最初出现的位置，将指向该位置的指针作为返回值返回。
 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，取得指定字符集中的字符最初出现前的字符数。

头文件 <string.h>

返回值 在 s1 字符串中，s2 指定字符串中的字符最初出现前的字符数

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 注 从头开始调查 s1 指定的字符串，取得 s2 指定字符串中的字符最初出现前的字符数，将该字符数作为返回值返回。
 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 在指定的字符串中，检索指定字符集中的字符最初出现的位置。

头文件 <string.h>

返回值 找到要检索的结果时 : 指向找到的字符的指针
 没有找到要检索的结果时 : NULL

参 数 s1 指向被检索的字符串的指针
 s2 指向表示在 s1 内要检索的字符的字符串指针

例

```
#include <string.h>
const char *s1, *s2;
char *ret;
ret=strpbrk(s1, s2);
```

备 注 在 s1 指定的字符串中，检索 s2 指定字符串中的某个字符最初出现的位置，将指向该位置的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 找到要检索的结果时：指向找到的字符的指针
 没有找到要检索的结果时：NULL

参 数 `s` 指向被检索的字符串的指针
 `c` 要检索的字符

例 `#include <string.h>`
 `const char *s;`
 `int c;`
 `char *ret;`
 `ret=strrchr(s,c);`

备 注 在 `s` 指定的字符串中，检索 `c` 指定的字符最后出现的位置，将指向该位置的指针作为返回值返回。
 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，取得指定字符集中的字符从头连续出现的字符数。

头文件 `<string.h>`

返回值 在 `s1` 字符串中，`s2` 指定字符中的字符从头连续出现的字符数

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 注 从头开始调查 `s1` 指定的字符串，取得 `s2` 指定字符串中的字符从头出现的字符数，将该字符数作为返回值返回。

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，检索指定字符串最初出现的位置。

头文件 <string.h>

返回值 找到要检索的结果时 : 指向找到的字符的指针
 没有找到要检索的结果时 : NULL

参 数 s1 指向被检索的字符串的指针
 s2 指向要检索的字符串的指针

例

```
#include <string.h>
const char *s1, *s2;
char *ret;
ret=strstr(s1,s2);
```

备 注 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，将指向该位置的指针作为返回值返回。

字句的拆分

char *strtok(char *s1, const char *s2)

说 明 将指定的字符串拆成若干个字句。

头文件 <string.h>

返回值 拆成字句时：指向已拆成的字句的起始位置的指针
没有拆成字句时：NULL

参 数 s1 指向被拆成若干个字句的字符串的指针
 s2 指向用于拆分字符串的字符串指针

例

```
#include <string.h>
char *s1, *ret;
const char *s2;
ret=strtok(s1,s2);
```

备 注 为了拆分字符串而连续调用 **strtok** 函数。

(a) 最初调用时
用 **s2** 指定的字符串中的字符，将 **s1** 指定的字符串从头开始拆成字句。如果拆成字句，就将指向该字句头的指针作为返回值返回；否则，返回 **NULL**。

(b) 第 2 次以后的调用时
用 **s2** 指定的字符串中的字符，从以前被拆成字句的下一个字符开始，将字符串拆成字句。如果拆成字句，就将指向该字句头的指针作为返回值返回；否则，返回 **NULL**。

第 2 次以后的调用时，给第 1 个参数指定 **NULL**，每次调用时的 **s2** 指定的字符串可不同。在被拆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】

此例是通过 **strtok** 函数将字符串 "a@b, @c/@d" 拆成 a、b、c、d 字句的程序。

在第 2 行，给字符串 **s1** 设定初始值的字符串 "a@b, @c/@d"。

在第 5 行，为了将 “@” 作为分割字符拆成字句，调用 **strtok** 函数。其结果，得到了作为返回值的指向字符 '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拆成字句，给第 1 个参数指定 **NULL**，调用 **strtok** 函数。结果，字符串依次被拆成 "b"、"c"、"d"。

字符的重复

void *memset(void *s, int c, size_t n)

说 明 从指定的存储区的头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 要设定的字符
 n 要设定的字符数

例

```
#include <string.h>
void *s, *ret;
int c;
size_t n;
ret=memset(s,c,n);
```

备 注 给 s 指定的存储区设定 n 个字符 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 注 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性定义。
 如果修正作为返回值返回的错误信息，就不保证运行。

字符串的字符数

size_t strlen(const char *s)

说 明	计算字符串的字符数。
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	<pre>#include <string.h> const char *s; size_t ret; ret=strlen(s);</pre>
备 注	表示 s 指定的字符串结束的空字符不为字符串的字符数的计算范围。

存储区的移动

void *memmove(void *s1, const void *s2, size_t n)

说 明	将指定容量的源存储区的内容复制到目标存储区。 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先进行复制，所以也能正常复制。
头文件	<string.h>
返回值	s1 的值
参 数	s1 指向目标存储区的指针 s2 指向源存储区的指针 n 要复制的字符数
例	<pre>#include <string.h> void *ret, *s1; const void *s2; size_t n; ret=memmove(s1, s2, n);</pre>

10.4.2 EC++ 类库

(1) 库的概要

以下说明在 C++ 程序中能标准利用的 EC++ 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 10.48 所示。

表 10.48 类库的种类和标准 include 文件的对应

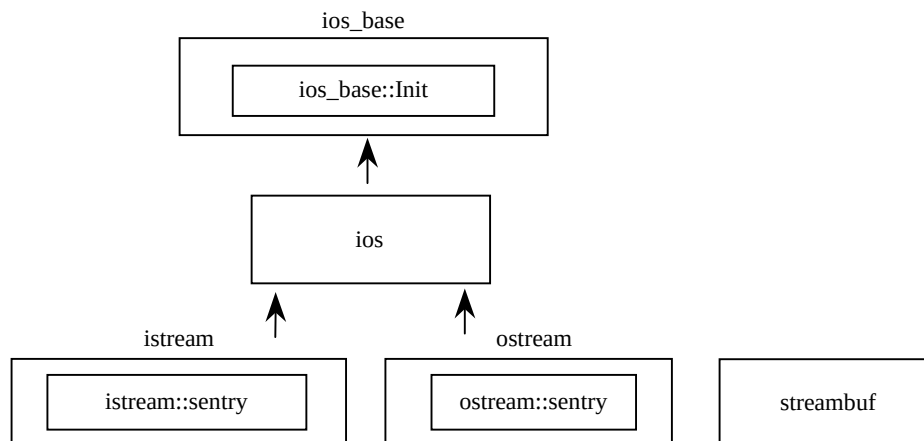
	库的种类	内容	标准 include 文件
1	流输入 / 输出类库	是进行输入 / 输出操作的库。	<ios>、<streambuf>、 <istream>、<ostream>、 <iostream>、<iomanip>
2	存储器操作库	是进行存储器确保和释放的库。	<new>
3	复数计算类库	是进行复数数据运算的库。	<complex>
4	字符串操作类库	是进行字符串操作的库。	<string>

(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称如下所示：

种类	定义名	说明
型	streamoff	是用 long 型定义的型。
	streamsize	是用 size_t 型定义的型。
	int_type	是用 int 型定义的型。
	pos_type	是用 long 型定义的型。
	off_type	是用 long 型定义的型。

(a) ios_base::Init 类

种类	定义名	说明
变量	init_cnt	是对流输入 / 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()	是构造函数。
	~Init()	是析构函数。

ios_base::Init::Init()

是类 Init 的构造函数。
使 init_cnt 递增。

ios_base::Init::~~Init()

是类 Init 的析构函数。
使 init_cnt 递减。

(b) ios_base 类

种类	定义名	说明
型	fmtflags	是表示格式控制信息的型。
	iostate	是表示流缓冲区的输入 / 输出状态的型。
	openmode	是表示文件打开模式的型。
	seekdir	是表示流缓冲区的查找状态的型。
变量	fmtfl	是格式标志。
	wide	是字段宽度。
	prec	是输出时的精度（小数的位数）。
	fillch	是填充字符。
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()	是构造函数。
	~ios_base()	是析构函数。
	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：

```

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：

```

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：

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。

决定继续输入或输出的流内的位置。

seekdir 的各位屏蔽的定义如下：

```

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()

是类 ios_base 的构造函数。

调用 Init::Init()。

ios_base::~~ios_base()

是类 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	是指向 streambuf 目标的指针。
	tiestr	是指向 ostream 目标的指针。
	state	是指向 streambuf 的状态标志。
函数	ios()	是构造函数。
	ios(streambuf* sbptr)	是构造函数。
	void init(streambuf* sbptr)	进行初始设定。
	virtual ~ios()	是析构函数。
	operator void*() const	判断有无错误 (!state&(badbit failbit))。
	bool operator!() const	判断有无错误 (state&(badbit failbit))。
	iosstate rdstate() const	参照状态标志 (state)。
	void clear(iosstate st = goodbit)	清除指定状态 (st) 以外的状态标志 (state)。
	void setstate(iosstate 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结尾 (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()

是类 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)

是类 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()

是类 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

iosstate ios::rdstate() const

参照状态标志（state）。

返回值为状态标志（state）。

void ios::clear(iosstate st = goodbit)

清除指定状态（st）以外的状态标志（state）。

当指向 streambuf 目标的指针（sb）为 0 时，将 badbit 设定为状态标志（state）。

void ios::setstate(iosstate 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结尾（state&eofbit）。

返回值如下：

为输入流的结尾时 : true
不为输入流的结尾时 : 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& noshowbase(ios_base& str)	清除基数显示前缀模式。
	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 + 符号生成模式。
	ios_base& noshowpos(ios_base& str)	清除 + 符号生成模式。
	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)	设定为内部填充模式。
	ios_base& left(ios_base& str)	设定为左侧填充模式。
	ios_base& right(ios_base& str)	设定为右侧填充模式。
	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)	设定为定点模式。
	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，在行头附加 0x；10 进制数时，直接输出；8 进制数时，在行头附加 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。

当没有指定精度时，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 + 符号生成输出模式（给正数附加 + 符号）。

返回值为 str。

ios_base& noshowpos(ios_base& str)

清除 + 符号生成输出模式。

返回值为 str。

ios_base& skipws(ios_base& str)

设定为空格跳读输入模式（跳读连续的空格）。

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)

设定为大写字母转换输出模式。

16 进制的基数表示为大写字母 0X，数值本身也为大写字母。浮点的指数表示也为大写字母 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)`

设定为定点输出模式。

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	是指向缓冲区有效数据长度的指针。
	B_beg_ptr	是指向缓冲区基址指针的指针。
	_B_len_ptr	是指向缓冲区长度的指针。
	B_next_ptr	是指向缓冲区的下一个读位置的指针。
	B_end_ptr	是指向缓冲区结束位置的指针。
	B_beg_pptr	是指向控制缓冲区的起始位置的指针。
	B_next_pptr	是指向缓冲区的下一个读位置的指针。
	C_flg_ptr	是指向文件输入 / 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()	是构造函数。
	virtual ~streambuf()	是析构函数。
	streambuf* pubsetbuf(char* s, streamsize n)	确保流输入 / 输出的缓冲区。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位置。
	int_type sgetc()	读一个字符。
	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)	返回读位置。
	int sungetc()	返回读位置。
	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。

种类	定义名	说明
函数	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)	给输入流的各指针赋值。
	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	对派生的各类，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	返回 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 此类没有定义处理。

streambuf::streambuf()

是构造函数。

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()

是析构函数。

streambuf* streambuf::pubsetbuf(char* s, streamsize n)

确保流输入 / 输出的缓冲区。

此函数调用 `setbuf(s,n)`。

返回值为 `*this`。

**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。

只将当前流指针移动 `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 : 读取的字符

为 `eof` 时 : `eof`

int_type streambuf::sbumpc()

读一个字符，并将指针设定到下一个位置。

返回值如下：

读位置有效时 : 读取的字符

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 : 读取的字符

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到 *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返回数据与 *c* 相同时，就返回读位置。
返回值如下：
能返回时 : *c* 的值
不能返回时 : *eof*

int streambuf::sungetc()

读位置正常时，返回读位置。
返回值如下：
能返回时 : 返回的值
不能返回时 : *eof*

int streambuf::putc(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。
当缓冲区小于 *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。

返回值为头指针。

```
char* streambuf::pptr() const
```

求输出流的 next 指针。

返回值为 next 指针。

```
char* streambuf::eptr() 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)
```

对从 streambuf 派生的各类，执行个别定义的运算。

返回值为 *this。此类没有定义处理。

```
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。此类没有定义处理。

```
virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode =
    (ios_base::openmode)(ios_base::in | ios_base::out))
```

更改流位置。

返回值为 -1。此类没有定义处理。

virtual int streambuf::sync()

转储清除输出流。

返回值为 0。此类没有定义处理。

virtual int streambuf::showmanyc()

求输入流的有效字符数。

返回值为 0。此类没有定义处理。

virtual streamsize streambuf::xsgetn(char* s, streamsize n)

将 *n* 个字符设定到 *s* 指向的存储区。

当缓冲区小于 *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*。此类没有定义处理。

virtual int_type streambuf::uflow()

读 *next* 指针的一个字符。

返回值为 *eof*。此类没有定义处理。

virtual int_type streambuf::pbackfail(int_type c = eof)

返回 *c* 表示的字符。

返回值为 *eof*。此类没有定义处理。

virtual streamsize streambuf::xsputn(const char* s, streamsize n)

插入 *s* 指向的 *n* 个字符。

当缓冲区小于 *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*。此类没有定义处理。

(f) `istream::sentry` 类

种类	定义名	说明
变量	<code>ok_</code>	是否为可输入状态。
函数	<code>sentry(istream& is, bool noskipws = false)</code>	是构造函数。
	<code>~sentry()</code>	是析构函数。
	<code>operator bool()</code>	参照 <code>ok_</code> 。

`istream::sentry::sentry(istream& is, bool noskipws = _false)`

是内部类 `sentry` 的构造函数。

当 `good()` 不为 0 时，能进行带格式或者不带格式的输入。

当 `tie()` 不为 0 时，转储清除有关的输出流。

`istream::sentry::~sentry()`

是内部类 `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	是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)	是构造函数。
	virtual ~istream()	是析构函数。
	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>>(long long& n)	
	istream& operator>>(unsigned long 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到 '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。如果在中途检测到字符 '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。如果在中途检测到字符 '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。如果在中途检测到字符 '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)	只将当前流指针移动 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)

是类 istream 的构造函数。

调用 ios::init(sb)。

进行 chcount=0 的设定。

virtual istream::~~istream()

是类 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>>(long long& n)
istream& istream::operator>>(unsigned long 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。

当没有抽出的字符时，调用 setstate(failbit)。

返回值为 *this。

```
streamsize istream::gcount() const
```

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：

当能抽出时：抽出的字符

当不能抽出时：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。当抽出的字符为 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。

当 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到 'delim'，就结束。

当 $ok\_==false$ 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。

当 $ok\_==false$ 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。

如果在中途检测到字符 'delim'，就结束。

当 $ok\_==false$ 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。

当 $ok\_==false$ 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。

如果在中途检测到字符 'delim'，就结束。

当 $ok\_==false$ 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。

如果在中途检测到字符 '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：

当 $ok\_==false$ 时 : streambuf::eof

当 $ok\_!=false$ 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)

当 `ok_!=false` 时，抽出长度为 `n` 的字符串，保存到 `s` 指向的存储区。

当抽出的字符数与 `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。当返回的字符为 `streambuf::eof` 时，设定 `badbit`。

返回值为 `*this`。

istream& istream::unget()

将输入流的指针返回一个位置。

当抽出的字符为 `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：

当无输入流时：`streambuf::eof`

当有输入流时：`0`

pos_type istream::tellg()

调查输入流的位置。

此函数调用 `streambuf::pubseekoff(0,cur,in)`。

返回值如下：

从流的起始位置的偏移量

但是，在输入处理发生错误时为 `-1`

istream& istream::seekg(pos_type pos)

只将当前流指针移动 `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)	跳读空格类字符。

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, signed 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、或者输入流中出现 streambuf::eof、或者下一个能输入的字符 c 为 isspace(c)=1 时，结束处理。当保存字符数为 0 时，设定 failbit。

返回值为 in。

istream& operator>>(istream& in, char& c)

istream& operator>>(istream& in, signed char& c)

istream& operator>>(istream& in, unsigned char& c)

抽出字符，保存到 c。

当没有抽出输入时，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)	是构造函数。
	~sentry()	是析构函数。
	operator bool()	参照 ok_。

ostream::sentry::sentry(ostream& os)

是内部类 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()

是内部类 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)	是构造函数。
	virtual ~ostream()	是析构函数。
	ostream& operator<<(bool n)	将 n 插入到输出流。
	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<<(long long n)	
	ostream& operator<<(unsigned long long n)	
	ostream& operator<<(float n)	
	ostream& operator<<(double n)	
	ostream& operator<<(long double n)	
	ostream& operator<<(void* n)	
	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)	将 s 的 n 个字符插入到输出流。
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	转储清除输出流。此函数调用 streambuf::pubsync()。
	pos_type tellp()	求当前的写位置。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，将当前流指针移动 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)

是构造函数。

调用 ios(sbptr)。

virtual ostream::~~ostream()

是析构函数。

```
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<<(long long n)
ostream& ostream::operator<<(unsigned long long n)
ostream& ostream::operator<<(float n)
ostream& ostream::operator<<(double n)
ostream& ostream::operator<<(long double n)
ostream& ostream::operator<<(void* n)
```

当 `sentry::ok_==true` 时，将 `n` 插入到输出流。

当 `sentry::ok_==false` 时，设定 `failbit`。

返回值为 `*this`。

```
ostream& ostream::operator<<(streambuf* sbptr)
```

当 `sentry::ok_==true` 时，将 `sbptr` 的输出行插入到输出流。

当 `sentry::ok_==false` 时，设定 `failbit`。

返回值为 `*this`。

```
ostream& ostream::put(char c)
```

当 `sentry::ok_==true` 并且 `rdbuf()->sputc(c)!=streambuf::eof` 时，将 `c` 插入到输出流。

在上述以外的情况时，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)
```

当 `sentry::ok_==true` 并且 `rdbuf()->sputn(s, n)==n` 时，将 `s` 的 `n` 个字符插入到输出流。

在上述以外的情况时，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()
```

求当前的写位置。

此函数调用 `streambuf::pubseekoff(0,cur,out)`。

返回值如下：

当前流的位置

但是，在处理中发生错误时为 `-1`

ostream& ostream::seekp(pos_type pos)

当没有错误时，求从流的起始位置到当前位置的偏移量。

另外，只将当前流指针移动 pos。

此函数调用 streambuf::pubseekpos(pos)。

返回值为 *this。

ostream& ostream::seekp(off_type off, seekdir dir)

当没有错误时，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)	插入换行字符，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)

给输出流插入换行字符。

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 signed 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 signed char* s)

ostream& operator<<(ostream& os, const unsigned char* s)

当 sentry::ok_==true 并且没有错误时，将 s 插入到输出流。

在上述以外的情况时，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)

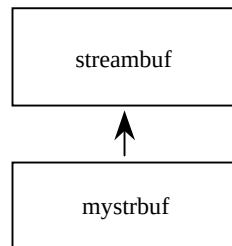
设定字段宽度（wide）。

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

在初始化 `istream`、`ostream` 的目标时，只要使用指向 `mystrbuf` 类（而不是 `streambuf`）目标的指针，就能使输入 / 输出流成为可使用状态。

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	<code>_file_Ptr</code>	是文件指针。
函数	<code>mystrbuf()</code>	是构造函数。进行 <code>streambuf</code> 缓冲区的初始化。
	<code>mystrbuf(void* ptr)</code>	(同上)
	<code>virtual ~mystrbuf()</code>	是析构函数。
	<code>void* myfptr() const</code>	返回指向 <code>FILE</code> 型结构体的指针。
	<code>mystrbuf* open(const char* filename, int mode)</code>	指定文件名和模式，打开文件。
	<code>mystrbuf* close()</code>	关闭文件。
	<code>virtual streambuf* setbuf(char* s, streamsize n)</code>	确保流输入 / 输出的缓冲区。
	<code>virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode)(ios_base::in ios_base::out))</code>	更改流指针的位置。
	<code>virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode)(ios_base::in ios_base::out))</code>	更改流指针的位置。
	<code>virtual int sync()</code>	转储清除流。
	<code>virtual int showmanyc()</code>	返回输入流的有效字符数。
	<code>virtual int_type underflow()</code>	不移动流位置，读一个字符。
	<code>virtual int_type pbackfail(int_type c = streambuf::eof)</code>	返回 <code>c</code> 表示的字符。
	<code>virtual int_type overflow(int_type c = streambuf::eof)</code>	插入 <code>c</code> 表示的字符。
	<code>void _Init(_f_type* fp)</code>	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++ Library." << 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和释放的函数。

通过给 `_ec2p_new_handler` 变量设定异常处理函数的地址，能在存储器确保失败时执行异常处理。
`_ec2p_new_handler` 为 `static` 变量，初始值为 `NULL`。如果使用此处理程序，就会失去可重入性。

异常处理函数所需的运行：

- 建立和返回可分配的区域。
- 没有规定不能建立时的运行。

种类	定义名	说明
型	<code>new_handler</code>	是指向返回 <code>void</code> 型的函数的指针型。
变量	<code>_ec2p_new_handler</code>	是指向异常处理函数的指针。
函数	<code>void* operator new(size_t size)</code>	确保 <code>size</code> 字节的区域。
	<code>void* operator new[](size_t size)</code>	确保 <code>size</code> 字节的数组区。
	<code>void* operator new(size_t size, void* ptr)</code>	将 <code>ptr</code> 指向的区域分配为存储区。
	<code>void* operator new[](size_t size, void* ptr)</code>	将 <code>ptr</code> 指向的区域分配为数组区。
	<code>void operator delete(void* ptr)</code>	释放区域。
	<code>void operator delete[](void* ptr)</code>	释放数组区。
	<code>new_handler set_new_handler(new_handler new_P)</code>	给 <code>_ec2p_new_handler</code> 设定异常处理函数地址 (<code>new_P</code>)。

`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` 指向的区域分配为存储区。

返回值为 `ptr`。

`void* operator new[ ](size_t size, void* ptr)`

将 `ptr` 指向的区域分配为数组区。

返回值为 `ptr`。

`void operator delete(void* ptr)`

释放 `ptr` 指向的存储区。当 `ptr` 为 `NULL` 时，不进行任何操作。

`void operator delete[ ](void* ptr)`

释放 `ptr` 指向的数组区。当 `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	<code>value_type</code>	是 <code>float</code> 型。
变量	<code>_re</code>	定义 <code>float</code> 精度的实部。
	<code>_im</code>	定义 <code>float</code> 精度的虚部。
函数	<code>float_complex(float re = 0.0f, float im = 0.0f)</code>	是构造函数。
	<code>float_complex(const double_complex& rhs)</code>	
	<code>float real() const</code>	求实部 (<code>_re</code>)。
	<code>float imag() const</code>	求虚部 (<code>_im</code>)。
	<code>float_complex& operator=(float rhs)</code>	将 <code>rhs</code> 复制到实部，虚部设定为 <code>0.0f</code> 。
	<code>float_complex& operator+=(float rhs)</code>	实部加上 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator-=(float rhs)</code>	实部减去 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator*=(float rhs)</code>	乘 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator/=(float rhs)</code>	除 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator=(const float_complex& rhs)</code>	复制 <code>rhs</code> 。
	<code>float_complex& operator+=(const float_complex& rhs)</code>	加上 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator-=(const float_complex& rhs)</code>	减去 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator*=(const float_complex& rhs)</code>	乘 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>float_complex& operator/=(const float_complex& rhs)</code>	除 <code>rhs</code> ，结果保存到 <code>*this</code> 。

`float_complex::float_complex(float re = 0.0f, float im = 0.0f)`

是类 `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)`

是类 `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`) 设定为 `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 (`_re`)，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)	lhs 除 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)	
	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。

种类	定义名	说明
函数	bool operator==(const float_complex& lhs, const float& rhs)	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	bool operator==(const float& lhs, const float_complex& 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)	
	istream& operator>>(	
	istream& is, float_complex& x)	以 u、(u) 或者 (u,v) (u: 实部, v: 虚部) 格式输入 x
	ostream& operator<<(	以 u、(u) 或者 (u,v) (u: 实部, v: 虚部) 格式输出 x
	ostream& os, float_complex& x)	
	float real(const float_complex& x)	求实部。
	float imag(const float_complex& x)	求虚部。
	float abs(const float_complex& x)	求绝对值。
	float arg(const float_complex& x)	求相位角。
	float norm(const float_complex& x)	求平方的绝对值。
	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 乘方。
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	

种类	定义名	说明
函数	<code>float_complex pow(const float& x, const float_complex& y)</code>	求 x 的 y 乘方。
	<code>float_complex sin(const float_complex& x)</code>	求复数正弦。
	<code>float_complex sinh(const float_complex& x)</code>	求复数双曲正弦。
	<code>float_complex sqrt(const float_complex& x)</code>	求右半空间范围内的平方根。
	<code>float_complex tan(const float_complex& x)</code>	求复数正切。
	<code>float_complex tanh(const float_complex& x)</code>	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)`

lhs 除 rhs，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，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，虚部假设为 float 型的 0.0f。

返回值为 $\text{lhs.real() != rhs.real() || lhs.imag() != rhs.imag()}$ 。

istream& operator>>(istream& is, float_complex& x)

以 u、(u) 或者 (u,v) (u 为实部，v 为虚部) 格式输入 x。输入值转换为 float_complex。

如果不以 u、(u)、(u,v) 格式输入数据，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 $(|x.\text{real()}|^2 + |x.\text{imag()}|^2)^{1/2}$ 。

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)

求平方的绝对值。

返回值为 $|x.\text{real()}|^2 + |x.\text{imag()}|^2$ 。

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 的复数的 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()))`。

返回值为 $\sinh(x)/\cosh(x)$ 。

(c) `double_complex` 类

种类	定义名	说明
型	<code>value_type</code>	是 <code>double</code> 型。
变量	<code>_re</code>	定义 <code>double</code> 精度的实部。
	<code>_im</code>	定义 <code>double</code> 精度的虚部。
函数	<code>double_complex(</code> <code>double re = 0.0,</code> <code>double im = 0.0)</code>	是构造函数。
	<code>double_complex(const float_complex&)</code>	
	<code>double real() const</code>	求实部。
	<code>double imag() const</code>	求虚部。
	<code>double_complex& operator=(double rhs)</code>	将 <code>rhs</code> 复制到实部，虚部设定为 0.0。
	<code>double_complex& operator+=(double rhs)</code>	实部加上 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator-=(double rhs)</code>	实部减去 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator*=(double rhs)</code>	乘 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator/=(double rhs)</code>	除 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator=(</code> <code>const double_complex& rhs)</code>	复制 <code>rhs</code> 。
	<code>double_complex& operator+=(</code> <code>const double_complex& rhs)</code>	加上 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator-=(</code> <code>const double_complex& rhs)</code>	减去 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator*=(</code> <code>const double_complex& rhs)</code>	乘 <code>rhs</code> ，结果保存到 <code>*this</code> 。
	<code>double_complex& operator/=(</code> <code>const double_complex& rhs)</code>	除 <code>rhs</code> ，结果保存到 <code>*this</code> 。

`double_complex::double_complex(double re = 0.0, double im = 0.0)`

是类 `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&)`

是类 `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`)，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)`

除 rhs，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)`

除 rhs，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)	lhs 除 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)	
	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。

种类	定义名	说明
函数	<pre>bool operator==(const double_complex& lhs, const double& rhs)</pre>	将 lhs 和 rhs 的实部和实部、虚部和虚部进行比较。
	<pre>bool operator==(const double& lhs, const double_complex& rhs)</pre>	
	<pre>bool operator!=(const double_complex& lhs, const double_complex& rhs)</pre>	
	<pre>bool operator!=(const double_complex& lhs, const double& rhs)</pre>	
	<pre>bool operator!=(const double& lhs, const double_complex& rhs)</pre>	
	<pre>istream& operator>>(istream& is, double_complex& x)</pre>	以 u、(u) 或者 (u,v) (u: 实部、v: 虚部) 格式输入 x。
	<pre>ostream& operator<<(ostream& os, const double_complex& x)</pre>	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_complex& x)	求绝对值。
	double arg(const double_complex& x)	求相位角。
	double norm(const double_complex& x)	求平方的绝对值。
	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)	求 x 的 y 乘方。

种类	定义名	说明
函数	double _complex pow(const double _complex& x, const double & y)	求 x 的 y 乘方。
	double _complex pow(const double _complex& x, const double _complex& y)	
	double _complex pow(const double & x, const double _complex& y)	
	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)
```

lhs 除 rhs，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，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，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)
```

以 u、(u) 或者 (u,v) (u 为实部、v 为虚部) 格式输入复数 x。输入值转换为 double_complex。

如果不以 u、(u)、(u,v) 格式输入数据，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 $(|x.real()|^2 + |x.imag()|^2)^{1/2}$ 。

```
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)`

求平方的绝对值。

返回值为 $|x.\text{real}()|^2 + |x.\text{imag}()|^2$ 。

`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)`

求自然对数（以 e 为底）。

返回值为 `double_complex(log(abs(x)), arg(x))`。

`double_complex log10(const double_complex& x)`

求常用对数（以 10 为底）。

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 乘方。

`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()), 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	<code>iterator</code>	是 <code>char*</code> 型。
	<code>const_iterator</code>	是 <code>const char*</code> 型。
常数	<code>npos</code>	是字符串的最大长度（ <code>UINT_MAX</code> 字符）。
变量	<code>s_ptr</code>	是指向目标保存字符串的区域的指针。
	<code>s_len</code>	是目标保存的字符串长度。
	<code>s_res</code>	是确保目标保存字符串的区域的容量。
函数	<code>string(void)</code>	是构造函数。
	<code>string::string(</code> <code>const string& str,</code> <code>size_t pos = 0,</code> <code>size_t n = npos)</code>	
	<code>string::string(const char* str, size_t n)</code>	
	<code>string::string(const char* str)</code>	
	<code>string::string(size_t n, char c)</code>	
	<code>~string()</code>	是析构函数。
	<code>string& operator=(const string& str)</code>	赋值 <code>str</code> 。
	<code>string& operator=(const char* str)</code>	
	<code>string& operator=(char c)</code>	赋值 <code>c</code> 。
	<code>iterator begin()</code>	求字符串的头指针。
	<code>const_iterator begin() const</code>	
	<code>iterator end()</code>	求字符串的尾指针。
	<code>const_iterator end() const</code>	
	<code>size_t size() const</code>	求被保存的字符串的长度。
	<code>size_t length() const</code>	
	<code>size_t max_size() const</code>	求被确保的区域的容量。
	<code>void resize(size_t n, char c)</code>	将可保存的字符串的字符数更改为 <code>n</code> 。
	<code>void resize(size_t n)</code>	将可保存的字符串的字符数更改为 <code>n</code> 。
	<code>size_t capacity() const</code>	求被确保的区域的容量。
	<code>void reserve(size_t res_arg = 0)</code>	重新分配区域。
	<code>void clear()</code>	清除被保存的字符串。
	<code>bool empty() const</code>	检查被保存的字符串的字符数是否为 0。
	<code>const char& operator[](size_t pos) const</code>	参照 <code>s_ptr[pos]</code> 。
	<code>char& operator[](size_t pos)</code>	
	<code>const char& at(size_t pos) const</code>	
	<code>char& at(size_t pos)</code>	

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)	
	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)	将从 str 字符串的位置 pos2 开始的 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())	在 p 指向的字符串前插入字符 c。
	void insert(iterator p, size_t n, char c)	在 p 指向的字符前插入 n 个字符 c。
	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 str 的位置 pos2 开始的 n2 个字符的字符串替换位置 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	参照指向字符串的保存区域的指针。
	const char* data() const	

种类	定义名	说明
函数	size_t find(const string& str, size_t pos = 0) const	检索在位置 pos 以后 str 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 str 的 n 个字符的字符串最初出现的位置。
	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 str 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 str 的 n 个字符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。

种类	定义名	说明
函数	size_t find_first_not_of(const string& str, size_t pos = 0) const	检索在位置 pos 以后非 str 中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 以后非 str 的前 n 个字符中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 以后非字符 c 的字符最初出现的位置。
	size_t find_last_not_of(const string& str, size_t pos = npos) const	检索在位置 pos 以前非 str 中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 以前非 str 的前 n 个字符中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 以前非字符 c 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	将位置 pos1 开始的 n1 个字符的字符串和 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。其中，s_len < n 时，s_len = n。

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()

是类 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的字符数更改为 n 。

$n \leq \text{size}()$ 时，用长度为 n 的原字符串进行替换。

$n > \text{size}()$ 时，用在原字符串的后面填入了 c （使其长度为 n ）的字符串进行替换。

需要 $n \leq \text{max\_size}()$ 。

$n > \text{max\_size}()$ 时，作为 $n = \text{max\_size}()$ 计算。

void string::resize(size_t n)

将目标能保存的字符串的字符数更改为 n 。

$n \leq \text{size}()$ 时，用长度为 n 的原字符串进行替换。

需要 $n \leq \text{max\_size}()$ 。

size_t string::capacity() const

求被确保的区域的容量。

返回值为确保的区域的容量。

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的字符数是否为 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)`

将从 `str` 字符串的 `pos2` 位置开始的 `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` 指向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` 指向的字符串前插入 `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	: <code>end()</code>

iterator string::erase(iterator first, iterator last)

删除范围 `[first, last]` 中的字符。

返回值如下：

有 <code>last</code> 的下一个 iterator 时	: <code>last</code> 的下一个 iterator
无 <code>last</code> 的下一个 iterator 时	: <code>end()</code>

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 `str` 的位置 `pos2` 开始的 `n2` 个字符的字符串替换位置 `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`

参照指向字符串的保存区域的指针。

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 `str` 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 `str` 的 `n` 个字符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 `str` 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` 以前字符串 `str` 的 `n` 个字符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` 以后非 `str` 中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` 以后非 `str` 的前 `n` 个字符中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` 以后非字符 `c` 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` 以前非 `str` 中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` 以前非 `str` 的前 `n` 个字符中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` 以前非字符 `c` 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
```

将位置 `pos1` 开始的 `n1` 个字符的字符串和 `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)	将 lhs 的字符串和 rhs 的字符串进行交换。
	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。如果在中途检测到字符 'delim'，就结束输入。
	istream& getline(istream& is, string& str)	从 is 抽出字符串，附加到 str。如果在中途检测到换行字符，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。
返回值为保存连接后的字符串的目标。

```
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)

将 lhs 字符串和 rhs 字符串进行交换。

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。

如果在中途检测到字符 '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。

如果在中途检测到换行字符，就结束输入。

返回值为 is。

10.4.3 可重入库

用标准库生成工具指定 `reent` 选项建立的库除 `rand`、`srand` 函数以外，都可重入。

不指定 `reent` 选项时的可重入库一览表如表 10.49 所示。在表中，用△表示的函数设定 `errno` 变量，如果在程序中不参照 `errno`，就可重入。

可重入栏 ○：可重入 ×：不可重入 △：设定 `errno`

表 10.49 可重入库一览表

标准 include 文件	函数名	可重入	标准 include 文件	函数名	可重入
<code>stddef.h</code>	<code>offsetof</code>	○	<code>mathf.h</code>	<code>atan2f</code>	△
<code>assert.h</code>	<code>assert</code>	×		<code>cosf</code>	△
<code>ctype.h</code>	<code>isalnum</code>	○		<code>sinf</code>	△
	<code>isalpha</code>	○		<code>tanf</code>	△
	<code>iscntrl</code>	○		<code>coshf</code>	△
	<code>isdigit</code>	○		<code>sinhf</code>	△
	<code>isgraph</code>	○		<code>tanhf</code>	△
	<code>islower</code>	○		<code>expf</code>	△
	<code>isprint</code>	○		<code>frexpf</code>	△
	<code>ispunct</code>	○		<code>ldexpf</code>	△
	<code>isspace</code>	○		<code>logf</code>	△
	<code>isupper</code>	○		<code>log10f</code>	△
	<code>isxdigit</code>	○		<code>modff</code>	△
	<code>tolower</code>	○		<code>powf</code>	△
	<code>toupper</code>	○		<code>sqrtf</code>	△
<code>math.h</code>	<code>acos</code>	△		<code>ceilf</code>	△
	<code>asin</code>	△		<code>fabsf</code>	△
	<code>atan</code>	△		<code>floorf</code>	△
	<code>atan2</code>	△		<code>fmodf</code>	△
	<code>cos</code>	△	<code>setjmp.h</code>	<code>setjmp</code>	○
	<code>sin</code>	△		<code>longjmp</code>	○
	<code>tan</code>	△	<code>stdarg.h</code>	<code>va_start</code>	○
	<code>cosh</code>	△		<code>va_arg</code>	○
	<code>sinh</code>	△		<code>va_end</code>	○
<code>math.h</code>	<code>tanh</code>	△	<code>stdio.h</code>	<code>fclose</code>	×
	<code>exp</code>	△		<code>fflush</code>	×
	<code>frexp</code>	△		<code>fopen</code>	×
	<code>ldexp</code>	△		<code>freopen</code>	×
	<code>log</code>	△		<code>setbuf</code>	×
	<code>log10</code>	△		<code>setvbuf</code>	×
	<code>modf</code>	△		<code>fprintf</code>	×
	<code>pow</code>	△		<code>fscanf</code>	×
	<code>sqrt</code>	△		<code>printf</code>	×
	<code>ceil</code>	△		<code>scanf</code>	×
	<code>fabs</code>	△		<code>sprintf</code>	△
	<code>floor</code>	△		<code>sscanf</code>	△
	<code>fmod</code>	△		<code>vfprintf</code>	×
<code>mathf.h</code>	<code>acosf</code>	△		<code>vprintf</code>	×
	<code>asinf</code>	△		<code>vsprintf</code>	△
	<code>atanf</code>	△		<code>fgetc</code>	×

标准 include 文件	函数名	可重入	标准 include 文件	函数名	可重入
stdio.h	fgets	×	stdlib.h	malloc__X	×
	fputc	×		realloc__X	×
	fputs	×		calloc__X	×
	getc	×		free__Y	×
	getchar	×		malloc__Y	×
	gets	×		realloc__Y	×
	putc	×		calloc__Y	×
	putchar	×		bsearch	○
	puts	×		qsort	○
	ungetc	×		abs	○
	fread	×		div	△
	fwrite	×		labs	○
	fseek	×		llabs	○
	ftell	×		ldiv	△
	rewind	×		lldiv	△
	clearerr	×	string.h	memcpy	○
	feof	×		memcpy__X__X	○
	ferror	×		memcpy__X__Y	○
	perror	×		memcpy__Y__X	○
stdlib.h	atof	△		memcpy__Y__Y	○
	atoi	△		strcpy	○
	atol	△		strncpy	○
	atoll	△		strcat	○
	atolfixed	△		strncat	○
	atolaccum	△		memcmp	○
	strtod	△		strcmp	○
	strtol	△		strncmp	○
	strtoul	△		memchr	○
	strtoll	△		strchr	○
	strtoull	△		strcspn	○
	strtolfixed	△		strpbrk	○
	strtolaccum	△		strrchr	○
	rand	×		strspn	○
	srand	×		strstr	○
	calloc	×		strtok	×
	free	×		memset	○
	malloc	×		strerror	○
	realloc	×		strlen	○
	free__X	×		memmove	○

10.4.4 不支持的库

在 C 语言规格定义的库中，本编译程序不支持的库如表 10.50 所示。

表 10.50 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	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。

10.4.5 DSP 库

(1) 概要

本节说明 SH2-DSP、SH3-DSP 和 SH4AL-DSP（以后合并称为 SH-DSP）中能使用的数字信号处理（DSP）库。本库含有标准的 DSP 函数，能通过单独或者连续使用此函数，进行 DSP 运算。

本库含有以下函数：

- 高速傅立叶转换
- 窗函数
- 滤波器
- 卷积和相互关系
- 其他

在本库函数中，除高速傅立叶转换和滤波器以外都可重入。

在使用本库时，必须包含表 10.51 所示的文件。另外，如表 10.52 所示，必须连接对应单片机和编译选项的库。

在调用本库时，如果函数正常结束，就返回 EDSP_OK；如果发生异常，就返回 EDSP_BAD_ARG 或者 EDSP_NO_HEAP。有关返回值的详细内容请参照各函数的说明。

表 10.51 DSP 库的 include 文件

库的种类	内容	include 文件
1 DSP 库	是进行 DSP 运算的库。	<ensigdsp.h> <filt_ws.h>*1

【注】*1 在使用滤波器函数时，只能在用户程序内包含 1 次此 include 文件。

表 10.52 DSP 库一览表

单片机	选项	库名
SH2-DSP	-pic=0	shdsplib.lib
	-pic=1	shdsppic.lib
SH3-DSP	-pic=0	-endian=big sh3dspnb.lib
	-pic=1	-endian=big sh3dsppb.lib
	-pic=0	-endian=little sh3dspnl.lib
	-pic=1	-endian=little sh3dsppl.lib
SH4AL-DSP	-pic=0	-endian=big sh4aldspnb.lib
	-pic=1	-endian=big sh4aldsppb.lib
	-pic=0	-endian=little sh4aldspnl.lib
	-pic=1	-endian=little sh4aldsppl.lib

(2) 数据格式

本库将数据作为带符号的 16 位定点数处理。带符号的 16 位定点数的数据格式如图 10.8(a) 所示，小数点被固定在最高位（MSB）右侧，能表示 $-1 \sim 1-2^{-15}$ 范围的值。

本库使用 short 型的数据格式进行数据的传递。因此，在 C/C++ 程序中使用本库时，需要用带符号的 16 位定点数表示数据。

(例) 在用带符号的 16 位定点数表示 +0.5，为 H'4000。因此，传递给库函数的 short 型实际参数为 H'4000。

本库的内部运算也使用带符号的 32 位定点数和带符号的 40 位定点数。带符号的 32 位定点数是图 10.8(b) 所示的数据格式，能表示 $-1 \sim 1-2^{-31}$ 范围的值；带符号的 40 位定点数是图 10.8(c) 所示的附加 8 位保护位的数据格式，能表示 $-2^8 \sim 2^8-2^{-31}$ 范围的值。

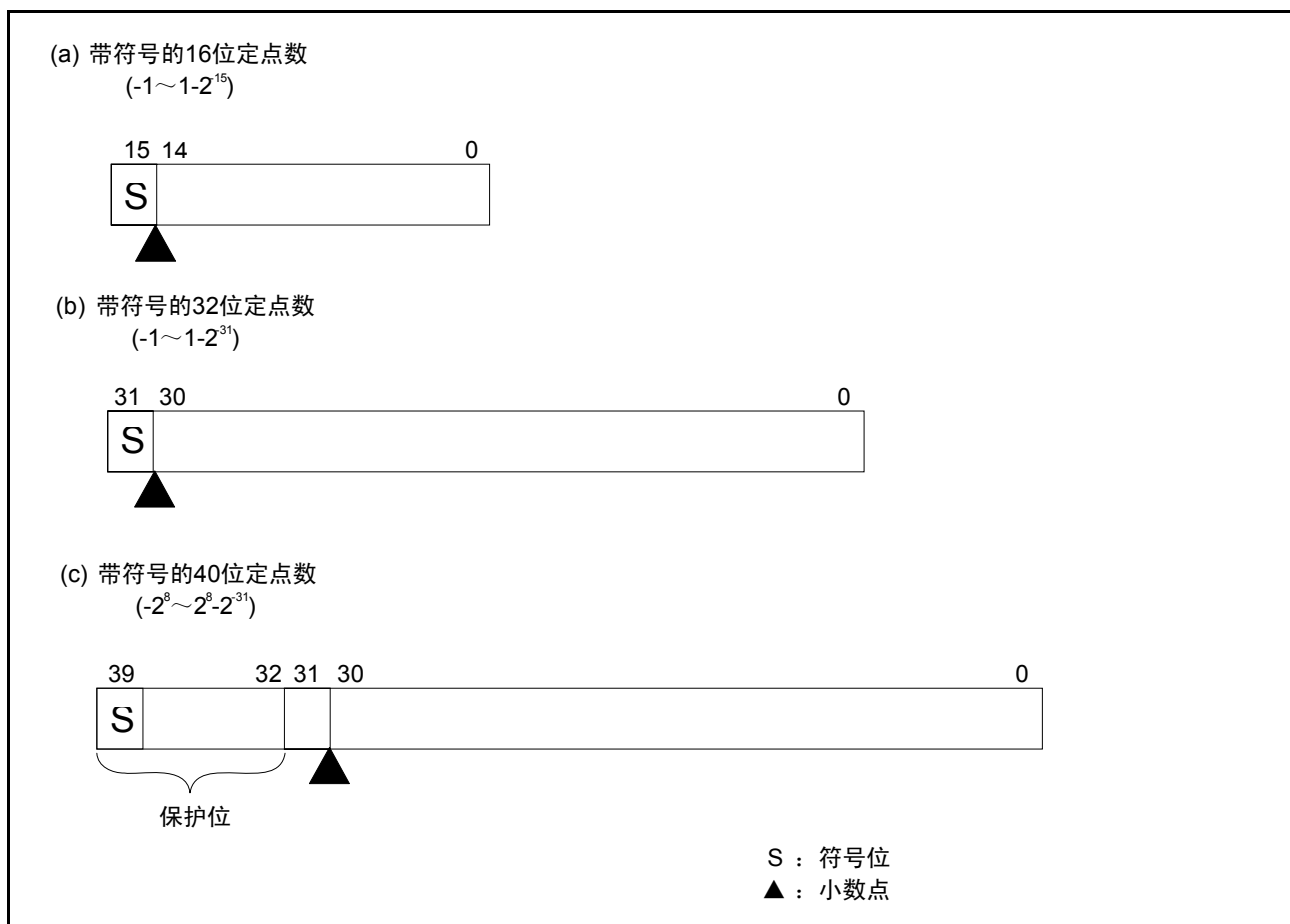


图 10.8 数据格式

用带符号的 32 位定点数保存带符号的 16 位定点数的乘法运算结果。必须注意：使用 DSP 指令的定点乘法运算只在 H'8000×H'8000 时才有可能发生上溢，并且乘法运算结果的最低位（LSB）总是 0，在将乘法运算结果用于下一次运算时，取出高 16 位，将其转换为带符号的 16 位定点数，此时有可能发生下溢或者降低精度。

本库的乘加运算用带符号的 40 位定点数保存加算结果。必须注意：不能在加法运算时发生上溢。

如果在运算时发生上溢，就有可能得不到正确的结果。为了防止上溢的发生，需要对系数或者输入数据进行定标。本库中有定标功能。有关定标的详细内容请参照各函数的说明。

(3) 效率

优化本库函数，使 SH-DSP 能高速执行。

为了高效率地使用库，在决定要开发的系统的存储器映像时，请尽量遵循以下 2 个建议事项：

- 将程序代码段分配到支持 1 个周期进行 32 位读操作的存储器。
- 将数据段分配到支持 1 个周期进行 16（或者 32）位读写操作的存储器。

如果使用的单片机内置了充分容量的 32 位存储器，最适合的方法就是将库代码和数据分配到该 32 位存储器；如果需要使用其他存储器，就请尽量遵循上述的建议事项。

(4) 和 DSP-C 一起使用

在 1 个程序中能同时使用 DSP 库和 DSP-C。如果在各函数的说明中记述了 2 种库函数的型，上段用于没有 dspc 选项时，下段用于指定 dspc 选项时；在只记述了 1 种函数时，和是否指定 dspc 选项无关，可以通用。

(5) 高速傅立叶转换

(a) 函数一览表

函数	说明
FftComplex	执行 not-in-place 复数 FFT。
FftReal	执行 not-in-place 实数 FFT。
IfftComplex	执行 not-in-place 复数逆 FFT。
IfftReal	执行 not-in-place 实数逆 FFT。
FftInComplex	执行 in-place 复数 FFT。
FftInReal	执行 in-place 实数 FFT。
IfftInComplex	执行 in-place 复数逆 FFT。
IfftInReal	执行 in-place 实数逆 FFT。
LogMagnitude	将复数数据转换为对数绝对值。
InitFft	生成 FFT 循环系数。
FreeFft	释放用于保存 FFT 循环系数的存储器。

【注】※ 有关 not-in-place 和 in-place，请参照“(5)(e) FFT 结构”。

这些函数使用用户定义的定标，执行正向高速傅立叶转换和逆向高速傅立叶转换。

用以下方程式定义正向傅立叶转换：

$$y_n = 2^{-s} \sum_{n=0}^N e^{-2j\pi n/N} \cdot x_n$$

在此，s 表示被定标的阶段数，N 表示数据的个数。

用以下方程式定义逆向傅立叶转换：

$$y_n = 2^{-s} \sum_{n=0}^N e^{2j\pi n/N} \cdot x_n$$

有关定标请参照“(5)(d) 定标”。

(b) 复数数据的数组格式

对于 FFT 和 IFFT 的复数数据数组，将实数分配到 X 存储器，虚数分配到 Y 存储器，但是实数 FFT 的输出数据和实数 IFFT 的输入数据的分配方法不同。假设保存实数和虚数的数组分别为 x 和 y，x[0] 中是 DC 成分的实数部分，y[0] 中不是 DC 成分的虚数部分而是 Fs/2 成分的实数部分（DC 成分和 Fs/2 成分都是实数，虚数部分为 0）。

(c) 实数数据的数组格式

FFT 和 IFFT 实数数据的数组格式有以下 3 种：

- 保存到单个数组，分配到任意存储块。
- 保存到单个数组，分配到 X 存储器。
- 分 2 个数组保存。各数组的大小为 $N/2$ ，数组的前半部分分配到 X 存储器，后半部分分配到 Y 存储器。

FftReal 只是第 1 种的指定方法。IfftReal、FftInReal 和 IfftInReal 是第 2 种或者第 3 种方法，由用户选择。

(d) 定标

在各阶段，基数 2 的 FFT 的信号强度成倍，并且峰值信号振幅也成倍。因此，在转换高强度信号时，有可能发生上溢。如果在各阶段将信号设定为 $1/2$ （称为定标），就能防止上溢的发生。但是，如果定标过多，也有可能发生不必要的量化噪声。

上溢或者量化噪声和定标的最适合的平衡取主要决于输入信号的特性。为了防止上溢的发生，对于大频谱峰值的信号，需要进行最大的定标，但是对于脉冲信号，不需要定标。

在全部阶段进行定标是最安全的方法。如果输入数据低于强度 2^{30} ，就能通过此方法防止上溢的发生。在本库中，能指定是否对各阶段进行定标。因此，通过精确地指定定标，能将上溢和量化噪声的影响控制在最小限。

为了指定定标的方法，各 FFT 函数的参数含有 scale。scale 从最低位开始，每一位对应每一个阶段。在对应的 scale 的位设定为 1 的各阶段，执行 2 的除法运算。

本库为了提高执行速度，使用基数 4 的 FFT。scale 从最低位开始，每 2 位对应各阶段。如果将任意 1 位设定为 1，就执行 2 的除法运算；如果将 2 位都设定为 1，就执行 4 的除法运算。即和 2 个基数 2 的 FFT 阶段能被 1 个基数 4 的 FFT 阶段替换相同，但是基数 4 的 FFT 比基数 2 的 FFT 更有可能发生量化噪声。

scale 的例子如下所示：

- scale = H'FFFFFFF（或者 size-1），在基数 2 的 FFT 的全部阶段进行定标。如果全部的输入数据的强度低于 2^{30} ，就不发生上溢。
- scale = H'55555555，在基数 2 的 FFT 的每隔 1 个阶段进行定标。
- scale = 0，不进行定标。

在 ensigdsp.h 中定义了这些 scale 的值，为 EFFTALLSCALE(H'FFFFFFF)、EFFTMIDSCALE(H'55555555)、EFFTNOSCALE(0)。

(e) FFT 结构

本库的 FFT 有 not-in-place FFT 和 in-place FFT 共 2 种结构。

not-in-place FFT 时，从 RAM 中取出输入数据，执行 FFT，输出结果保存到 RAM 的其他位置（用户指定）。

in-place FFT 时，从 RAM 中取出输入数据，执行 FFT，输出结果保存到 RAM 的相同位置。如果使用此方法，就会增加 FFT 的执行时间，但是能减少存储器的使用空间。

如果将输入数据用于其他 FFT 函数，就必须使用 not-in-place FFT。另外，如果想节约存储器空间，就必须使用 in-place FFT。

not-in-place 复数 FFT

```
int FftComplex (short op_x[ ], short op_y[ ], const short ip_x[ ],
               const short ip_y[ ], long size, long scale)
int FftComplex (_X_ _fixed op_x[ ], _Y_ _fixed op_y[ ],
               const _X_ _fixed ip_x[ ], const _Y_ _fixed ip_y[ ], long size, long scale)
```

说 明 执行复数的高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 4
 • size 不是 2 的乘方
 • size > max_fft_size

参 数 op_x[] 输出数据的实数部分
 op_y[] 输出数据的虚数部分
 ip_x[] 输入数据的实数部分
 ip_y[] 输入数据的虚数部分
 size FFT 的长度
 scale 定标的指定

备 注 因为本函数通过 **not-in-place** 进行转换，所以必须分别准备输入数组和输出数组。有关复数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”。
 必须在调用本函数前调用 **InitFft**，然后对循环系数和 **max_fft_size** 进行初始化。
 有关定标的指定请参照“(5)(d) 定标”。
 scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

not-in-place 实数 FFT***int FftReal (short op_x[], short op_y[], const short ip[], long size, long scale)******int FftReal (_X_fixed op_x[], _Y_fixed op_y[], const _fixed ip[], long size, long scale)***

说 明 执行实数的高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 8
 • size 不是 2 的乘方
 • size > max_fft_size

参 数 op_x[] 正的输出数据的实数部分
 op_y[] 正的输出数据的虚数部分
 ip[] 实数的输入数据
 size FFT 的长度
 scale 定标的指定

备 注 将 size/2 的正的输出数据保存到 op_x 和 op_y。负的输出数据是正的输出数据的共轭复数。另外，因为 0 和 $F_S/2$ 的输出数据的值是实数，所以 $F_S/2$ 的实数输出数据保存到 op_y[0]。
 因为本函数通过 not-in-place 进行转换，所以必须分别准备输入数组和输出数组。有关复数和实数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”和“(5)(c) 实数数据的数组格式”。
 必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。
 有关定标的指定请参照“(5)(d) 定标”。
 scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

not-in-place 复数逆 FFT

```
int IfftComplex (short op_x[ ], short op_y[ ], const short ip_x[ ],
                const short ip_y[ ], long size, long scale)
int IfftComplex (_X_fixed op_x[ ], _Y_fixed op_y[ ],
                const _X_fixed ip_x[ ], const _Y_fixed ip_y[ ], long size, long scale))
```

说 明 执行复数的逆高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 4
 • size 不是 2 的乘方
 • size > max_fft_size

参 数 op_x[] 输出数据的实数部分
 op_y[] 输出数据的虚数部分
 ip_x[] 输入数据的实数部分
 ip_y[] 输入数据的虚数部分
 size 逆 FFT 的长度
 scale 定标的指定

备 注 因为本函数通过 **not-in-place** 进行转换，所以必须分别准备输入数组和输出数组。复数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”。
 必须在调用本函数前调用 **InitFft**，然后对循环系数和 **max_fft_size** 进行初始化。
 有关定标的指定参照“(5)(d) 定标”。
 Scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

not-in-place 实数逆 FFT

```
int IfftReal (short op_x[], short scratch_y[], const short ip_x[],
             const short ip_y[], long size, long scale, int op_all_x)
int IfftReal (_X_ _fixed op_x[], _Y_ _fixed scratch_y[],
             const _X_ _fixed ip_x[], const _Y_ _fixed ip_y[],
             long size, long scale, int op_all_x)
```

说 明 执行实数的逆高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 8
 • size 不是 2 的乘方
 • size > max_fft_size
 • op_all_x ≠ 0 或者 1

参 数 op_x[] 实数的输出数据
 scratch_y[] 暂存存储器或者实数的输出数据
 ip_x[] 正的输入数据的实数部分
 ip_y[] 正的输入数据的虚数部分
 size 逆 FFT 长度
 scale 定标的指定
 op_all_x 输出数据的分配指定

备 注 必须将 size/2 的正的输入数据保存到 ip_x 和 ip_y。负的输出数据是正的输出数据的共轭复数。另外，因为 0 和 $F_S/2$ 的输出数据的值是实数，所以 $F_S/2$ 的实数输出数据保存到 ip_y[0]。

通过 op_all_x 指定输出数据的格式。当 op_all_x=1 时，将全部输出数据保存到 op_x；当 op_all_x=0 时，将最初的 size/2 的输出数据保存到 op_x，剩余的 size/2 的输出数据保存到 scratch_y。

因为本函数通过 not-in-place 进行转换，所以必须分别准备输入数组和输出数组。有关复数和实数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”和“(5)(c) 实数数据的数组格式”。

ip_x 和 ip_y 必须分别保存 size/2 的数据。根据 op_all_x 的值，op_x 保存 size 或者 size/2 的数据。

必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。

有关定标的指定请参照“(5)(d) 定标”。

scale 使用低 $\log_2(\text{size})$ 位。

本函数不是可重入。

in-place 复数 FFT

int FftInComplex (short data_x[], short data_y[], long size, long scale)
int FftInComplex (_X_fixed data_x[], _Y_fixed data_y[], long size, long scale)

说 明 执行 in-place 复数的高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 4
 • size 不是 2 的乘方
 • size > max_fft_size

参 数 data_x[] 输入 / 输出数据的实数部分
 data_y[] 输入 / 输出数据的虚数部分
 size FFT 的长度
 scale 定标的指定

备 注 有关复数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”。
 必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。
 有关定标的指定请参照“(5)(d) 定标”。
 Scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

in-place 实数 FFT

int FftInReal (short data_x[], short data_y[], long size, long scale, int ip_all_x)
int FftInReal (_X__fixed data_x[], _Y__fixed data_y[], long size, long scale, int ip_all_x)

说 明 执行 in-place 实数的高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 8
 • size 不是 2 的乘方
 • size > max_fft_size
 • ip_all_x ≠ 0 或者 1

参 数 data_x[] 输入时是实数数据，输出时是正的输出数据的实数部分
 data_y[] 输入时是实数数据或者未使用、输出时是正的输出数据的虚数部分
 size FFT 的长度
 scale 定标的指定
 ip_all_x 输入数据的分配指定

备 注 通过 ip_all_x 指定输入数据的格式。当 ip_all_x=1 时，从 data_x 中取出全部输入数据；当 ip_all_x=0 时，从 data_x 中取出前半部分的 size/2 的输入数据，从 data_y 中取出后半部分的 size/2 的输入数据。
 在执行本函数后，将 size/2 正的输出数据保存到 data_x 和 data_y。负的输出数据是正的输出数据的共轭复数。另外，因为 0 和 $F_S/2$ 的输出数据的值是实数，所以 $F_S/2$ 的实数输出数据保存到 data_y[0]。
 有关复数和实数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”和“(5)(c) 实数数据的数组格式”。
 data_y 保存 size/2 的数据。根据 ip_all_x 的值，data_x 保存 size 或者 size/2 的数据。
 必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。
 有关定标的指定请参照“(5)(d) 定标”。
 scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

in-place 复数逆 FFT

int IfftInComplex (short data_x[], short data_y[], long size, long scale)
int IfftInComplex (_X_fixed data_x[], _Y_fixed data_y[], long size, long scale)

说 明 执行 in-place 复数的逆高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 4
 • size 不是 2 的乘方
 • size > max_fft_size

参 数 data_x[] 输入 / 输出数据的实数部分
 data_y[] 输入 / 输出数据的虚数部分
 size 逆 FFT 的长度
 scale 定标的指定

备 注 有关复数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”。
 必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。
 有关定标的指定请参照“(5)(d) 定标”。
 scale 使用低 $\log_2(\text{size})$ 位。
 本函数不是可重入。

in-place 实数逆 FFT

int IfftInReal (short data_x[], short data_y[], long size, long scale, int op_all_x)
int IfftInReal (_X_fixed data_x[], _Y_fixed data_y[], long size, long scale, int op_all_x)

说 明 执行 in-place 实数的逆高速傅立叶转换。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 8
 • size 不是 2 的乘方
 • size > max_fft_size
 • op_all_x ≠ 0 或者 1

参 数 data_x[] 输入时是正的输入数据的实数部分，输出时是实数数据
 data_y[] 输入时是正的输入数据的虚数部分，输出时是实数数据或者未使用
 size 逆 FFT 的长度
 scale 定标的指定
 op_all_x 输出数据的分配指定

备 注 必须将 size/2 的正的输入数据保存到 data_x 和 data_y。负的输入数据是正的输入数据的共轭复数。另外，因为 0 和 $F_S/2$ 的输入数据的值是实数，所以 $F_S/2$ 的实数输入数据保存到 data_y[0]。

通过 op_all_x 指定输出数据的格式。当 op_all_x=1 时，将全部输出数据保存到 data_x；当 op_all_x=0 时，将前半部分的 size/2 的输出数据保存到 data_x，后半部分的 size/2 的输出数据保存到 data_y。

有关复数和实数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”和“(5)(c) 实数数据的数组格式”。

data_y 保存 size/2 的数据。根据 op_all_x 的值，data_x 保存 size 或者 size/2 的数据。

必须在调用本函数前调用 InitFft，然后对循环系数和 max_fft_size 进行初始化。

有关定标的指定请参照“(5)(d) 定标”。

scale 使用低 $\log_2(\text{size})$ 位。

本函数不是可重入。

对数的绝对值

```
int LogMagnitude (short output[ ], const short ip_x[ ], const short ip_y[ ],
                  long no_elements, float fscale)
```

```
int LogMagnitude ( _fixed output[ ], const _X_ _fixed ip_x[ ],
                  const _Y_ _fixed ip_y[ ], long no_elements, float fscale)
```

说 明 以分贝为单位，计算复数输入数据的对数绝对值，将定标结果写到输出数组

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 1
 • no_elements > 32767
 • |fscale| ≥ $2^{15}/(10\log_{10}2^{31})$

参 数 output[] 实数输出 z
 ip_x[] 输入的实数部分 x
 ip_y[] 输入的虚数步分 y
 no_elements 输出数据的个数 N
 fscale 输出定标系数

备 注 $z(n)=10fscale \cdot \log_{10}(x(n)^2+y(n)^2) \quad 0 \leq n < N$
 有关复数数据的数组分配，请参照“(5)(b) 复数数据的数组格式”。

循环系数的生成

int InitFft (long max_size)

说 明	生成 FFT 函数使用的循环系数（1/4 长度）。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_NO_HEAP	能用 malloc 确保的存储空间不足
	EDSP_BAD_ARG	以下任一情况： • max_size < 2 • max_size 不是 2 的乘方 • max_size > 32768
参 数	max_size	需要的 FFT 的最大长度
备 注	将循环系数保存到通过 malloc 确保的存储器。 如果生成循环系数，就更新 max_fft_size 全局变量。max_fft_size 表示 FFT 的最大容许长度。 必须在最初调用 FFT 函数前调用一次本函数。 max_size 不能小于 8。 以 max_size 指定的转换长度生成循环系数。在执行长度小于 max_size 的 FFT 函数时也使用相同的循环系数。 将循环系数的地址保存到内部变量，此时不能在用户程序中存取。 本函数不是可重入。	

循环系数的释放

void FreeFft (void)

说 明	释放用于保存循环系数的存储器。	
头文件	<ensigdsp.h>	
备 注	将 max_fft_size 全局变量置 0。在执行 FreeFft 后再次执行 FFT 函数前，必须执行 InitFft。 本函数不是可重入。	

(6) 窗函数

(a) 函数一览表

函数名	说明
GenBlackman	生成布莱克曼窗。
GenHamming	生成汉明窗。
GenHanning	生成汉宁窗。
GenTriangle	生成三角窗口。

布莱克曼窗

int GenBlackman (short output[], long win_size)

int GenBlackman (_fixed output[], long win_size)

说 明 生成布莱克曼窗，输出到 output。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG win_size ≤ 1

参 数 output[] 输出数据 w(n)
 win_size 窗大小 N

备 注 在给实际的数据添加此窗时，使用 VectorMult。
 使用的函数如下所示：

$$w(n) = (2^{15} - 1) \left[0.42 - 0.5 \cos\left(\frac{2\pi n}{N}\right) + 0.08 \cos\left(\frac{4\pi n}{N}\right) \right] \quad 0 \leq n < N$$

汉明窗口

int GenHamming (short output[], long win_size)
int GenHamming (_fixed output[], long win_size)

说 明 生成汉明窗口，输出到 output。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG win_size ≤ 1

参 数 output[] 输出数据 W(n)
 win_size 窗大小 N

备 注 在给实际的数据添加此窗时，使用 VectorMult。
 使用的函数如下所示：

$$W(n) = (2^{15} - 1) \left[0.54 - 0.46 \cos\left(\frac{2\pi n}{N}\right) \right] \quad 0 \leq n < N$$

汉宁窗

int GenHanning (short output[], long win_size)
int GenHanning (_fixed output[], long win_size)

说 明 生成汉宁窗，输出到 output。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG win_size ≤ 1

参 数 output[] 输出数据 W(n)
 win_size 窗大小 N

备 注 在给实际的数据添加此窗时，使用 VectorMult。
 使用的函数如下所示：

$$W(n) = \left(\frac{2^{15} - 1}{2} \right) \left[1 - \cos\left(\frac{2\pi n}{N}\right) \right] \quad 0 \leq n < N$$

三角窗口

int GenTriangle (short output[], long win_size)
int GenTriangle (_fixed output[], long win_size)

说 明 生成三角窗口，输出到 output。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG win_size ≤ 1

参 数 output[] 输出数据 w(n)
 win_size 窗大小 N

备 注 在给实际的数据添加此窗时，使用 VectorMult。
 使用的函数如下所示：

$$w(n) = (2^{15} - 1) \left[1 - \left\lfloor \frac{|2n - N + 1|}{N + 1} \right\rfloor \right] \quad 0 \leq n < N$$

(7) 滤波器

(a) 函数一览表

函数名	说明
Fir	执行有限冲激响应滤波器的处理。
Fir1	单数据执行有限冲激响应滤波器的处理。
lir	执行无限冲激响应滤波器的处理。
lir1	单数据执行无限冲激响应滤波器的处理。
Dlir	执行双精度无限冲激响应滤波器的处理。
Dlir1	单数据执行双精度无限冲激响应滤波器的处理。
Lms	执行适应的 FIR 滤波器的处理。
Lms1	单数据执行适应 FIR 滤波器的处理。
InitFir	分配用于 FIR 滤波器的工作区。
Initlir	分配用于 IIR 滤波器的工作区。
InitDlir	分配用于 DIIR 滤波器的工作区。
InitLms	分配用于 LMS 滤波器的工作区。
FreeFir	释放在 InitFir 分配的工作区。
Freelir	释放在 Initlir 分配的工作区。
FreeDlir	释放在 InitDlir 分配的工作区。
FreeLms	释放在 InitLms 分配的工作区。

【注】使用本函数时，只能在用户程序内包含 1 次 `filt_ws.h`。

(b) 系数的定标

如果进行滤波器处理，就有可能发生饱和或者量化噪声。通过滤波器系数的定标，能将他们控制到最小程度，但是在进行定标时，必须充分考虑饱和和量化噪声的影响。系数太大，就可能发生饱和；系数太小，有可能发生量化噪声。

在 FIR（有限冲激响应）滤波器情况下，要使以下表达式成立，只要设定滤波器系数，就不会发生饱和。

$\text{coeff}[i] \neq H'8000$ （对于全部的 i ）

$\sum |\text{coeff}| < 2^{24}$

$\text{res\_shift} = 24$

coeff 是滤波器系数， res_shift 是通过输出进行右移的位数。

但是，在多个输入信号的情况下，即使使用更小 res_shift 的值（或者更大的 coeff 的值），也可减少发生饱和的可能性，并且也能大幅度减少量化噪声。如果输入值有可能包含 $H'8000$ ，就必须将全部的 coeff 值设定在 $H'8001 \sim H'7FFF$ 范围内。

IIR（无限冲激响应）滤波器是递归结构，因此，上述的定标方法不适用。

LMS（最小均方）适应的滤波器和 FIR 滤波器一样，但是在适应系数时，有可能发生饱和。此时，必须进行系数不包含 $H'8000$ 的设定。

(c) 工作区

数字滤波器必须保持从某个处理到下一个处理的信息，这些信息保存在能以最小系统开销存取的存储器中。本库将 Y-RAM 区用作工作区。必须在执行滤波器处理前调用 `Init` 函数，初始化工作区。

通过库函数存取工作区存储器，不能从用户程序直接存取工作区。

(d) 存储器的使用

为了高效率使用 SH-DSP，必须将滤波器系数分配到 X 存储器。能将输入 / 输出数据分配到任意的存储段。

必须用 #pragma section 将滤波器系数分配到 X 存储器。

在指定 dspc 选项时，必须使用 __X 存储器型限定词。此时，不需要指定 #pragma section。

必须用 Init 函数将各滤波器从全局缓冲区分配到工作区，而全局缓冲区分配在 Y 存储器中。

FIR

```
int Fir(short output[ ], const short input[ ], long no_samples, const short coeff[ ],
        long no_coeffs, int res_shift, short *workspace)
int Fir (__fixed output[ ], const __fixed input[ ], long no_samples,
        const __X __fixed coeff[ ], long no_coeffs, int res_shift, __Y __fixed *workspace)
```

说 明 执行有限冲激响应（FIR）滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：

- no_samples < 1
- no_coeffs ≤ 6
- res_shift < 0
- res_shift > 25

参 数

output[]	输出数据 y
input[]	输入数据 x
no_samples	输入数据的个数 N
coeff[]	滤波器系数 h
no_coeffs	系数的个数（滤波器的长度）K
res_shift	适用于各输出的右移
workspace	指向工作区的指针

备 注 最新的输入数据保存在工作区，input 的滤波器处理结果写到 output。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k) x(n - k) \right] \cdot 2^{-\text{res\_shift}}$$

用 39 位保存乘加运算的结果，输出 y(n) 是右移 res_shift 位后的结果的低 16 位，在发生上溢时为正或负的最大值。

有关系数的定标请参照“(7)(b) 系数的定标”。

必须在调用本函数前调用 InitFir，然后对滤波器的工作区进行初始化。

如果给 output 指定和 input 相同的数组，input 就被盖写。

本函数不是可重入。

用于单数据的 FIR

int Fir1 (short *output, short input, const short coeff[], long no_coeffs, int res_shift, short *workspace)

说 明 对单数据执行有限冲激响应 (FIR) 滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况:
 • no_coeffs ≤ 6
 • res_shift < 0
 • res_shift > 25

参 数 output 指向输出数据 y(n) 的指针
 input 输入数据 x(n)
 coeff[] 滤波器系数 h
 no_coeffs 系数的个数 (滤波器的长度) K
 res_shift 适用于各输出的右移
 workspace 指向工作区的指针

备 注 最新的输入数据保存在工作区, input 的滤波器处理结果写到 *output。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k) x(n - k) \right] \cdot 2^{-\text{res\_shift}}$$

用 39 位保存乘加运算的结果, 输出 y(n) 是右移 res_shift 位后的结果的低 16 位, 在发生上溢时为正或负的最大值。

有关系数的定标请参照 “(7)(b) 系数的定标”。

必须在调用本函数前调用 InitFir, 然后对滤波器的工作区进行初始化。

本函数不是可重入。

IIR

```
int Iir (short output[ ], const short input[ ], long no_samples, const short coeff[ ],
        long no_sections, short *workspace)
int Iir ( _fixed output[ ], const _fixed input[ ], long no_samples,
        const _X_ _fixed coeff[ ], long no_sections, _Y_ _fixed *workspace)
```

说 明 执行无限冲激响应 (IIR) 滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况:
 • no_samples < 1
 • no_sections < 1
 • a_{0k} < 0
 • a_{0k} > 16

参 数 output[] 输出数据 y_{k-1}
 input[] 输入数据 x₀
 no_samples 输入数据的个数 N
 coeff[] 滤波器系数
 no_sections 2 阶滤波器段的个数 K
 workspace 指向工作区的指针

备 注 滤波器的结构是将双 2 阶滤波器连接成 K 列。对于各双 2 阶滤波器的输出, 进行附加的定标。用带符号的 16 位定点数指定滤波器系数。
 由以下方程式得到各双 2 阶滤波器的输出:

$$D_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)]2^{-15}$$

$$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)]2^{-a_{0k}}$$

 第 k 个段的输入 x_k(n) 是前一个段的输出 y_{k-1}(n)。从 input 读取最初段 (k=0) 的输入, 将最后段 (k=K-1) 的输出写到 output。
 必须按以下顺序设定系数 Coeff:
 a₀₀、a₁₀、a₂₀、b₀₀、b₁₀、b₂₀、a₀₁、a₁₁、a₂₁、b₀₁、...、b_{2K-1}
 a_{0k} 项是对第 k 个双 2 阶滤波器的输出进行右移的位数。
 用 32 位对各双 2 阶滤波器进行饱和运算。各双 2 阶滤波器的输出是右移 15 位或者 a_{0k} 位后的结果的低 16 位, 在发生上溢时为正或负的最大值。
 必须在调用本函数前调用 InitIir, 然后对滤波器的工作区进行初始化。
 如果给 output 指定和 input 相同的数组, input 就被盖写。
 本函数不是可重入。

单数据的 IIR

int Iir1 (short *output, short input, const short coeff[], long no_sections, short *workspace)

说 明 对单数据执行无限冲激响应 (IIR) 滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况:
 • no_sections < 1
 • $a_{0k} < 0$
 • $a_{0k} > 16$

参 数 output 指向输出数据 $y_{k-1}(n)$ 的指针
 input 输入数据 $x_0(n)$
 coeff[] 滤波器系数
 no_sections 2 阶滤波器段的个数 K
 workspace 指向工作区的指针

备 注 滤波器的结构是将双 2 阶滤波器连接成 K 列。对于各双 2 阶滤波器的输出, 进行附加的定标。用带符号的 16 位定点数指定滤波器系数。
 由以下方程式得到各双 2 阶滤波器的输出:
 $D_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{15}x(n)]2^{-15}$
 $y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)]2^{-a_{0k}}$
 第 k 个段的输入 $x_k(n)$ 是前一个段的输出 $y_{k-1}(n)$ 。从 input 读取最初段 ($k=0$) 的输入, 将最后段 ($k=K-1$) 的输出写到 output。
 必须按以下顺序设定系数 Coeff:
 a_{00} 、 a_{10} 、 a_{20} 、 b_{00} 、 b_{10} 、 b_{20} 、 a_{01} 、 a_{11} 、 a_{21} 、 b_{01} 、...、 b_{2K-1}
 a_{0k} 项是对第 k 个双 2 阶滤波器的输出进行右移的位数。
 用 32 位对各双 2 阶滤波器进行饱和运算。各双 2 阶滤波器的输出是右移 15 位或者 a_{0k} 位后的结果的低 16 位, 在发生上溢时为正或负的最大值。
 必须在调用本函数前调用 InitIir, 然后对滤波器的工作区进行初始化。
 本函数不是可重入。

双精度 IIR

```
int DIir (short output[ ], const short input[ ], long no_samples,
         const long coeff[ ], long no_sections, long *workspace)
int DIir (_fixed output[ ], const _fixed input[ ], long no_samples,
         const _X long _fixed coeff[ ], long no_sections, _Y long _fixed *workspace)
```

说 明 执行双精度无限冲激响应滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：

- no_samples < 1
- no_sections < 1
- a_{0k} < 3
- 在 k < K-1 时 a_{0k} > 32
- 在 k = K-1 时 a_{0k} > 48

参 数 output[] 输出数据 y_{K-1}
 input[] 输入数据 x
 no_samples 输入数据的个数 N
 coeff[] 滤波器系数
 no_sections 2 阶滤波器段的个数 K
 workspace 指向工作区的指针

备 注 滤波器的结构是将双 2 阶滤波器连接成 K 列。对于各双 2 阶滤波器的输出，进行附加的定标。用带符号的 32 位定点数指定滤波器系数。

由以下方程式得到各双 2 阶滤波器的输出：

$$D_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{29}x(n)]2^{-31}$$

$$y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)]2^{-a_{0k}2^2}$$

第 k 个段的输入 x_k(n) 是前一个段的输出 y_{k-1}(n)。从将 input 左移 16 位后的值读取最初段 (k=0) 的输入，将最后的段 (k=K-1) 的输出写到 output。

必须按以下顺序设定系数 Coeff：

a₀₀、a₁₀、a₂₀、b₀₀、b₁₀、b₂₀、a₀₁、a₁₁、a₂₁、b₀₁、...、b_{2K-1}

a_{0k} 项是对第 k 个双 2 阶滤波器的输出进行右移的位数。

DIir 和 Iir 不同的是：DIir 不是用 16 位值而是用 32 位值指定滤波器系数。用 64 位保存乘加运算的结果。中间阶段的输出是右移 a_{0k} 位后的结果的低 32 位，在发生上溢时为正或负的最大值。最后阶段的输出是右移 a_{0K-1} 位后的结果的低 16 位，在发生上溢时为正或负的最大值。

必须在调用本函数前调用 InitDIir，然后对滤波器的工作区进行初始化。

延迟节点 d_k(n) 被舍入成 30 位的值，在发生上溢时为正或负的最大值。

必须在用带符号的 32 位定点数指定系数后使用 DIir。此时，必须在 k<K-1 时将 a_{0k} 置 31，而在 k=K-1 时将 a_{0k} 置 47。

DIir 的执行速度比 Iir 慢，所以如果需要双精度计算，就请使用 Iir。

如果给 output 指定和 input 相同的数组，input 就被盖写。

本函数不是可重入。

用于单数据的双精度 IIR

int Ddir1 (short *output, const short input, const long coeff[], long no_sections, long *workspace)

说 明	对单数据进行双精度无限冲激响应滤波器的处理。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	以下任一情况： • no_sections < 1 • a_{0k} < 3 • 在 k < K-1 时 a_{0k} > 32 • 在 k = K-1 时 a_{0k} > 48
参 数	output	指向输出数据 y _{k-1} (n) 的指针
	input	输入数据 x ₀ (n)
	coeff[]	滤波器系数
	no_sections	2 阶滤波器段的个数 K
	workspace	指向工作区的指针
备 注	滤波器的结构是将双 2 阶滤波器连接成 K 列。对于各双 2 阶滤波器的输出，进行附加的定标。用带符号的 32 位定点数指定滤波器系数。 由以下方程式得到各双 2 阶滤波器的输出： $D_k(n) = [a_{1k}d_k(n-1) + a_{2k}d_k(n-2) + 2^{29}x(n)]2^{-31}$ $y_k(n) = [b_{0k}d_k(n) + b_{1k}d_k(n-1) + b_{2k}d_k(n-2)]2^{-a_{0k}2^2}$ 第 k 个段的输入 x_k(n) 是前一个段的输出 y_{k-1}(n)。从将 input 左移 16 位后的值读取最初段 (k=0) 的输入，将最后的段 (k=K-1) 的输出写到 output。 必须按以下顺序设定系数 Coeff： a₀₀、a₁₀、a₂₀、b₀₀、b₁₀、b₂₀、a₀₁、a₁₁、a₂₁、b₀₁、...、b_{2K-1} a_{0k} 项是对第 k 个双 2 阶滤波器的输出进行右移的位数。 Ddir1 和 Iir1 不同的是：Ddir1 不是用 16 位值而是用 32 位值指定滤波器系数。用 64 位保存乘加运算的结果。中间阶段的输出是右移 a_{0k} 位后的结果的低 32 位，在发生上溢时为正或负的最大值。最后阶段的输出是右移 a_{0K-1} 位后的结果的低 16 位，在发生上溢时为正或负的最大值。 必须在调用本函数前调用 InitDIir，然后对滤波器的工作区进行初始化。 延迟节点 d_k(n) 被舍入成 30 位的值，在发生上溢时为正或负的最大值。 必须在用带符号的 32 位定点数指定系数后使用 Ddir1。此时，必须在 k<K-1 时将 a_{0k} 置 31，而在 k=K-1 时将 a_{0k} 置 47。 Ddir1 的执行速度比 Iir1 慢，所以如果需要双精度计算，就请使用 Iir1。 本函数不是可重入。	

适应 FIR

```

int Lms (short output[ ], const short input[ ], const short ref_output[ ],
        long no_samples, short coeff[ ], long no_coeffs, int res_shift,
        short conv_fact, short *workspace)
int Lms (_fixed output[ ], const _fixed input[ ], const _fixed ref_output[ ],
        long no_samples, __X__fixed coeff[ ], long no_coeffs, int res_shift,
        short conv_fact, __Y__fixed *workspace)

```

说 明 使用最小均方算法 (LMS)，执行实数适应 FIR 滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_samples < 1
 • no_coeffs ≤ 6
 • res_shift < 0
 • res_shift > 25

参 数 output[] 输出数据 y
 input[] 输入数据 x
 ref_output[] 期望的输出值 d
 no_samples 输入数据的个数 N
 coeff[] 适应滤波器系数 h
 no_coeffs 系数的个数 K
 res_shift 适应于各输出的右移
 conv_fact 收敛系数 2μ
 workspace 指向工作区的指针

备 注 用以下方程式定义 FIR 滤波器：

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k) x(n-k) \right] \cdot 2^{-\text{res\_shift}}$$

用 39 位保存乘法运算的结果，输出 y(n) 是右移 res_shift 位后的结果的低 16 位，在发生上溢时为正或负的最大值。

使用 Widrow-Hoff 算法更新滤波器系数。

$$h_{n+1}(k) = h_n(k) + 2\mu e(n)x(n-k)$$

在此，e(n) 是期望的信号和实际输出的误差。

$$e(n) = d(n) - y(n)$$

在计算 2μe(n)x(n-k) 时，进行 2 次 16 位 × 16 位的乘法运算，2 次乘法运算的结果都保存高 16 位，在发生上溢时为正或负的最大值。如果更新的系数值为 H'8000，就有可能在乘法运算中发生上溢，所以必须将系数的值设定在 H'8001 ~ H'7FFF 范围内。

有关系数的定标请参照“(7)(b) 系数的定标”。因为通过 LMS 滤波器让系数适应，最安全的定标设定方法是将系数不超过 256 个并将 res_shift 置 24。

通常，必须将 conv_fact 设定为正，而且不能置为 H'8000。

必须在调用本函数前调用 InitLms，然后对滤波器进行初始化。

如果给 output 指定与 input 或者 ref_output 相同的数组，input 或者 ref_output 就被盖写。

本函数不是可重入。

用于单数据的适应 FIR

```
int Lms1 (short *output, short input, short ref_output, short coeff[],
         long no_coeffs, int res_shift, short conv_fact, short *workspace)
```

说 明 使用最小均方算法 (LMS)，对单数据执行实数适应 FIR 滤波器的处理。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_coeffs ≤ 6
 • res_shift < 0
 • res_shift > 25

参 数 output 指向输出数据 $y(n)$ 的指针
 input 输入数据 $x(n)$
 ref_output 期望的输出值 $d(n)$
 coeff[] 适应滤波器系数 h
 no_coeffs 系数的个数 K
 res_shift 适用于各输出的右移
 conv_fact 收敛系数 2μ
 workspace 指向工作区的指针

备 注 通过以下方程式定义 FIR 滤波器。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k) \right] \cdot 2^{-\text{res\_shift}}$$

用 39 位保存乘法运算的结果，输出 $y(n)$ 是右移 res_shift 位后的结果的低 16 位，在发生上溢时为正或负的最大值。

使用 Widrow-Hoff 算法更新滤波器系数。

$$h_{n+1}(k) = h_n(k) + 2\mu e(n)x(n-k)$$

在此， $e(n)$ 是期望的信号和实际输出的误差。

$$e(n) = d(n) - y(n)$$

在计算 $2\mu e(n)x(n-k)$ 时，进行 2 次 16 位 × 16 位的乘法运算，2 次乘法运算的结果都保存高 16 位，在发生上溢时为正或负的最大值。如果更新的系数值为 H'8000，就有可能在乘法运算中发生上溢，所以必须将系数的值设定在 H'8001 ~ H'7FFF 范围内。

有关系数的定标请参照“(7)(b) 系数的定标”。因为通过 LMS 滤波器让系数适应，最安全的定标设定方法是将系数不超过 256 个并将 res_shift 置 24。

通常，必须将 conv_fact 设定为正，而且不能置为 H'8000。

必须在调用本函数前调用 InitLms，然后对滤波器进行初始化。

本函数不是可重入。

FIR 工作区的分配

int InitFir (short **workspace, long no_coefs)
int InitFir (_Y_ _fixed **workspace, long no_coefs)

说 明	分配 Fir 和 Fir1 使用的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_NO_HEAP	workspace 能使用的存储空间不足
	EDSP_BAD_ARG	no_coefs ≤ 2
参 数	workspace	指向工作区指针的指针
	no_coefs	系数的个数 K
备 注	将已输入的数据全部初始化为 0。 只有 Fir、Fir1、Lms 和 Lms1 能操作由 InitFir 分配的工作区，不能从用户程序直接存取工作区。 本函数不是可重入。	

IIR 工作区的分配

int InitIir (short **workspace, long no_sections)
int InitIir (_Y_ _fixed **workspace, long no_sections)

说 明	分配 Iir 和 Iir1 使用的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_NO_HEAP	workspace 能使用的存储空间不足
	EDSP_BAD_ARG	no_sections < 1
参 数	workspace	指向工作区指针的指针
	no_sections	2 阶滤波器段的个数 K
备 注	将已输入的数据全部初始化为 0。 只有 Iir 和 Iir1 能操作由 InitIir 分配的工作区，不能从用户程序直接存取工作区。 本函数不是可重入。	

双精度 IIR 工作区的分配

int InitDIir(long **workspace, long no_sections)
int InitDIir (_Y long __fixed **workspace, long no_sections)

说 明	分配 Diir 和 DIir1 使用的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_NO_HEAP	workspace 能使用的存储空间不足
	EDSP_BAD_ARG	no_sections < 1
参 数	workspace	指向工作区指针的指针
	no_sections	2 阶滤波器段的个数 K
备 注	将已输入的数据全部初始化为 0。 只有 Diir 和 DIir1 能操作由 InitDIir 分配的工作区。 本函数不是可重入。	

适应 FIR 工作区的分配

int InitLms(short **workspace, long no_coeffs)
int InitLms (_Y long __fixed **workspace, long no_coeffs)

说 明	分配 Lms 和 Lms1 使用的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_NO_HEAP	workspace 能使用的存储空间不足
	EDSP_BAD_ARG	no_coeffs ≤ 2
参 数	workspace	指向工作区指针的指针
	no_coeffs	系数的个数 K
备 注	将已输入的数据全部初始化为 0。 只有 Fir、Fir1、Lms 和 Lms1 能操作由 InitLms 分配的工作区，不能从用户程序直接存取工作区。 本函数不是可重入。	

FIR 工作区的释放

int FreeFir (short **workspace, long no_coeffs)
int FreeFir (_Y_ _fixed **workspace, long no_coeffs)

说 明	释放由 InitFir 分配的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	no_coeffs ≤ 2
参 数	workspace	指向工作区指针的指针
	no_coeffs	系数的个数 K
备 注	本函数不是可重入。	

IIR 工作区的释放

int FreeIir (short **workspace, long no_sections)
int FreeIir (_Y_ _fixed **workspace, long no_sections)

说 明	释放由 InitIir 分配的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	no_sections < 1
参 数	workspace	指向工作区指针的指针
	no_sections	2 阶滤波器段的数 K
备 注	本函数不是可重入。	

双精度 IIR 工作区的释放

int FreeDIir (long **workspace, long no_sections)
int FreeDIir (_Y long _fixed **workspace, long no_sections)

说 明	释放由 InitDIir 分配的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	$\text{no\_section} \leq 2$
参 数	workspace	指向工作区指针的指针
	no_sections	2 阶滤波器段的个数 K
备 注	本函数不是可重入。	

适应 FIR 工作区释放

int FreeLms (short **workspace, long no_coefs)
int FreeLms (_Y _fixed **workspace, long no_coefs)

说 明	释放由 InitLms 分配的工作区。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	$\text{no\_coefs} < 1$
参 数	workspace	指向工作区指针的指针
	no_coefs	系数的个数 K
备 注	本函数不是可重入。	

(8) 卷积和相互关系

(a) 函数一览表

函数名	说明
ConvComplete	计算 2 个数组的完全卷积。
ConvCyclic	计算 2 个数组的周期性卷积。
ConvPartial	计算 2 个数组的部分卷积。
Correlate	计算 2 个数组的相互关系。
CorrCyclic	计算 2 的数组的周期性相互关系。

在使用这些函数时，必须将 2 个输入数组中的 1 个数组分配到 X 存储器，而另一个数组分配到 Y 存储器。输出数组可分配到任何一个存储器。

完全卷积

```
int ConvComplete (short output[ ], const short ip_x[ ], const short ip_y[ ],
                 long x_size, long y_size, int res_shift)
int ConvComplete ( _ _fixed output[ ], const _ _X _ _fixed ip_x[ ],
                 const _ _Y _ _fixed ip_y[ ], long x_size, long y_size, int res_shift)
```

说 明 求 2 个输入数组 x 和 y 的完全卷积，结果写到输出数组 z。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • x_size < 1
 • y_size < 1
 • res_shift < 0
 • res_shift > 25

参 数 output[] 输出 z
 ip_x[] 输入 x
 ip_y[] 输入 y
 x_size ip_x 的大小 X
 y_size ip_y 的大小 Y
 res_shift 适用于各输出的右移

备 注

$$z(m) = \left[\sum_{i=0}^{x-1} x(i) y(m-i) \right] \cdot 2^{-\text{res\_shift}} \quad 0 \leq m < X+Y-1$$

将输入数组以外的数据读取为 0。

不能将输出数组的大小设定为小于 X+Y-1。

周期性卷积

```
int ConvCyclic (short output[ ], const short ip_x[ ], const short ip_y[ ], long size, int res_shift)
```

```
int ConvCyclic (_fixed output[ ], const _X_fixed ip_x[ ], const _Y_fixed ip_y[ ], long size, int res_shift)
```

说 明 周期性地进行 2 个输入数组 x 和 y 的卷积计算，结果写到输出数组 z。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 6
 • res_shift < 0
 • res_shift > 25

参 数 output[] 输出 z
 ip_x[] 输入 x
 ip_y[] 输入 y
 size 数组的大小 N
 res_shift 适用于各输出的右移

备 注

$$z(m) = \left[\sum_{i=0}^{N-1} x(i) y(|m - i + N|_N) \right] \cdot 2^{-res\_shift} \quad 0 \leq m < N$$

在此， $|i|_N$ 表示余数 $(i \% N)$ 。

部分卷积

```
int ConvPartial (short output[ ], const short ip_x[ ], const short ip_y[ ],
               long x_size, long y_size, int res_shift)
int ConvPartial (_fixed output[ ], const _X_ _fixed ip_x[ ],
               const _Y_ _fixed ip_y[ ], long x_size, long y_size, int res_shift)
```

说 明 本函数求 2 个输入数组 x 和 y 的卷积，结果写到输出数组 z。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • x_size < 5
 • y_size < 1
 • res_shift < 0
 • res_shift > 25

参 数 output[] 输出 z
 ip_x[] 输入 x
 ip_y[] 输入 y
 x_size ip_x 的大小 X
 y_size ip_y 的大小 Y
 res_shift 适用于各输出的右移

备 注 不包含由输入数组以外的数据导致的输出。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i) b(m + A - 1 - i) \right] \cdot 2^{-res\_shift} \quad 0 \leq m \leq |A-B|$$

在此，数组的个数 $a < b$ ，A 是 a 的大小，B 是 b 的大小。

不能将输出数组的大小设定为小于 $|X-Y|+1$ 。

将输入数组以外的数据读取为 0。

相互关系

```
int Correlate (short output[ ], const short ip_x[ ], const short ip_y[ ], long x_size,
              long y_size, long no_corr, int x_is_larger, int res_shift)
int Correlate (_fixed output[ ], const _X_ _fixed ip_x[ ],
              const _Y_ _fixed ip_y[ ], long x_size, long y_size, long no_corr,
              int x_is_larger, int res_shift)
```

说 明 求 2 个输入数组 x 和 y 的相互关系，结果写到输出数组 z。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • x_size < 1
 • y_size < 1
 • no_corr < 1
 • res_shift < 0
 • res_shift > 25
 • x_is_larger ≠ 0 或者 1

参 数 output[] 输出 z
 ip_x[] 输入 x
 ip_y[] 输入 y
 x_size ip_x 的大小 X
 y_size ip_y 的大小 Y
 no_corr 要计算的相关的个数 M
 x_is_larger X=Y 时的数组指定
 res_shift 适用于各输出的右移

备 注 在以下方程式中，数组的个数 a > b，A 是 a 的大小。当 X=Y 时，如果
 x_is_larger=1，
 x 就为 a，如果 x_is_larger=0，x 就为 b。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i) b(i + m) \right] \cdot 2^{-res\_shift} \quad 0 \leq m < M$$

即使 A < X + M 也没关系。此时，将输入数组以外的数据读取为 0。

res_shift = 0 相当于通常的整数计算，res_shift = 15 相当小数计算。

周期性相互关系

```
int CorrCyclic (short output[ ], const short ip_x[ ], const short ip_y[ ], long size,
               int reverse, int res_shift)
int CorrCyclic (_fixed output[ ], const _X _fixed ip_x[ ],
               const _Y _fixed ip_y[ ], long size, int reverse, int res_shift)
```

说 明 周期性地进行数组 x 和 y 的相互关系计算，结果写到输出数组 z。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • size < 5
 • res_shift < 0
 • res_shift > 25
 • reverse ≠ 0 或者 1

参 数 output[] 输出 z
 ip_x[] 输入 x
 ip_y[] 输入 y
 size 数组的大小 N
 reverse 反相标志
 res_shift 适用于各输出的右移

备 注

$$z(m) = \left[\sum_{i=0}^{N-1} x(i) y(|i + m|_N) \right] \cdot 2^{-\text{res\_shift}} \quad 0 \leq m < N$$

在此， $|i|_N$ 表示余数 ($i \% N$)。在 reverse=1 时，将输出的数据反相，实际计算如下：

$$z(m) = \left[\sum_{i=0}^{N-1} y(i) x(|i + m|_N) \right] \cdot 2^{-\text{res\_shift}} \quad 0 \leq m < N$$

(9) 其他

(a) 函数一览表

函数名	说明
Limit	将 H'8000 的数据替换为 H'8001。
CopyXtoY	将数组从 X 存储器复制 Y 存储器。
CopyYtoX	将数组从 Y 存储器复制到 X 存储器。
CopyToX	将数组从指定的位置复制到 X 存储器。
CopyToY	将数组从指定的位置复制到 Y 存储器。
CopyFromX	将数组从 X 存储器复制到指定的位置。
CopyFromY	将数组从 Y 存储器复制到指定的位置。
GenGNoise	生成高斯白噪声。
MatrixMult	进行 2 个矩阵的乘法运算。
VectorMult	进行 2 个数据的乘法运算。
MsPower	求均方值。
Mean	求平均值。
Variance	求平均值和偏差。
MaxI	求整数数组的最大值。
MinI	求整数数组的最小值。
PeakI	求整数数组的最大绝对值。

H'8000→H'8001 的替换

int Limit (short data_xy[], long no_elements, int data_is_x)

int Limit (_fixed data_xy[], long no_elements, int data_is_x)

说 明	将 H8000 的输入数据替换为 H'8001，这样可以在进行 DSP 指令的定点乘法运算时不发生上溢。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	以下任一情况： no_elements < 1 data_is_x ≠ 0 或者 1
参 数	data_xy[]	数据数组
	no_elements	数据的个数
	data_is_x	数据的分配指定
备 注	即使进行此处理，也有可能在乘加运算的加法运算中发生上溢。 必须在 data_is_x=1 时，将数据分配到 X 存储器，而在 data_is_x=0 时，将数据分配到 Y 存储器。	

X 存储器→Y 存储器的复制

```
int CopyXtoY (short op_y[ ], const short ip_x[ ], long n)
```

```
int CopyXtoY (_ _Y_ _fixed op_y[ ], const _ _X_ _fixed ip_x[ ], long n)
```

说 明	将数组从 ip_x 复制到 op_y。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	n < 6
参 数	op_y[]	输出数组
	ip_x[]	输入数组
	n	数据的个数
备 注	必须将 ip_x 分配到 X 存储器， op_y 分配到 Y 存储器。	

Y 存储器→X 存储器的复制

```
int CopyYtoX (short op_x[ ], const short ip_y[ ], long n)
```

```
int CopyYtoX (_ _X_ _fixed op_x[ ], const _ _Y_ _fixed ip_y[ ], long n)
```

说 明	将数组从 ip_y 复制到 op_x。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	n < 1
参 数	op_x[]	输出数组
	ip_y[]	输入数组
	n	数据的个数
备 注	必须将 ip_x 分配到 X 存储器， op_y 分配到 Y 存储器。	

复制到 X 存储器

int CopyToX (short op_x[], const short input[], long n)
int CopyToX (_X_ _fixed op_x[], const _fixed input[], long n)

说 明 将数组 input 复制到 op_x。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG n < 1

参 数 op_x[] 输出数组
 input[] 输入数组
 n 数据的个数

备 注 必须将 op_x 分配到 X 存储器，而 input 可分配到任意的存储器。

复制到 Y 存储器

int CopyToY (short op_y[], const short input[], long n)
int CopyToY (_Y_ _fixed op_y[], const _fixed input[], long n)

说 明 将数组 input 复制到 op_y。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG n < 1

参 数 op_y[] 输出数组
 input[] 输入数组
 n 数据的个数

备 注 必须将 op_y 分配到 Y 存储器，而 input 可分配到任意的存储器。

从 X 存储器复制

int CopyFromX (short output[], const short ip_x[], long n)
int CopyFromX (_fixed output[], const _X_ _fixed ip_x[], long n)

说 明	将数组 ip_x 复制到 output。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	n < 1
参 数	output[]	输出数组
	ip_x[]	输入数组
	n	数据的个数
备 注	必须将 ip_x 分配到 X 存储器，而 output 可分配到任意的存储器。	

从 Y 存储器复制

int CopyFromY (short output[], const short ip_y[], long n)
int CopyFromY (_fixed output[], const _Y_ _fixed ip_y[], long n)

说 明	将数组 ip_y 复制到 output。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	n < 1
参 数	output[]	输出数组
	ip_y[]	输入数组
	n	数据的个数
备 注	必须将 ip_y 分配到 Y 存储器，而 output 可分配到任意的存储器。	

高斯白噪声

int GenGWnoise (short output[], long no_samples, float variance)
int GenGWnoise (_fixed output[], long no_samples, float variance)

说 明	生成平均值为 0 且用户指定偏差的白色高斯噪声。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	以下任一情况： • no_samples < 1 • variance ≤ 0.0
参 数	output[]	白噪声数据的输出
	no_samples	输出数据的个数
	variance	高斯分布的偏差 σ^2
备 注	以 2 个数据为 1 组，生成输出数据。为了生成 1 组输出数据，在 x 平方的合计小于 1 前，使用 rand 函数生成 1 组 -1 ~ 1 之间的随机数 γ_1 和 γ_2。通过以下方程式计算 1 组的输出数据 o_1 和 o_2。 $o_1 = \sigma \gamma_1 \sqrt{-2 \ln(x)/x}$ $o_2 = \sigma \gamma_2 \sqrt{-2 \ln(x)/x}$ 如果将数据的个数设定为奇数，就取消最后 1 组的第 2 个数据。 因为本函数调用的标准库的 rand 函数不是可重入，所以生成的随机数 γ_1 和 γ_2 的顺序不一定总是相同。但是，不会影响生成的白噪声 o_1 和 o_2 的特性。 本函数使用浮点运算，因为浮点运算的处理速度慢，所以建议在将本函数用于评估。	

矩阵的乘法运算

```
int MatrixMult (void *op_matrix, const void *ip_x, const void *ip_y, long m,
               long n, long p, int x_first, int res_shift)
```

```
int MatrixMult (void *op_matrix, const __X void *ip_x, const __Y void *ip_y,
               long m, long n, long p, int x_first, int res_shift)
```

说 明 进行 2 个矩阵 x 和 y 的乘法运算，结果保存到 op_matrix。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • m, n, 或者 p < 5
 • res_shift < 0
 • res_shift > 25
 • x_first ≠ 0 或者 1

参 数 op_matrix 指向输出的第一数据的指针
 ip_x 指向输入 x 的第一数据的指针
 ip_y 指向输入 y 的第一数据的指针
 m 矩阵 1 的行数
 n 矩阵 1 的列数、矩阵 2 的行数
 p 矩阵 2 的列数
 x_first 矩阵乘法运算的顺序指定
 res_shift 适用于各输出的右移

备 注 在 x_first=1 时，计算 xy。此时，ip_x 为 m×n，ip_y 为 n×p、op_matrix 为 m×p。
 在 x_first=0 时，计算 yx。此时，ip_y 为 m×n，ip_x 为 n×p，op_matrix 为 m×p。
 用 39 位保存乘加运算的结果。输出 y(n) 是右移 res_shift 位后的结果的低 16 位，在
 发生上溢时为正或负的最大值。
 按照通常的 C 样式（行优先顺序）分配各矩阵。

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

为了能指定任意的数组大小，通过 void* 指定数组参数，这些参数必须指向 short 型变量。

必须分别准备输入数组 ip_x 和 ip_y 及输出数组 op_matrix。

乘法运算

```
int VectorMult (short output[ ], const short ip_x[ ], const short ip_y[ ],
               long no_elements, int res_shift)
```

```
int VectorMult (_fixed output[ ], const _X_ fixed ip_x[ ],
               const _Y_ fixed ip_y[ ], long no_elements, int res_shift)
```

说 明	从 ip_x 和 ip_y 中各取出 1 个数据，进行乘法运算，结果保存到 output。	
头文件	<ensigdsp.h>	
返回值	EDSP_OK	成功
	EDSP_BAD_ARG	以下任一情况： • no_elements < 4 • res_shift < 0 • res_shift > 16
参 数	output[]	输出
	ip_x[]	输入 1
	ip_y[]	输入 2
	no_elements	数据的个数
	res_shift	适用于各输出的右移
备 注	输出是右移 res_shift 位后的结果的低 16 位，在发生上溢时为正或负的最大值。 本函数进行数据的乘法运算。计算内积时，必须在将 m 和 p 置 1 后使用 MatrixMult。	

均方值

```
int MsPower (long *output, const short input[ ], long no_elements, int src_is_x)
```

```
int MsPower (long __fixed *output, const __fixed input[ ], long no_elements, int src_is_x)
```

说 明 求输入数据的均方值。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 6
 • src_is_x ≠ 0 或者 1

参 数 output 指向输出的指针
 input[] 输入 x
 no_elements 数据的个数 N
 src_is_x 数据的分配指定

备 注

$$\text{均方值} = \frac{1}{N} \sum_{i=0}^{N-1} x(i)^2$$

除法运算结果向最近的整数值方向舍入。

用 63 位保存运算结果。如果 no_elements 大于等于 2³²，就有可能发生上溢。

在 src_is_x=1 时，必须将数据分配到 X 存储器，而在 src_is_x=0 时，将数据分配到 Y 存储器。

平均值

int Mean (short *mean, const short input[], long no_elements, int src_is_x)
int Mean (_fixed *mean, const _fixed input[], long no_elements, int src_is_x)

说 明 求输入数据的平均值。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 6
 • src_is_x ≠ 0 或者 1

参 数 mean 指向 input 平均 $\bar{x}$ 的指针
 input[] 输入 x
 no_elements 数据的个数 N
 src_is_x 数据的分配指定

备 注

$$\bar{x} = \frac{1}{N} \sum_{i=0}^{N-1} x(i)$$

除法运算结果向最近的整数值方向舍入。

用 32 位保存运算结果。如果 no_elements 大于 $2^{16}-1$ ，就有可能发生上溢。

在 src_is_x=1 时，必须将数据分配到 X 存储器，而在 src_is_x=0 时，将数据分配到 Y 存储器。

平均值和偏差

```
int Variance (long *variance, short *mean, const short input[ ],
             long no_elements, int src_is_x)
```

```
int Variance (long __fixed *variance, __fixed *mean,
             const __fixed input[ ], long no_elements, int src_is_x)
```

说 明 求 input 的平均值和偏差。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 4
 • src_is_x ≠ 0 或者 1

参 数 variance 指向输入偏差 σ^2 的指针
 mean 指向数据平均 $\bar{x}$ 的指针
 input[] 输入 x
 no_elements 数据的个数 N
 src_is_x 数据的分配指定

备 注

$$\bar{x} = \frac{1}{N} \sum_{i=0}^{N-1} x(i)$$

$$\sigma^2 = \frac{1}{N} \sum_{i=0}^{N-1} x(i)^2 - \bar{x}^2$$

除法运算结果向最近的整数值方向舍入。

用 32 位保存 $\bar{x}$ ，不检查是否发生上溢。

如果 no_elements 大于 $2^{16}-1$ ，就有可能发生上溢。

用 63 位保存 σ^2 ，不检查是否发生溢出。

在 src_is_x=1 时，必须将数据分配到 X 存储器，而在 src_is_x=0 时，将数据分配到 Y 存储器。

最大值

```
int MaxI (short **max_ptr, short input[ ], long no_elements, int src_is_x)
```

```
int MaxI (_fixed **max_ptr, _fixed input[ ], long no_elements, int src_is_x)
```

说 明 检索数组 input 的最大值，将该地址返回到 max_ptr。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 1
 • src_is_x ≠ 0 或者 1

参 数 max_ptr 指向最大数据指针的指针
 input[] 输入
 no_elements 数据的个数
 src_is_x 数据的分配指定

备 注 如果多个数据有相同的最大值，就将最近的数据地址返回到 input 的开头。
 在 src_is_x=1 时，必须将数据分配到 X 存储器，而在 src_is_x=0 时，将数据分配到 Y 存储器。

最小值

```
int MinI (short **min_ptr, short input[ ], long no_elements, int src_is_x)
```

```
int MinI (_fixed **min_ptr, _fixed input[ ], long no_elements, int src_is_x)
```

说 明 检索数组 input 的最小值，将该地址返回到 min_ptr。

头文件 <ensigdsp.h>

返回值 EDSP_OK 成功
 EDSP_BAD_ARG 以下任一情况：
 • no_elements < 1
 • src_is_x ≠ 0 或者 1

参 数 min_ptr 指向最小数据指针的指针
 input[] 输入
 no_elements 数据的个数
 src_is_x 数据的分配指定

备 注 如果多个数据有相同的最小值，将最近的数据地址返回到 input 的开头。
 在 src_is_x=1 时，必须将数据分配到 X 存储器，而在 src_is_x=0 时，将数据分配到 Y 存储器。

最大绝对值

```
int PeakI (short **peak_ptr, short input[ ], long no_elements, int src_is_x)
```

```
int PeakI ( _fixed **peak_ptr, _fixed input[ ], long no_elements, int src_is_x)
```

说 明 检索数组 `input` 的最大绝对值，将该地址返回到 `peak_ptr`。

头文件 `<ensigdsp.h>`

返回值 `EDSP_OK` 成功
 `EDSP_BAD_ARG` 以下任一情况：
 • `no_elements < 1`
 • `src_is_x ≠ 0` 或者 `1`

参 数 `peak_ptr` 指向最大绝对值数据指针的指针
 `input[ ]` 输入
 `no_elements` 数据的个数
 `src_is_x` 数据的分配指定

备 注 如果多个数据有相同的最大绝对值，就将最近的数据地址返回到 `input` 的开头。
 在 `src_is_x=` 时，必须将数据分配到 X 存储器，而在当 `src_is_x=0` 时，将数据分配到 Y 存储器。

第 11 章 汇编程序的语言规格

11.1 程序的元素

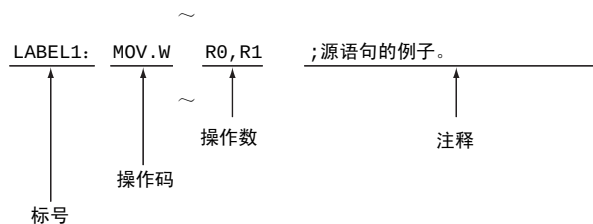
11.1.1 源语句

(1) 源语句的结构

源语句的结构如下所示：

[标号][Δ 操作][Δ 操作数][注释]

■ 编程例



(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、DSP 指令、扩展指令、汇编程序控制指令和各种控制语句的助记符（名称）。执行指令和 DSP 指令是单片机的指令。扩展指令是汇编程序定义的指令，被扩展为执行指令和常数数据（文字）或者多个执行指令。汇编程序控制指令是对汇编程序给予指示的指令。控制语句为有关文件包含功能、条件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能记述假名 / 日语汉字（shift JIS 码和 EUC 码）、LATIN 码字符。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) 源语句的记述方法

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（:）。

■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 (:),
; 所以汇编程序不认为是标号。

(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。

■编程例

ADD R0, R1 ; 没有记述标号时。
LABEL1: ADD 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。

■编程例

ADD R0, R1 ; ADD 指令的操作数为 2 个。
SHAL 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。

■编程例

ADD 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, 能将 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) 的步骤, 记述跨多行的源语句:

- 将分开操作数和操作数的逗号 (,) 作为一个段落进行换行。
- 在下一行的第 1 列写加号 (+)。
- 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。

■编程例 1

```

.DATA.L  H'FFFF0000,
+        H'FF00FF00,
+        H'FFFFFFFF
; 跨 3 行写 1 条源语句的例子。

```

■编程例 2

```

.DATA.L  H'FFFF0000,      ; 初始值 1
+        H'FF00FF00,      ; 初始值 2
+        H'FFFFFFFF        ; 初始值 3
; 给每个操作数加注释的例子。

```

11.1.2 关键字

关键字是汇编程序作为具有特别含义的符号而准备的名称。关键字有寄存器名、运算符和定位计数器。寄存器名因各单片机种类而不同，请参照各单片机的《编程手册》。关键字不能用作符号。

- 寄存器名
R0 ~ R15、FR0 ~ FR15、DR0 ~ DR14（只限偶数号）、XD0 ~ XD14（只限偶数号）、FV0 ~ FV12（只限 4 的倍数序号）、R0_BANK ~ R7_BANK、SP*、SR、GBR、VBR、MACH、MACL、PR、PC、SSR、SPC、FPUL、FPSCR、MOD、RE、RS、DSR、A0、A0G、A1、A1G、M0、M1、X0、X1、Y0、Y1、XMTRX、DBR、SGR、TBR
- 运算符
STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD、\$EVEN、\$ODD、\$EVEN2、\$ODD2
- 定位计数器
\$

【注】 * R15 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。
- 寄存器别名 : 表示通用寄存器和浮点寄存器。
- 段名 : 表示段的名称。

符号的使用例子如下：

■编程例 1

```

~
BRA      SUB1                ; BRA 是转移指令。
~
SUB1:    ; SUB1 表示转移目标的地址符号。
~

```


(d) 不能用作符号的名称

关键字不能用于符号，并且汇编程序将以下所示的名称用作内部符号*。程序员不能使用相同的名称。

`_$Ommmmm` (m 是数字 (0 ~ F))

`_$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

通过标号的记述进行符号的定义，在参照时记述为操作数。作为例外，将 `.SECTION` 和 `.MACRO` 定义的符号记述为操作数。另外，将 `.MACRO` 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，有时参照出现在定义前，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，需要参照跨文件的符号。能从其他源文件参照定义的符号的定义称为外部定义。反之，参照其他源文件定义的符号的参照称为外部参照。外部定义由 `.EXPORT`、`.GLOBAL` 声明，外部参照由 `.IMPORT`、`.GLOBAL` 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

整数常数用基数表示。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 进制表示的数值记述
- 8 进制数 用基数 Q' 和 8 进制表示的数值记述
- 10 进制数 用基数 D' 和 10 进制表示的数值记述
- 16 进制数 用基数 H' 和 16 进制表示的数值记述

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` 指定为几进制数）。

■编程例

```
.DATA.B B'10001000      ;
.DATA.B Q'210           ; 这些源语句都表示相同的数值
.DATA.B D'136           ;
.DATA.B H'88            ;
```

【补充】

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)。

(2) 字符常数

字符常数是将字符码作为数值的常数。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 / 日语汉字、LATIN 码字符。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 / 日语汉字，就必须在 shift JIS 码时指定 sjis 选项、在 EUC 码时指定 euc 选项；如果记述 LATIN 码字符，就必须指定 latin1 选项。另外，不能混合使用 shift JIS 码、EUC 码和 LATIN 码。

■编程例 1

```
.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
```

■编程例 2

```
.DATA.B ""         ; 为 1 个双引号字符的字符常数。
.DATA.L "汉字"    ; 汉字
```

(3) 浮点常数

能通过确保浮点常数的汇编程序控制指令进行浮点常数的指定。

(a) 浮点常数的记述方法

浮点常数的记述有 10 进制记述和 16 进制记述共 2 种。

• 10 进制记述

$$F' [\{\pm\}] \left\{ \begin{matrix} n [.m] \\ .m \end{matrix} \right\} [t [\{\pm\}] xx]$$

F' : 表示 10 进制的浮点常数，不能省略。

$[\{\pm\}] \left\{ \begin{matrix} n [.m] \\ .m \end{matrix} \right\}$: 用 10 进制指定整数部分 n 和小数部分 m，能省略整数部分或者小数部分的值。另外，如果省略 $\pm$ ，就假定是 +。

t : 指定的精度代码有以下 2 种：

- S 单精度
- D 双精度

如果省略，就遵循汇编程序控制指令的操作长度。

$[\{\pm\}]xx$: 用 10 进制指定指数部 (10 的乘方) 的值。

如果省略，就假定是 0；如果省略 $\pm$ ，就假定是 +。

■例

$F' 0.5S - 2 = 0.5 \times 10^{-2} = 0.005 = H' 3BA3D70A$

$F' .123D3 = 0.123 \times 10^3 = 123 = H' 405EC00000000000$

- 16 进制记述

H' xxxx [.t]

H' : 表示 16 进制, 不能省略。

xxxx : 用 16 进制指定浮点的位模式。在指定短于数据长的情况下, 确保右边, 左边为所需个数的 0; 在指定长于数据长的情况下, 确保右边的有效数据。

t : 指定的精度代码有以下 2 种:

- S 单精度
- D 双精度

如果省略, 就遵循汇编程序控制指令的操作长度。

对于在 10 进制中难以记述的精度为 0 或无穷大的数据, 此格式可直接指定其浮点常数的位模式。

■ 例

H' 0123456789ABCDEF.S → H' 89ABCDEF

H' FFFF.D → H' 000000000000FFFF

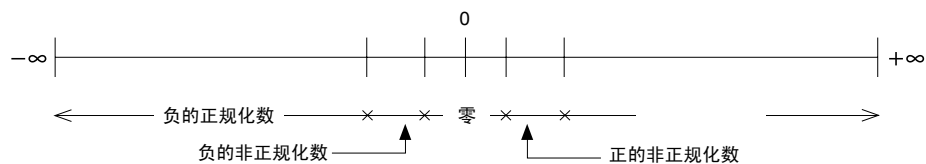
(b) 浮点值的范围

浮点值的数据型如表 11.1 所示。

表 11.1 浮点值的数据型

数据型	内容
正规化数 (Normalized Number)	下溢边界的值 ≤ 绝对值 ≤ 上溢边界的值
非正规化数 (Denormalized Number)	0 < 绝对值 < 下溢边界的值
零	绝对值为 0
无穷大	绝对值 > 上溢边界的值
非数值 (Not a Number: NaN)	不为数值的值 有 sNaN (signaling NaN) 和 qNaN (quiet NaN)

用数的直线表示如下。但是, 非数值不作为数值处理, 所示不表示在数的直线上。



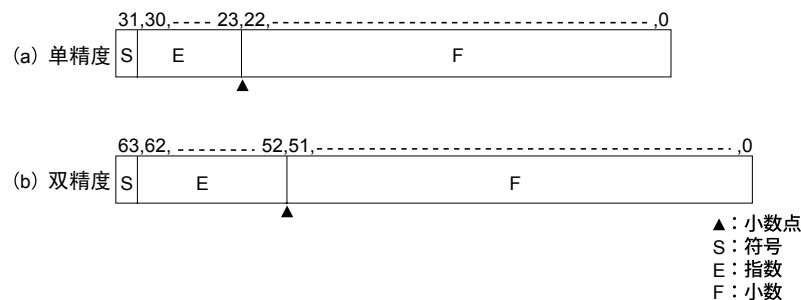
汇编程序能记述的数值范围如表 11.2 所示。

表 11.2 数据格式和数值范围（绝对值）

数据格式		单精度	双精度
正规化数	最大值	3.40×10^{38}	1.79×10^{308}
	最小值	1.18×10^{-38}	2.23×10^{-308}
非正规化数	最大值	1.17×10^{-38}	2.22×10^{-308}
	最小值	1.40×10^{-45}	4.94×10^{-324}

(c) 浮点数据格式

表示 FPU 的浮点数据格式。



- 符号（S）
表示数值的符号。0 为正数，1 为负数。
- 指数（E）
表示指数。数据格式中指数的值减某一定的偏差值（bias）为实际的指数。
- 小数（F）
小数各位有各自的含义，从起始位开始对应 2^{-1} 、 2^{-2} 、……、 2^{-n} （n 为小数的位长）。

数据格式的长度如表 11.3 所示。

表 11.3 数据格式的长度

参数	单精度	双精度
位长	32 位	64 位
符号位（S）	1 位	1 位
指数（E）	8 位	11 位
小数（F）	23 位	52 位
指数的偏差值（bias）	127	1023

用表 11.3 的符号表示浮点的数值如下：

$$2^{E-bias} \cdot (-1)^S \cdot \begin{cases} (1.F) & \text{：正规化数} \\ (0.F) & \text{：非正规化数} \end{cases}$$

$$(1.F) = 1 + b_0 \times 2^{-1} + b_1 \times 2^{-2} + \dots + b_{n-1} \times 2^{-n}$$

$$(0.F) = b_0 \times 2^{-1} + b_1 \times 2^{-2} + \dots + b_{n-1} \times 2^{-n}$$

b：小数的位的位置
n：小数的位长

各数据型的浮点表现如表 11.4 所示。因为非数值不为数值，所示不表示。

表 11.4 数据型的浮点表示

数据型	单精度	双精度
正规化数	$(-1)^S \cdot 2^{E-127} \cdot (1.F)$	$(-1)^S \cdot 2^{E-1023} \cdot (1.F)$
非正规化数	$(-1)^S \cdot 2^{-126} \cdot (0.F)$	$(-1)^S \cdot 2^{-1022} \cdot (0.F)$
零	$(-1)^S \cdot 0$	
无穷大	$(-1)^S \cdot \infty$	
非数值	quiet NaN、signaling NaN	

(d) 浮点常数的有效数字

在确保浮点常数的控制指令中，本汇编程序在生成目标码时支持以下 2 种舍入方法，并设定有效数字。

- (a) Round to Nearest even(RN) : 将目标码的最低位舍入为最近的绝对值。
当两绝对值距离相等时，使最低位为零。
- (b) Round to Zero(RZ) : 舍去目标码的最低位后置 0。

■例

```
.FDATA.S F' 1S-1 的目标码
RN: H' 3DCCCCCD
RZ: H' 3DCCCCC
```

(e) 非正规化数的处理

非正规化数的处理因单片机种类而不同。在单片机不处理非正规数的情况下，对于非正规化数范围的值，发生警告 841，输出 0 的目标码；在单片机处理非正规化数的情况下，对于非正规化数范围的值，发生警告 842，输出非正规化数的目标码。能通过选项更改非正规化数的处理。

■例

- 单片机不处理非正规数的情况
.FDATA.S F' 1S-40 : 警告 841、目标码 H' 00000000
- 单片机处理非正规化数的情况
.FDATA.S F' 1S-40 : 警告 842、目标码 H' 000116C2

(4) 浮点运算（四则运算）的规格

在确保浮点常数的控制指令中，能记述四则运算。

(a) 浮点数的 10 进制记述方法

- 四则运算的第 2 个数据以后的 "F'" 可以省略，但是为了区别指数部分，在省略 "F'" 时必须记述小数部分。

■例

```
F' 0.5+2.0      : F' 0.5+F' 2.0
F' 0.5+2        : F' 0.5+2    2 被解释为指数
F' 0.5*2        : 出错
```

- 不能在 "F" 的后面记述 "("。

■ 例

```
(F' 0.5+0.6)*0.7      :   OK
F' (0.5+0.6)*0.7      :   NG
```

- 在对 .FDATA 用逗号分开指定多个数据时，不能省略逗号后面的 "F"。

■ 例

```
.FDATA.S F' 0.5, 0.6   :   NG
```

【注】浮点数运算的运算对象限于 10 进制记述的数值，可进行四则运算。10 进制记述和 16 进制记述的数值运算、16 进制记述的数值会出错。

另外，在运算对象为 10 进制记述的数值并且精度不一样时，输出警告信息(W)826，操作长度指定的精度有效。

■ 例

```
F' 1.5S + F' 0.5S      OK
F' 1.5S + 0.5S         OK: "F" 可省略
F' 1.5S + H' 3BA3D70A   NG: (E)101 SYNTAX ERROR IN SOURCE STATEMENT
H' 3BA3D70A + H' 3BA3D70A NG: (E)101 SYNTAX ERROR IN SOURCE STATEMENT
F' 1.5S + 0.5D         OK: (W)826 ILLEGAL PRECISION
                        ( 进行 F' 1.5S + F' 0.5S 的运算 )
```

(b) 结果值的舍入方法

在浮点的四则运算中，如果其结果超过内部表示的小数部分的有效数字，就按照以下规则进行舍入：

- 结果值向接近该值的 2 个浮点数的内部表示中的较近的方向舍入。
- 结果值在接近该值的 2 个浮点数的正中央时，向小数的最后位为 0 的方向舍入。
- 在单片机为 SH-2E 的情况下，舍去超过有效数字的部分。
- 在单片机为 SH-4、SH-4A、SH2A-FPU 的情况下，如果指定 round = nearest，就将超过有效数字的部分进行四舍五入；如果指定 round = zero，就舍去超过有效数字。

(c) 上溢、下溢和无效运算时的处理

对执行时的上溢、下溢和无效运算进行以下的处理：

- 上溢时，根据结果的符号为正无穷大或者负无穷大。
- 下溢时，根据结果的符号为正零或者负零。
- 如果加上符号相反的无穷大相或者减去符号相同的无穷大，就产生无效运算。此时，结果为非数值。

【注】有关常数表达式，在汇编时进行运算。

此时，如果检测到上溢、下溢或者无效运算，就发生警告级错误。

(5) 定点常数

能通过确保定点常数的汇编程序控制指令的操作数进行定点常数的指定。

(a) 定点常数的记述方法

定点常数作为 -1.0 ~ 1.0 范围的实数，用 10 进制数记述。定点常数有字和长字 2 种。

- 字的定点常数

用带符号的 2 个字节整数表示 -1.0 ~ 1.0 范围的实数。带符号的 2 个字节整数的值 x ($-32,768 \leq x \leq 32,767$) 表示的实数值为 $x/32768$ 。

■例

定点常数	字数据的表现
-1.0	H'8000
-0.5	H'C000
0.0	H'0000
0.5	H'4000
1.0	H'7FFF

• 长字的定点常数

用带符号的 4 个字节整数表示 -1.0 ~ 1.0 范围的实数。带符号的 4 个字节整数值 x ($-2,147,483,648 \leq x \leq 2,147,483,647$) 表示的实数值为 $x/2147483648$ 。

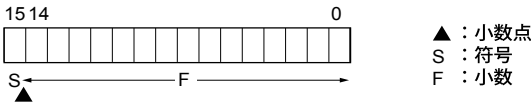
■例

定点常数	长字数据的表现
-1.0	H'80000000
-0.5	H'C0000000
0.0	H'00000000
0.5	H'40000000
1.0	H'7FFFFFFF

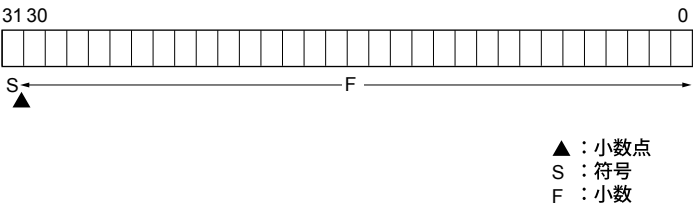
(b) 定点常数的数据格式

在字的情况下，定点常数的数据格式由 1 位符号和 15 位数值构成；在长字的情况下，定点常数的数据格式由 1 位符号和 31 位数值构成。所以认为小数点总是在符号的右边。

字



长字



• 符号 (S)

表示数值的符号。0 为正数，1 为负数。

• 小数 (F)

小数的各位有各自的含义，从起始位开始对应 2^{-1} 、 2^{-2} 、……、 2^{-31} 。

(c) 定点常数的有效数字

在长字的情况下，31 位能表示的值为 10 进制的 9 位数，但是汇编程序以 10 进制的小数处理有效数字，用 RN（向接近绝对值为 0 的方向舍入）舍入第 32 位，并将计算结果的高 31 位作为定点数据。

【注】实际的定点数据范围为 -1.0 ~ 0.9999999999，而将 1.0 假定为 0.9999999999，既 H'7FFFFFFF。

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（位置）。定位计数器的值根据目标码的输出自动变化，并且能通过汇编程序控制指令故意地改变定位计数器值。

■编程例

```

~
.ORG      H'00001000    ; 给定位计数器设定 H'00001000。
.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'00001002。
.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'00001004。
.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'00001006。
.ALIGN    4             ; 将定位计数器值调整为 4 的倍数。
                        ; 定位计数器值变为 H'00001008。
.DATA.L   H'FFFFFFFF    ; 此汇编程序控制指令的目标码长度为 4 个字节。
                        ; 定位计数器值变为 H'0000100C。
                        ; .ORG 是设定定位计数器值的汇编程序控制指令。
                        ; .ALIGN 是调整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。
                        ; .L 指定以长字（=4 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。

■编程例

```

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、符号和运算符的组合，进行运行结果的计算，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（寄存器的别名除外）
 - 根据上述的项和运算符得出的运算结果
- 单独的项也是一种表达式。

(b) 运算符

运算符一览表如表 11.5 所示。

表 11.5 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	求段集合的起始地址	STARTOF 段名
	SIZEOF	以字节为单位求段集合的大小	SIZEOF 段名
奇偶运算	\$EVEN	在 2 的倍数时为 1，否则为 0	\$EVEN 符号
	\$ODD	在 2 的倍数时为 0，否则为 1	\$ODD 符号
	\$EVEN2	在 4 的倍数时为 1，否则为 0	\$EVEN2 符号
	\$ODD2	在 4 的倍数时为 0，否则为 1	\$ODD2 符号
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

(c) 括号

能用圆括号 () 更改运算的优先顺序。

(2) 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 11.6 所示。

表 11.6 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 \$EVEN \$ODD \$EVEN2 \$ODD2 HIGH LOW HWORD LWORD *	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】 * 优先顺序 1 的运算符为一元运算符。

表达式的记述例子如下所示：

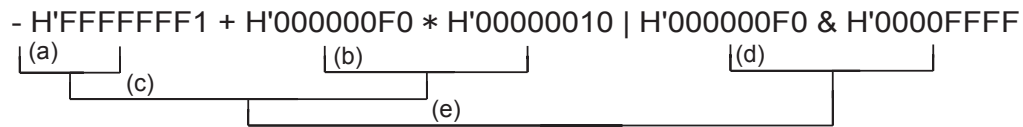
■ 编程例 1

$$1 + (2 - (3 + (4 - 5)))$$

汇编程序按(a)~(d)的顺序计算。

(a)的结果	 -1	} 最后结果为1。
(b)的结果	 2	
(c)的结果	 0	
(d)的结果	 1	

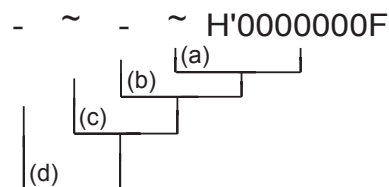
■ 编程例 2



汇编程序按(a)~(e)的顺序计算。

(a)的结果	H'0000000F	} 最后结果为H'0000FFF。
(b)的结果	H'00000F00	
(c)的结果	H'00000F0F	
(d)的结果	H'000000F0	
(e)的结果	H'00000FFF	

■ 编程例 3



汇编程序按(a)~(d)的顺序计算。

(a)的结果	H'FFFFFFF0	} 最后结果为H'00000011。
(b)的结果	H'00000010	
(c)的结果	H'FFFFFFEF	
(d)的结果	H'00000011	

(3) 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的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。

■ 编程例

```

        .CPU          SH1
        .SECTION      INIT_RAM, DATA, ALIGN=4
        .RES.B        H'100
        .SECTION      INIT_DATA, DATA, ALIGN=4
INIT_BGN .DATA.L      (STARTOF INIT_RAM)          ; [1]
INIT_END .DATA.L      (STARTOF INIT_RAM) + (SIZEOF INIT_RAM) ; [2]
;
;
        .SECTION      MAIN, CODE, ALIGN=4
INITIAL:
        MOV.L         DATA1, R6
        MOV           #0, R5
        MOV.L         DATA1+4, R3
        BRA           LOOP2
        MOV.L         @R3, R4
LOOP1:
        MOV.L         R5, @R4
        ADD           #4, R4
LOOP2:
        MOV.L         @R6, R3
        CMP/Hi        R3, R4
        BF            LOOP1
        RTS
        NOP
DATA1:
        .DATA.L       INIT_END
        .DATA.L       INIT_BGN
        .END

```

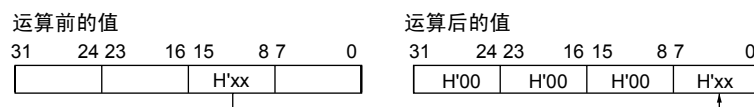
用零初始化段 (INIT_RAM) 的数据区。

[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。



■ 编程例

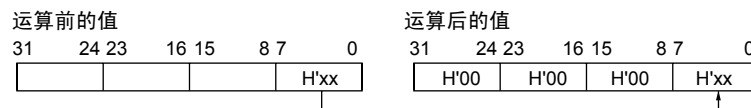
```

LABEL   .EQU          H'00007FFF
        .DATA          HIGH LABEL          ; 将整数数据确保在存储器上。

```

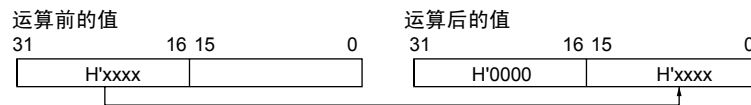
(d) LOW 运算

抽出 4 个字节值的最低位字节。



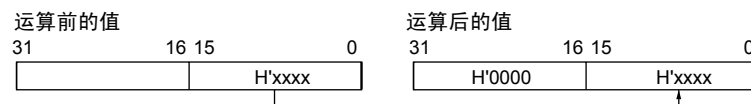
(e) HWORD 运算

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。



(g) 奇偶运算

判断地址符号的值为 2 的倍数或者 4 的倍数。

奇偶运算符的运算内容如表 11.7 所示。

表 11.7 奇偶运算符的运算内容

运算符	运算内容
\$EVEN	在 2 的倍数时为 1，否则为 0
\$ODD	在 2 的倍数时为 0，否则为 1
\$EVEN2	在 4 的倍数时为 1，否则为 0
\$ODD2	在 4 的倍数时为 0，否则为 1

■编程例

用 \$ODD2 运算符求当前程序计数的例子。

LAB:

```

MOVA    @(0, PC), R0
ADD     #-4+2*$ODD2 LAB, R0      ; 如果 LAB 为 4 的倍数,
                                   ; 就将 $ODD2 置 0, 否则置 1。

```

(4) 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 位带符号的处理。

■ 编程例

`~H'F0`

汇编程序将 `H'F0` 解释为 `H'000000F0`，因此 `~H'F0` 为 `H'FFFFFF0F` 而不为 `H'0000000F`。

(b) 算术运算

对于必须确定值的位置，在乘法或者除法运算时不能将相对地址或外部参照符号用作项。另外，在除法运算时不能将 0 用作除数。

■ 编程例

<code>.IMPORT</code>	<code>SYM</code>	
<code>.DATA</code>	<code>SYM/10</code>	; 正常
<code>.ORG</code>	<code>SYM/10</code>	; 错误

(c) 逻辑运算

在逻辑运算时不能将相对地址或者外部参照符号用作项。

11.1.7 字符串

字符串是一串字符的数据。在字符串中能使用以下 ASCII 字符：

ASCII 码	[	H'09 (制表符)
		H'20 (空白) ~ H'7E (~)

字符串中的 1 个字符表示具有该字符的 ASCII 码值的字节数据。另外，能记述 shift JIS 码、EUC 码的假名 / 日语汉字、LATIN 码字符。如果将假名 / 日语汉字作为字符记述，就必须在 shift JIS 码时指定 sjis 选项、在 EUC 码时指定 euc 选项。如果不指定，就取决于主机的日语代码。如果记述 LATIN 码字符，就必须指定 latin1 选项。

必须用双引号 (") 将字符排列括起来记述字符串。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。

■ 编程例

```
.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 个字节或者 4 个字节。

字符串不作为数值处理，数据的长度为 1 个字节 ~ 255 个字节。

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。因为局部标号不与有效范围外的其他符号冲突，所以能不必在意其他符号名而进行局部控制。和通常的地址符号一样，能通过标号的记述进行局部标号的定义，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，还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、.ASSIGN、.REG、.FREG 标号
- .EXPORT、.IMPORT、.GLOBAL 的操作数

局部标号的使用例子如下所示：

■编程例

```

LABEL1:                                ; 局部块 1 的开始
?0001:
    ~
    CMP/EQ      R1, R2
    BT          ?0002      ; 转移到局部块 1 的 ?0002。
    BRA        ?0001      ; 转移到局部块 1 的 ?0001。
?0002:
    ~
LABEL2:                                ; 局部块 2 的开始
?0001:
    ~
    CMP/GE      R1, R2
    BT          ?0002      ; 转移到局部块 2 的 ?0002。
    BRA        ?0001      ; 转移到局部块 2 的 ?0001。
?0002:
LABEL3:                                ; 局部块 3 的开始

```

(2) 局部标号的命名方法

(a) 头字符

局部标号是以头字符“?”开始的字符串。

(b) 能用于局部标号的字符

除了头字符以外，局部标号的字符由以下 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，由地址符号或者 .SECTION 进行局部块的分隔。在局部块内能参照该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，不发生错误。

【注】 不认为 .ASSIGNA、ASSIGNC、.EQU、.ASSIGN、.REG、.FREG 定义的地址符号是有效范围的分隔符。

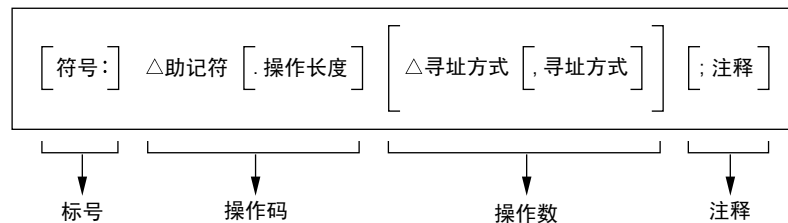
11.2 执行指令

11.2.1 执行指令的概要

执行指令是单片机的指令。

单片机在解读并执行存储器中的执行指令的目标码。

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	数据长度
B	字节
W	字（2 个字节）
L	长字（4 个字节）
S	单精度（4 个字节）
D	双精度（8 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 11.8 所示：

表 11.8 寻址方式一览表

寻址方式	名 称	解 说
Rn	寄存器直接	寄存器的内容。
@Rn	寄存器间接	存储区。Rn 的值表示区域的起始地址。
@Rn+	后增寄存器间接	存储区。递增 *1 前的 Rn 的值表示区域的起始地址。 单片机先参照 Rn，然后递增。
@-Rn	先减寄存器间接	存储区。递减 *2 后的 Rn 的值表示区域的起始地址。 单片机先递减 Rn，然后参照。
@(disp,Rn)	带位移量的寄存器间接 *3	存储区。区域的起始地址为 Rn 的值 + 位移量 (disp)。 Rn 的内容不变。
@(R0,Rn)	带变址的寄存器间接	存储区。区域的起始地址为 Rn 的值 + R0 的值。 Rn 和 R0 的内容不变。
@(disp,GBR)	带位移量的 GBR 间接	存储区。区域的起始地址为 GBR 的值 + 位移量 (disp)。 GBR 的内容不变。
@(R0,GBR)	带变址的 GBR 间接	存储区。区域的起始地址为 GBR 的值 + R0 的值。 GBR 和 R0 的内容不变。
@(disp,PC)	带位移量的 PC 相对	存储区。区域的起始地址为 PC 的值 + 位移量 (disp)。
@@(disp,TBR)	带位移量的 TBR 双重间接	存储区。区域的起始地址为 TBR 的值 + 位移量 (disp)。 TBR 的内容不变。
symbol	符号指定的 PC 相对	【转移指令的操作数】 symbol 表示转移目标。汇编程序从 symbol 和 PC 的值推导出位移量 (disp)。disp=symbol-PC。 【数据传送指令的操作数】 存储区。Symbol 表示区域的起始地址。 汇编程序从 symbol 和 PC 的值中推导出位移量 (disp)。disp=symbol-PC。 【指定 RS,RE 寄存器的指令 (LDRS,LDRE) 的操作数】 存储区。Symbol 表示区域的起始地址。 汇编程序从 symbol 和 PC 的值推导出位移量 (disp)。disp=symbol-PC。
#imm	立即数	表示常数。

【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示位移量。

位移量的允许值因操作数的寻址方式和操作长度而不同。

表 11.9 位移量的允许值

寻址方式	位移量 (单位为字节, () 内是 10 进制数)
@(disp,Rn)	操作长度为字节 (B) 时 H'00000000 ~ H'0000000F (0 ~ 15) 操作长度为字 (W) 时 H'00000000 ~ H'0000001E (0 ~ 30) 操作长度为长字 (L) 时 H'00000000 ~ H'0000003C (0 ~ 60)
@(disp:12,Rn)	操作长度为字节 (B) 时 H'00000000 ~ H'00000FFF (0 ~ 4095) 操作长度为字 (W) 时 H'00000000 ~ H'00001FFE (0 ~ 8190) 操作长度为长字 (L) 时 H'00000000 ~ H'00003FFC (0 ~ 16380)
@(disp,GBR)	操作长度为字节 (B) 时 H'00000000 ~ H'000000FF (0 ~ 255) 操作长度为字 (W) 时 H'00000000 ~ H'000001FE (0 ~ 510) 操作长度为长字 (L) 时 H'00000000 ~ H'000003FC (0 ~ 1020)
@(disp,PC)	【数据传送指令的操作数】 操作长度为字 (W) 时 H'00000000 ~ H'000001FE (0 ~ 510) 操作长度为长字 (L) 时 H'00000000 ~ H'000003FC (0 ~ 1020) 【设定 RS、RE 寄存器的指令 (LDRS、LDRE) 的操作数】 H'FFFFFF00 ~ H'000000FE (-256 ~ 254)
@@(disp,TBR)	H'00000000 ~ H'000003FC (0 ~ 1020)
symbol	【转移指令的操作数】 条件转移指令 (BT、BF、BF/S、BT/S) 的操作数 H'00000000 ~ H'000000FF (0 ~ 255) H'FFFFFF00 ~ H'FFFFFFF (-256 ~ -1) 无条件转移指令 (BRA、BSR) 的操作数 H'00000000 ~ H'00000FFF (0 ~ 4095) H'FFFFFF00 ~ H'FFFFFFF (-4096 ~ -1) 【数据传送指令的操作数】 操作长度为字 (W) 时 H'00000000 ~ H'000001FE (0 ~ 510) 操作长度为长字 (L) 时 H'00000000 ~ H'000003FC (0 ~ 1020) 【设定 RS、RE 寄存器的指令 (LDRS、LDRE) 的操作数】 H'FFFFFF00 ~ H'000000FE (-256 ~ 254)

立即数的允许值因执行指令而不同。

表 11.10 立即数的允许值

执行指令	立即数
TST,AND,OR,XOR	H'00000000 ~ H'000000FF (0 ~ 255)
MOV	H'00000000 ~ H'000000FF (0 ~ 255) H'FFFFFF80 ~ H'FFFFFFF (-128 ~ -1) *1
ADD,CMP/EQ	H'00000000 ~ H'000000FF (0 ~ 255) H'FFFFFF80 ~ H'FFFFFFF (-128 ~ -1) *1
TRAPA	H'00000000 ~ H'000000FF (0 ~ 255)
SETRC,LDRCL	H'00000001 ~ H'000000FF (1 ~ 255) *2
MOVI20	H'00000000 ~ H'000FFFFF (0 ~ 1048575)
MOVI20S*3	H'00000000 ~ H'0FFFFFF0 (0 ~ 268435200)
位操作指令	H'00000000 ~ H'00000007 (0 ~ 7)

【注】 *1 也可用正的 10 进制数记述 H'FFFFFF80 ~ H'FFFFFFF 的范围。

*2 如果给 SETRC 和 LDRCL 指令的立即数置 0，就发生警告 835，并将目标码置 0。此时，执行 1 次重复的范围。另外，如果给 SETRC 和 LDRCL 指令的立即数值指定外部参照符号，优化连接编辑程序就检查 H'00000000 ~ H'000000FF (0 ~ 255) 的范围。

*3 如果立即数的低 8 位不为 0，就发生警告 845，舍去立即数的低 8 位并调整为 0。

【注】 汇编程序根据条件调整位移量。

条件	调整内容
操作长度为字	→ 舍去位移量的低 1 位，
位移量不为 2 的倍数	→ 调整为 2 的倍数
操作长度为长字	→ 舍去位移量的低 2 位，
位移量不为 4 的倍数	→ 调整为 4 的倍数
转移指令	→ 舍去位移量的低 1 位，
位移量不为 2 的倍数	→ 调整为 2 的倍数

作为操作数，记述 @(disp,Rn)、@(disp,GBR)、@@(disp,TBR)、@(disp,PC) 时，必须考虑调整汇编程序的位移量。

■编程例

```
MOV.L @(63,PC),R0
```

汇编程序在将位移量从 63 调整为 60 后生成和 MOV.L @(60,PC),R0 相同的目标码。另外，发出警告 870。

11.2.2 有关执行指令的注意事项

(1) 操作长度的注意事项

能指定的操作因助记符和寻址方式的组合而不同。

(a) SH-1 的执行指令和操作长度的组合

SH-1 的执行指令和操作长度的组合如表 11.11 所示。

表 11.11 执行指令和操作长度的组合 (1)

数据传送指令		操作长度			
助记符	寻址方式	B	W	L	省略时
MOV	#imm,Rn	○	△	△	B *1
MOV	@(disp,PC),Rn	×	○	○	L
MOV	symbol,Rn	×	○	○	L
MOV	Rn,Rm	×	×	○	L
MOV	Rn,@Rm	○	○	○	L
MOV	@Rn,Rm	○	○	○	L
MOV	Rn, @-Rm	○	○	○	L
MOV	@Rn+, Rm	○	○	○	L
MOV	R0, @(disp,Rn)	○	○	○	L
MOV	Rn, @(disp,Rm)	×	×	○	L *2
MOV	@(disp,Rn), R0	○	○	○	L
MOV	@(disp,Rn), Rm	×	×	○	L *3
MOV	Rn, @(R0,Rm)	○	○	○	L
MOV	@(R0,Rn), Rm	○	○	○	L
MOV	R0, @(disp,GBR)	○	○	○	L
MOV	@(disp,GBR), R0	○	○	○	L
MOVA	#imm, R0	×	×	△	L
MOVA	@(disp,PC), R0	×	×	○	L
MOVA	symbol,R0	×	×	○	L
MOV T	Rn	×	×	○	L
SWAP	Rn,Rm	○	○	×	W
XTRCT	Rn,Rm	×	×	○	L

【注】 *1 在为长度选择模式的情况下，汇编程序由 imm 的值决定操作长度。

*2 此时的 Rn 为 R1 ~ R15

*3 此时的 Rm 为 R1 ~ R15

表 11.11 执行指令和操作长度的组合 (2)

算术运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
ADD	Rn,Rm	×	×	○	L
ADD	#imm,Rn	×	×	○	L
ADDC	Rn,Rm	×	×	○	L
ADDV	Rn,Rm	×	×	○	L
CMP/EQ	#imm,R0	×	×	○	L
CMP/EQ	Rn,Rm	×	×	○	L
CMP/HS	Rn,Rm	×	×	○	L
CMP/GE	Rn,Rm	×	×	○	L
CMP/HI	Rn,Rm	×	×	○	L
CMP/GT	Rn,Rm	×	×	○	L
CMP/PZ	Rn	×	×	○	L
CMP/PL	Rn	×	×	○	L
CMP/STR	Rn,Rm	×	×	○	L
DIV1	Rn,Rm	×	×	○	L
DIV0S	Rn,Rm	×	×	○	L
DIV0U	(无操作数)	×	×	×	—
EXTS	Rn,Rm	○	○	×	W
EXTU	Rn,Rm	○	○	×	W
MAC	@Rn+,@Rm+	×	○	×	W
MULS	Rn,Rm	×	○	○	L *
MULU	Rn,Rm	×	○	○	L *
NEG	Rn,Rm	×	×	○	L
NEGC	Rn,Rm	×	×	○	L
SUB	Rn,Rm	×	×	○	L
SUBC	Rn,Rm	×	×	○	L
SUBV	Rn,Rm	×	×	○	L

【注】* W 和 L 为相同的目标码。

表 11.11 执行指令和操作长度的组合 (3)

逻辑运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
AND	Rn,Rm	×	×	○	L
AND	#imm,R0	×	×	○	L
AND	#imm,@(R0,GBR)	○	×	×	B
NOT	Rn,Rm	×	×	○	L
OR	Rn,Rm	×	×	○	L
OR	#imm,R0	×	×	○	L
OR	#imm,@(R0,GBR)	○	×	×	B
TAS	@Rn	○	×	×	B
TST	Rn,Rm	×	×	○	L
TST	#imm,R0	×	×	○	L
TST	#imm,@(R0,GBR)	○	×	×	B
XOR	Rn,Rm	×	×	○	L
XOR	#imm,R0	×	×	○	L
XOR	#imm,@(R0,GBR)	○	×	×	B

表 11.11 执行指令和操作长度的组合 (4)

移位指令		操作长度			
助记符	寻址方式	B	W	L	省略时
ROTL	Rn	×	×	○	L
ROTR	Rn	×	×	○	L
ROTCL	Rn	×	×	○	L
ROTCR	Rn	×	×	○	L
SHAL	Rn	×	×	○	L
SHAR	Rn	×	×	○	L
SHLL	Rn	×	×	○	L
SHLR	Rn	×	×	○	L
SHLL2	Rn	×	×	○	L
SHLR2	Rn	×	×	○	L
SHLL8	Rn	×	×	○	L
SHLR8	Rn	×	×	○	L
SHLL16	Rn	×	×	○	L
SHLR16	Rn	×	×	○	L

表 11.11 执行指令和操作长度的组合 (5)

转移指令		操作长度			
助记符	寻址方式	B	W	L	省略时
BF	symbol	×	×	×	—
BT	symbol	×	×	×	—
BRA	symbol	×	×	×	—
BSR	symbol	×	×	×	—
JMP	@Rn	×	×	×	—
JSR	@Rn	×	×	×	—
RTS	(无操作数)	×	×	×	—

表 11.11 执行指令和操作长度的组合 (6)

系统控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
CLRT	(无操作数)	×	×	×	—
CLRMAC	(无操作数)	×	×	×	—
LDC	Rn,SR	×	×	○	L
LDC	Rn,GBR	×	×	○	L
LDC	Rn,VBR	×	×	○	L
LDC	@Rn+,SR	×	×	○	L
LDC	@Rn+,GBR	×	×	○	L
LDC	@Rn+,VBR	×	×	○	L
LDS	Rn,MACH	×	×	○	L
LDS	Rn,MACL	×	×	○	L
LDS	Rn,PR	×	×	○	L
LDS	@Rn+,MACH	×	×	○	L
LDS	@Rn+,MACL	×	×	○	L
LDS	@Rn+,PR	×	×	○	L
NOP	(无操作数)	×	×	×	—
RTE	(无操作数)	×	×	×	—
SETT	(无操作数)	×	×	×	—
SLEEP	(无操作数)	×	×	×	—
STC	SR,Rn	×	×	○	L
STC	GBR,Rn	×	×	○	L
STC	VBR,Rn	×	×	○	L
STC	SR,@-Rn	×	×	○	L
STC	GBR,@-Rn	×	×	○	L
STC	VBR,@-Rn	×	×	○	L
STS	MACH,Rn	×	×	○	L
STS	MACL,Rn	×	×	○	L
STS	PR,Rn	×	×	○	L
STS	MACH,@-Rn	×	×	○	L
STS	MACL,@-Rn	×	×	○	L
STS	PR,@-Rn	×	×	○	L
TRAPA	#imm	×	×	○	L

【符号的含义】

Rm	: 通用寄存器 (R0 ~ R15)
Rn	: 通用寄存器 (R0 ~ R15)
R0	: 通用寄存器 (固定 R0)
SR	: 状态寄存器
GBR	: 全局基址寄存器
VBR	: 向量基址寄存器
MACH	: 乘加寄存器 (高位)
MACL	: 乘加寄存器 (低位)
PR	: 过程寄存器
PC	: 程序计数器
imm	: 立即数
disp	: 位移量
symbol	: 符号
B	: 字节
W	: 字 (2 个字节)
L	: 长字 (4 个字节)
○	: 指定有效
×	: 指定无效, 为和省略操作长度的指定相同的结果
△	: 将汇编语言程序解释为扩展指令

(b) SH-2 的执行指令和操作长度的组合

对于 SH-1, 由 SH-2 追加的执行指令和操作长度的组合如表 11.12 所示。

表 11.12 执行指令和操作长度的组合 (1)

算术运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
MAC	@Rn+, @Rm+	×	○	○	W
MUL	Rn, Rm	×	×	○	L
DMULS	Rn, Rm	×	×	○	L
DMULU	Rn, Rm	×	×	○	L
DT	Rn	×	×	×	—

表 11.12 执行指令和操作长度的组合 (2)

转移指令		操作长度			
助记符	寻址方式	B	W	L	省略时
BF/S	symbol	×	×	×	—
BT/S	symbol	×	×	×	—
BRAF	Rn	×	×	×	—
BSRF	Rn	×	×	×	—

(c) SH-2E 的执行指令和操作长度的组合

对于 SH-2，由 SH-2E 追加的执行指令和操作长度的组合如表 11.13 所示。

表 11.13 执行指令和操作长度的组合 (1)

数据传送指令		操作长度				
助记符	寻址方式	B	W	L	S	省略时
FLDI0	FRn	×	×	×	○	S
FLDI1	FRn	×	×	×	○	S
FMOV	@Rm,FRn	×	×	×	○	S
FMOV	FRn,@Rm	×	×	×	○	S
FMOV	@Rm+,FRn	×	×	×	○	S
FMOV	FRn,@-Rm	×	×	×	○	S
FMOV	@(R0,Rm),FRn	×	×	×	○	S
FMOV	FRm,@(R0,Rm)	×	×	×	○	S
FMOV	FRm,FRn	×	×	×	○	S

表 11.13 执行指令和操作长度的组合 (2)

算术运算指令		操作长度				
助记符	寻址方式	B	W	L	S	省略时
FABS	FRn	×	×	×	○	S
FADD	FRm,FRn	×	×	×	○	S
FCMP/EQ	FRm,FRn	×	×	×	○	S
FCMP/GT	FRm,FRn	×	×	×	○	S
FDIV	FRm,FRn	×	×	×	○	S
FMAC	FR0,FRm,FRn	×	×	×	○	S
FMUL	FRm,FRn	×	×	×	○	S
FNEG	FRn	×	×	×	○	S
FSUB	FRm,FRn	×	×	×	○	S

表 11.13 执行指令和操作长度的组合 (3)

系统控制指令		操作长度				
助记符	寻址方式	B	W	L	S	省略时
FLDS	FRm,FPUL	×	×	×	○	S
FLOAT	FPUL,FRn	×	×	×	○	S
FSTS	FPUL,FRn	×	×	×	○	S
FTRC	FRm,FPUL	×	×	×	○	S
LDS	Rm,FPUL	×	×	○	×	L
LDS	@Rm+,FPUL	×	×	○	×	L
LDS	Rm,FPSCR	×	×	○	×	L
LDS	@Rm+,FPSCR	×	×	○	×	L
STS	FPUL,Rn	×	×	○	×	L
STS	FPUL,@-Rn	×	×	○	×	L
STS	FPSCR,Rn	×	×	○	×	L
STS	FPSCR,@-Rn	×	×	○	×	L

【符号的含义】

FRm : 浮点寄存器
 FRn : 浮点寄存器
 FR0 : 浮点寄存器 (固定FR0)
 FPUL : 浮点通信寄存器
 FPSCR : 浮点状态/控制寄存器
 S : 单精度 (4个字节)

(d) SH-3 的执行指令和操作长度的组合

对于 SH-2, 由 SH-3 追加的执行指令和操作长度的组合如表 11.14 所示。

表 11.14 执行指令和操作长度的组合 (1)

数据传送指令		操作长度			
助记符	寻址方式	B	W	L	省略时
PREF	@Rn	×	×	×	—

表 11.14 执行指令和操作长度的组合 (2)

移位运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
SHAD	Rn, Rm	×	×	○	L
SHLD	Rn, Rm	×	×	○	L

表 11.14 执行指令和操作长度的组合 (3)

系统控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
CLRS	(无操作数)	×	×	×	—
SETS	(无操作数)	×	×	×	—
LDC	Rm, SSR	×	×	○	L
LDC	Rm, SPC	×	×	○	L
LDC	Rm, Rn_BANK	×	×	○	L
LDC	@Rm+, SSR	×	×	○	L
LDC	@Rm+, SPC	×	×	○	L
LDC	@Rm+, Rn_BANK	×	×	○	L
STC	SSR, Rn	×	×	○	L
STC	SPC, Rn	×	×	○	L
STC	Rm_BANK, Rn	×	×	○	L
STC	SSR, @-Rn	×	×	○	L
STC	SPC, @-Rm	×	×	○	L
STC	Rm_BANK, @-Rn	×	×	○	L
LDTLB	(无操作数)	×	×	×	—

【符号的含义】

Rn_BANK: 通用寄存器组
 SSR : 保存数据寄存器
 SPC : 保存程序计数器

(e) SH-4 的执行指令和操作长度的组合

对于 SH-3，由 SH-4 追加的执行指令和操作长度的组合如表 11.15 所示。

表 11.15 执行指令和操作长度的组合 (1)

数据传送指令		操作长度					省略时
助记符	寻址方式	B	W	L	S	D	
FLDI0	FRn	×	×	×	○	×	S
FLDI1	FRn	×	×	×	○	×	S
FMOV	FRm,FRn	×	×	×	○	×	S
FMOV	FRn,@Rm	×	×	×	○	×	S
FMOV	FRn,@-Rm	×	×	×	○	×	S
FMOV	FRn,@(R0,Rm)	×	×	×	○	×	S
FMOV	@Rm,FRn	×	×	×	○	×	S
FMOV	@Rm+,FRn	×	×	×	○	×	S
FMOV	@(R0,Rm),FRn	×	×	×	○	×	S
FMOV	DRm,DRn	×	×	×	×	○	D
FMOV	DRm,@Rn	×	×	×	×	○	D
FMOV	DRm@-Rn	×	×	×	×	○	D
FMOV	DRm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,DRn	×	×	×	×	○	D
FMOV	@Rm+,DRn	×	×	×	×	○	D
FMOV	@(R0,Rm),DRn	×	×	×	×	○	D
FMOV	DRm,XDn	×	×	×	×	○	D
FMOV	XDm,DRn	×	×	×	×	○	D
FMOV	XDm,XDn	×	×	×	×	○	D
FMOV	XDm,@Rn	×	×	×	×	○	D
FMOV	XDm,@-Rn	×	×	×	×	○	D
FMOV	XDm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,XDn	×	×	×	×	○	D
FMOV	@Rm+,XDn	×	×	×	×	○	D
FMOV	@(R0,Rm),XDn	×	×	×	×	○	D

表 11.15 执行指令和操作长度的组合 (2)

算术运算指令		操作长度					省略时
助记符	寻址方式	B	W	L	S	D	
FABS	FRn	×	×	×	○	×	S
FABS	DRn	×	×	×	×	○	D
FADD	FRm,FRn	×	×	×	○	×	S
FADD	DRm,DRn	×	×	×	×	○	D
FCMP/EQ	FRm,FRn	×	×	×	○	×	S
FCMP/EQ	DRm,DRn	×	×	×	×	○	D
FCMP/GT	FRm,FRn	×	×	×	○	×	S
FCMP/GT	DRm,DRn	×	×	×	×	○	D
FDIV	FRm,FRn	×	×	×	○	×	S
FDIV	DRm,DRn	×	×	×	×	○	D
FIPR	FVm,FVn	×	×	×	○	×	S
FMAC	FR0,FRm,FRn	×	×	×	○	×	S
FMUL	FRm,FRn	×	×	×	○	×	S
FMUL	DRm,DRn	×	×	×	×	○	D
FNEG	FRn	×	×	×	○	×	S
FNEG	DRn	×	×	×	×	○	D
FSQRT	FRn	×	×	×	○	×	S
FSQRT	DRn	×	×	×	×	○	D
FSUB	FRm,FRn	×	×	×	○	×	S
FSUB	DRm,DRn	×	×	×	×	○	D
FTRV	XMTRX,FVn	×	×	×	○	×	S

表 11.15 执行指令和操作长度的组合 (3)

系统控制指令		操作长度					省略时
助记符	寻址方式	B	W	L	S	D	
FCNVDS	DRm,FPUL	×	×	×	×	○	D
FCNVSD	FPUL,DRn	×	×	×	×	○	D
FLDS	FRm,FPUL	×	×	×	○	×	S
FLOAT	FPUL,FRn	×	×	×	○	×	S
FLOAT	FPUL,DRn	×	×	×	×	○	D
FRCHG	(无操作数)	×	×	×	×	×	—
FSCHG	(无操作数)	×	×	×	×	×	—
FSTS	FPUL,FRn	×	×	×	○	×	S
FTRC	FRm,FPUL	×	×	×	○	×	S
FTRC	DRm,FPUL	×	×	×	×	○	D
LDC	Rm,DBR	×	×	○	×	×	L
LDC	@Rm+,DBR	×	×	○	×	×	L
LDS	Rm,FPUL	×	×	○	×	×	L
LDS	@Rm+,FPUL	×	×	○	×	×	L
LDS	Rm,FPSCR	×	×	○	×	×	L
LDS	@Rm+,FPSCR	×	×	○	×	×	L
OCBI	@Rn	×	×	×	×	×	—
OCBP	@Rn	×	×	×	×	×	—
OCBWB	@Rn	×	×	×	×	×	—
STC	DBR,Rn	×	×	○	×	×	L
STC	DBR,@-Rn	×	×	○	×	×	L
STC	SGR,Rn	×	×	○	×	×	L
STC	SGR,@-Rn	×	×	○	×	×	L
STS	FPUL,Rm	×	×	○	×	×	L
STS	FPUL,@-Rm	×	×	○	×	×	L
STS	FPSCR,Rm	×	×	○	×	×	L
STS	FPSCR, @-Rm	×	×	○	×	×	L

【符号的含义】

- DRm : 双精度浮点寄存器
- DRn : 双精度浮点寄存器
- XDm : 双精度浮点扩展寄存器
- XDn : 双精度浮点扩展寄存器
- FVm : 单精度浮点向量寄存器
- FVn : 单精度浮点向量寄存器
- XMTRX : 单精度浮点扩展寄存器行列
- DBR : 调试向量基址寄存器
- SGR : 保存通用寄存器 15
- D : 双精度 (8个字节)

(f) SH2-DSP、SH3-DSP 的执行指令和操作长度的组合

对于 SH-2、SH2-DSP、SH-3，由 SH3-DSP 追加的执行指令和操作长度的组合如表 11.16 所示。

表 11.16 执行指令和操作长度的组合 (1)

重复控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
LDRS	@(disp, PC)	×	×	○	L
LDRS	symbol	×	×	○	L
LDRE	@(disp, PC)	×	×	○	L
LDRE	symbol	×	×	○	L
SETRC	Rn	×	×	×	—
SETRC	#imm	×	×	×	—

表 11.16 执行指令和操作长度的组合 (2)

系统控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
LDC	Rn, MOD	×	×	○	L
LDC	Rn, RS	×	×	○	L
LDC	Rn, RE	×	×	○	L
LDC	@Rn+, MOD	×	×	○	L
LDC	@Rn+, RS	×	×	○	L
LDC	@Rn+, RE	×	×	○	L
LDS	Rn, DSR	×	×	○	L
LDS	Rn, A0	×	×	○	L
LDS	@Rn+, DSR	×	×	○	L
LDS	@Rn+, A0	×	×	○	L
STC	MOD, Rn	×	×	○	L
STC	RS, Rn	×	×	○	L
STC	RE, Rn	×	×	○	L
STC	MOD, @-Rn	×	×	○	L
STC	RS, @-Rn	×	×	○	L
STC	RE, @-Rn	×	×	○	L
STS	DSR, Rn	×	×	○	L
STS	A0, Rn	×	×	○	L
STS	DSR, @-Rn	×	×	○	L
STS	A0, @-Rn	×	×	○	L

【符号的含义】

- MOD : 模寄存器
- RS : 重复开始寄存器
- RE : 重复结束寄存器
- DSR : DSP 状态寄存器
- A0 : DSP 数据寄存器 (除了 A0 以外, 能指定 A1、X0、X1、Y0、Y1)

(g) SH4AL-DSP 的执行指令和操作长度的组合

对于 SH3-DSP，由 SH4AL-DSP 追加的执行指令和操作长度的组合如表 11.17 所示。

表 11.17 执行指令和操作长度的组合 (1)

数据传送指令		操作长度			
助记符	寻址方式	B	W	L	省略时
CLRDMXY	(无操作数)	×	×	×	—
MOVCA	R0,@Rn	×	×	○	L
MOVCO	R0,@Rn	×	×	○	L
MOVLI	@Rn,R0	×	×	○	L
MOVUA	@Rn,R0	×	×	○	L
MOVUA	@Rn+,R0	×	×	○	L
PREFI	(无操作数)	×	×	×	—
SETDMX	(无操作数)	×	×	×	—
SETDMY	(无操作数)	×	×	×	—

表 11.17 执行指令和操作长度的组合 (2)

系统控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
ICBI	@Rn	×	×	○	L
LDC	Rn,DBR	×	×	○	L
LDC	@Rn+,DBR	×	×	○	L
LDC	Rn,SGR	×	×	○	L
LDC	@Rn+,SGR	×	×	○	L
OCBI	@Rn	×	×	×	—
OCBP	@Rn	×	×	×	—
OCBWB	@Rn	×	×	×	—
STC	DBR,Rn	×	×	○	L
STC	DBR,@-Rn	×	×	○	L
STC	SGR,Rn	×	×	○	L
STC	SGR,@-Rn	×	×	○	L
SYNCO	@Rn	×	×	○	L

表 11.17 执行指令和操作长度的组合 (3)

重复控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
LDRC	Rn	×	×	×	—

(h) SH-4A 的执行指令和操作长度的组合

对于 SH-4，由 SH-4A 追加的执行指令和操作长度的组合如表 11.18 所示。

表 11.18 执行指令和操作长度的组合 (1)

数据传送指令		操作长度			
助记符	寻址方式	B	W	L	省略时
MOVCO	R0,@Rn	×	×	○	L
MOVLI	@Rn,R0	×	×	○	L
MOVUA	@Rn,R0	×	×	○	L
MOVUA	@Rn+,R0	×	×	○	L
PREFI	(无操作数)	×	×	×	—

表 11.18 执行指令和操作长度的组合 (2)

系统控制指令		操作长度					
助记符	寻址方式	B	W	L	S	D	省略时
FPCHG	（无操作数）	×	×	×	×	×	—
ICBI	@Rn	×	×	×	×	×	—
LDC	Rn,SGR	×	×	○	×	×	L
LDC	@Rn+,SGR	×	×	○	×	×	L
FSCA	FPUL,DRn	×	×	×	○	×	S
FSRRA	FRn	×	×	×	○	×	S
SYNCO	@Rn	×	×	×	×	×	—

(i) SH-2A 的执行指令和操作长度的组合

对于 SH-2，由 SH-2A 追加的执行指令和操作长度的组合如表 11.19 所示。

表 11.19 执行指令和操作长度的组合 (1)

数据传送指令		操作长度			
助记符	寻址方式	B	W	L	省略时
MOV	R0,@Rn+	○	○	○	L
MOV	@-Rm,R0	○	○	○	L
MOV	Rm,@(disp:12,Rn)	○	○	○	L
MOV	@(disp:12,Rm),Rn	○	○	○	L
MOVI20	#imm20,Rn	×	×	○	L
MOVI20S	#imm20,Rn	×	×	○	L
MOVML	Rm,@-R15	×	×	○	L
MOVML	@R15+,Rn	×	×	○	L
MOVMU	Rm,@-R15	×	×	○	L
MOVMU	@R15+,Rn	×	×	○	L
MOVRT	Rn	×	×	○	L
MOVU	@(disp:12,Rm),Rn	○	○	×	W
NOTT	(无操作数)	×	×	×	—
PREF	@Rn	×	×	×	—

表 11.19 执行指令和操作长度的组合 (2)

算术运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
CLIPS	Rn	○	○	×	W
CLIPU	Rn	○	○	×	W
DIVS	R0,Rn	×	×	○	L
DIVU	R0,Rn	×	×	○	L
MULR	R0,Rn	×	×	○	L

表 11.19 执行指令和操作长度的组合 (3)

移位运算指令		操作长度			
助记符	寻址方式	B	W	L	省略时
SHAD	Rn, Rm	×	×	○	L
SHLD	Rn, Rm	×	×	○	L

表 11.19 执行指令和操作长度的组合 (4)

转移指令		操作长度			
助记符	寻址方式	B	W	L	省略时
JSR/N	@Rn	×	×	×	—
JSR/N	@@(disp,TBR)	×	×	×	—
RTS/N	(无操作数)	×	×	×	—
RTV/N	Rn	×	×	×	—

表 11.19 执行指令和操作长度的组合 (5)

系统控制指令		操作长度			
助记符	寻址方式	B	W	L	省略时
LDBANK	@Rm,R0	×	×	○	L
LDC	Rm,TBR	×	×	○	L
RESBANK	(无操作数)	×	×	×	—
STBANK	R0,@Rn	×	×	○	L
STC	TBR,Rn	×	×	○	L

表 11.19 执行指令和操作长度的组合 (6)

位操作指令		操作长度			
助记符	寻址方式	B	W	L	省略时
BAND	#imm,@(disp:12,Rn)	○	×	×	B
BANDNOT	#imm,@(disp:12,Rn)	○	×	×	B
BCLR	#imm,@(disp:12,Rn)	○	×	×	B
BCLR	#imm,Rn	×	×	○	L
BLD	#imm,@(disp:12,Rn)	○	×	×	B
BLD	#imm,Rn	×	×	○	L
BLDNOT	#imm,@(disp:12,Rn)	○	×	×	B
BOR	#imm,@(disp:12,Rn)	○	×	×	B
BORNOT	#imm,@(disp:12,Rn)	○	×	×	B
BSET	#imm,@(disp:12,Rn)	○	×	×	B
BSET	#imm,Rn	×	×	○	L
BST	#imm,@(disp:12,Rn)	○	×	×	B
BST	#imm,Rn	×	×	○	L
BXOR	#imm,@(disp:12,Rn)	○	×	×	B

【符号的含义】

TBR : 转移表基址寄存器

(j) SH2A-FPU 的执行指令和操作长度的组合

对于 SH-2E 和 SH-2A，由 SH2A-FPU 追加的执行指令和操作长度的组合如表 11.20 所示。

表 11.20 执行指令和操作长度的组合 (1)

数据传送指令		操作长度					
助记符	寻址方式	B	W	L	S	D	省略时
FMOV	DRm,DRn	×	×	×	×	○	D
FMOV	DRm,@Rn	×	×	×	×	○	D
FMOV	DRm@-Rn	×	×	×	×	○	D
FMOV	DRm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,DRn	×	×	×	×	○	D
FMOV	@Rm+,DRn	×	×	×	×	○	D
FMOV	@(R0,Rm),DRn	×	×	×	×	○	D
FMOV	@(disp:12,Rm),FRn	×	×	×	○	×	S
FMOV	@(disp:12,Rm),DRn	×	×	×	×	○	D
FMOV	FRm,@(disp:12,Rn)	×	×	×	○	×	S
FMOV	DRm,@(disp:12,Rn)	×	×	×	×	○	D

表 11.20 执行指令和操作长度的组合 (2)

算术运算指令		操作长度					
助记符	寻址方式	B	W	L	S	D	省略时
FABS	DRn	×	×	×	×	○	D
FADD	DRm,DRn	×	×	×	×	○	D
FCMP/EQ	DRm,DRn	×	×	×	×	○	D
FCMP/GT	DRm,DRn	×	×	×	×	○	D
FDIV	DRm,DRn	×	×	×	×	○	D
FMUL	DRm,DRn	×	×	×	×	○	D
FNEG	DRn	×	×	×	×	○	D
FSQRT	FRn	×	×	×	○	×	S
FSQRT	DRn	×	×	×	×	○	D
FSUB	DRm,DRn	×	×	×	×	○	D

表 11.20 执行指令和操作长度的组合 (3)

系统控制指令		操作长度					
助记符	寻址方式	B	W	L	S	D	省略时
FCNVDS	DRm,FPUL	×	×	×	×	○	D
FCNVSD	FPUL,DRn	×	×	×	×	○	D
FLOAT	FPUL,DRn	×	×	×	×	○	D
FSCHG	(无操作数)	×	×	×	×	×	—
FTRC	DRm,FPUL	×	×	×	×	○	D

(2) 有关延迟指令的注意事项

无条件的转移指令为延迟转移指令。

单片机在执行延迟转移指令前执行延迟槽指令（在存储器内位于紧接着转移指令之后的指令）。

在延迟槽指令无效的情况下，汇编程序发出错误 150 或者 151 的通知。

延迟转移指令和延迟槽指令的关系如表 11.21 所示。

表 11.21 延迟转移指令和延迟槽指令的关系

延迟槽指令		延迟转移指令									
		BF/S	BT/S	BRAF	BSRF	BRA	BSR	JMP	JSR	RTS	RTE
BF		×	×	×	×	×	×	×	×	×	×
BT		×	×	×	×	×	×	×	×	×	×
BF/S		×	×	×	×	×	×	×	×	×	×
BT/S		×	×	×	×	×	×	×	×	×	×
BRAF		×	×	×	×	×	×	×	×	×	×
BSRF		×	×	×	×	×	×	×	×	×	×
BRA		×	×	×	×	×	×	×	×	×	×
BSR		×	×	×	×	×	×	×	×	×	×
JMP		×	×	×	×	×	×	×	×	×	×
JSR		×	×	×	×	×	×	×	×	×	×
JSR/N		×	×	×	×	×	×	×	×	×	×
RTS		×	×	×	×	×	×	×	×	×	×
RTS/N		×	×	×	×	×	×	×	×	×	×
RTE		×	×	×	×	×	×	×	×	×	×
RTV/N		×	×	×	×	×	×	×	×	×	×
TRAPA		×	×	×	×	×	×	×	×	×	×
DIVS		×	×	×	×	×	×	×	×	×	×
DIVU		×	×	×	×	×	×	×	×	×	×
LDC	Rn,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
	@Rn+,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
MOV	@(disp,PC),Rn	×	×	×	×	×	×	×	×	×	×
	symbol,Rn	×	×	×	×	×	×	×	×	×	×
MOVA	@(disp,PC),R0	×	×	×	×	×	×	×	×	×	×
	symbol,R0	×	×	×	×	×	×	×	×	×	×
LDRS	@(disp,PC)	×	×	×	×	×	×	×	×	×	×
	symbol	×	×	×	×	×	×	×	×	×	×
LDRE	@(disp,PC)	×	×	×	×	×	×	×	×	×	×
	symbol	×	×	×	×	×	×	×	×	×	×
扩展指令	MOV.L #imm,Rn	×	×	×	×	×	×	×	×	×	×
	MOV.W #imm,Rn	×	×	×	×	×	×	×	×	×	×
	MOVA #imm,R0	×	×	×	×	×	×	×	×	×	×
32 位指令 *2		×	×	×	×	×	×	×	×	×	×
寄存器组关联指令 *3		×	×	×	×	×	×	×	×	×	×
上述以外的指令		○	○	○	○	○	○	○	○	○	○

【符号的含义】

- : 正常 汇编程序按照指定生成目标码
- × : 错误 150 或者 151
延迟槽指令无效
汇编程序生成 NOP 指令的目标码 (H'0009)
- *1 : 在单片机种类为 SH-1、SH-2、SH-2E、SH2-DSP、SH-2A、SH2A-FPU 时正常
否则为错误 150 或者 151
- *2 : 32 位指令
给操作数指定带 12 位位移量的寄存器间接或者 20 位立即数的指令
MOVI20、MOVI20S 指令
- *3 : 寄存器组关联指令
RESBANK、LDBANK、STBANK 指令

【注】 在延迟转移指令和延迟槽指令附属于不同的段时，汇编程序就不检查延迟槽指令的合法性。

(3) 有关地址计算的注意事项

在操作数的寻址方式为带位移量的 PC 相对 @(disp,PC) 时，需要考虑 PC 的值后进行编程。
PC 值的变化因情况而不同。

(a) 通常

PC 的值为 (执行中的指令的起始地址 +4) 字节。

■例

假设正在执行绝对地址 H'00001000 的 MOV 指令。

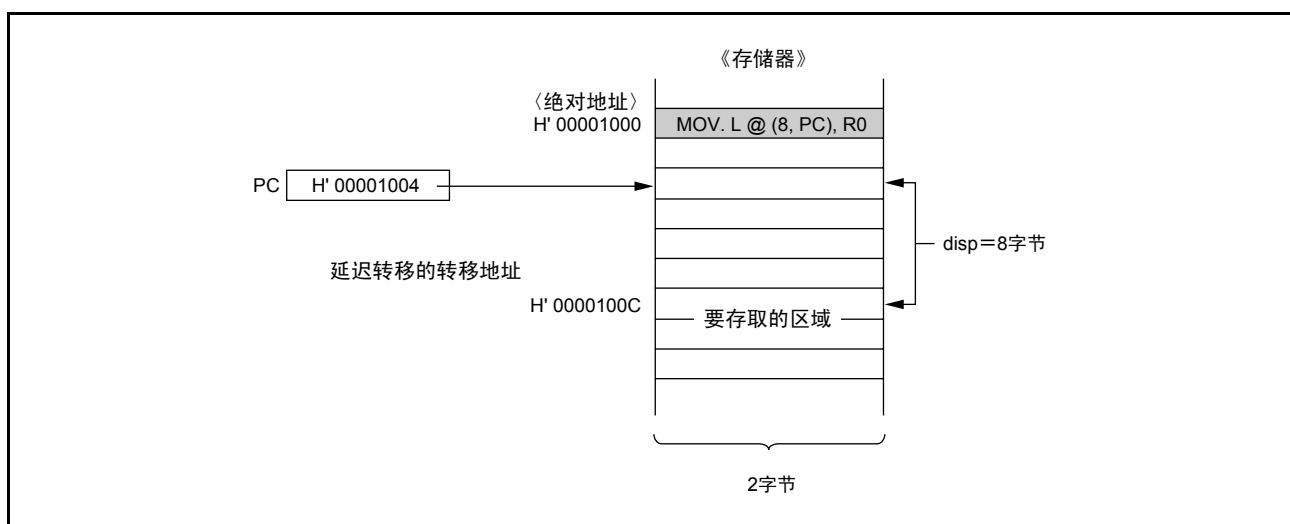


图 11.1 地址计算的例子 (通常)

(b) 延迟槽指令执行中

PC 的值为（延迟转移指令的转移地址 +2）字节。

■例

假设正在执行绝对地址 H'00001000 的 MOV 指令。

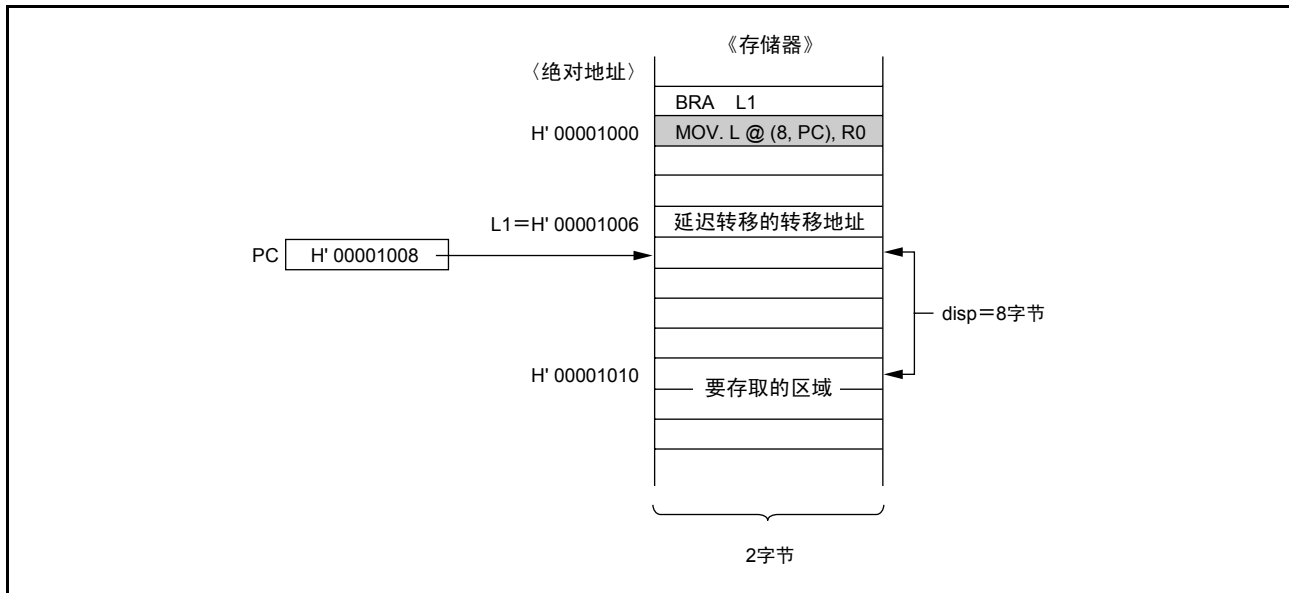


图 11.2 地址计算的例子（因转移而发生 PC 的变化）

【补充】 在操作数为符号指定的 PC 相对（symbol）时，汇编程序根据 PC 的值推导位移量，生成目标码。

(c) 执行 MOV.L @(disp,PC),Rn 或者 MOVA @(disp,PC),R0

在 PC 的值不为 4 的倍数时，单片机舍去低 2 位，调整为 4 的倍数，并计算地址。

■例 1 单片机调整 PC

假设正在执行地址 H'00001002 的 MOV 指令。

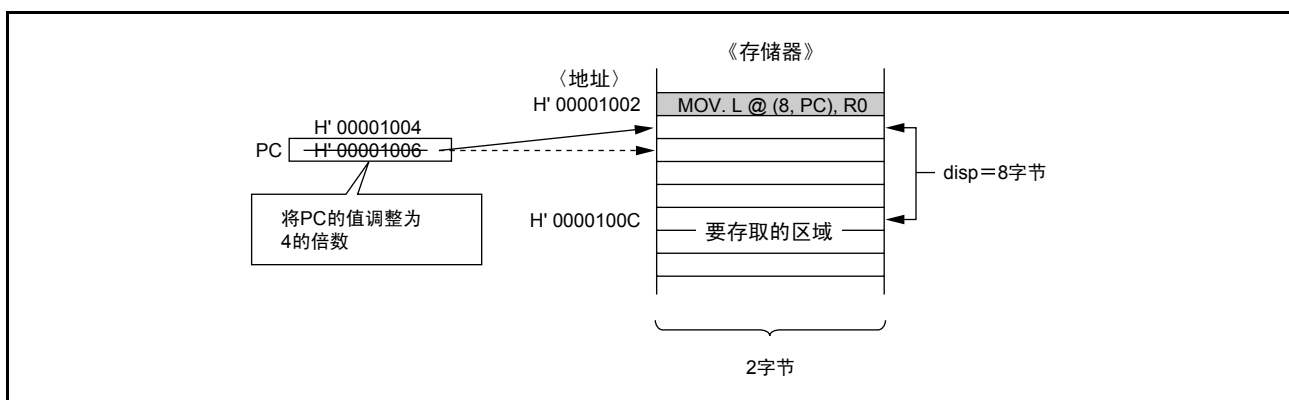


图 11.3 地址计算的例子（单片机调整 PC）

■例 2 单片机调整 PC

假设正在执行地址 H'00001000 的 MOV 指令。

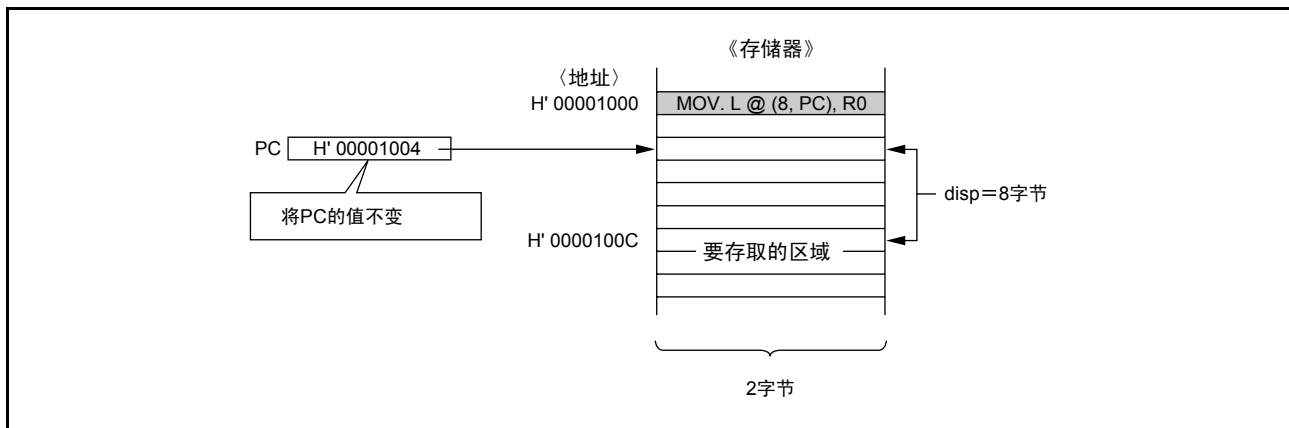


图 11.4 地址计算的例子（单片机不调整 PC）

【补充】 在操作数为符号指定的 PC 相对（symbol）时，汇编程序根据 PC 的值推导位移量，并生成目标码。

11.3 DSP 指令

11.3.1 程序的元素

(1) 源语句

SH2-DSP、SH3-DSP、SH4AL-DSP 的指令能分为执行指令和 DSP 指令。DSP 指令是操作 DSP 寄存器的指令。DSP 指令的指令体系和执行指令不同，记述方法也不同。

DSP 指令能在 1 条语句内记述多个操作。

DSP 指令的操作有以下几种：

(a) DSP 运算操作

指定 DSP 寄存器之间运算的操作如下：

PABS、PADD、PADDC、PAND、PCLR、PCMP、PCOPY、PDEC、PDMSB、PINC、PLDS、PMULS、PNEG、POR、PRND、PSHA、PSHL、PSTS、PSUB、PSUBC、PSWAP、PXOR

(b) X 数据的传送操作

指定 DSP 寄存器和 X 数据存储器之间数据传送的操作如下：

MOVX、NOPX

(c) Y 数据传送操作

指定 DSP 寄存器和 Y 数据存储器之间数据传送的操作如下：

MOVY、NOPY

(d) 单数据的传送操作

指定没有限定 X 数据存储器、Y 数据存储器的一般存储器和 DSP 寄存器之间数据传送的操作如下：

MOVS

(2) 并行运算指令

并行运算指令是在和 DSP 运算的同时进行 DSP 寄存器和 X 数据存储器、Y 数据存储器之间的数据传送。指令的长度为 32 位。

并行运算指令的构成如下所示：

[< 标号 >][△ <DSP 运算部 >][△ < 数据传送部 >][< 注释 >]

(a) DSP 运算部的记述方法

DSP 运算部的结构如下所示：

[< 条件 > △]<DSP 运算操作 > △ < 操作数 >[△…]

• 条件

条件是执行并行运行指令的条件。

条件如下：

DCT、DCF

DCT 在 DC 位是 1 时执行指令。

DCF 在 DC 位是 0 时执行指令。

- DSP 运算操作

指定 DSP 运算。只在 PADD 和 PMULS、PCLR 和 PMULS、PSUB 和 PMULS 组合时，才能指定 2 个 DSP 运算操作。

(b) 数据传送部的记述方法

数据传送部的结构如下所示：

[<X 数据传送操作>[△<操作数>]]

[△<Y 数据传送操作>[△<操作数>]]

按照此顺序指定 X 数据存储器的传送和 Y 数据存储器的传送。

在数据传送的操作为 NOPX 或者 NOPY 时能省略。

■ 编程例

```

LABEL1: PADD  A0,M0,A0  PMULS  X0,Y0,M0  MOVX.W  @R4+,X0  MOVY.W  @R6+,Y0 ; DSP 指令
标号      DSP 运算部      数据传送部      注释

```

```

      DCT  PINC X1,A1  MOVX.W @R4,X0  MOVY.W @R6+,Y0
      DSP  运算部      数据传送部

```

```

      PCMP X1,M0  MOVX.W @R4,X0      ; 省略 Y 存储器
      DSP  运算部  数据传送部      注释

```

(3) 数据传送指令

数据传送指令指定 X 数据存储器传送和 Y 数据存储器传送的组合或者单数据传送。

(a) X 数据存储器传送和 Y 数据存储器传送的组合

X 数据存储器传送和 Y 数据存储器传送的组合结构如下所示：

[<标号>][△<X 数据传送操作>[△<操作数>]]

[△<Y 数据传送操作>[△<操作数>]][<注释>]

按照此顺序，指定 X 数据存储器的传送和 Y 数据存储器的传送。在数据传送的操作为 NOPX 或者 NOPY 时能省略。但是，和并行运算指令不同，不能将 X 数据存储器的传送和 Y 数据存储器的传送都省略。

X 数据存储器的传送指令和 Y 数据存储器的传送指令的记述例子如下所示。

■ 编程例

```

LABEL2:  MOVX.W @R4,X0      ; 数据传送指令（省略 Y 数据存储器的传送）
      MOVX.W @R4,X0  MOVY.W @R6+,Y0

```

(b) 单数据传送指令

单数据传送指令的结构如下所示。

[< 标号 >][△ < 单数据传送操作 > △ < 操作数 >][< 注释 >]

指定 MOVS 指令。

单数据传送指令的记述例子如下所示。

■ 编程例

LABEL3: MOVS.W @-R2,A0 ; 单数据传送

(4) 跨多行的源语句的记述方法

因为 DSP 指令 1 行能记述多个操作，所以源语句变长，变为难读的程序。在此，DSP 指令除了分开操作数的逗号 (,) 以外，还能分开操作数和操作。

必须按照以下 (a) ~ (c) 的步骤记述跨多行的源语句：

- (a) 在操作数和操作之间作为一个段落进行换行。
- (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。

■ 编程例

	PADD	A0, M0, X0
+	PMULS	A1, Y1, M0
+	MOVX	@R4, X0
+	MOVY	@R6, Y1

; 跨 4 行写 1 条源语句的例子。

11.3.2 DSP 指令

(1) DSP 运算指令

DSP 运算指令的助记符如表 11.22 所示。

表 11.22 DSP 运算指令助记符的分类

分类	助记符
DSP 算术运算指令	PADD、PSUB、PCOPY、PDMSB、PINC、PNEG、PMULS、PADDC、PSUBC、PCMP、PDEC、PABS、PRND、PCLR、PLDS、PSTS、PSWAP
DSP 逻辑运算指令	POR、PAND、PXOR
DSP 移位运算命令	PSHA、PSHL

(a) 操作长度

不能给 DSP 运算指令指定操作长度。

(b) 寻址方式

DSP 运算指令的寻址方式如表 11.23 所示。

表 11.23 DSP 运算指令的寻址方式

寻址方式	记述方法
DSP 寄存器直接	Dp(DSP 寄存器名)
立即数	#imm

- DSP 寄存器直接

能指定为 DSP 寄存器直接的寄存器如表 11.24 所示。

表中 Sx、Sy、Dz、Du、Se、Sf、Dg 请参照“表 11.26 DSP 运算指令一览表”。

表 11.24 能指定为 DSP 寄存器直接的寄存器

		DSP 寄存器							
		A0	A1	M0	M1	X0	X1	Y0	Y1
Dp	Sx	○	○			○	○		
	Sy			○	○			○	○
	Dz	○	○	○	○	○	○	○	○
	Du	○	○			○		○	
	Se		○			○	○	○	
	Sf		○			○		○	○
	Dg	○	○	○	○				

- 立即数

只能将立即数指定为 PSHA、PSHL 指令的第 1 个操作数，能指定的值如下：

值的种类：能指定常数、符号或者表达式。

符号种类：立即数包含相对地址符号和外部参照符号，并能指定任意符号。

值的范围：能指定值的范围如表 11.25 所示。

表 11.25 立即数值的范围

指令	数值的范围
PSHA 指令	H'FFFFFFE0 ~ H'00000020 (-32 ~ 32)
PSHL 指令	H'FFFFFFF0 ~ H'00000010 (-16 ~ 16)

(c) 多条 DSP 运算指令的组合

能指定 PADD 指令和 PMULS 指令、PCLR 指令和 PMULS 指令或者 PSUB 指令和 PMULS 指令的组合。

此指令的组合原来是一条 DSP 运算指令，但是为了使程序易读，可分开 PADD 指令、PCLR 指令、PSUB 指令的操作数和 PMULS 指令的操作数进行记述。

多条 DSP 运算指令的组合例子如下所示：

■编程例

PADD A0, M0, A0	PMULS X0, Y0, M0	NOPX	MOVY.W@R6+, Y0
PCLR A1	PMULS X0, Y0, M0	NOPX	MOVY.W@R6+, Y0
PSUB A1, M1, A1	PMULS X1, Y1, M1	MOVX @R4+, X0	NOPY

【注】 如果通过多条 DSP 运算指令的组合将同一寄存器指定为目标寄存器，就发生警告 701。

PADD A0, M0, A0 PMULS X0, Y0, A0 → 警告 701

(d) 带条件的 DSP 运算指令

DSP 运算指令有带条件的运算指令，能根据 DSR 寄存器的 DC 位的状态决定是否执行。条件 DCT 只在 DC 位为 1 时执行指令；条件 DCF 只在 DC 位为 0 时执行指令。

带条件的 DSP 运算指令如下：

PABS、PADD、PAND、PCLR、PCOPY、PDEC、PDMSB、PINC、PLDS、PNEG、POR、PRND、PSHA、PSHL、PSTS、PSUB、PSWAP、PXOR

(e) DSP 运算指令一览表

SH2-DSP、SH3-DSP 的 DSP 运算指令如表 11.26 所示，对于 SH2-DSP、SH3-DSP，由 SH4AL-DSP 追加的 DSP 运算指令的一览表如表 11.27 所示。能指定为表中 Sx、Sy、Dz、Du、Se、Sf、Dg 的寄存器请参照“表 11.24 能指定为 DSP 寄存器直接的寄存器”。

表 11.26 DSP 运算指令一览表

助记符	寻址方式	助记符	寻址方式
PABS	Sx,Dz	—	—
PABS	Sy,Dz	—	—
PADD	Sx,Sy,Dz	—	—
PADD	Sx,Sy,Du	PMULS	Se,Sf,Dg
PADDC	Sx,Sy,Dz	—	—
PAND	Sx,Sy,Dz	—	—
PCLR	Dz	—	—
PCMP	Sx,Sy	—	—
PCOPY	Sx,Dz	—	—
PCOPY	Sy,Dz	—	—
PDEC	Sx,Dz	—	—
PDEC	Sy,Dz	—	—
PDMSB	Sx,Dz	—	—
PDMSB	Sy,Dz	—	—
PINC	Sx,Dz	—	—
PINC	Sy,Dz	—	—
PLDS	Dz,MACH	—	—
PLDS	Dz,MACL	—	—
PMULS	Se,Sf,Dg	—	—
PNEG	Sx,Dz	—	—
PNEG	Sy,Dz	—	—
POR	Sx,Sy,Dz	—	—
PRND	Sx,Dz	—	—
PRND	Sy,Dz	—	—
PSHA	#imm,Dz	—	—
PSHA	Sx,Sy,Dz	—	—
PSHL	#imm,Dz	—	—
PSHL	Sx,Sy,Dz	—	—
PSTS	MACH,Dz	—	—
PSTS	MACL,Dz	—	—
PSUB	Sx,Sy,Dz	—	—
PSUB	Sx,Sy,Du	PMULS	Se,Sf,Dg
PSUBC	Sx,Sy,Dz	—	—
PXOR	Sx,Sy,Dz	—	—

表 11.27 DSP 运算命令一览

助记符	寻址方式	助记符	寻址方式
PCLR	Du	PMULS	Se,Sf,Dg
PSUB	Sy,Sx,Dz	—	—
PSWAP	Sx,Dz	—	—
PSWAP	Sy,Dz	—	—

(2) 数据传送指令

(a) 助记符

数据传送指令有双存储器传送和单存储器传送。双存储器传送同时进行 X 存储器和 DSP 寄存器之间、Y 存储器和 DSP 寄存器之间的数据传送。单存储器传送进行任意存储器和 DSP 寄存器之间的数据传送。

数据传送指令的助记符一览表表 11.28 所示。

表 11.28 数据传送指令的助记符一览表

	分类	助记符
双存储器传送	X 存储器传送	NOPX
		MOVX
	Y 存储器传送	NOPY
		MOVY
单存储器传送		MOVS

(b) 操作长度

NOPX、NOPY 指令不能指定长度。

MOVX、MOVY 指令能指定字 (W) 和长字 (L)。如果不指定长度, 就解释为字。

MOVS 指令能指定字 (W) 和长字 (L)。如果不指定长度, 就解释为长字。

(c) 寻址方式

数据传送指令可指定的寻址方式如表 11.29 所示。

表 11.29 数据传送指令的寻址方式

寻址方式	记述方法
DSP 寄存器直接	Dz
寄存器间接	@Az
后增寄存器间接	@Az+
带变址的寄存器间接 / 后增	@Az+Iz
先减寄存器间接	@-Az

在寻址方式中, 带变址的寄存器间接 / 后增是 DSP 数据传送指令特有的寻址方式。此寻址方式表示在参照寄存器 Az 所指的内容后只对 Az 的内容增加 Iz 的值。

(d) 能指定为寻址方式的寄存器

能指定为 DSP 寄存器直接、寄存器间接、后增寄存器间接、带变址的寄存器间接 / 后增、先减间接的寄存器如表 11.30 所示。

有关表中的 Dx、Dy、Dxy、Ds、Da、Ax、Ax2、Ay、Ay2、As、Ix、Iy、Is，请参照“表 11.31 数据传送指令一览表”。

表 11.30 能指定为数据传送指令寻址方式的寄存器

		通用寄存器										DSP 寄存器									
		R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	A0	A1	M0	M1	X0	X1	Y0	Y1	A0G	A1G
Dz	Dx															○	○				
	Dy																	○	○		
	Dxy															○	○	○	○		
	Ds											○	○	○	○	○	○	○	○	○	○
	Da											○	○								
Az	Ax					○	○														
	Ax2	○	○			○	○														
	Ay							○	○												
	Ay2			○	○			○	○												
	As			○	○	○	○														
Iz	Ix									○											
	Iy										○										
	Is									○											

【注】如果在同一指令内的 DSP 运算指令和数据传送指令之间指定同一个目标寄存器，就发生警告 703。

PADD A0,M0,Y0 NOPX MOVY.W @R6+,Y0 → 警告 703

(e) 数据传送指令一览表

SH2-DSP、SH3-DSP 的数据传送指令如表 11.31 所示，对于 SH2-DSP、SH3-DSP，由 SH4AL-DSP 追加的数据传送指令一览表如表 11.32 所示。

能指定为表中 Dx、Dy、Dxy、Ds、Da、Ax、Ax2、Ay、Ay2、As、Ix、Iy、Is 的寄存器请参照“表 11.30 能指定为数据传送指令寻址方式的寄存器”。

表 11.31 数据传送指令一览表

种类	助记符	寻址方式
X 数据传送指令	NOPX	—
	MOVX.W	@Ax,Dx
	MOVX.W	@Ax+,Dx
	MOVX.W	@Ax+Ix,Dx
	MOVX.W	Da,@Ax
	MOVX.W	Da,@Ax+
	MOVX.W	Da,@Ax+Ix
Y 数据传送指令	NOPY	—
	MOVY.W	@Ay,Dy
	MOVY.W	@Ay+,Dy
	MOVY.W	@Ay+Iy,Dy
	MOVY.W	Da,@Ay
	MOVY.W	Da,@Ay+
	MOVY.W	Da,@Ay+Iy
单数据传送指令	MOVS.W	@-As,Ds
	MOVS.W	@As,Ds
	MOVS.W	@As+,Ds
	MOVS.W	@As+Is,Ds
	MOVS.W	Ds,@-As
	MOVS.W	Ds,@As
	MOVS.W	Ds,@As+
	MOVS.W	Ds,@As+Is
	MOVS.L	@-As,Ds
	MOVS.L	@As,Ds
	MOVS.L	@As+,Ds
	MOVS.L	@As+Is,Ds
	MOVS.L	Ds,@-As
	MOVS.L	Ds,@As
	MOVS.L	Ds,@As+
	MOVS.L	Ds,@As+Is

表 11.32 数据传送指令一览表

种类	助记符	寻址方式
X 数据传送指令	MOVX.W	@Ax2,Dxy
	MOVX.W	@Ax2+,Dxy
	MOVX.W	@Ax2+R8,Dxy
	MOVX.L	@Ax2,Dxy
	MOVX.L	@Ax2+,Dxy
	MOVX.L	@Ax2+R8,Dxy
Y 数据传送指令	MOVY.W	@Ay2,Dxy
	MOVY.W	@Ay2+,Dxy
	MOVY.W	@Ay2+R9,Dxy
	MOVY.L	@Ay2,Dxy
	MOVY.L	@Ay2+,Dxy
	MOVY.L	@Ay2+R9,Dxy

11.4 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 11.33 所示。

表 11.33 汇编程序控制指令一览表

分类	助记符	功能
有关单片机的指令	.CPU	指定单片机种类。
有关段或者 定位计数器的指令	.SECTION	声明段。
	.ORG	设定定位计数器值。
	.ALIGN	将定位计数器值调整为调整数的倍数。
有关符号的指令	.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。
	.FREG	定义浮点寄存器名。
确保数据或者数据区的指令	.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。
	.FDATA	确保浮点数据。
	.FDATAB	确保浮点块。
	.XDATA	确保定点数据。
	.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。
	.FRES	确保浮点数据区。
有关外部定义或者 外部参照的指令	.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。
有关目标模块的指令	.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。
	.ENDIAN	指定 Endian 种类。
	.LINE	更改行号。
有关汇编列表的指令	.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	设定源程序列表的页眉。
	.PAGE	将源程序列表换页。
	.SPACE	将空行输出到源程序列表。
其他	.PROGRAM	设定目标模块名。
	.RADIX	指定无基数的整数常数为几进制数。
	.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。

【注】 给编译程序输出的汇编源程序记述 .FILE。 .FILE 只在编译程序输出的汇编源程序中有效。不能在用户建立的汇编程序中使用。

.CPU

格 式 △ .CPU △ < 单片机种类 >

单片机种类	对象单片机
SH1	汇编的源程序对象为 SH-1。
SH2	汇编的源程序对象为 SH-2。
SH2E	汇编的源程序对象为 SH-2E。
SH2A	汇编的源程序对象为 SH-2A。
SH2AFPU	汇编的源程序对象为 SH2A-FPU。
SHDSP	汇编的源程序对象为 SH2-DSP。
SH3	汇编的源程序对象为 SH-3。
SH3DSP	汇编的源程序对象为 SH3-DSP。
SH4	汇编的源程序对象为 SH-4。
SH4A	汇编的源程序对象为 SH-4A。
SH4ALDSP	汇编的源程序对象为 SH4AL-DSP。

不能记述标号。

说 明

指定要汇编的源程序对象的单片机种类。

必须在源程序开始处记述 .CPU。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 .CPU，就发生错误。

.CPU 的指定只限 1 次有效。单片机种类的优先顺序为 cpu 选项、.CPU、SHCPU 环境变量的顺序。如果省略指定，SHCPU 环境变量设定的单片机种类有效。

例

```
.CPU SH2      ; 汇编的源程序对象为 SH-2。
.SECTION A, CODE, ALIGN=4
MOV.L        R0, R1
MOV.L        R0, R2
```

段的声明

.SECTION

格 式 Δ .SECTION Δ < 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 = < 调整数 > }

不能记述标号。

说 明

指定段的声明和重新开始。

段是程序的 1 个分隔，是连接的处理单位。

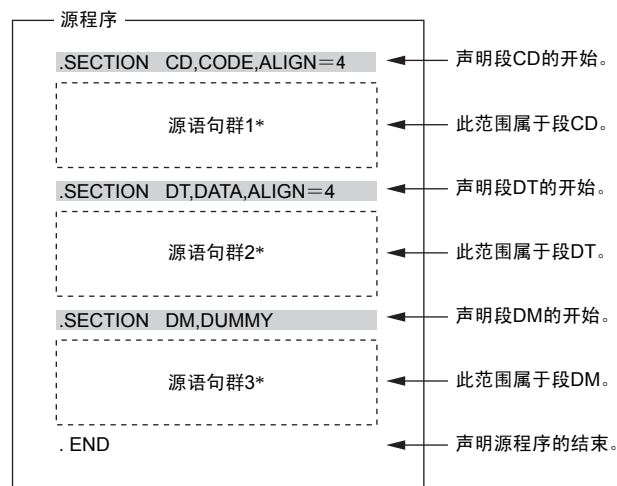
段属性如下：

- CODE 代码段
- DATA 数据段
- STACK 堆栈段
- DUMMY 虚设段

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=< 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调整数的倍数的绝对地址。在没有指定格式种类时，设定 ALIGN=4。

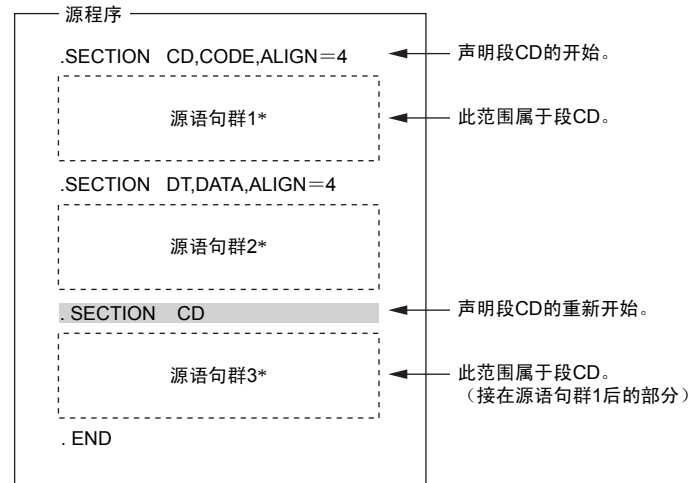
简单举例说明有关段的声明。



【注】* 在此例中，假设“源语句群1~3”中不出现 SECTION。

在同文件中，能重新声明和重新开始已声明的段。

简单地举例说明有关段的重新开始



【注】* 在此例中，假设“源语句群1~3”中不出现SECTION。

在重新开始段时，必须省略第 2 操作数和第 3 操作数的指定（开始该段时的指定有效）。在开始绝对地址的段时，必须给第 3 操作数指定“LOCATE= 起始地址”。起始地址是该段开始的绝对地址。起始地址的指定如下：

- 指定常数值。

并且

- 指定时，不使用前方参照符号。

起始地址的允许值为 H'00000000 ~ H'FFFFFFFF（10 进制表示为 0 ~ 4,294,967,295）。

在开始相对地址的段，必须给第 3 操作数指定“ALIGN= 调整数”。优化连接编辑程序调整段的起始地址，使段的起始地址为调整数的倍数相对地址。调整数的指定如下：

- 指定常数值。

并且

- 指定时，不使用前方参照符号。

调整数的允许值为 2 的乘方（ 2^0 、 2^1 、 2^2 、……、 2^{31} ）。

代码段的情况为不小于 4 的值（ 2^2 、 2^3 、 2^4 、……、 2^{31} ）。

在以下任一情况下，汇编程序提供默认段。

- 在不声明段时记述了执行指令、扩展指令和 DSP 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。
- 在不声明段时记述了 .ORG。
- 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。

默认段为如下的段：

- 段名：P
- 段的种类：代码段、相对地址段（调整数 = 4）

例

<pre> ALIGN 4 DATA.L H'11111111 ~ </pre>	<pre> ; 此范围属于默认段P。 ; 默认段P为代码段，并且为 ; 调整数=4的相对地址段。 </pre>
<pre>SECTION CD, CODE, ALIGN=2</pre>	
<pre> MOV R0, R1 MOV R0, R2 ~ </pre>	<pre> ; 此范围属于段CD。 ; 段CD为代码段，并且为 ; 调整数=4的相对地址段。 </pre>
<pre>SECTION DT, DATA, LOCATE=H' 00001000</pre>	
<pre> X1: DATA.L H'22222222 DATA.L H'33333333 ~ </pre>	<pre> ; 此范围属于段DT。 ; 段DT为数据段，并且为 ; 起始地址 = H'00001000的绝对地址段。 </pre>

在此例中，假设“~”的部分不出现 `SECTION`。

定位计数器值的设定

.ORG

格 式 Δ .ORG Δ < 定位计数器值 >
不能记述标号。

说 明 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

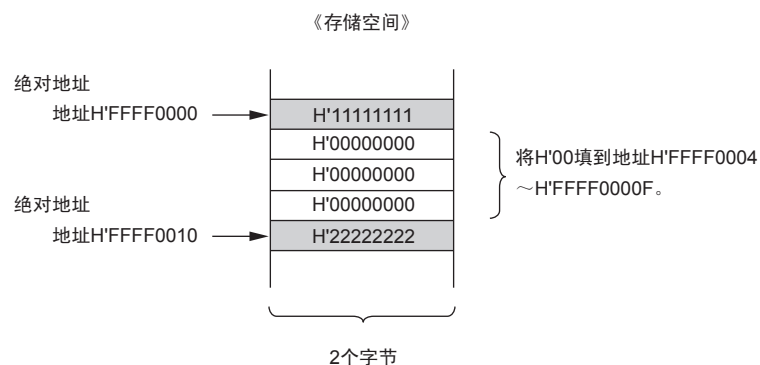
- 指定时，不使用前方参照符号。

定位计数器的允许值为 H'00000000 ~ H'FFFFFFF (10 进制表示为 0 ~ 4,294,967,295)。

在用绝对地址段指定时，必须设定为定位计数器值 $\geq$ 段的起始地址 (比较无符号的值)。
汇编程序将定位计数器的值视为如下的内容：

- 绝对地址段内的绝对地址
- 相对地址段内的相对地址 (从段起始地址开始的相对距离)

例 .SECTION DT, DATA, LOCATE=H'FFFF0000
 .DATA.L H'11111111
 .ORG H'FFFF0010 ; 设定定位计数器值。
 .DATA.L H'22222222 ; 将整数数据 H'22222222 保存在
 ~ ; 绝对地址 H'FFFF0010 中。



定位计数器值的调整

.ALIGN

格 式 Δ .ALIGN Δ < 调整数 >
不能记述标号。

说 明 将段内的定位计数器值调整为调整数的倍数。
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。

调整数的允许值为 2 的乘方（ 2^0 、 2^1 、 2^2 、……、 2^{31} ）。
在相对地址段使用 .ALIGN 时，必须设定为
 . SECTION 指定的调整数 $\geq$.ALIGN 指定的调整数。
如果在代码段、数据段中记述 .ALIGN，汇编程序就将 NOP 指令的目标码 * 填到存储器，
调整定位计数器值。将 H'09 填到不完全的字节长的区域。
【注】* 这样的目标码不表示在汇编列表中。

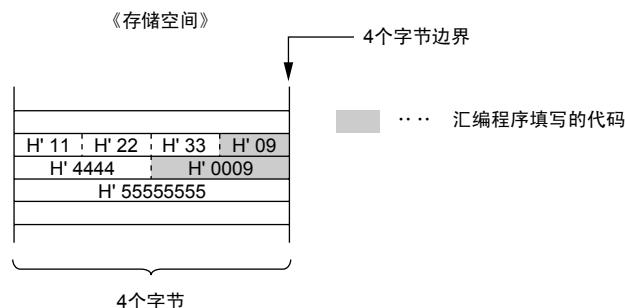
例

```

SECTION P, CODE, ALIGN=4
.DATA.B      H'11
.DATA.B      H'22
.DATA.B      H'33
.ALIGN 2     ; 将定位计数器值调整为 2 的倍数。
.DATA.W      H'4444
.ALIGN 4     ; 将定位计数器值调整为 4 的倍数。
.DATA.L      H'55555555
~

```

假设字节整数数据 H'11 原来就位于 4 个字节的边界。
汇编程序按下图填写目标码并进行调整。



符号值的设定

.EQU

格 式 < 符号 > [:] △ .EQU △ < 符号值 >

说 明 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'FFFFFFF（10 进制表示为 -2,147,483,648 ~ 4,294,967,295）。
 【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/EQ     #X1,R0      ; 和 CMP/EQ #10,R0 相同。
        BT         LABEL1
        CMP/EQ     #X2,R0      ; 和 CMP/EQ #20,R0 相同。
        BT         LABEL2
~

```

符号值的设定

.ASSIGN

格 式 < 符号 > [:] △ .ASSIGN △ < 符号值 >

说 明 给符号设定值。
 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'FFFFFFF（10 进制表示为 -2,147,483,648 ~ 4,294,967,295）。
 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/EQ        #X1,R0          ; 和 CMP/EQ #1,R0 相同。
        BT            LABEL1
        CMP/EQ        #X2,R0          ; 和 CMP/EQ #2,R0 相同。
        BT            LABEL2
~
X1:      .ASSIGN      3
X2:      .ASSIGN      4
        CMP/EQ        #X1,R0          ; 和 CMP/EQ #3,R0 相同。
        BT            LABEL3
        CMP/EQ        #X2,R0          ; 和 CMP/EQ #4,R0 相同。
        BF            LABEL4
~

```

寄存器别名的定义

.REG

格 式 < 符号 > [:] △ .REG △ < 寄存器名 >
 或者
 < 符号 > [:] △ .REG △ (< 寄存器名 >)

说 明 定义寄存器的别名。
 能和原寄存器名完全同等地使用 .REG 定义的寄存器别名。
 用 .REG 定义的寄存器别名不能重新定义。
 能命名的寄存器只限通用寄存器（R0 ～ R15、SP）。
 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。

例

```

~
MIN:      .REG      R10
MAX:      .REG      R11
MOV       #0,MIN    ; 和 MOV  #0,R10 相同。
MOV       #99,MAX   ; 和 MOV  #99,R11 相同。
CMP/HS    MIN,R1
BF        LABEL
CMP/HS    R1,MAX
BF        LABEL
~

```

浮点寄存器别名的定义

.FREG

格 式 < 符号 > [:] △ .FREG △ < 浮点寄存器名 >
 或者
 < 符号 > [:] △ .FREG △ (< 浮点寄存器名 >)

说 明 定义浮点寄存器的别名。
 能和原寄存器名完全同等地使用 .FREG 定义的浮点寄存器别名。
 用 .FREG 定义的寄存器别名不能重新定义。能命名的寄存器为以下浮点寄存器：
 • FRm (m=0 ~ 15)
 • DRn (n=0、2、4、6、8、10、12、14)
 • XDn (n=0、2、4、6、8、10、12、14)
 • Fvi (i=0、4、8、12)
 从定义的位置开始，.FREG 定义有效。.FREG 定义的符号有以下使用限制：
 • 不能外部参照或者外部定义。
 在单片机种类为 SH2E、SH4、SH4A、SH2AFPU 时，能使用 .FREG。

例

```

~
MAX:  .FREG      FR11
      FMOV      FR1,MAX      ; 和 FMOV FR1,FR11 相同。
      FCMP/EQ   MAX,FR2      ; 和 FCMP/EQ FR11,FR2 相同。
      BF        LABEL
~

```

.DATA

说明 将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，就为长字。
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。
操作长度和整数数据的范围如下：

长度	数据长度	整数数据的范围（10 进制表示）
B 字节	（1 个字节）	H'00000000~H'000000FF （0~255）
		H'FFFFFF80 ~ H'FFFFFFF （-128~-1）
W 字	（2 个字节）	H'00000000 ~ H'0000FFFF （0~65,535）
		H'FFFF8000 ~ H'FFFFFFF （-32,768~-1）
L 长字	（4 个字节）	H'00000000 ~ H'FFFFFFF （0 ~ 4,294,967,295）
		H'80000000 ~ H'FFFFFFF （-2,147,483,648 ~ -1）

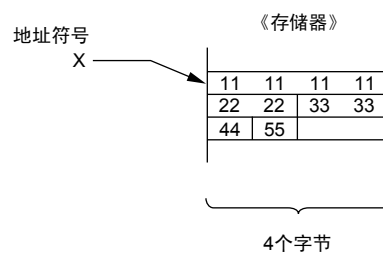
```

~
.ALIGN    4

X:

.DATA.L    H'11111111          ;
.DATA.W    H'2222,H'3333      ; 确保整数数据
.DATA.B    H'44,H'55          ;
~

```



【注】数据为16进制数。

.DATAB

【注】数据为16进制数。

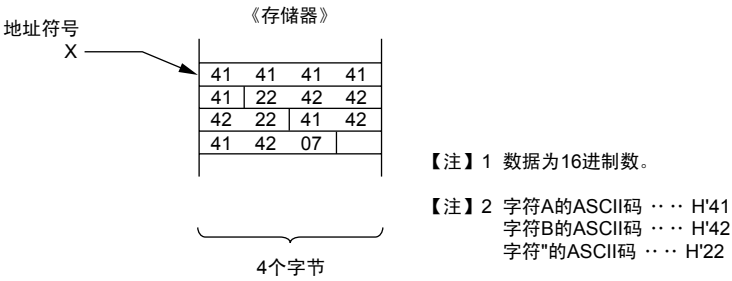
字符串数据的确保

.SDATA

格 式 [< 符号 >[:]] △ .SDATA △ "< 字符串 >"[, ...]

说 明 将字符串数据确保在存储器中。
 能给字符串附加控制字符。
 记述格式如下：
 " 字符串 "< 控制字符的 ASCII 码 >
 控制字符的 ASCII 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 ~
 .ALIGN 4
X: ~
 .SDATA "AAAAA" ; 确保字符串数据。
 .SDATA """"BBB"" ; 含有双引号的例子。
 .SDATA "ABAB"<H'07> ; 附加控制字符的例子。
 ~



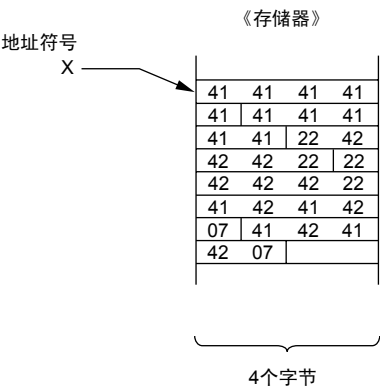
字符串数据块的确保

.SDATAB

格 式 [

说 明 只将指定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。
块数的上限值由字符串数据的长度决定（字符串数据的长度 × 块数 ≤ H'FFFFFFF
（4,294,967,295）字节）。
能给字符串附加控制字符。
记述格式如下：
 " 字符串 "<控制字符的 ASCII 码>
控制字符的 ASCII 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 ~
 .ALIGN 4
X:
 .SDATAB 2,"AAAAA" ; 确保字符串数据。
 .SDATAB 2,"""BBB"" ; 含有双引号的例子。
 .SDATAB 2,"ABAB"<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

带计数的字符串数据的确保

.SDATAC

格 式 [`<符号>[:]`] Δ `.SDATAC` Δ "`<字符串>`" [`,...`]

说明 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。
计数是以字节为单位表示字符串数据（不包含计数的数据）的长度。
能给字符串附加控制字符。
记述格式如下：

" 字符串 "< 控制字符的 ASCII 码 >

控制字符的 ASCII 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

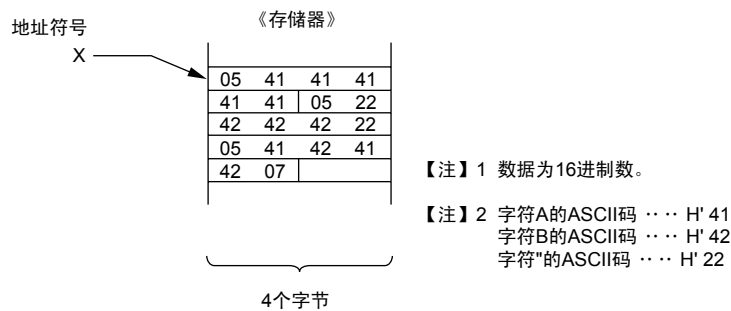
```

~
.ALIGN    4

X:

.SDATAAC  "AAAAA"           ; 确保带计数的字符串数据。
.SDATAAC  """"BBB""""       ; 含有双引号的例子。
.SDATAAC  "ABAB"<H'07>      ; 附加控制字符的例子。
~

```



以零结束的字符串数据的确保

.SDATAZ

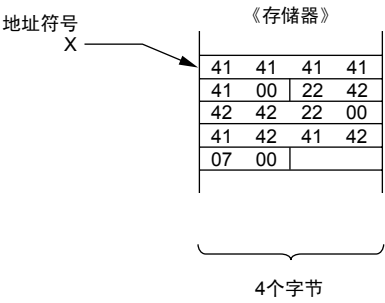
格 式 [`<符号>[:]`] Δ `.SDATAZ` Δ "`<字符串>`"[, ...]

说 明 将以零结束的字符串数据确保在存储器中。
以零结束的字符串是在字符串的末尾附加 1 个字节零的字符串。
能给字符串附加控制字符。
记述格式如下：

- "字符串" `<控制字符的 ASCII 码>`
控制字符的 ASCII 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

```
~
.ALIGN 4
X:
.SDATAZ "AAAAA"           ; 确保以零结束的字符串数据。
.SDATAZ """"BBB""""       ; 含有双引号的例子。
.SDATAZ "ABAB"<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

.FDATA

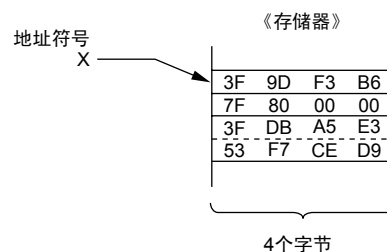
格 式 [<操作长度> = { S | D }

说 明 将浮点常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，就为单精度。

操作长度如下：

指定内容	数据长度
S	单精度 (4 个字节)
D	双精度 (8 个字节)

例

[illegible]

备 注 操作数不能记述 H' 形式（浮点 16 进制记述）的表达式，能记述 F' 形式（浮点 10 进制记述）的表达式。

浮点的记述方法和汇编程序的浮点常数相同。

"F'"["+|"-"](n["." [m]]|". "m)[("S"|"D")["+|"-"]x] (n,m,x 为 10 进制数)

在没有指数符号（S 或者 D）时，按照 .FDATA 的长度指定。

在 .FDATA.S 中出现双精度常数时转换为单精度，在 .DATA.D 中出现单精度常数时转换为双精度。

浮点表达式的语法如下:

$$\langle \text{式} \rangle ::= ["+" | "-"] \langle \text{项} \rangle [("+" | "-") \langle \text{项} \rangle]^*$$
$$\langle \text{项} \rangle ::= \langle \text{因子} \rangle [("*" | "/") \langle \text{因子} \rangle]$$

<因子> ::= <常数> | "(" <式> ")"

被 0.0 除会出错。

不能和非浮点常数（整数、定点）进行混合运算。

浮点数据块的确保

.FDTAB[illegible]

说明

只将指定的浮点常数数据连续确保在存储器中。

确保的数据长度取决于操作长度。如果省略操作长度，就为单精度。

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、外部参照符号和相对地址符号。

能指定的块数范围因操作长度而不同。

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 进制表示）	
S	单精度	（4 个字节）	H'00000001 ~ H'3FFFFFFF	（1 ~ 1,073,741,823）
D	双精度	（8 个字节）	H'00000001 ~ H'1FFFFFFFFF	（1 ~ 536,870,911）

```

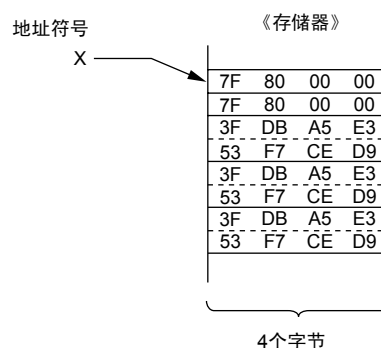
~
.ALIGN    4

X:
.FDATAB.S    2,H'7F800000.S    ; 确保 2 个 4 字节区域
                                ; "7F800000" (H'7F800000.S)。

.FDATAB.D    3,F'4.32D-1      ; 确保 3 个 8 字节的区域
                                ; "3FDBA5E353F7CED9"
                                (F'4.32D-1)。

~

```



.XDATA

格 式 [**<符号>[:]**] Δ .XDATA[.**<操作长度>**] Δ **<定点常数>**[, $\dots$]
 <操作长度> = { W | L }

说 明 将定点常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，就为长字。

操作长度如下：

指定内容	数据长度
W	字 (2 个字节)
L	长字 (4 个字节)

例

```

~
.ALIGN    4

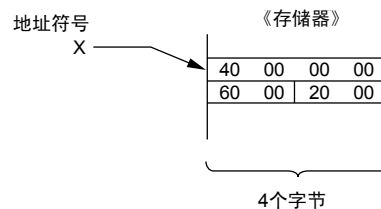
X:

.XDATA.L      0.5           ; 确保 4 个字节的区域 (H'40000000)。

.XDATA.W      0.75,0.25    ; 确保 2 个字节的区域 (H'6000) 和
                           ; (H'2000)。

~

```



.RES

格 式 [$\langle$ 符号 $\rangle$ [:]] Δ .RES[. $\langle$ 操作长度 $\rangle$] Δ $\langle$ 区域数 $\rangle$
 $\langle$ 操作长度 $\rangle$ = { B | W | L }

说明

将数据区确保在存储器中。

确保的数据区的单位容量取决于操作长度。如果省略操作长度，就为长字。

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。

能指定的区域数范围因操作长度而不同。

数据长度和区域数的范围如下:

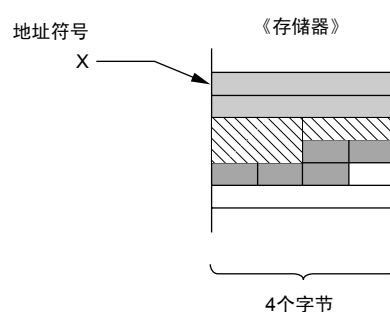
长度	数据长度	块数的范围（10 进制表示）
B 字节	（1 个字节）	H'00000001 ~ H'FFFFFFF （1 ~ 4,294,967,295）
W 字	（2 个字节）	H'00000001 ~ H'7FFFFFFF （1 ~ 2,147,483,647）
L 长字	（4 个字节）	H'00000001 ~ H'3FFFFFFF （1 ~ 1,073,741,823）

```

例      ~
        .ALIGN    4

X:
        .RES.L      2          ; 确保 2 个长字的区域。
        .RES.W      3          ; 确保 3 个字的区域。
        .RES.B      5          ; 确保 5 个字节的区域。
        ~

```



字符串数据区的确保

.SRES

格 式 [`<符号>[:]`] Δ .SRES Δ `<字符串区容量>[, ...]`

说 明 确保用于字符串的数据区。

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 进制表示为 1 ~ 4,294,967,295)。

例

2

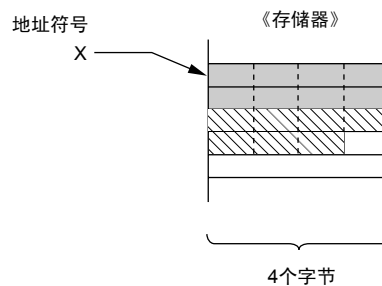
```
.ALIGN 4
```

X:

.SRES 7 ; 确保 7 个字节的区域。

.SRES 6 ; 确保 6 个字节的区域。

~



带计数的字符串数据区的确保

.SRESC

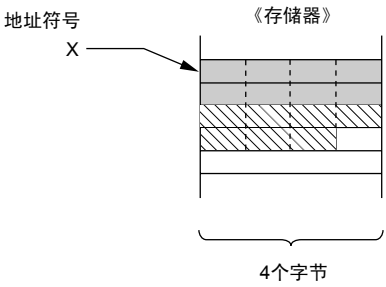
格 式 [

说 明 将带计数的字符串数据区确保在存储器中。
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。
计数是以字节为单位表示字符串的数据区（不包含计数的区域）的大小。
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 进制表示为 0 ~ 255）
被确保在存储器中的区域容量为字符串区容量 +1 个字节的计数。

例 ~
 .ALIGN 4
X: .SRESC 7 ; 确保 7 个字节 +1 个字节的计数。
 .SRESC 6 ; 确保 6 个字节 +1 个字节的计数。
 ~



以零结束的字符串数据区的确保

.SRESZ

格 式 [`<符号>[:]`] Δ .SRESZ Δ `<字符串区容量>[, ...]`

说明 将以零结束的字符串数据区确保在存储器中。
以零结束的字符串是字符串的末尾附加 1 个字节零的字符串。
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 进数表示为 0 ~ 255)。被确保在存储器上的区域容量为字符串区容量 + 末尾 1 个字节的零。

例

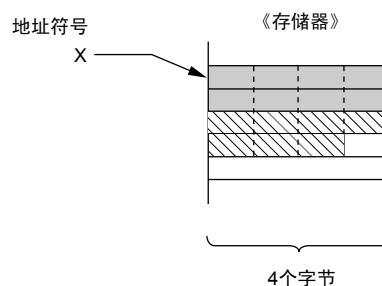
```

~
.ALIGN    4

X:

.SRESZ    7           ; 确保 7 个字节 + 末尾 1 个字节的零。
.SRESZ    6           ; 确保 6 个字节 + 末尾 1 个字节的零。
~

```



浮点数据区的确保

.FRES

格 式 [$\langle$ 符号 $\rangle$ [:]] Δ .FRES[. $\langle$ 操作长度 $\rangle$] Δ $\langle$ 区域确保数 $\rangle$
 $\langle$ 操作长度 $\rangle$ = { S | D }

说 明

将浮点数据区确保在存储器中。

确保的数据区的单位容量取决于操作长度。如果省略操作长度，就为单精度。

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、外部参照符号和相对地址符号。

操作长度如下：

指定内容	数据长度
S	单精度 (4 个字节)
D	双精度 (8 个字节)

例

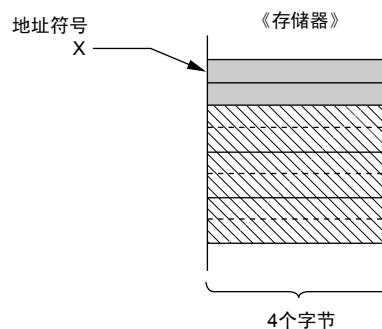
```

~
.ALIGN    4

X:

.FRES.S    2           ; 确保 2 个区域。
.FRES.D    3           ; 确保 3 个区域。
~

```



外部定义符号的声明

.EXPORT

格 式 Δ .EXPORT Δ < 符号 >[, ...]
不能记述标号。

说 明 声明外部定义符号。
为了从其他文件参照当前文件内定义的符号，需要声明外部定义符号。
能声明为外部定义的符号如下：

- 常数符号（用 .ASSIGN 定义的符号除外）
- 绝对地址符号（虚拟段的地址符号除外）
- 相对地址符号

 要对符号进行外部参照时，必须在将符号声明为外部定义符号的同时声明为外部参照符号。
 在参照符号的文件中，通过 .IMPORT 或者 .GLOBAL 声明外部参照符号。

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。
        ~
        .ALIGN      4
        .DATA.L      X                ; 参照 X。
        ~
    
```

```

        .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。
        ~
        .ALIGN      4
        .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 Δ .GLOBAL Δ < 符号 >[, ...]
不能记述标号。

说 明 声明外部定义符号或者外部参照符号。
为了从其他文件参照当前文件内定义的符号，需要声明外部定义符号；为了从当前文件参照其他文件内定义的符号，需要声明外部参照符号。
如果通过 .GLOBAL 声明文件内定义的符号，该符号就为外部定义符号；如果通过 .GLOBAL 声明文件内未定义的符号，该符号就为外部参照符号。
能声明为外部定义的符号如下：

- 常数符号（.ASSIGN 定义的符号除外）
- 绝对地址符号（虚拟段的地址符号除外）
- 相对地址符号

 要对符号进行外部参照时，必须在将符号声明为外部定义符号的同时声明为外部参照符号。
 在定义符号的文件中，通过 .EXPORT 或者 .GLOBAL 声明外部定义符号。
 在参照符号的文件中，通过 .IMPORT 或者 .GLOBAL 声明外部参照符号。

例 在文件 B 中参照文件 A 定义的符号的例子。

• 文件 A

```

        .GLOBAL      X                ; 将 X 声明为外部定义符号。
        ~
X:      .EQU    H'10000000            ; 定义 X。
        ~
    
```

• 文件 B

```

        .GLOBAL      X                ; 将 X 声明为外部参照符号。
        ~
        .ALIGN    4
        .DATA.L      X                ; 参照 X。
        ~
    
```

目标模块或者调试信息的输出控制

.OUTPUT

格 式

$\Delta .\text{OUTPUT} \triangle <\text{输出指定}> [, \dots]$

<输出指定> = { OBJ | NOOBJ |
 DBG | NODBG }

不能记述标号

说 明 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
<u>OBJ</u>	输出
NOOBJ	抑止输出

(2) 调试信息的输出

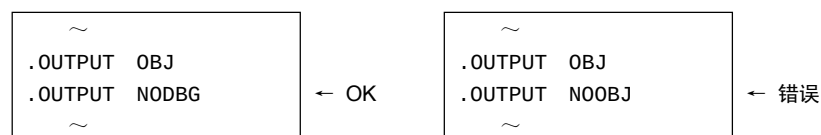
控制调试信息的输出。

输出种类如下:

输出种类	输出控制
DBG	输出
<u>NODBG</u>	抑止输出

如果 2 次以上使用 `.OUTPUT` 并且指定的内容出现矛盾，就出错。

【例】



只在输出目标模块时，有关调试信息输出的指定才有效。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。

例 在下例中，对于目标模块和调试信息的输出，假设没有指定选项。

• 例 1

```
.OUTPUT      OBJ          ; 输出目标模块。
               ; 不输出调试信息。
```

2

• 例 2

```
.OUTPUT      OBJ,DBG      ; 输出目标模块和
                        ; 调试信息。
```

2

• 例 3

```
.OUTPUT    OBJ,NDBG    ; 输出目标模块。  
           ~           ; 不输出调试信息。
```

备 注 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 $\triangle$.DEBUG $\triangle$ < 输出指定 >
 < 输出指定 > = { ON | OFF }
 不能记述标号。

说 明 控制符号调试信息的部分输出。
 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
<u>ON</u>	输出
OFF	抑止输出

例 ~
 .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。
 ~

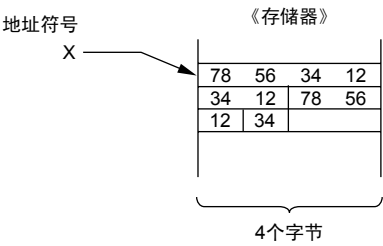
备 注 符号调试信息是有关调试信息中的符号的信息。

指定内容	输出的控制
BIG	通过 Big Endian 进行汇编。
LITTLE	通过 Little Endian 进行汇编。

【注】数据为16进制数。

```
• 例 2（指定 Little Endian）
    .CPU      SH3
    .ENDIAN   LITTLE           ; 指定 Little Endian。
    ~

X:
    .DATA.L   H'12345678       ;
    .DATA.W   H'1234, H'5678   ; 确保整数数据。
    .DATA.B   H'12, H'34       ;
    ~
```



【注】数据为16进制数。

.LINE

格 式 Δ .LINE Δ ["< 文件名 >",]< 行号 >
不能记述标号。

说 明 更改汇编程序的错误信息或者调试时参照的行号和文件名。
不更改程序中第一个 .LINE 到下一个 .LINE 的行号和文件。
在指定调试选项，输出汇编源程序时，编译程序（V3.0 以上）生成对应源文件行的 .LINE。
如果省略文件名，就不更改文件而更改行号。

例 shc - code=asmcode - debug test.c

源程序(test.c)

```
int func()  
{  
    int i,j;  
  
    j=0;  
    for (i=1;i<=10;i++){  
        j+=i;  
    }  
    return(j);  
}
```

汇编源程序(test.src)

```
.FILE      "C:\asm\test.c"  
.EXPORT    _func  
.SECTION   P,CODE,ALIGN=4  
_func:     ; function: func  
           ; frame size=0  
           .LINE      "C:\asm\test.c",5  
MOV        #0,R5      ; H'00000000  
           .LINE      "C:\asm\test.c",6  
MOV        #1,R2      ; H'00000001  
MOV        #10,R6     ; H'0000000A  
  
L11:  
ADD        #-1,R6  
TST        R6,R6  
           .LINE      "C:\asm\test.c",7  
ADD        R2,R5  
ADD        #1,R2  
BF         L11  
RTS  
MOV        R5,R0  
.END
```

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 通过输出的指定控制以下各列表的输出或者抑止输出:

- (1) 汇编列表
- (2) 源程序列表
- (3) 对照表
- (4) 段信息列表

根据各输出的指定控制的内容如下:

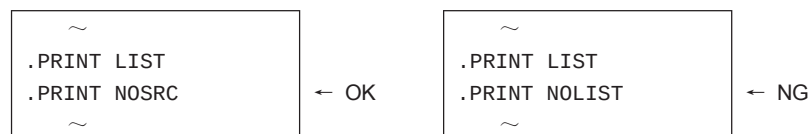
项目	输出指定 *1		含义	控制内容
	输出	输出抑止		
(1)	LIST	<u>NOLIST</u>	汇编列表的输出控制 *2	输出或者抑止输出汇编列表
(2)	<u>SRC</u>	NOSRC	源程序列表的输出控制 *3 *4	输出或者抑止输出源程序列表
(3)	<u>CREF</u>	NOCREF	对照表的输出控制 *3 *5	输出或者抑止输出对照表
(4)	<u>SCT</u>	NOSCT	段信息列表的输出控制 *3 *6	输出或者抑止输出段信息列表

【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` 并且指定的内容出现矛盾，就出错。

【例】



例 在下例中，对于汇编列表的输出，假设没有指定选项。

• 例 1

```
.PRINT LIST ; 输出全部种类的汇编列表。
```

• 例 2

.PRINT LIST,NOSRC,NOCREF ; 只输出段信息列表。

源程序列表的部分输出控制

.LIST

格 式 Δ .LIST Δ < 输出指定 > [, ...]

< 输出指定 > = { ON | OFF |
 COND | NOCOND | DEF | NODEF | CALL | NOCALL |
 EXP | NOEXP | CODE | NOCODE }

不能记述标号。

说 明

根据输出指定进行如下控制：

- (1) 控制源语句的部分输出或者输出抑止
- (2) 控制有关条件汇编功能和宏功能的源语句的部分输出或者输出抑止
- (3) 控制目标码显示行的部分输出或者输出抑止

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	抑止输出		
(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 .ASSIGNA、.ASSIGNC
	CALL	NOCALL	调用的输出控制 *1	宏调用语句 .AIF、.AIFDEF、.AENDI
	EXP	NOEXP	展开的输出控制 *1	宏展开部分 .AREPEAT、.AWHILE 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行数的部分

【注】*1 本指定在没有指定 show/noshow 选项时有效。

.LIST 的指定只在输出源程序列表时才有效。

有关源程序列表的部分输出，汇编程序优先指定选项。

.LIST 本身不显示在源程序列表。

例

在下例中，对于源程序列表的部分输出，假设没有指定选项。

	.LIST	NOCOND, NODEF	控制源程序列表的部分输出。
	.MACRO	SHLRN COUNT, Rd	
SHIFT	.ASSIGNA	¥COUNT	
	.AIF	¥&SHIFT GE 16	
	SHLR16	¥Rd	
SHIFT	.ASSIGNA	¥&SHIFT-16	
	.AENDI		
	.AIF	¥&SHIFT GE 8	
	SHLR8	¥Rd	
SHIFT	.ASSIGNA	¥&SHIFT-8	
	.AENDI		
	.AIF	¥&SHIFT GE 4	对通用多位的移位进行宏定义。
	SHLR2	¥Rd	
	SHLR2	¥Rd	
SHIFT	.ASSIGNA	¥&SHIFT-4	
	.AENDI		
	.AIF	¥&SHIFT GE 2	
	SHLR2	¥Rd	
SHIFT	.ASSIGNA	¥&SHIFT-2	
	.AENDI		
	.AIF \&SHIFT GE 1		
	SHLR	\Rd	
	.AENDI		
	.ENDM		
SHLRN	23, R0	宏调用	

编程例的源程序列表的输出结果

通过 .LIST 抑止宏定义、.ASSIGNA、.ASSIGNC、.AIF 或者 .AIFDEF 的不成立部分的输出。

31	31	
32	32	SHLRN 23, R0
33	M	
35	M	
36	M	.AIF 23 GE 16
37 00000000 4029	C	SHLR16 R0
39	M	.AENDI
40	M	
41	M	.AIF 7 GE 8
45	M	
46	M	.AIF 7 GE 4
47 00000002 4009	C	SHLR2 R0
48 00000004 4009	C	SHLR2 R0
50	M	.AENDI
51	M	
52	M	.AIF 3 GE 2
53 00000006 4009	C	SHLR2 R0
55	M	.AENDI
56	M	
57	M	.AIF 1 GE 1
58 00000008 4001	C	SHLR R0
59	M	.AENDI

格式

< 长度指定 > = { LIN = < 行数 > | COL = < 列数 > | TAB = {4|8}}

不能记述标号。

说 明

- 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	没有指定时
LIN=<行数>	每1页的行数 *1	20 ~ 255 *4	60
COL=<列数>	每1行的列数 *2	79 ~ 255 *4	132
TAB={4 8}	制表符长度 *3	4,8 *5	8

【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 在没有指定 show 选项的 tab 子选项时有效。
- *4 在指定的值超出范围的情况下，小于下限时指定下限值，大于上限时指定上限值。此时，不显示错误。
- *5 在指定了无效值时，指定 8。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。

例

```
.FORM    LIN=60,COL=200,TAB=4      ; 从此页开始将汇编列表的每 1 页设定为
; 60 行。
; 从此行开始将汇编列表的每 1 行设定为
; 200 列。
; 制表符长度为 4 的汇编列表输出。
;
```

```
.FORM LIN=55,COL=150 ; 从此页开始将汇编列表的每 1 页设定为
; 55 行。
; 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置

.HEADING

格 式 △ .HEADING △ "< 字符串 >"
不能记述标号。

说 明 设定源程序列表的页眉。
页眉能设定 60 个字符以内的字符串。
在 1 个源程序中，能多次使用 .HEADING。 .HEADING 设定的有效范围如下：

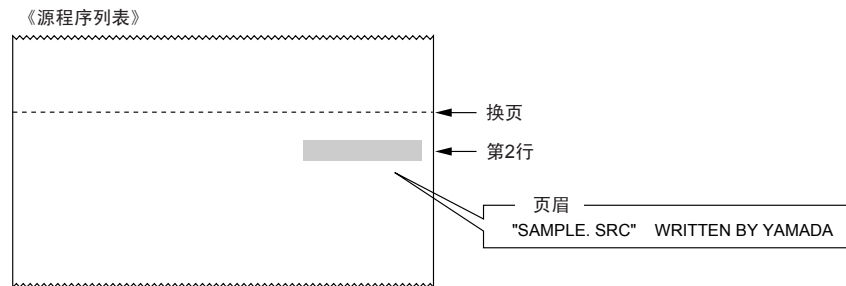
- 设定在页的第 1 行时，该页有效
- 设定在页的第 2 行以后时，下一页有效

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 在任意位置进行源程序列表的换页。
 .PAGE 在列表的第 1 行时，此换页指定无效。
 .PAGE 本身不显示在源程序列表。

例 ~
 MOV R0, R1
 RTS
 MOV R0, R2
 .PAGE ; 因为段会转换，所以能指定换页。
 .SECTION DT, DATA, ALIGN=4
 .DATA.L H'11111111
 .DATA.L H'22222222
 .DATA.L H'33333333
 ~

《源程序列表》

18 00000022 6103	18	MOV	R0, R1	
19 00000024 000B	19	RTS		
20 00000026 6203	20	MOV	R0, R2	

*** SuperH RISC engine ASSEMBLER Ver. 5.1 *** 06/01/01 10:15:30				
PROGRAM NAME =				
22 00000000	22	.SECTION	DT, DATA, ALIGN	
23 00000000 11111111	23	.DATA.L	H'11111111	
24 00000004 22222222	24	.DATA.L	H'22222222	
25 00000008 33333333	25	.DATA.L	H'33333333	

← 换页

源程序列表的空行输出

.SPACE

格 式 Δ .SPACE[Δ < 行数 >]
不能记述标号。

说 明 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。
.SPACE 输出的空行没有行号等显示。
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。
在 .SPACE 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 .SPACE 本身。

例

```
.SECTION    DT1, DATA, ALIGN=4
.DATA.L     H'11111111
.DATA.L     H'22222222
.DATA.L     H'33333333
.DATA.L     H'44444444      ; 在替换段的位置,
.SPACE      5                ; 插入 5 行空行
.SECTION    DT2, DATA, ALIGN=4
~
```

《源程序列表》

```
*** SuperH RISC engine ASSEMBLER Ver. 5.1 ***    06/01/01 10:15:30
PROGRAM NAME =
  1 00000000          1      .SECTION DT1, DATA, ALIGN=4
  2 00000000 11111111    2      .DATA.L  H'11111111
  3 00000004 22222222    3      .DATA.L  H'22222222
  4 00000008 33333333    4      .DATA.L  H'33333333
  5 0000000C 44444444    5      .DATA.L  H'44444444

  7 00000000          7      .SECTION DT2, DATA, ALIGN=4
~
```

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

设定目标模块名。

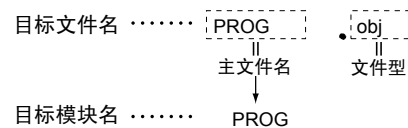
目标模块名是优化连接编辑程序为了识别目标模块所需要的名称。

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。

.PROGRAM 的最初设定有效，汇编程序忽视第 2 次以后的设定。

没有 **.PROGRA** 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      .PROGRAMPROG1          ; 将 PROG1 设定为目标模块名。
                                ;
```


源程序结束和入口点的指定

.END

格 式 Δ .END [Δ < 符号 >]

说 明 表示源程序的结束。
 在 .END 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 Δ .STACK Δ < 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。
 能对 1 个符号定义的堆栈值只限 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'FFFFFFFC 范围内的 4 的倍数，否则该定义无效。
 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
 ~

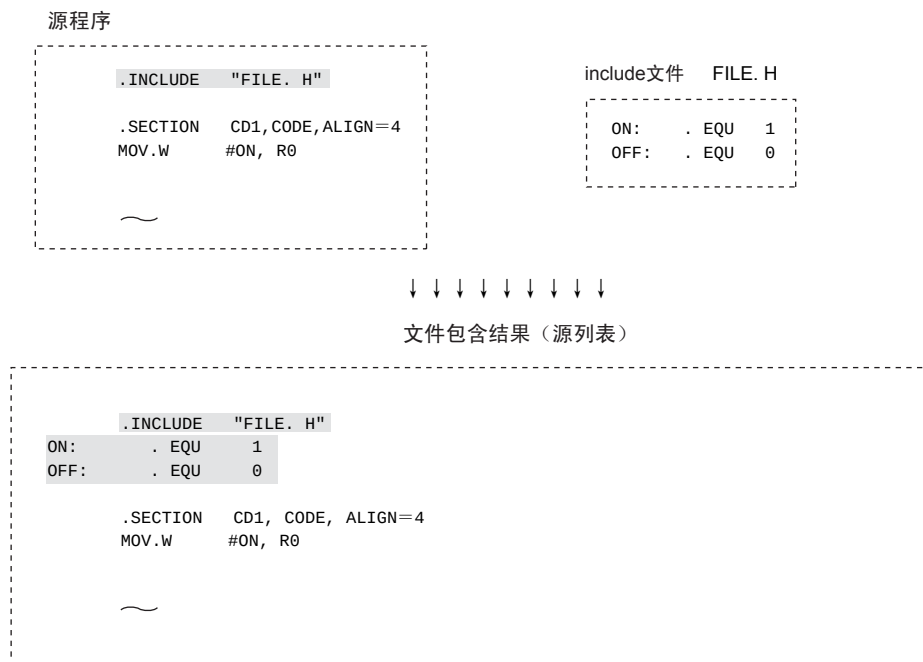
11.5 文件包含功能

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）。

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。

■ 编程例



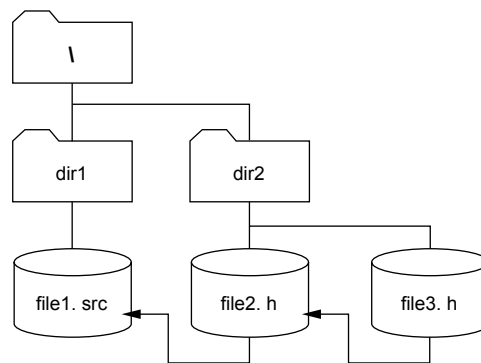
指定 include 文件

.INCLUDE

格 式 △ .INCLUDE △ "< 文件名 >"
不能记述标号。

说 明 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）。
能以包含文件夹的形式指定文件名。
以绝对路径（根文件夹开始的路径）或相对路径（当前文件夹开始的路径）指定文件夹。
也能在 **include** 文件中取入其他文件，包含能嵌套 30 层。
另外，源文件中的 **.INCLUDE** 的当前文件夹为启动汇编程序时的文件夹。**include** 文件中的 **.INCLUDE** 的当前文件夹为该 **include** 文件存在的文件夹。
能通过 **include** 选项更改由 **.INCLUDE** 指定的文件夹名。

例 文件夹为下图结构时，进行以下操作：



从根文件夹（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：

```
>asmsh c:\dir1\file1.src [RET]
```

file1.src 需要以下 **include** 控制语句。

```
.INCLUDE "dir2\file2.h" ; \ 为当前文件夹（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 为当前文件夹（指定相对路径）。
```

或者

```
.INCLUDE "\dir2\file3.h" ; 指定绝对路径
```

【注】 UNIX 时，必须将反斜线（\）改为斜线（/）。

11.6 条件汇编功能

11.6.1 条件汇编功能的概要

条件汇编功能简单实现如下汇编：

- 将源程序的字符串替换为其他字符串
- 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并进行汇编

(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数型和字符型。

(a) 整数型预处理器变量

用 .ASSIGNA 定义（能重新定义），并能通过在预处理器变量的前头加反斜线 \ 和 and 的符号（&）参照内容。

■编程例

```
FLAG: .ASSIGNA 1
~
.AIF \&FLAG EQ 1 ; FLAG 为 1 时
MOV R0,R1; 汇编 MOV R0,R1。
.AENDI
~
```

(b) 字符型的预处理器变量

能用 .ASSIGNC 重新定义，并能在预处理器变量的前头加反斜线（\）和符号 &，进行内容的参照。

■编程例

```
FLAG: .ASSIGNC "ON"
~
.AIF "\&FLAG" EQ "ON" ; FLAG 为 "ON" 时
MOV R0,R1; 汇编 MOV R0,R1。
.AENDI
~
```

(2) 置换符号

用 .DEFINE 定义，能通过指定进行一部分源程序的置换。编程如下：

■编程例

```
SYM1: .DEFINE "R1"
~
MOV.L SYM1,R0; 被置换为 MOV.L 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
    条件成立时的汇编部分
.AELIF    比较型条件
    条件成立时的汇编部分
.AELSE
    全部条件都不成立时的汇编部分
.AENDI
~

```

可省略此部分

■ 编程例

```

~
.AIF  "\\&FLAG" EQ "ON"
MOV   R0, R10      ; FLAG 为
MOV   R1, R11      ; "ON" 时
MOV   R2, R12      ; 进行汇编。
.AELSE
MOV   R10, R0      ; FLAG 不为
MOV   R11, R1      ; "ON" 时
MOV   R12, R2      ; 进行汇编。
.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的汇编部分
.AELSE
    不定义条件置换符号时的汇编部分
.AENDI
~

```

可省略此部分

■ 编程例

```

~
.AIFDEF FLAG
MOV      R0,R10          ; 在用 .AIFDEF 控制语句参照前,
MOV      R1,R11          ; 如果用 .DEFINE 控制语句定义了 FLAG,
MOV      R2,R12          ; 就进行汇编。
.AELSE
MOV      R10,R0          ; 在用 .AIFDEF 控制语句参照前,
MOV      R11,R1          ; 如果没有用 .DEFINE 控制语句定义 FLAG,
MOV      R12,R2          ; 就进行汇编。
.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，并汇编被展开的部分。编程如下：

```

~
.AREPEAT 重复次数
    重复对象部分
.AENDR
~

```

■ 编程例

；以 64 位 ÷ 32 位的除法为例。

；R1:R2(64 位) ÷ R0(32 位) = R2(32 位)：无符号

```

TST      R0,R0          ; 被零除的检查
BT       zero_div
CMP/HS   R0,R1          ; 上溢的检查
BT       over_div
DIV0U                      ; 标志的初始化
.AREPEAT 32
ROTCL    R2              ; 重复 32 次后进行汇编。
DIV1     R0,R1           ;
.AENDR
ROTCL    R2              ; R2= 商

```

(5) 条件重复展开

条件成立时，能对一部分源程序重复展开并进行汇编。编程如下：

```

~
.AWHILE 重复条件
    重复对象部分
.AENDW
~

```

■ 编程例

; 以乘加运算为例。

```
TblSiz: .ASSIGNA 50; TblSiz: 数据表的大小
        MOV      A_Tbl1,R1          ; R1: 数据表 1 的起始地址
        MOV      A_Tbl2,R2          ; R2: 数据表 2 的起始地址
        CLRMAC      ; MAC 寄存器的初始化
        .AWHILE \&TblSiz GT 0      ; 在 TblSiz 大于 0 的期间,
        MAC.W      @R1+,@R2+      ; 重复展开乘加运算, 然后进行汇编。
TblSiz: .ASSIGNA \&TblSiz-1      ; TblSiz 减 1。
        .AENDW
        STS      MACL,R0          ; 结果为 R0。
```

11.6.2 有关条件汇编功能的控制语句

条件汇编功能的控制语句如下：

分类	助记符	功能
有关变量定义	.ASSIGNA	定义整数型预处理器变量。 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	根据条件决定是否对一部分源程序进行汇编。
	.AELIF	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。
	.AELSE	
	.AENDI	
	.AIFDEF	根据置换符号的定义决定是否对一部分源程序进行汇编。
	.AELSE	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	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。
	.AENDR	
	.AWHILE	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。
	.AENDW	
其他	.EXITM	中断 .AREPEAT、.AWHILE 的重复展开。
	.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。

整数型预处理器变量的定义

.ASSIGNA

格 式 < 预处理器变量名 > [:] △ .ASSIGNA △ < 值 >

说 明 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
- .AIF
- .AELIF
- .AREPEAT
- .AWHILE
- 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 ; 建立通用多位的移位指令。
 ; 只将 SHIFT 的值右移位。

```

Rn:      .REG   R0                      ; 给 Rn 设定 R0。
SHIFT:   .ASSIGNA 27                    ; 给 SHIFT 设定 27。

        .AIF \&SHIFT GE 16              ; 条件: SHIFT ≥ 16
        SHLR16      Rn                  ; 如果条件成立, 就将 Rn 右移 16 位。
SHIFT:   .ASSIGNA \&SHIFT-16            ; SHIFT 减 16。
        .AENDI

        .AIF \&SHIFT GE 8               ; 条件: SHIFT ≥ 8
        SHLR8       Rn                  ; 如果条件成立, 就将 Rn 右移 8 位。
SHIFT:   .ASSIGNA \&SHIFT-8             ; SHIFT 减 8。
        .AENDI

        .AIF \&SHIFT GE 4               ; 条件: SHIFT ≥ 4
        SHLR2       Rn                  ; 如果条件成立, 就将 Rn 右移 4 位。
        SHLR2       Rn                  ;
SHIFT:   .ASSIGNA \&SHIFT-4             ; SHIFT 减 4。
        .AENDI

        .AIF \&SHIFT GE 2               ; 条件: SHIFT ≥ 2
        SHLR2       Rn                  ; 如果条件成立, 就将 Rn 右移 2 位。
SHIFT:   .ASSIGNA \&SHIFT-2            ; SHIFT 减 2。
        .AENDI

        .AIF \&SHIFT EQ 1               ; 条件: SHIFT=1
        SHLR        Rn                  ; 如果条件成立, 就将 Rn 右移 1 位。
        .AENDI

```

展开结果如下:

```

        SHLR16      R0                  ; 如果条件成立, 就将 Rn 右移 16 位。
        SHLR8       R0                  ; 如果条件成立, 就将 Rn 右移 8 位。
        SHLR2       R0                  ; 如果条件成立, 就将 Rn 右移 2 位。
        SHLR        R0                  ; 如果条件成立, 就将 Rn 右移 1 位。

```

.ASSIGNC

```
例      FLAG: .ASSIGNC "ON" ; 给 FLAG 设定 "ON"。
        ~
        .AIF "\&FLAG" EQ "ON" ; FLAG 为 "ON" 时，
MOV      R0,R1 ; 汇编 MOV R0,R1。
        .AENDI
        ~
FLAG:    .ASSIGNC "\&FLAG " ; 给 FLAG 附加空格 (" ")。
FLAGA:   .ASSIGNC "OFF" ; 给 FLAGA 设定 "OFF"。
FLAG:    .ASSIGNC "\&FLAG'AND \&FLAGA"
        ; 为了明确区别 FLAG 和 AND，使用 "'"。
        ; FLAG 结果为 "ON AND OFF"。
        ~
```


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 指定将符号置换成对应的字符串。

.DEFINE 和 .ASSIGNC 的不同点如下：

- 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。
- 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。

不能重新定义 .DEFINE 定义的符号。

用选项定义了置换符号时，对同名的符号的 .DEFINE 无效。

另外，不能置换 ".AENDI"、".AENDR"、".AENDW"、".AIFDEF"、".END"、".ENDM" 共 6 个控制指令。

例

```
SYM1: .DEFINE  "R1"
      ~
      MOV.L  SYM1,R0          ; 置换为 MOV.L  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。

```
C0: .DEFINE  "0"
    MOV.B    #H'C0,R0        ; 置换为 MOV.B  #H'0,R0
    MOV.B    #H'0C0,R0       ; 没被置换。
```

用 DEFINE 定义了同名符号时，以基数（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.B    #B'10,R0        ; 置换为 MOV.H  #H'10,R0。
```

比较型条件汇编

.AIF、.AELIF、.AELSE、.AENDI

格 式 Δ .AIF Δ < 项 1> Δ < 关系运算符> Δ < 项 2>
 <.AIF 条件成立时汇编的源语句>
 [Δ .AELIF Δ < 项 1> Δ < 关系运算符> Δ < 项 2>
 <.AELIF 条件成立时汇编的源语句>]
 [Δ .AELSE
 <全部条件不成立时汇编的源语句>]
 Δ .AENDI

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。
 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	不可记述
.AENDI	

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，条件始终不成立。

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：

关系运算符	条件
EQ	项 1 = 项 2
NE	项 1 $\neq$ 项 2
GT	项 1 > 项 2
LT	项 1 < 项 2
GE	项 1 $\geq$ 项 2
LE	项 1 $\leq$ 项 2

【注】 将值作为带 32 位的符号进行比较。
 字符串比较只在 EQ、NE 时有效。

```

例    ～
      .AIF \&TYPE EQ 1
      MOV      R0,R3          ;TYPE 为
      MOV      R1,R4          ; 1 时
      MOV      R2,R5          ; 进行汇编。
      .AELIF \&TYPE EQ 2
      MOV      R0,R6          ;TYPE 为
      MOV      R1,R7          ; 2 时
      MOV      R2,R8          ; 进行汇编。
      .AELSE
      MOV      R0,R9          ;TYPE 既不为
      MOV      R1,R10         ; 1 也不为 2 时
      MOV      R2,R11         ; 进行汇编。
      .AENDI
      ～

```

定义型条件汇编

.AIFDEF、.AELSE、.AENDI

格 式 $\triangle$.AIFDEF $\triangle$ < 置换符号 >
 < 定义了置换符号时汇编的源语句 >
 [$\triangle$.AELSE
 < 没有定义置换符号时汇编的源语句 >]
 $\triangle$.AENDI

不能记述标号。

说 明 .AIFDEF、.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 控制语句定义置换符号。

在用选项定义了所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。在参照后用控制语句定义所记述的置换符号时或者没有定义时，条件不成立

例

```

~
.AIFDEF  FLAG
MOV     R0, R3           ; 在用 .DEFINE 控制语句定义 FLAG 时,
MOV     R1, R4           ; 进行汇编。
.AELSE
MOV     R0, R6           ; 在没有用 .DEFINE 控制语句定义 FLAG 时,
MOV     R1, R7           ; 进行汇编。
.AENDI
~

```

重复展开

.AREPEAT、.AENDR

格 式 Δ .AREPEAT Δ < 次数 >
 < 重复展开后汇编的源语句 >
 Δ .AENDR

不能记述标号。

说 明 只进行指定次数的重复展开，然后进行汇编。
 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，然后进行汇编（和重复复制源语句相同，不在执行时循环）。

用常数或者预处理器变量指定次数。

如果给次数指定 0 以下的值，就不进行展开。

例 ; 以 64 位 ÷ 32 位的除法为例。
 ; R1:R2(64 位) ÷ R0(32 位) = R2(32 位) : 无符号
 TST R0,R0 ; 被零除的检查
 BT zero_div
 CMP/HS R0,R1 ; 上溢的检查
 BT over_div
 DIV0U ; 标志的初始化
 .AREPEAT 32
 ROTCL R2 ; 重复 32 次后进行汇编。
 DIV1 R0,R1 ;
 .AENDR
 ROTCL R2 ; R2= 商

例	;	以乘法运算为例。	
	TblSiz: .ASSIGNA	50	; TblSiz: 数据表的大小
	MOV	A_Tbl1,R1	; R1: 数据表 1 的起始地址
	MOV	A_Tbl2,R2	; R2: 数据表 2 的起始地址
	CLRMAC		; MAC 寄存器的初始化
	.AWHILE	&TblSiz GT 0	; 在 TblSiz 大于 0 的期间,
	MAC.W	@R0+,@R1+	; 重复乘加运算, 然后进行汇编。
	TblSiz: .ASSIGNA	&TblSiz-1	; TblSiz 减 1。
	.AENDW		
	STS	MACL,R0	; 结果为 R0。

展开的中断

.EXITM

格 式 △ .EXITM
不能记述标号。

说 明 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      R0, R1
        ADD      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 被更新并 .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      R0, R1    ... COUNT 为 0 时
        ADD      R2, R3
        ADD      R0, R1    ... COUNT 为 1 时
        ADD      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 667，结束汇编程序。 .AERROR 能用于检查预处理器变量值等。
例	<pre> ~ .AIF \&FLG EQ 1 MOV R1,R10 MOV R2,R11 .AELSE .AERROR ; \&FLAG 不为 1 时，发生错误。 .AENDI ~ </pre>

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	<pre> .ALIMIT 20 ~ FLG: .ASSIGNA 0 .AWHILE \&FLG EQ 0 ; 展开 20 次后，中断展开 NOP ; 发生警告。 .AENDW ~ </pre>

11.7 宏功能

11.7.1 宏功能的概要

本汇编语言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。

使用被定义的宏指令进行的调用称为宏调用。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：

■ 编程例

```

~
.MACRO  SUM                                ; 进行 R0、R1、R2、R3 合计计算的处理
MOV     R0,R10                            ; 定义为宏指令 SUM。
ADD     R1,R10
ADD     R2,R10
ADD     R3,R10
.ENDM
~

SUM                                         ; 调用宏指令 SUM。
                                           ; 展开宏体
                                           ;      MOV     R0,R10
                                           ;      ADD     R1,R10
                                           ;      ADD     R2,R10
                                           ;      ADD     R3,R10

```

能更改一部分定义的宏指令后进行展开，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。

展开宏指令时，将形式参数置换为对应的宏参数。

■编程例

```

~
.MACRO  SUM ARG1                ; 定义形式参数 ARG1。
MOV     R0, \ARG1               ; 使用 ARG1 记述宏体。
ADD     R1, \ARG1
ADD     R2, \ARG1
ADD     R3, \ARG1
.ENDM
~
SUM     R10                     ; 指定宏参数 R10 调用宏 SUM。
                                   ; 用宏参数置换宏体中的形式参数。
                                   ; 展开
                                   ;      MOV     R0, R10
                                   ;      ADD     R1, R10
                                   ;      ADD     R2, R10
                                   ;      ADD     R3, R10

```

11.7.2 有关宏功能的控制语句

宏功能的控制语句如下：

助记符	功能
.MACRO	定义宏指令。
.ENDM	
.EXITM	中断宏指令的展开。 请参照 11.6.2 的 .EXITM 。

宏指令的定义

.MACRO、.ENDM

格 式 Δ .MACRO Δ < 宏名 > [Δ < 形式参数 > [, ...]]
 Δ .ENDM
 < 形式参数 >: < 形式参数 > [= < 形式参数的默认值 >]
 不能记述标号。

说 明 定义宏（给一连串的源语句命名后进行归纳处理）。
 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。

在展开宏指令时要置换一部分宏体进行展开的情况下，进行指定。

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 [']

在要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

在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。否则，就忽视 `.ENDM`，不显示错误。

例

```

~
.MACRO  SUM                                ; 进行 R0、R1、R2、R3 合计计算的处理
MOV     R0,R10                            ; 定义为宏指令 SUM。
ADD     R1,R10
ADD     R2,R10
ADD     R3,R10
.ENDM
~
SUM                                           ; 调用宏指令 SUM。
                                           ; 展开宏体
                                           ;      MOV      R0,R10
                                           ;      ADD      R1,R10
                                           ;      ADD      R2,R10
                                           ;      ADD      R3,R10

```

11.7.3 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（'）。

■编程例

```
.MACRO  PLUS1 P,P1                ; P,P1 为形式参数。
ADD     #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。
```

展开结果如下：

```
ADD     #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（'）。

■编程例

```
.MACRO  PLUS1
ADD     #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"                 ; 参照预处理器变量 V。
V1: .ASSIGNA  1                   ; 参照预处理器变量 V1。
    PLUS1                          ; 展开 PLUS1。
```

展开结果如下：

```
ADD     #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

在宏体中有标号时，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），宏生成号的记述方法如下。

\@

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，因为宏生成号被展开为数字，所以不能记述在符号名的前头。

■编程例

```

        .MACRO  RES_STR STR,Rn
        MOV.L   #str\@,\Rn
        BRA     end_str\@
        NOP
str\@   .SDATA "\STR"
        .ALIGN  2
end_str\@
        .ENDM
RES_STR "ONE",R0          ; 每次展开 RES_STR 时，
RES_STR "TWO",R1          ; 生成不同的符号

```

展开结果如下：

```

        MOV.L   #str00000,R0
        BRA     end_str00000
        NOP
str00000 .SDATA "ONE"
        .ALIGN  2
end_str00000
        MOV.L   #str00001,R1
        BRA     end_str00001
        NOP
str00001 .SDATA "TWO"
        .ALIGN  2
end_str00001

```

(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。宏处理除外的记述方法如下：

\\(宏处理除外的字符串)

在宏展开中去掉反斜线“\”和括号。

■ 编程例

```
.MACRO BACK_SLASH_SET
\ (MOV    #"\", R0)          ; 将 “\” 展开为 ASCII 字符。
.ENDM
```

展开结果如下：

```
MOV      #"\", 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（要避免在列表文件中多次出现相同的注释），记述宏内注释。宏内注释的写法如下：

```
\; 注释
```

■ 编程例

```
.MACRO  PUSH  Rn
MOV.L   \Rn, @-R15      \; \Rn 为寄存器。
.ENDM
PUSH    R0
```

展开结果如下：（不展开注释）。

```
MOV.L   R0, @-R15
```

(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：字符串的字符数
- .INSTR 函数：字符串的检索
- .SUBSTR 函数：字符串的抽出

11.7.4 宏调用

展开定义的宏指令（由宏定义进行定义）称为宏调用，宏调用的记述方法如下：

[<符号>[:]] △ <宏名> [△ <宏参数>[, ...]]
<宏参数>: [= <形式参数名>] = <字符串>

在调用宏以前对宏名进行宏定义（.MACRO）。

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

关键字指定是用等号（=）对宏定义（.MACRO）声明的形式参数的形式参数名进行分开指定的方法。

(4) 宏参数的记述

在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。

■ 编程例

```

.MACRO SUM FROM=0,T0=9 ; 定义宏指令SUM和形式参数FROM, T0。
MOV    R\FROM,R10
COUNT .ASSIGNA \FROM+1
.AWHILE \&COUNT LE \T0
MOV    R\&COUNT,R10
COUNT .ASSIGNA \&COUNT+1
.AENDW
.ENDM
SUM    0,5
SUM    T0=5

```

用形式参数记述宏体。

任何一个结果都相同。

用宏参数置换宏体中的形式参数，展开结果如下：

```

MOV    R0,R10
MOV    R1,R10
MOV    R2,R10
MOV    R3,R10
MOV    R4,R10
MOV    R5,R10

```

11.7.5 字符串操作函数

能在宏体中使用的字符串操作函数如下：

字符串操作函数	功能
.LEN	返回字符串的字符数。
.INSTR	检索字符串。
.SUBSTR	抽出字符串。

字符串的字符数

.LEN

格 式 .LEN[△]("< 字符串 ")

说 明 计算字符串的字符数，将结果置换为省略基数的 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
 .ALIGN 4
 .SRES .LEN("\P1")
 .ENDM
 ~
 RESERVE_LENGTH ABCDEF
 RESERVE_LENGTH ABC

展开结果如下：
 .ALIGN 4
 .SRES 6 ; "ABCDEF" 的字符数是 6。
 .ALIGN 4
 .SRES 3 ; "ABC" 的字符数是 3。

字符列的检索

.INSTR

格 式 `.INSTR[  $\Delta$  ] ("< 字符串 1 >", "< 字符串 2 >" [, < 检索开始位置 >])`

说 明 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 的检索位置替换为省略基数的 10 进制数。

字符串 1 中不含有字符串 2 时，替换为 -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，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) 才能记述本函数。

例

```
~
.MACRO  FIND_STR P1
.DATA.W      .INSTR("ABCDEFGH", "\P1", 0)
.ENDM
~
FIND_STR    CDE
FIND_STR    H
```

展开结果如下：

```
.DATA.W      2          ; 从 "ABCDEFGH" 的第 2 个字符 (起始位置为 0)
                  ; 起有 "CDE"
.DATA.W      -1         ; "ABCDEFGH" 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 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，并替换为用双引号 (") 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。

能指定的抽出开始位置 ≥ 0 。

能指定的抽出长度 ≥ 1 。

在抽出开始位置和抽出长度不适当时，替换为空字符 ("")。

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`) 才能记述本函数。

例

```

~
.MACRO  RESERVE_STR P1=0,P2
.SDATA      .SUBSTR("ABCDEFGH",\P1,\P2)
.ENDM
~
RESERVE_STR  2,2
RESERVE_STR  ,3      ; 省略宏参数 P1。

```

展开结果如下：

```

.SDATA      "CD"
.SDATA      "ABC"

```

11.8 文字池的自动生成功能

11.8.1 文字池的自动生成功能的概要

要将 2 个字节或者 4 个字节长的常数数据（以下称为文字）传送到寄存器时，必须确保文字池（文字的收集），并以 PC 相对寻址方式进行参照。在配置文字池时，需要考虑以下几点：

- 数据是在数据传送指令能参照的范围内吗？
- 2 个字节长的数据是位于 2 字节边界吗？4 个字节长的数据是位于 4 字节边界吗？
- 能通过多个数据传送指令共享 1 个数据吗？
- 在程序中的何处配置文字池？

文字池自动生成功能是从 1 条指令自动生成对应常数数据寄存器传送的 PC 相对 MOV 指令（或者 MOVA 指令）和 .DATA 的功能。例如，如果使用本功能，就能将以下程序 (a) 记述为 (b)。

■程序 (a)

```

MOV.L      DATA1, R0
MOV.L      DATA2, R1
~
.ALIGN     4
DATA1      .DATA.L   H'12345678
DATA2      .DATA.L   5000000

```

■程序 (b)

```

MOV.L      #H'12345678, R0
MOV.L      #5000000, R1
~

```

11.8.2 有关文字池自动生成功能的扩展指令

对于扩展指令（MOV.W #imm,Rn、MOV.L #imm,Rn、MOVA #imm,R0）的记述，汇编程序自动生成需要的文字池，计算 PC 相对位移量的值。

扩展指令的源语句及其展开结果如表 11.34 所示。将扩展指令的源语句展开为 1 条执行指令和 1 个文字数据。

表 11.34 扩展指令的种类和展开结果

扩展指令		展开结果	
MOV.W	#imm,Rn	MOV.W	@(disp,PC),Rn 和 2 个字节长的文字数据
MOV.L	#imm,Rn	MOV.L	@(disp,PC),Rn 和 4 个字节长的文字数据
MOVA	#imm,R0	MOVA	@(disp,PC),R0 和 4 个字节长的文字数据

11.8.3 文字池自动生成功能的大小模式

文字池自动生成的大小模式有大小指定模式（通过指定操作长度的数据传送指令（扩展指令）自动生成文字池）和大小选择模式（对没有指定操作长度的数据传送指令，汇编程序通过 imm 的值选择适当的指令）2 种。

数据传送指令和大小模式的关系如表 11.35 所示。

表 11.35 数据传送指令和大小模式的关系

数据传送指令		大小指令模式	大小选择模式
MOV	#imm,Rn	执行指令	由汇编程序选择
MOV.B	#imm,Rn	执行指令	执行指令
MOV.W	#imm,Rn	扩展指令	扩展指令
MOV.L	#imm,Rn	扩展指令	扩展指令

(1) 大小指定模式

在大小指定模式中，将没有指定操作长度的数据传送指令（MOV #imm,Rn）作为通常执行指令处理。大小指定模式在没有指定 auto_literal 选项时有效。

(2) 大小选择模式

在大小选择模式中，除了扩展指令以外，对于没有指定操作长度的数据传送指令（MOV #imm,Rn）的记述，汇编程序判断 imm 值的范围，如果需要，就自动生成文字池。在带符号的范围内判断 imm 值。大小选择模式在指定 auto_literal 选项时有效。

在大小选择模式中，根据 imm 值的范围选择的指令如表 11.36 所示。

表 11.36 大小选择模式中选择的指令

imm 的指定方法	imm 值的范围 *	选择的指令
常数 参照前定义的常数符号 参照前定义的绝对地址符号	H'FFFFFF80 ~ H'0000007F (-128 ~ 127)	MOV.B #imm,Rn
	H'FFFF8000 ~ H'FFFFFF7F (-32,768 ~ -129)	MOV.W #imm,Rn
	H'00000080 ~ H'00007FFF (128 ~ 32,767)	展开结果 (MOV.W @(disp,PC),Rn 和 2 个字节长的文字数据)
	H'80000000 ~ H'FFFF7FFF (-2,147,483,648 ~ -32,769)	MOV.L #imm,Rn
	H'00008000 ~ H'7FFFFFFF (32,768 ~ 2,147,483,647)	展开结果 (MOV.L @(disp,PC),Rn 和 4 个字节长的文字数据)
相对地址符号 外部参照符号 参照后定义的常数符号 参照后定义的绝对地址符号	不依存 imm 的值	MOV.L #imm,Rn 展开结果 (MOV.L @(disp,PC),Rn 和 4 个字节长的文字数据)

【注】 * () 内为 10 进制数

11.8.4 文字池的输出

文字池的输出点如下：

- 无条件转移和紧接着延迟槽指令后的位置
- 程序员记述 .POOL 的位置

另外，能通过 `literal` 选项选择输出点。

汇编程序在扩展指令的记述位置之后输出与最近的输出点对应的文字。汇编程序在归纳相同输出点的文字数据后生成 1 个文字池。

【注】 在给延迟槽指令指定标号时，不会成为文字池的输出点。

(1) 利用无条件转移的文字池输出

输出例子如下所示：

■ 编程例

源程序

```
. SECTION CD1, CODE, LOCATE=H' 0000F000
CD1_START:
MOV. L   #H' FFFF0000, R0
MOV. W   #H' FF00, R1
MOV. L   #CD1_START, R2
MOV      #H'FF, R3
RTS
MOV      R0, R10
.END
```

↓↓↓↓

文字池自动生成结果（源程序列表）

1	0000F000	1	. SECTION CD1, CODE, LOCATE=H' 0000F000
2	0000F000	2	CD1_START
3	0000F000 D003	3	MOV. L #H' FFFF0000, R0
4	0000F002 9103	4	MOV. W #H' FF00, R1
5	0000F004 D203	5	MOV. L #CD1_START, R2
6	0000F006 E3FF	6	MOV #H'FF, R3
7	0000F008 000B	7	RTS
8	0000F00A 6A03	8	MOV R0, R10
9			***** BEGIN-POOL *****
10	0000F00C FF00		DATA FOR SOURCE-LINE 4
11	0000F00E 0000		ALIGNMENT CODE
12	0000F010 FFFF0000		DATA FOR SOURCE-LINE 3
13	0000F014 0000F000		DATA FOR SOURCE-LINE 5
14			***** END-POOL *****
15		9	. END

(2) 文字池输出到 .POOL 位置

如果在位移量的范围内不进行无条件转移的文字池输出（程序中无条件转移较少的情况等），汇编程序就输出错误 402。此时，必须在位移量的范围内记述 .POOL。

位移量的范围如下：

- 当操作长度为字（W）时：0 ～ 510 字节
- 当操作长度为长字（L）时：0 ～ 1020 字节

在文字池输出到 .POOL 位置时，跳过该文字池的转移指令也一起被输出。

■ 编程例

源程序

```
.SECTION CD1, CODE, LOCATE=H' 0000F000
CCD1_START
MOV. L   #H' FFFF0000, R0
MOV. W   #H' FF00, R1
MOV. L   #CD1_START, R2
MOV      #H' FF, R3
.POOL
.END
```

↓↓↓↓↓

文字池自动生成结果（源程序列表）

1	0000F000	1	.SECTION CD1, CODE, LOCATE=H' 0000F000
2	0000F000	2	CD1_START:
3	0000F000 D003	3	MOV. L #H' FFFF0000, R0
4	0000F002 9103	4	MOV. W #H' FF00, R1
5	0000F004 D203	5	MOV. L #CD1_START, R2
6	0000F006 E3FF	6	MOV #H' FF, R3
7	0000F008	7	.POOL
8			***** BEGIN-POOL *****
9	0000F008 A006		BRA TO END-POOL
10	0000F00A 0009		NOP
11	0000F00C FF00		DATA FOR SOURCE-LINE 4
12	0000F00E 0000		ALIGNMENT CODE
13	0000F010 FFFF0000		DATA FOR SOURCE-LINE 3
14	0000F014 0000F000		DATA FOR SOURCE-LINE 5
15			***** END-POOL *****
16		8	.END

11.8.5 文字的共享

汇编程序进行收集在同一文字池的多个扩展指令 `imm` 的共享处理。共享以下所示的同一文字数据：

- (1) 符号
- (2) 常数
- (3) 符号 ± 常数

除了上述以外，如果在汇编时具有相同值，就有可能共享可判断的表达式。即使 `imm` 值相同，如果扩展指令的操作长度不同，就不共享文字数据。另外，在输出目标的文字池不同时，也不共享数据。多条扩展指令共享 1 个文字数据的例子如下所示：

■ 编程例

源程序

```

SECTION CD1, CODE, LOCATE=H' 0000F000
CD1_START:
MOV. L  #H' FFFF0000, R0
MOV. W  #H' FF00, R1
MOV. L  #H' FFFF0000, R2
MOV     #H' FF, R3
RTS
MOV     R0, R10
.END

```

↓↓↓↓

文字池自动生成结果（源程序列表）

1	0000F000	1	.SECTION CD1, CODE, LOCATE=H' 0000F000
2	0000F000	2	CD1_START:
3	0000F000 D003	3	MOV. L #H' FFFF0000, R0
4	0000F002 9103	4	MOV. W #H' FF00, R1
5	0000F004 D202	5	MOV. L #H' FFFF0000, R2
6	0000F006 E3FF	6	MOV #H' FF, R3
7	0000F008 000B	7	RTS
8	0000F00A 6A03	8	MOV R0, R10
9			***** BEGIN-POOL *****
10	0000F00C FF00		DATA FOR SOURCE-LINE 4
11	0000F00E 0000		ALIGNMENT CODE
12	0000F010 FFFF0000		DATA FOR SOURCE-LINE 3, 5
13			***** END-POOL *****
14		9	.END

11.8.6 文字池输出的抑止

如果程序中过多地进行无条件转移，就会产生以下问题：

- 输出几个小的文字池。
- 不能共享文字。

此时，必须用以下的方法抑止文字池的输出。

```
~
延迟转移指令
延迟槽指令
.NOPOOL
~
```

■ 编程例

源程序

```
~
CASE1:
    MOV. L    #H' FFFF0000, R0  --- 扩展指令1
    RTS
    NOP
    .NOPOOL                    --- 不在此点输出文字池。
CASE2:
    MOV. L    #H' FFFF0000, R0  --- 扩展指令2
    RTS
    NOP
~
```

↓↓↓↓

文字池自动生成结果（源程序列表）

20	0000F000	20	CASE1:
21	0000F000 D002	21	MOV. L #H' FFFF0000, R0
22	0000F002 000B	22	RTS
23	0000F004 0009	23	NOP
24		24	.NOPOOL
25	0000F006	25	CASE2:
26	0000F006 D001	26	MOV. L #H' FFFF0000, R0
27	0000F008 000B	27	RTS
28	0000F00A 0009	28	NOP
29			***** BEGIN-POOL *****
30	0000F00C FFFF0000		DATA FOR SOURCE-LINE 21, 26
31			***** END-POOL *****

11.8.7 有关文字池自动生成的注意事项

(1) 记述扩展指令后发生错误的情况

不能给延迟槽指令记述扩展指令（错误 151）。

不能给调整数未满 2 的相对地址段记述扩展指令（错误 152）。

不能给调整数未满 4 的相对地址段记述 `MOV.L #imm,Rn`、`MOVA #imm,R0`（错误 152）。

(2) 记述 `.POOL` 后发生错误的情况

不能紧接着转移指令之后记述 `.POOL`（错误 522）。

(3) 记述 `.NOPOOL` 后发生错误的情况

在紧接着延迟槽指令后记述 `.NOPOOL` 时有效。否则，就发生错误 521。

(4) 作为扩展指令的展开结果，其他执行指令不在范围内的情况

汇编程序生成文字池，位移量超出范围的指令发生错误 402。

必须用 `.NOPOOL` 等移动文字池的输出点，或者更改错误指令的位置或寻址方式。

(5) 没有找到文字池的输出位置的情况

从扩展指令的位置来看，当没有找到满足以下条件文字池的输出点时

- 同文件
- 同段
- 扩展指令的记述位置以后

汇编程序将文字池及跳过该文字池的该 `BRA` 指令（延迟槽指令为 `NOP`）输出到该扩展指令存在的段的最后，发出警告 876。

(6) 扩展指令的位移量不在范围内的情况

生成文字池，但是如果扩展指令的位移量在范围以外，该扩展指令就发生错误 402。

必须使用 `.POOL` 等，在范围内的位置生成文字池。

(7) 大小指定模式和大小选择模式的不同

Ver.2.0 的文字池功能只支持大小指定模式，但是在 Ver.3.1 以后追加了大小选择模式。如果在 Ver.3.1 以上的大小选择模式中汇编 Ver.2.0 建立的源程序，对于没有指定操作长度的数据传送指令，`imm` 值为 `H'00000080` ~ `H'000000FF`（128 ~ 255）的范围。

大小指定模式和大小选择模式的输出例子如下所示：

■ 编程例

源程序

```

SECTION CD1, CODE, LOCATE=H' 0000F000
MOV. L   #H' FF, R0
MOV. W   #H' FF, R1
MOV. B   #H' FF, R2
MOV      #H' FF, R3
RTS
MOV      R0, R10
.END

```

↓↓↓↓↓

大小指定模式的文字池自动生成结果（源程序列表）

1	0000F000	1	. SECTION CD1, CODE, LOCATE=H' 0000F000
2	0000F000 D003	2	MOV. L #H' FF, R0
3	0000F002 9103	3	MOV. W #H' FF, R1
4	0000F004 E2FF	4	MOV. B #H' FF, R2
5	0000F006 E3FF	5	MOV #H' FF, R3
6	0000F008 000B	6	RTS
7	0000F00A 6A03	7	MOV R0, R10
8			***** BEGIN-POOL *****
9	0000F00C 00FF		DATA FOR SOURCE-LINE 3
10	0000F00E 0000		ALIGNMENT CODE
11	0000F010 000000FF		DATA FOR SOURCE-LINE 2
12			***** END-POOL *****
13		8	. END

R3的内容为H'FFFFFFFF。

大小选择模式的文字池自动生成结果（源程序列表）

1	0000F000	1	. SECTION CD1, CODE, LOCATE=H' 0000F000
2	0000F000 D003	2	MOV. L #H' FF, R0
3	0000F002 9103	3	MOV. W #H' FF, R1
4	0000F004 E2FF	4	MOV. B #H' FF, R2
5	0000F006 9301	5	MOV #H' FF, R3
6	0000F008 000B	6	RTS
7	0000F00A 6A03	7	MOV R0, R10
8			***** BEGIN-POOL *****
9	0000F00C 00FF		DATA FOR SOURCE-LINE 3, 5
10	0000F00E 0000		ALIGNMENT CODE
11	0000F010 000000FF		DATA FOR SOURCE-LINE 2
12			***** END-POOL *****
13		8	. END

R3的内容为H'000000FF。

11.9 重复循环指令的自动生成功能

11.9.1 重复循环指令自动生成功能的概要

在 SH-DSP、SH3-DSP、SH4AL-DSP 中，通过 LDRS 指令和 LDRE 指令将重复的起始地址和重复的结束地址分别设定到 RS 寄存器和 RE 寄存器，该地址的设定值因重复循环之间的指令数而不同。在记述地址时，需要考虑表 11.37 所示的内容。

表 11.37 重复循环之间的指令数和地址的设定值

指令数	1 条指令	2 条指令	3 条指令	4 条指令以上
RS 寄存器	s_addr0+8	s_addr0+6	s_addr0+4	s_addr
RE 寄存器	s_addr0+4	s_addr0+4	s_addr0+4	e_addr3+4

- s_addr0 : 重复起始地址的 1 条指令之前的地址
- s_addr : 重复起始地址
- e_addr3 : 重复结束地址的 3 条指令之前的地址

重复循环命令自动生成功能是从 1 条指令自动生成 PC 相对的 LDRS、LDRE 指令和 SETRC 指令的功能。其中，LDRS 和 LDRE 指令将重复循环之间的指令数的地址值传送给 RS 和 RE 寄存器，SETRC 指令设定重复次数。

例如，如果使用本功能，就能将以下程序 (a) 记述为 (b)。

■程序 (a)

```

        LDRS s_addr0+6
        LDRE s_addr0+4
        SETRC #10
s_addr0: NOP
        PADD A0,M0,A0          ; 重复起始地址
        PCMP x1,M0            ; 重复结束地址

```

■程序 (b)

```

        REPEAT s_addr,e_addr,#10
        NOP
s_addr:  PADD A0,M0,A0          ; 重复起始地址
e_addr:  PCMP X1,M0            ; 重复结束地址

```

11.9.2 有关重复循环指令的自动生成功能的扩展指令

对于扩展指令（REPEAT s_label,e_label,#imm、REPEAT s_label,e_label,Rn、REPEAT s_label,e_label）的记述，汇编程序自动生成需要的指令，计算 PC 相对位移量的值。扩展指令的源语句及其展开结果如表 11.38 所示。将扩展指令的源语句展开为 2 条或者 3 条执行指令。

表 11.38 扩展指令的种类和展开结果

扩展指令	展开结果
REPEAT s_label,e_label,#imm	LDRS @(disp,PC)、LDRE @(disp,PC) 和 SETRC #imm
REPEAT s_label,e_label,Rn	LDRS @(disp,PC)、LDRE @(disp,PC) 和 SETRC Rn
REPEAT s_label,e_label	LDRS @(disp,PC) 和 LDRE @(disp,PC)

11.9.3 REPEAT 的记述方法

REPEAT 的记述方法如下：

[< 符号>[:]] △ REPEAT △ < 起始地址>,< 结束地址>[,< 重复次数>]

(1) 语句的元素

- (a) < 起始地址>、< 结束地址>
用标号记述重复循环的起始地址和结束地址。
- (b) < 重复次数>
用立即或者通用寄存器记述重复次数。

(2) 说明

- (a) REPEAT 自动生成执行指令（LDRS、LDRE），用于重复起始地址到结束地址的指令。
- (b) 如果指定重复次数，就生成 SETRC；如果省略重复次数，就不生成 SETRC。

11.9.4 编程例

(1) 基本例（重复的指令数在 4 条指令以上时）

```

                REPEAT RptStart,RptEnd,#5
                PCLR Y0
                PCLR A0
RptStart:      MOVX @R4+,X1  MOVY @R6+,Y1
                PADD A0,Y0,Y0  PMULS X1,Y1,A0
                DCT PCLR A0
                AND  R0,R4
RptEnd:        AND  R0,R6

```

此程序 5 次重复执行 RptStart 到 RptEnd 的 5 条指令。

典型的展开

```

                LDRS RptStart
                LDRE RptEnd3+4
                SETRC #5
                PCLR Y0
                PCLR A0
RptStart:      MOVX @R4+,X1  MOVY @R6+,Y1
RptEnd3:       PADD A0,Y0,Y0  PMULS X1,Y1,A0 ; 实际不生成标号
                DCT PCLR A0
                AND  R0,R4
RptEnd:        AND  R0,R6

```

(2) 重复的指令数为 1 条指令时

必须用相同标号指定起始地址和结束地址。

```

                REPEAT Rpt,Rpt,R0
                MOVX @R4+,X1  MOVY @R6,Y1
Rpt:           PADD A0,Y0,Y0  PMULS X1,Y1,A0  MOVX @R4+,X1  MOVY @R6+,Y1

```

典型的展开

```

                LDRS RptStart0+8
                LDRE RptStart0+4
                SETRC R0
RptStart0:     MOVX @R4+,X1  MOVY @R6,Y1 ; 实际不生成标号
Rpt:           PADD A0,Y0,Y0  PMULS X1,Y1,A0  MOVX @R4+,X1  MOVY @R6+,Y1

```

(3) 重复的指令数为 2 条指令时

```

                REPEAT RptStart,RptEnd,#10
                PCLR Y0
RptStart:      MOVX @R4+,X1  MOVY @R6+,Y1
RptEnd:        PADD A0,Y0,Y0  PMULS X1,Y1,A0

```

典型的展开

```

                LDRS RptStart0+6
                LDRE RptStart0+4
                SETRC #10
RptStart0:     PCLR Y0 ; 实际不生成标号
RptStart:      MOVX @R4+,X1  MOVY @R6+,Y1
RptEnd:        PADD A0,Y0,Y0  PMULS X1,Y1,A0

```


(4) 重复的指令数为 3 条指令时

```

                REPEAT RptStart,RptEnd,R0
                PCLR Y0
RptStart:      MOVX @R4+,X1  MOVY  @R6+,Y1
                PMULS X1,Y1,A0
RptEnd:        PADD A0,Y0,Y0
典型的展开
                LDRE RptStart0+4
                LDRS RptStart0+4
                SETRC R0
RptStart0:     PCLR Y0      ; 实际不生成标号
RptStart:      MOVX @R4+,X1  MOVY  @R6+,Y1
                PMULS X1,Y1,A0
RptEnd:        PADD A0,Y0,Y0

```

(5) 省略重复次数时

如果省略重复次数，汇编程序就不展开 SETRC。在要分离 LDRS、LDRE 和 SETRC 时使用。

```

                REPEAT RptStart,RptEnd
                ; 对此部分进行 LDRS 和 LDRE 的展开。
                MOV #10,R0
OuterLoop:
                SETRC #16
                PCLR Y0
                PCLR A0
RptStart:      MOVX @R4+,X1  MOVY  @R6+,Y1
                PADD A0,Y0,Y0  PMULS X1,Y1,A0
                DCT PCLR A0
                AND  R0,R4
RptEnd:        AND  R0,R6
                DT  R0
                BF  OuterLoop

```

11.9.5 有关 REPEAT 扩展指令的注意事项

(1) 有关起始地址和结束地址的注意事项

能指定为起始地址和结束地址的标号是同段内的标号或者是同局部块内的局部标号。
起始地址必须为 REPEAT 扩展指令后的地址，结束地址必须为起始地址之后的地址。

(2) 有关循环内记述指令的注意事项

- (a) 如果在循环内记述要确保数据或者数据区的控制指令或者 .ORG，汇编程序就输出警告，将 1 条控制指令作为 1 条指令进行重复指令数的计数。如果记述 .ALIGN 后生成调整，汇编程序就输出警告，并将 .ALIGN 作为 1 指令进行重复指令数的计数。该控制指令如下：

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ALIGN、.ORG

- (b) 汇编程序在循环内抑止文字池的自动生成。因此，即使有无条件转移指令，也不为文字池的输出对象。另外，如果记述 .POOL，汇编程序就输出警告，忽视 .POOL。

(3) 有关循环前指令的注意事项

如果重复的指令数不超过 3 条指令，循环前的指令就必须为执行指令或者 DSP 指令。因此，如果重复的指令数不超过 3 条指令并且在起始地址之前的指令为以下的情况，汇编程序就输出错误。

- (a) 确保数据或者数据区的控制指令或者 .ORG

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ORG

- (b) 文字池自动生成功能生成的文字池

在起始地址之前为无条件转移指令 + 延迟槽指令或者 .POOL 的情况下，有可能会自动生成文字池。
为了抑止文字池的生成，必须紧接着延迟槽指令之后记述 .NOPOOL。

- (c) 通过 .ALIGN 生成 1 个字节的调整

指定 .ALIGN 时，如果在 .ALIGN 之前为奇数地址，就有可能生成 1 个字节的调整（例如，定位计数器用 3 指定 .ALIGN 4 时）。此时，在循环前输出不是执行指令的数据，发生错误。如果输出 2 个字节以上的调整，之前的指令就为 NOP，并能正常运行。

(4) 其他注意事项

- (a) 必须在 REPEAT 扩展指令～起始地址之间记述 1 条指令（或者 1 条以上）的执行指令或者 DSP 指令。否则，编程序就输出错误。
- (b) 不能在 REPEAT 扩展指令～结束地址之间记述其他 REPEAT 扩展指令。如果嵌套记述 REPEAT 扩展指令，汇编程序就输出错误。
- (c) 不能在 REPEAT 扩展指令～结束地址之间记述转移指令、TRAPA 指令（未指定 cpu=sh4aldsp 时）和 SR、RS、RE 的装入指令。否则，汇编程序就输出错误。

11.10 扩展重复循环指令的自动生成功能

11.10.1 扩展重复循环命令的自动生成功能的概要

在 SH4AL-DSP 中，通过 LDRS 指令和 LDRE 指令将重复起始地址和重复结束地址设定到 RS 寄存器和 RE 寄存器。

扩展重复循环指令的自动生成功能是从 1 条指令自动生成 PC 相对 LDRS、LDRE 指令和 LDRC 指令的功能。其中，LDRS 和 LDRE 指令将重复的起始地址和重复的结束地址传送到 RS 和 RE 寄存器，LDRC 指令设定重复次数。

例如，如果使用本功能，就能将以下程序 (a) 记述为 (b)。

■程序 (a)

```

                LDRS s_addr
                LDRE e_addr
                LDRC #10
                NOP
s_addr:         PADD A0,M0,A0          ; 重复起始地址
e_addr:         PCMP X1,M0            ; 重复结束地址

```

■程序 (b)

```

                EREPEAT s_addr,e_addr,#10
                NOP
s_addr:         PADD A0,M0,A0          ; 重复起始地址
e_addr:         PCMP X1,M0            ; 重复结束地址

```

11.10.2 有关扩展重复循环指令的自动生成功能的扩展指令

对于扩展指令（EREPEAT s_label,e_label,#imm、EREPEAT s_label,e_label,Rn、EREPEAT s_label,e_label）的记述，汇编程序自动生成需要的指令。

扩展指令的源语句及其展开结果如表 11.39 所示。将扩展指令的源语句展开为 2 个或者 3 个执行指令。

表 11.39 扩展指令的种类和展开结果

扩展指令	展开结果
EREPEAT s_label,e_label,#imm	LDRS @(disp,PC)、LDRE @(disp,PC) 和 LDRC #imm
EREPEAT s_label,e_label,Rn	LDRS @(disp,PC)、LDRE @(disp,PC) 和 LDRC Rn
EREPEAT s_label,e_label	LDRS @(disp,PC) 和 LDRE @(disp,PC)

11.10.3 EREPEAT 的记述方法

EREPEAT 的记述方法如下：

[< 符号 >[:]] △ EREPEAT △ < 起始地址 >,< 结束地址 >[,< 重复次数 >]

(1) 语句的元素

- (a) < 起始地址 >、< 结束地址 >
用标号记述重复循环的起始地址和结束地址。
- (b) < 重复次数 >
用立即或者通用寄存器记述重复次数。

(2) 说明

- (a) EREPEAT 自动生成执行指令（LDRS、LDRE），用于重复起始地址到结束地址的指令。
- (b) 如果指定重复次数，就生成 LDRC；如果省略重复次数，就不生成 LDRC。

11.10.4 编程例

(1) 基本例

```

EREPEAT RptStart,RptEnd,#5
PCLR Y0
PCLR A0
RptStart:  MOVX @R4+,X1  MOVY @R6+,Y1
           PADD A0,Y0,Y0  PMULS X1,Y1,A0
DCT PCLR A0
           AND  R0,R4
RptEnd:    AND  R0,R6

```

此程序 5 次重复执行 RptStart 到 RptEnd 的 5 条指令。

典型的展开

```

LDRS RptStart
LDRE RptEnd3+4
LDRC #5
PCLR Y0
PCLR A0
RptStart:  MOVX @R4+,X1  MOVY @R6+,Y1
           PADD A0,Y0,Y0  PMULS X1,Y1,A0
DCT PCLR A0
           AND  R0,R4
RptEnd:    AND  R0,R6

```

(2) 省略重复次数

如果省略重复次数，汇编程序就不展开 LDRC。在要分离 LDRS、LDRE 和 LDRC 时使用。

```

EREPEAT RptStart,RptEnd
; 对此部分进行 LDRS 和 LDRE 的展开
MOV #10,R0

OuterLoop:
    LDRC #16
    PCLR Y0
    PCLR A0
RptStart: MOVX @R4+,X1  MOVY @R6+,Y1
    PADD A0,Y0,Y0  PMULS X1,Y1,A0
    DCT  PCLR A0
    AND  R0,R4
RptEnd:  AND  R0,R6
    DT  R0
    BF OuterLoop

```

11.10.5 有关扩展 REPEAT 指令（EREPEAT 指令）的注意事项

(1) 有关起始地址和结束地址的注意事项

能指定为起始地址和结束地址的标号是同段内的标号或者是同局部块内的局部标号。
起始地址必须为 EREPEAT 扩展指令后的地址，结束地址必须为起始地址之后的地址。

(2) 有关循环内记述指令的注意事项

- (a) 如果在循环内记述要确保数据或者数据区的控制指令或者 .ORG，汇编程序就输出警告，将 1 条控制指令作为 1 条指令进行重复指令数的计数。如果记述 .ALIGN 后生成调整，汇编程序就输出警告，并将 .ALIGN 作为 1 条指令进行重复指令数的计数。该控制指令如下：
.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ALIGN、.ORG
- (b) 汇编程序在循环内抑止文字池的自动生成。因此，即使是无条件转移指令，也不为文字池的输出对象。另外，如果记述 .POOL，汇编程序就输出警告，忽视 .POOL。

(3) 有关循环前指令的注意事项

循环前的指令必须为执行指令或者 DSP 指令。因此，如果起始地址之前的指令为以下的情况，汇编程序就输出错误。

- (a) 确保数据或者数据区的控制指令或者令 .ORG
.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ORG
- (b) 文字池自动生成功能生成的文字池
在起始地址之前为无条件转移指令 + 延迟槽指令或者 .POOL 的情况下，有可能会自动生成文字池。
为了抑止文字池的生成，必须紧接着延迟槽指令之后记述 NOPOOL。

(c) 通过 .ALIGN 生成 1 个字节的调整

指定 .ALIGN 时，如果在 .ALIGN 之前为奇数地址，就有可能生成 1 个字节的调整（例如，定位计数器用 3 指定 .ALIGN 4 时）。此时，在循环前输出不是执行指令的数据，发生错误。如果输出 2 个字节以上的调整，之前的指令就为 NOP，并能正常运行。

(4) 其他注意事项

- (a) 不能给 EREPEAT 的最后指令记述其他 EREPEAT 扩展指令。如果嵌套记述 EREPEAT 扩展指令，汇编程序就输出错误。
- (b) 不能给 EREPEAT 的最后指令记述延迟转移指令和 SR、RS、RE 的装入指令。否则，汇编程序就输出错误。

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。
(W)	警告	继续处理。
(E)	错误	继续选项的解析处理，中断处理。
(F)	致命错误	中断处理。
(-)	内部错误	中断处理。

12.2 信息一览

C0001 (I) Character combination "String literal" in comment

在注释中有 " 字符串 "。

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

对于赋值表达式，在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函数原型指定的参数型。

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 " in " 函数名 "

参照没有设定值的局部变量。

- C0012 (I) Unused variable " 变量名 "**
有不使用的变量。
- C0015 (I) No return value**
在返回型不是 `void` 型的函数中，返回语句没有返回值。
- C0016 (I) Conversion in case constant expression**
将 `Case` 标号的常数表达式转换为控制表达式的通用整数扩展的型。
- C0017 (I) Missing return statement**
在返回型不是 `void` 型的函数中，存在没有返回语句的控制路径。
- 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**
没有函数的函数原型声明。
- 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，而另一边是指向不是 `0` 的标量型。
- 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**
字符常数的长度超过了 1 个字符。
- C1008 (W) Identifier too long**
标识符的长度超过了 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了 4 个字符。

C1012 (W) Floating point constant overflow

浮点常数的值超出了范围。根据符号，假设与 $+\infty$ 或者 $-\infty$ 对应的内部表示值。

C1013 (W) Integer constant overflow

整数型常数的值超出了 `unsigned long` 型的范围。假设上溢的高位被忽视的值。

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示的最小值。假设常数值为 0.0。

C1016 (W) Argument mismatch

函数原型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示。

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示。

C1019 (W) Illegal constant expression

在常数表达式中，关系运算符 "`<`"、"`>`"、"`<=`" 或者 "`>=`" 的操作数是指向不同型的指针型。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 "-" 的操作数是指向不同型的指针型。假设结果值为 0。

C1021 (W) Register saving pragma conflicts in interrupt function "函数名"

与对于 "函数名" 所示的中断函数，控制寄存器保存和恢复的 `#pragma` 不适当。忽视 `#pragma` 指定。

C1022 (W) First operand of "运算符" is not lvalue

第 1 操作数的 "运算符" 不能是左边值。

C1023 (W) Can not convert Japanese code "代码" to output type

日语代码不能转换为指定的输出代码。

C1024 (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。

C1027 (W) Invalid #pragma gbr_base/gbr_base1

同时指定 `gbr=auto` 和 `#pragma gbr_base`、`#pragma gbr_base1`。忽视 `#pragma` 指定。

- C1028 (W)#pragma " 识别符 " has no effect**
指定的 `#pragma` 识别符无效。
- C1029 (W)Function with ifunc calls " 函数名 " without ifunc**
从 `ifunc` 指定的函数中调用了无 `ifunc` 指定的函数。
- C1030 (W) Bit order mismatch**
在结构体和结构体成员中 `bit_order` 不同。
- C1031 (W) Multiple #pragma for one function**
对 1 个函数重复指定了 `#pragma`。
- C1200 (W) Division by floating point zero**
在常数表达式中, 进行了以浮点型 `0.0` 为除数的除法运算。根据符号, 假设与 $+\infty$ 或者 $-\infty$ 对应的内部表示值。
- C1201 (W) Ineffective floating point operation**
在常数表达式中, 进行了 " $\infty-\infty$ "、"`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示值。
- 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。
- C1301 (W) " 选项 " option ignored**
忽视 " 选项 " 进行编译。
- C1302 (W) "double=float" option ignored**
同时指定了 `double=float`、`cpu=sh2afpu|sh4|sh4a`。忽视 `double=float` 而解释为 `fpu=single` 后进行编译。
- C1304 (W) " 单片机种类 1 " is interpreted as " 单片机种类 2 "**
`cpu=< 单片机种类 1 >` 无效。在解释为 `cpu=< 单片机种类 2 >` 后进行编译。
- C1308 (W) Duplicate number specified in option " 选项 " : " 序号 "**
在 " 选项 " 中指定了相同的序号。
- C1309 (W) Section name "S" specified**
给段名指定了 `S`。有可能和编译程序生成的堆栈区的段名重叠。
- C1310 (W) "repeat" option ignored**
忽视 `repeat` 选项进行编译。
- C1311 (W) "softpipe" option ignored**
忽视 `softpipe` 选项进行编译。

C1312 (W) "fdiv" option ignored

忽视 `fdiv` 选项进行编译。

**C1314 (W) File_inline " 文件名 " ignored by #pragma global_register mismatch
#pragma global_register 的指定有矛盾。**

忽视 `file_inline` 选项进行编译。

C1315 (W) File_inline " 文件名 " ignored by same file as source file

通过 `file_inline` 选项指定了编译对象文件。

忽视 `file_inline` 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 `inline` 展开。忽视 `#pragma inline` 的指定。

C1402 (W) #pragma " 识别符 " ignored

忽视 `#pragma` " 识别符 " 的指定。

C1405 (W) Illegal #pragma syntax

作为关键字指定了不存在的 `#pragma`。

C1501 (W) Division by zero

在常数计算中，进行了被零除的运算。

C1600 (W) Debugging information describing location of " 名称 " is lost

不输出 " 名称 " 的符号信息。

C1700 (W) Memory qualifier ignored

忽视存储器型的限定词。

**C1701 (W) Conversion from pointer without memory qualifier to pointer with
memory qualifier**

进行了从没有存储器型限定词的指针到有存储器型限定词的指针的转换。存储器型限定词无效。

**C1702 (W) Conversion from pointer with circular qualifier to pointer without
circular qualifier**

进行了从有 `_ _circ` 型限定词的指针到没有 `_ _circ` 型限定词的指针的转换。`_ _circ` 型限定词无效。

C1703 (W) Fixed point constant overflow

极限常数值超出了值的范围。

C1704 (W) Out of Fixed point

极限常数的有效位数超过了 17 位。

C1705 (W) Modulo addressing may be illegal in function " 函数名 "

有可能在 " 函数名 " 所示的函数中进行了错误的模寻址。

- C1800 (W) Variable " 变量名 " type mismatch in files**
用 " 变量名 " 表示的变量的型在文件之间为不同。
不能指定 `file_inline` 选项。
- 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误。
- C2002 (E) Missing ", "**
在有参数的 `#define` 语句中, 没有分隔参数排列的逗号 ", "。
- C2003 (E) Missing ")"**
在判断是否用 `#define` 语句定义名称的 `defined` 表达式中, 名称的后面没有右括弧 ")"。
- C2004 (E) Missing ">"**
在 `#include` 语句的文件名的指定中, 文件名的后面没有 ">"。
- C2005 (E) Cannot open include file " 文件名 "**
不能打开 `#include` 语句指定的文件名的文件。
- C2006 (E) Multiple #define's**
由 `#define` 语句重新定义了相同的宏名。
- C2008 (E) Processor directive #elif mismatches**
没有对应 `#elif` 语句的 `#if` 语句、`#ifdef` 语句、`#ifndef` 语句、`#elif` 语句。
- C2009 (E) Processor directive #else mismatches**
没有对应 `#else` 语句的 `#if` 语句、`#ifdef` 语句、`#ifndef` 语句。
- C2010 (E) Macro parameters mismatch**
宏调用的参数个数和宏定义的参数个数不同。
- C2011 (E) Line too long**
宏展开后的源程序的行超过了极限值。
- C2012 (E) Keyword as a macro name**
将处理器规定的关键字定义为 `#define` 语句或者 `#undef` 语句的宏名。
- C2013 (E) Processor directive #endif mismatches**
没有对应 `#endif` 行的 `#if` 语句、`#ifdef` 语句、`#ifndef` 语句。
- C2014 (E) Missing #endif**
在没有对应 `#if` 语句、`#ifdef` 语句、`#ifndef` 语句文的 `#endif` 行的情况下, 文件结束。

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极限值。

C2017 (E) Missing "

在 `#include` 语句的文件名的指定中，文件名的后面没有 “”。

C2018 (E) Illegal #line

`#line` 语句指定的行数超过了极限值。

C2019 (E) File name too long

文件名的字符数超过了极限值。

C2020 (E) System identifier "名称" redefined

定义了和内部函数同名的符号。

C2021 (E) Invalid number specified in option "选项" : "序号"

通过 "选项" 指定了无效值。请确认值的范围。

C2022 (E) Error level message cannot be changed : "change_message"

对于 **Error** 级的信息，不能更改信息级。

C2100 (E) Multiple storage classes

在声明中指定的存储区类说明符超过 1 个。

C2101 (E) Address of register

对于具有 **register** 存储类的变量，使用了一元运算符 `&`。

C2102 (E) Illegal type combination

型说明符的组合有误。

C2103 (E) Bad self reference structure

用与主结构体或者联合体相同的型，声明了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是整数型，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

指定了元素的型为函数型的数组型。

C2108 (E) Function returning array

指定了返回值的型为数组型的函数型。

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，将 `auto` 或者 `register` 指定为存储类。

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`、`long long`、`unsigned long long`、`bool`、`enum` 中的任一型、或者这些型和 `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、64 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 " 引数名 "

对函数的参数声明列中没有的标识符进行了参数声明。

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

数组元素个数的值指定了小于 1 或者大于 2147483647 的整数值。

C2127 (E) Missing array size

没有指定数组元素的个数。

C2128 (E) Illegal pointer declaration for "*"

在表示指针型的声明 "*" 的后，立即指定了 `const`、`volatile` 以外的型说明符。

C2129 (E) Illegal initializer type

在变量初始值的指定中，初始值的型不是能赋值给变量的型。

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变量的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，还进行了参数声明。

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` 型的指定方法有误。在以下三种情况下能指定 `void` 型：

- (1) 指定为指针的指向型；
- (2) 指定为函数的返回型；
- (3) 明确指定函数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。

C2144 (E) Type mismatch

具有 `Extern` 存储类的相同名称的变量或者函数的型不一致。

C2145 (E) Const/volatile specified for incomplete type

给不完全型指定了 `const` 或者 `volatile`。

C2200 (E) Index not integer

数组的下标表达式不为整数型。

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函数原型的参数型。

C2202 (E) Number of parameters mismatch

函数调用中参数的个数和函数原型的参数的个数不一致。

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 `"++"`、`"--"` 用于不可赋值的左边值（数组型、`const` 型除外的左边值）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 "~" 用于标量型的表达式。

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 "%=" 用于标量型的表达式。

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数型时，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。在以下三种情况下允许二元运算符 " - " 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 个操作数是标量型。

C2218 (E) Scalar required

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。在以下六种情况下允许条件运算符 "?" : " 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` 的整数型常数或者将 `0` 的整数型常数转换为指向 `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不可赋值的左边值（数组型、`const` 型除外的左边值）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 "=" 运算符 "

赋值运算符 `"="` 两边的表达式不一致。在以下五种情况下允许赋值运算符 `"="` 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` 的整数型常数或者将 `0` 的整数型常数转换为指向 `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。在以下三种情况下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` 的整数型常数或者指向 `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。在以下二种情况下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了数组。

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或者极限型的型转换。

C2236 (E) Illegal cast in constant

在常数表达式中，将指针型转换为 char 型或者 short 型。

C2237 (E) Illegal constant expression

在常数表达式中，对指针型的常数 $\&$ 整数型的数据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

段的命名有误。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 标号的表达式不是标量型的常数表达式。

C2307 (E) Multiple case labels

在 1 个 **switch** 语句中重复指定了同值的 **case** 标号。

C2308 (E) Multiple default labels

在 1 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 语句、**for** 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）不是标量型。

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 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。

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

注释的嵌套超过了 255 层。

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，整数型或者极限型数据被零除。

C2600 (E) 字符串

如果没有指定 `nolistfile` 选项，就将 `#error` 的字符串指定的错误信息显示在列表文件。

C2650 (E) Invalid pointer reference

所指定的地址值和调整数不一致。

C2700 (E) Function " 函数名 " in #pragma interrupt already declared

在中断函数声明 `#pragma interrupt` 中指定的函数全部被声明为通常的函数。

C2701 (E) Multiple interrupt for one function

对于 1 个函数，重复声明了中断函数声明的 `#pragma interrupt`。

C2702 (E) Multiple #pragma interrupt options

重复指定了同类中断规格。

C2703 (E) Illegal #pragma interrupt declaration

中断函数声明 `#pragma interrupt` 的规格有误。

C2704 (E) Illegal reference to interrupt function

不正确地参照了中断函数。

C2705 (E) Illegal parameter in interrupt function

在中断函数中，使用的参数的型不一致。

C2706 (E) Missing parameter declaration in interrupt function

没有中断函数的选项指定所使用的变量声明。

C2707 (E) Parameter out of range in interrupt function

中断函数的参数 `tn` 的值超过了 256。

C2709 (E) Illegal section name declaration

`#pragma section` 的指定有误。

C2710 (E) Section name too long

所指定的段名的字符数超过了 31 个字符。

C2711 (E) Section name table overflow

所指定的段数在 1 个文件中超过了 64 个。

C2712 (E) GBR based displacement overflow

`#pragma gbr_base`、`#pragma gbr_base1` 声明的变量区发生了溢出。

C2713 (E) Illegal #pragma interrupt function type

`#pragma interrupt` 指定的函数的型不正确。

C2799 (E) GBR used in-line function

在指定 `gbr=auto` 时，不能使用 GBR 关联的内部函数。

C2800 (E) Illegal parameter number in in-line function

在内部函数中，使用的参数的个数不一致。

C2801 (E) Illegal parameter type in in-line function

在内部函数中，参数的型不一致。

C2802 (E) Parameter out of range in in-line function

在内部函数中，参数大小超过了可指定的范围。

C2803 (E) Invalid offset value in in-line function

在内部函数中，参数的指定不正确。

C2804 (E) Illegal in-line function

在所指定的 `cpu` 选项中，有不能使用的内部函数。

C2805 (E) Function " 函数名 " in #pragma inline/inline_asm already declared

在指定 `#pragma` 指定前，有 " 函数名 " 所示的函数。

C2806 (E) Multiple #pragma for one function

对于 1 个函数，指定了多种矛盾的 `#pragma`。

C2807 (E) Illegal #pragma inline/inline_asm declaration

`#pragma inline` 或者 `#pragma inline_asm` 的指定方法有误。

- C2808 (E) Illegal option for #pragma inline_asm
与 #pragma inline_asm 的指定无关, 指定了 code=machinecode。
- C2809 (E) Illegal #pragma inline/inline_asm function type
指定 #pragma inline 或者 #pragma inline_asm 的识别符的种类有误。
- C2810 (E) Global variable " 变量名 " in #pragma gbr_base/gbr_base1 already declared
在指定 #pragma 前, 定义了 " 变量名 " 所示的变量。
- C2811 (E) Multiple #pragma for one global variable
对于变量, 指定了多种矛盾的 #pragma。
- C2812 (E) Illegal #pragma gbr_base/gbr_base1 declaration
#pragma gbr_base、#pragma gbr_base1 的指定方法有误。
- C2813 (E) Illegal #pragma gbr_base/gbr_base1 global variable type
指定 #pragma gbr_base、#pragma gbr_base1 的识别符的种类有误。
- C2814 (E) Function " 函数名 " in #pragma noregsave/noregalloc/regsave already declared
在指定 #pragma 前, 有 " 函数名 " 所示的函数。
- C2815 (E) Illegal #pragma noregsave/noregalloc/regsave declaration
#pragma noregsave、#pragma noregalloc、#pragma regsave 的指定方法有误。
- C2816 (E) Illegal #pragma noregsave/noregalloc/regsave function type
指定 #pragma noregsave、#pragma noregalloc、#pragma regsave 的识别符的种类有误。
- C2817 (E) Symbol " 识别符 " in #pragma abs16/abs20/abs28/abs32 already declared
在指定 #pragma 前, 声明了 " 识别符 " 所示的名称。
- C2818 (E) Multiple #pragma for one symbol
对于同一识别符, 指定了多种矛盾的 #pragma。
- C2819 (E) Illegal #pragma abs16/abs20/abs28/abs32 declaration
#pragma abs16/abs20/abs28/abs32 的指定方法有误。
- C2820 (E) Illegal #pragma abs16/abs20/abs28/abs32 symbol type
指定 #pragma abs16/abs20/abs28/abs32 的识别符的种类有误。
- C2821 (E) Global variable " 变数名 " in #pragma global_register already declared
指定 #pragma global_register 的变数名全部被定义。
- C2822 (E) Illegal register " 寄存器 " in #pragma global_register
指定 #pragma global_register 的寄存器不正确。

- C2823 (E) Illegal #pragma global_register declaration
#pragma global_register 的指定方法有误。
- C2824 (E) Illegal #pragma global_register type
指定了不能指定 #pragma global_register 的变量。
- C2828 (E) Illegal #pragma entry declaration
#pragma entry 声明有语法错误。
- C2829 (E) Function " 函数名 " in #pragma entry already declared
在声明 #pragma entry 前，有同名符号或者已指定了 #pragma。
- C2830 (E) Illegal #pragma entry function type
所指定的符号不是函数。
- C2831 (E) Multiple #pragma entry declaration
有多个 #pragma entry 声明。
- C2832 (E) Illegal #pragma stacksize declaration
#pragma stacksize 声明有语法错误。
- C2833 (E) Multiple #pragma stacksize declaration
有多个 #pragma stacksize 声明。
- C2840 (E) Illegal #pragma ifunc declaration
#pramga ifunc 声明有语法错误。
- C2841 (E) Illegal #pragma ifunc function type
#pragma ifunc 指定的识别符的种类有误。
- C2842 (E) Function " 函数名 " in #pragma ifunc already declared
在指定 #pragma 前，声明了 " 函数名 " 所示的名称。
- C2843 (E) Illegal floating type used in function
在函数中使用了 float 和 double。在声明 #pragma ifunc 时，不能使用 float 和 double。
- C2844 (E) Illegal #pragma pack/unpack declaration
#pragma pack/unpack 声明有语法错误。
- C2845 (E) Illegal #pragma bit_order declaration
#pragma bit_order 声明有语法错误。
- C2846 (E) Packed structure used in in-line function
在内部函数中，使用了 pack 指定的结构体。

- C2847 (E) Illegal #pragma tbr declaration
#pragma tbr 的指定方法有误。
- C2848 (E) Function " 函数名 " in #pragma tbr already declared
在指定 #pragma tbr 前, 有 " 函数名 " 所示的函数。
- C2849 (E) Illegal offset in #pragma tbr
偏移量的指定不正确。
- C2850 (E) Illegal #pragma tbr function type
所指定的符号不是函数。
- C2851 (E) Too many function in tbr
Tbr 指定的函数个数超过了极限值。
- C2852 (E) Variable " 变量名 " in #pragma address already declared
指定 pragma address 的变量名已被定义。
- C2853 (E) Illegal #pragma address symbol type
通过 #pragma address 指定了结构体 / 联合体的成员或者变量名以外的符号。
- C2854 (E) Illegal address in #pragma address
指定的地址为以下任意一种情况:
(1) 给调整数 4 的变量、结构体指定了 4 的倍数以外的地址;
(2) 给调整数 2 的变量、结构体指定了 2 的倍数以外的地址;
(3) 给不同的变量指定了相同的地址;
(4) 对于不同的变量, 变量的地址重叠;
(5) 指定了 #pragma abs16/abs20/abs28, 绝对地址不在该范围内;
(6) 指定了 abs16/abs20/abs28 选项, 绝对地址不在该范围内。
- C2855 (E) All registers are used in #pragma global_register
由 #pragma global_register 指定的寄存器个数不够。
- C2856 (E) Illegal usage of in-line function " 函数名 "
内部函数 " 函数名 " 的使用方法有误。
- C2900 (E) Incompatible memory qualifiers
要将存储器型限定词转换为不同的指针型。
- C2901 (E) Illegal type qualifier
型限定词的指定有误。
- C2902 (E) Illegal arithmetic conversion
算术转换有误。

C2903 (E) Illegal circ specification

`_circ` 的指定有误。

C3000 (F) Statement nest too deep

`If` 语句、`while` 语句、`for` 语句、`do` 语句和 `switch` 语句的嵌套超过了极限值。

C3001 (F) Block nest too deep

复句的嵌套超过了极限值。

C3006 (F) Too many parameters

在函数声明或者调用中，参数的个数超过了极限值。

C3007 (F) Too many macro parameters

在宏定义或调用中，参数的个数超过了极限值。

C3008 (F) Line too long

宏展开后的 1 行的长度超过了极限值。

C3009 (F) String literal too long

字符串的字符数超过了 32766 个字符。字符串的字符数是连接连续指定的字符串后的字节数。此字符串的字符数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` 语句的个数超过了极限值。

C3014 (F) For nest too deep

`for` 语句的嵌套超过了极限值。

C3015 (F) Symbol table overflow

编译程序生成的符号的个数超过了极限值。

C3016 (F) Internal label overflow

编译程序生成的内部标号的个数超过了极限值。

C3017 (F) Too many case labels

在 1 个 `switch` 语句中 `case` 标号的个数超过了极限值。

C3018 (F) Too many goto labels

在 1 个函数中定义的 `goto` 标号的个数超过了极限值。

C3019 (F) Cannot open source file " 文件名 "

不能打开源文件。

C3020 (F) Source file input error " 文件名 "

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 语句的嵌套超过了极限值。

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 2147483647 个字节。

C3100 (F) Misaligned pointer access

要使用调整不正确的指针进行参照或者设定。

C3201 (F) Object size overflow

目标大小超过了 4G 字节。

C3203 (F) Assembly source line too long

输出的汇编源程序的 1 行过长。

C3204 (F) Illegal stack access

函数内使用的堆栈大小（包含局部变量区、寄存器保存区以及其他函数调用的参数进栈区等）或者该函数调用的参数区超过了 2G 个字节。

C3205 (F) Cannot apply repeat operation for a loop

对于循环，不向 DSP 扩展重复循环进行展开。必须使循环变小或者解除 repeat 选项的指定。

C3300 (F) Cannot open internal file

在以下 3 种情况下有可能发生错误:

- (1) 不能打开编译程序内部生成的中间文件;
- (2) 中间文件和同名的文件已存在;
- (3) 不能打开编译程序内部使用的文件。

C3301 (F) Cannot close internal file

不能关闭编译程序内部生成的中间文件。请确认编译程序的安装步骤是否有误。

C3302 (F) Cannot input internal file

不能读编译程序内部生成的中间文件。请确认编译程序的安装步骤是否有误。

C3303 (F) Cannot output internal file

不能写编译程序内部生成的中间文件。请增大磁盘空间的容量。

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误。

C3306 (F) Interrupt in compilation

在编译处理中从标准输入终端检测到 (Ctrl)+C 命令的中断。

C3307 (F) Compiler version mismatch

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。

C3308 (F) Cannot create file " 文件名 "

不能建立编译程序生成的文件。

C3320 (F) Command parameter buffer overflow

命令行的指定超过了 4096 个字符。

C3321 (F) Illegal environment variable

在以下 4 种情况下发生错误:

- (1) 没有设定环境变量 SHC_LIB ;
- (2) 没有给环境变量 SHC_LIB 指定编译程序的执行文件名;
- (3) 在设定环境变量 SHC_LIB 时, 是否指定了违反规约的文件名或者路径名的大小超过了 118 个字符;
- (4) 不给环境变量 SHCPU 设定 "SH1"、"SH2"、"SH2E"、"SH2DSP"、"SHDSP"、"SH2A"、"SH2AFPU"、"SH3"、"SH3DSP"、"SH4"、"SH4A"、"SH4ALDSP"。

C3322 (F) Current directory cannot be read to get its name

不能取得当前文件夹的信息。

C4000-C4999 (-) Internal error

在编译程序的内部处理中发生某种障碍。请与购入本编译程序的营业部门或者代理店联系，反映错误的发生情况。

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他应用程序。

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

有无法认识的字符 (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 进制数的记述有误。
- C5024 (E) Invalid octal digit
8 进制数的记述有误。
- 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
浮点型的值超出了范围。
- C5031 (E) Expression must have integral type
表达式的型必须是整数型。
- 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 语句的指定方法有误。

C5037 (E) The #endif for this directive is missing

#endif 行的指定方法有误。

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
在常数表达式中，不能指定此运算符。
- C5058 (E) This operator is not allowed in a preprocessing expression
在预处理器的表达式中，不能指定此运算符。
- C5059 (E) Function call is not allowed in a constant expression
在常数表达式中，不能调用函数。
- C5060 (E) This operator is not allowed in an integral constant expression
在整数型常数表达式中，不能指定此运算符。
- C5061 (W) Integer operation result is out of range
整数的运算结果超出了值的范围。假设为上溢的高位被忽视的值。
- 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
枚举型成员的值超出了 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数型转换。照样设定位串。
- C5069 (W) Integer conversion resulted in truncation
进行了舍去高位字节的整数型转换。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
不能指定以函数为元素的数组。
- C5089 (E) Array of void is not allowed
不能指定以 void 型为元素的数组。
- C5090 (E) Function returning function is not allowed
不能指定以函数型为返回型的函数。
- C5091 (E) Function returning array is not allowed
不能指定以数组为返回型的函数。
- 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
不能将此型作为数组的元素。
- 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
指定了带符号整数型的长度为 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，有可能被优化删除。
- 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
只在循环中，continue 语句才有效。
- C5116 (E) A break statement may only be used within a loop or switch
只在循环或者 switch 语句中，break 语句才有效。
- 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
在形式参数中不能重复声明型名。
- 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 class 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
表达式必须是整数型的常数表达式。

- 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
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
在将浮点型的值转换为整数型时, 超出了值的范围。

C5174 (I) Expression has no effect

无效果的表达式。有可能被优化删除。

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。请确认文件夹是否存在。

C5183 (E) Type of cast must be integral

型转换的型必须是整数型。

C5184 (E) Type of cast must be arithmetic or pointer

数据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

对 0 无符号整数进行了无意义的比较。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
实际参数的个数少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枚举型的值。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 1 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)

有基类的存取控制的指定。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

构造函数和析构造函数没有返回值。

C5279 (E) Invalid destructor declaration

析构造函数的声明不正确。

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
不能将虚拟基类的 "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"
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
重新定义了默认参数。
- 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 "名称" 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

只在函数声明中，inline 说明符才有效。

C5326 (E) "inline" is not allowed

不允许 inline 的指定。

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够。
- C5346 (E) Nonmember operator requires a parameter with class type
非成员函数的运算符函数需要 class 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 "演算子" 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
成员名或者基类有误。
- 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 (W) 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:
不允许抽象类的 "型" 的型转换。
- 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

在经过指向 "型" 的指针或目标后，不能存取限定公开名 "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"

没有适当的从 "型 1" 转换到 "型 2" 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

不允许枚举型声明。

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" 的实际参数不够。

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
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"。以后的 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
不能对模板 "名称" 进行实体化。
- 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
不能对 inline 函数 "名称" 进行实体化。
- C5488 (E) Pure virtual entity-kind "名称" cannot be explicitly instantiated
不能对纯虚拟函数 "名称" 进行实体化。
- C5489 (E) Entity-kind "名称" cannot be instantiated -- no template definition was supplied
由于没有定义模板, 所以不能对 "名称" 进行实体化。

- C5490 (E) Entity-kind "名称" cannot be instantiated -- it has been explicitly specialized
不能对 "名称" 进行实体化。
- 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
模板的参数不够。
- 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

不允许枚举型。

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"

不能进行转换到不完全型 "型" 的型转换。

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 "
有可能根据以前的 " 型 " 的处理程序，进行了处理程序的屏蔽。
- 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
执行异常处理程序。
- 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 / 内部连接的指定发生冲突。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"
"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" 有误。
- 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" 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 inline 指定的模板函数。

C5614 (F) Invalid error number: " 指定序号 "

通过 change_message 选项指定了无效值。请确认值的范围。

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 " 不是正确的文件夹。

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 ")

inline 的指定和以前的声明 " 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。

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 指定的语法有误。

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 inline 展开函数调用 "名称"。

C5679 (I) Entity-kind "名称" cannot be inlined

不能 inline 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

失去了 "名称" 的数据型转换的结果 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 class 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 class 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 参照的操作数必须是完全 class 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 class 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
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 型或者能转换为 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) Redclaration 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
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
在定义模板函数后, 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 template<> 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。忽视默认参数。
- 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 的异常指定和 "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 inline 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他模板的形式参数。
- 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数型或者枚举型。
- C5848 (E) Expression must have arithmetic or enum type
表达式的型必须是算术型或者枚举型。

- C5849 (E) Expression must have arithmetic, enum, or pointer type
表达式的型必须是算术型、枚举型或者指针型。
- C5850 (E) Type of cast must be integral or enum
型转换的型必须是整数型或者枚举型。
- C5851 (E) Type of cast must be arithmetic, enum, or pointer
型转换的型必须是算术型、枚举型或者指针型。
- 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
在包含该成员的 namespace 中，必须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，不允许非整数型的运算。

C5875 (E) Embedded C++ does not support templates
Embedded C++ 规格不支持模板功能。

C5876 (E) Embedded C++ does not support exception handling
Embedded C++ 规格不支持异常处理功能。

C5877 (E) Embedded C++ does not support namespaces
Embedded C++ 规格不支持 namespace 功能。

C5878 (E) 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 (E) Embedded C++ does not support the new cast syntax
Embedded C++ 规格不支持 new 的数据型转换功能。

C5880 (E) Embedded C++ does not support using-declarations
Embedded C++ 规格不支持 using 声明功能。

C5881 (E) Embedded C++ does not support "mutable"
Embedded C++ 规格不支持 mutable 功能。

C5882 (E) 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枚举型的形式参数。

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 的使用方法有误。

C5929 (E) Incorrect use of va_arg

va_arg 的使用方法有误。

C5930 (E) Incorrect use of va_end

va_end 的使用方法有误。

C5935 (E) "typedef" may not be specified here

不能指定 typedef。

- C5936 (W) Redclaration of entity-kind "名称" alters its access
通过 "名称" 的重新声明更改了存取的指定。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
重复指定了 using 声明 "名称"。重复的 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 class 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"。按照位字段的型的长度，继续处理位数。
- 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
用未连接的型声明了有连接的变量。作为有连接的变量。
- C5962 (W) Use of a type with no linkage to declare a function
用未连接的型声明了有连接的函数。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。
- C5986 (E) Expected a section name string
_ _sectop/_ _secend/_ _seclsize 没有段名。
- C5988 (E) Invalid pragma declaration
#pragma 的语法不正确。
- C5989 (E) "名称" has already been specified by other pragma
此符号已被其他 #pragma 指定。
- C5990 (E) Pragma may not be specified after definition
不能只给定义符号后的声明指定 #pragma。
- C5991 (E) Invalid kind of pragma is specified to this symbol
指定了不正确的 #pragma。

- C5994 (W) Operator new and operator delete cannot be given internal linkage
用 static 定义了 operator new/operator delete。
- C5995 (E) Storage class "mutable" is not allowed for anonymous unions
不能将 mutable 指定为无名联合体。
- C5997 (E) Abstract class type "型" is not allowed as catch type:
不能用 catch 接受抽象类。
- C5998 (E) A qualified function type cannot be used to declare a nonmember function or a static member function
带限定的函数型不能用于非成员函数和 static 成员函数的声明。
- C5999 (E) A qualified function type cannot be used to declare a parameter
带限定的函数型不能用于函数参数的指定。
- C6000 (E) Cannot create a pointer or reference to qualified function type
不能建立指向带限定的函数型的指针型和参照型。
- C6001 (W) Extra braces are nonstandard
集合型的初始化子列表中有多余的 '{'。
- C6002 (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。
- C6005 (E) Expected "class"
指定为模板参数的类模板需要类。
- C6006 (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。
- 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 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。
- C6009 (E) "型 1" is not compatible with "型 2"
指定的类模板与模板参数的形式不一致。
- C6010 (W) A storage class may not be specified here
在此不能指定存储区类说明符。

C6011 (E) Class member designated by a using-declaration must be visible in a direct base class

在参照可能的直接基底类中，必须指定类成员的 using。

C6016 (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。

C6017 (E) Recursive instantiation of default argument

递归生成模板函数的默认值参数的示例。

C6018 (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他型参数。

C6019 (E) " 示例名 " is not an entity that can be defined

要生成没有实体的示例。

C6023 (E) A qualified friend template declaration must refer to a specific previously declared template

必须在参照前定义限定的友元模板。

C6028 (E) " 类名 " has no member class " 成员名 "

使用了非类的成员。

C6029 (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。

C6030 (E) Recursive instantiation of template default argument

生成了模板的默认参数的递归示例。

C6031 (E) Access declarations and using-declarations cannot appear in unions

在 union 中不能使用 using 指定。

C6032 (E) " 名称 " is not a class member

不是类的成员。

C6038 (W) Invalid redeclaration of nested class

在类中二重定义了类。

C6045 (E) " 模板名 " cannot be declared in this scope

在此作用域中不能声明 template。

12.3 C 标准库函数的错误信息

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 <errno.h> 定义的宏 `errno`。在错误号中定义了对应的错误信息，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>
#include      <errno.h>

main()
{
    FILE *fp;

    fp=fopen("file", "w");
    fp=NULL;

    fclose(fp);                                /* error occurred */

    printf("%s \n", strerror(errno)); /* print error message */
}
```

说明

- (1) 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。
- (2) 如果将错误号作为实际参数传递给 `strerror` 函数，`strerro` 函数就返回对应的错误信息的字符串指针。通过 `printf` 函数的字符串的输出指定，输出错误信息。

标准库的错误信息一览表

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、atof、atoi、atol、atoll、atolfixed、atolaccum、strtod、strtol、strtoul、strtoll、stroull、strtolfixed、strtolaccum、perror、fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf、acos、acosh、asin、asinh、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 没有定义数学函数参数的结果值。	acos、acosh、asin、asinh、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
1104 (ESTRN)	Too long string 字符串的字符数超过了 512 个字符。	atof、atoi、atol、atoll、atolfixed、atolaccum、strtod、strtol、strtoul、strtoll、stroull、strtolfixed、strtolaccum
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 基数的指定有误。	strtol、strtoul、strtoll、strtoull
1202 (ETLN)	Number too long 表示数值的字符串的字符数超过了有效位数。	atof、atolfixed、atolaccum、strtod、strtolfixed、strtolaccum、fscanf、scanf、sscanf
1204 (EEXP)	Exponent too large 指数的位数超过了 3 位。	atof、strtod、fscanf、scanf、sscanf
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。	atof、strtod、fscanf、scanf、sscanf
1210 (EFLOATO)	Overflow out of float float 型的 10 进制数值超出了 float 型的范围（上溢）。	fscanf、scanf、sscanf
1220 (EFLOATU)	Underflow out of float float 型的 10 进制数值超出了 float 型的范围（下溢）。	fscanf、scanf、sscanf
1250 (EDBLO)	Overflow out of double double 型的 10 进制数值超出了 double 型的范围（上溢）。	fscanf、scanf、sscanf
1260 (EDBLU)	Underflow out of double double 型的 10 进制数值超出了 double 型的范围（下溢）。	fscanf、scanf、sscanf

错误号	错误信息 / 说明	设定错误号的函数
1270 (ELDBLO)	Overflow out of long double long double 型的 10 进制数值超出了 long double 型的范围（上溢）。	fscanf、scanf、sscanf
1280 (ELDBLU)	Underflow out of long double long double 型的 10 进制数值超出了 long double 型的范围（下溢）。	fscanf、scanf、sscanf
1300 (NOTOPN)	File not open 不能打开文件。	fclose、fflush、setbuf、setvbuf、fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf、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 带格式的输入 / 输出函数指定的格式有误。	fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf、perror
1400 (EFIXEDO)	Overflow out of __fixed __fixed 型的 10 进制数值超出了 __fixed 型的范围（上溢）。	fscanf、scanf、sscanf
1410 (EFIXEDU)	Underflow out of __fixed __fixed 型的 10 进制数值超出了 __fixed 型的范围（下溢）。	fscanf、scanf、sscanf
1420 (EACCUMO)	Overflow out of __accum __accum 型的 10 进制数值超出了 __accum 型的范围（上溢）。	fscanf、scanf、sscanf
1430 (EACCUMU)	Underflow out of __accum __accum 型的 10 进制数值超出了 __accum 型的范围（下溢）。	fscanf、scanf、sscanf
1440 (ELFIXEDO)	Overflow out of long __fixed long __fixed 型的 10 进制数值超出了 long __fixed 型的范围（上溢）。	fscanf、scanf、sscanf
1450 (ELFIXEDU)	Underflow out of long __fixed long __fixed 型的 10 进制数值超出了 long __fixed 型的范围（下溢）。	fscanf、scanf、sscanf
1460 (ELACCUMO)	Overflow out of long __accum long __accum 型的 10 进制数值超出了 long __accum 型的范围（上溢）。	fscanf、scanf、sscanf
1470 (ELACCUMU)	Underflow out of long __accum long __accum 型的 10 进制数值超出了 long __accum 型的范围（下溢）。	fscanf、scanf、sscanf
2100 (EMALRESM)	Error in waiting semaphore malloc 的信标资源的确保持失败。	calloc、free、malloc、realloc, calloc__X、free__X、malloc__X、realloc__X、 calloc__Y、free__Y、malloc__Y、realloc__Y
2101 (EMALFRSM)	Error in signaling semaphore malloc 的信标资源的确保持失败。	calloc、free、malloc、realloc calloc__X、free__X、malloc__X、realloc__X、 calloc__Y、free__Y、malloc__Y、realloc__Y

错误号	错误信息 / 说明	设定错误号的函数
2110 (ETOKRESM)	Error in waiting semaphore strtok 的信标资源的确保失败。	strtok
2111 (ETOKFRSM)	Error in signaling semaphore strtok 的信标资源的确保失败。	strtok
2120 (EIOBRESM)	Error in waiting semaphore _job 的信标资源的确保失败。	fopen
2121 (EIOBFRSM)	Error in signaling semaphore _job 的信标资源的确保失败。	fopen

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、文件夹名等。

30 (E) INVALID COMMAND PARAMETER

选项有误。
请检查选项。

40 (E) CANNOT ALLOCATE MEMORY

在处理中存储器不足。
在用户能够使用的存储量极少时，发生此错误。
如果有其他执行中的处理，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

包含文件夹的文件名太长或者文件名有误。
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。

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。

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

在堆栈段内记述了执行指令、DSP 指令、扩展指令、确保数据的汇编程序控制指令。
请删除执行指令、DSP 指令、扩展指令、确保数据的汇编程序控制指令。

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误。
请检查记述的方法。

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，修改延迟槽指令。

151 (E) ILLEGAL EXTENDED INSTRUCTION POSITION

给延迟槽指令记述了扩展指令。
请给延迟槽指令记述执行指令。

152 (E) ILLEGAL BOUNDARY ALIGNMENT VALUE

记述扩展指令的段调整数不正确。
请将记述扩展指令的段调整数至少指定为 2。

160 (E) REPEAT LOOP NESTING

在 REPEAT 扩展指令～结束地址之间存在其他 REPEAT 扩展指令。
请修正 REPEAT 扩展指令的位置。

161 (E) ILLEGAL START ADDRESS FOR REPEAT LOOP

在 REPEAT 扩展指令～结束地址之间不存在执行指令或者 DSP 指令。
请在 REPEAT 扩展指令和起始地址之间至少记述 1 条执行指令或者 DSP 指令。

162 (E) ILLEGAL DATA BEFORE REPEAT LOOP

在 REPEAT 扩展指令的循环之前存在不正确的数据。
在循环前为编程控制指令时，请修正汇编程序控制指令；在循环前为文字池时，请通过 .NOPOL 抑止文字池的输出。
在重复传送指令不超过 3 条指令时，在循环前必须为执行指令或者 DSP 指令。

163 (E) ILLEGAL INSTRUCTION IN REPEAT LOOP

在重复循环内配置了不正确的指令。

在 REPEAT 扩展指令～结束地址之间不能配置转移指令、TRAPA 指令（未指定 `cpu=sh4aldsp` 时）或者对于 SR、RS、RE 的装入指令。

164 (E) ILLEGAL INSTRUCTION IN REPEAT LOOP

给重复结束指令配置了不正确的指令。

不能给重复结束指令配置延迟转移指令或者对于 SR、RS、RE 的装入指令。

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）指定了关键字。

请修正符号（包含段名）。

202 (E) ILLEGAL SYMBOL OR SECTION NAME

符号（包含段名）有误。

请修正符号（包含段名）。

203 (E) ILLEGAL LOCAL LABEL

局部标号的指定有误。

请修正局部标号的指定。

300 (E) ILLEGAL MNEMONIC

操作码有误。

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，或者注解有误。

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。

307 (E) ILLEGAL ADDRESSING MODE

指定了操作数不允许的寻址方式。

请修正操作数。

308 (E) SYNTAX ERROR IN OPERAND

操作数有语法错误。

请修正操作数。

309 (E) FLOATING POINT REGISTER MISMATCH

对单精度的操作指定了双精度的浮点寄存器，或者对双精度的操作指定了单精度浮点寄存器。
请修正操作长度或者浮点寄存器。

350 (E) SYNTAX ERROR IN SOURCE STATEMENT (助记符)

DSP 指令的源语句有语法错误。
请检查源语句。

351 (E) ILLEGAL COMBINATION OF MNEMONICS (助记符 , 助记符)

指定了没被允许的 DSP 运算指令的组合。
请修正指令的组合。

352 (E) ILLEGAL CONDITION (助记符)

对 DSP 运算指令指定了不正确的条件。
请删除条件的指定或者更改 DSP 运算指令。

353 (E) ILLEGAL POSITION OF INSTRUCTION (助记符)

DSP 运算指令的指定位置有误。
请将 DSP 运算指令指定到正确的位置。

354 (E) ILLEGAL ADDRESSING MODE (助记符)

DSP 运算指令的地址格式有误。
请修正操作数。

355 (E) ILLEGAL REGISTER NAME (助记符)

DSP 运算指令的寄存器的指定有误。
请修正寄存器名。

357 (E) ILLEGAL COMBINATION OF MNEMONICS (助记符)

数据传送指令的指定不正确。
请修正数据传送指令。

371 (E) ILLEGAL COMBINATION OF MNEMONICS (助记符 , 助记符)

数据传送指令的组合有误。
请修正数据传送指令的组合。

372 (E) ILLEGAL ADDRESSING MODE (助记符)

指定了数据传送指令的操作数不允许的地址格式。
请修正操作数。

373 (E) ILLEGAL REGISTER NAME (助记符)

数据传送指令的寄存器的指定有误。
请修正寄存器名。

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

操作数的值超出了范围。
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

给立即值指定了不正确的值（前方参照符号、外部参照符号和相对地址符号）。
请修正立即值。

407 (E) MEMORY OVERFLOW

在表达式的计算中，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，指定了错误的段名。
请修正段名。

412 (E) ILLEGAL SYMBOL IN EXPRESSION

给移位数指定了相对地址。
请修正运算内容。

450 (E) ILLEGAL DISPLACEMENT VALUE

位移量的值不正确（指定了负值）。
请修正位移量的值。

452 (E) ILLEGAL DATA AREA ADDRESS

PC 相对数据传送指令的数据区的地址不正确。
请对符合各指令的操作长度的边界地址进行存取。
（MOV.L 和 MOVA 指令为 4 个字节边界、MOV.W 为 2 个字节边界）。

453 (E) LITERAL POOL OVERFLOW

文字未输出的扩展指令超过了 510 个。
请使用 .POOL 等输出文字池。

460 (E) ILLEGAL SYMBOL

给 REPEAT 扩展指令的操作数指定了无前方参照的标号、未定义的符号、非标号的符号，或者起始地址的值为大于结束地址的值。
请修正操作数。

461 (E) SYNTAX ERROR IN OPERAND

指定了不正确的操作数。
请修正操作数。

462 (E) ILLEGAL VALUE IN OPERAND

REPEAT 扩展指令和标号的距离超出了范围。
请修正 REPEAT 扩展指令或者标号的位置。

463 (E) NO INSTRUCTION IN REPEAT LOOP

在重循环内或者在指定为结束地址的位置不存在指令。
请在起始地址～结束地址之间记述指令或者将指令存在的地址指定为结束地址。

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或者定位计数器值的指定有误。
请修正起始地址或者定位计数器的值。

502 (E) ILLEGAL SYMBOL IN OPERAND

给操作数指定了不正确的值（前方参照符号、外部参照符号、相对地址符号、未定义的符号）。
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

给操作数指定了不正确的值（前方参照符号、外部参照符号）。
请修正操作数。

505 (E) ILLEGAL OPERAND

操作数的名称有误。
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

调整数的指定有误。
请修正调整数。

512 (E) ILLEGAL EXECUTION START ADDRESS

执行的起始地址有误。
请修正执行的起始地址。

513 (E) ILLEGAL REGISTER NAME

寄存器名有误。
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

给操作数指定了不正确的值（前方参照符号、外部参照符号、其他段的相对地址符号）。
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。

519 (E) ILLEGAL SYMBOL IN OPERAND

在单片机种类为 SH2A、SH2A-FPU 时，给指定常数值的操作数指定了具有地址值的符号或者定位计数器的值。
在单片机种类为 SH2A、SH2A-FPU 时，给操作数指定了具有地址值的符号或者定位计数器的值。

520 (E) ILLEGAL .CPU DIRECTIVE POSITION

.CPU 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。

521 (E) ILLEGAL .NOP00L DIRECTIVE POSITION

.NOP00L 的记述位置不正确。

请紧接在延迟槽指定之后记述 .NOP00。

522 (E) ILLEGAL .P00L DIRECTIVE POSITION

紧接在延迟转移指令之后记述了 .P00L。

请在延迟转移指令之后记述延迟槽指令。

523 (E) ILLEGAL OPERAND

.LINE 的操作数有误。

请修正 .LINE 的操作数。

525 (E) ILLEGAL .LINE DIRECTIVE POSITION

在宏展开或者带条件重复展开中，指定了 .LINE。

请改变 .LINE 的指定位置。

526 (E) STRING TOO LONG

操作数的字符串超过了 255 个字符。

.SDATA、.SDATAB、.SDATAC、.SDATAZ 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 5

给段属性指定了 COMMON。

公共段无法使用。

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 和 .ORG 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 和 .ORG 的指定内容。

530 (E) ILLEGAL OPERAND

.FILE 的操作数有误。

请修正 .FILE 的操作数。

531 (E) ILLEGAL .FILE DIRECTIVE POSITION

在宏展开或者带条件的重复展开中指定了 .FILE。

请改变 .FILE 的指定位置。

532 (E) ILLEGAL OPERAND

.STACK 的操作数有误。

请将堆栈值修正为 4 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

源程序中有不正确的字符。
请修正不正确的字符。

601 (E) INVALID DELIMITER

分隔符不正确。
请修正分隔符。

602 (E) INVALID CHARACTER STRING FORMAT

字符串有误。
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

控制指令的操作数不正确。
不能给本控制指令的操作数记述符号和定位计数器 (\$)。

610 (E) MULTIPLE MACRO NAMES

要用 .MACRO 定义的宏名已被定义。
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。

613 (E) ILLEGAL .MACRO DIRECTVE POSITION

在宏体 (.MACRO ~ .ENDM 之间) 中、.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

在宏体（.MACRO ~ .ENDM 之间）中有 .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）中、.AREPEAT ~ .AENDR 之间、.AWHILE ~ .AENDW。
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误，或者宏体（.MACRO ~ .ENDM 之间）的形式参数名有误。
请修正形式参数名。
如果宏体的形式参数名有误，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

在宏展开中有 .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。

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。

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误。
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 或者 .AWHILE 配对的 .AENDR 或者 .AENDW。
请检查控制语句。

644 (E) ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION

在 .AIF ~ .AENDI 之间有 .AENDR 或者 .AENDW。
请删除 .AENDR 或者 .AENDW。

645 (E) EXPANSION TOO LONG

在 .AREPEAT 或者 .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 的文件名有误。
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。

662 (E) ILLEGAL .END DIRECTIVE POSITION

在 .AIF ~ .AENDI 之间有 .END。
请删除 .END。

663 (E) ILLEGAL .END DIRECTIVE POSITION

在文件中有 .END。
请删除 .END。

664 (E) ILLEGAL .END DIRECTIVE POSITION

在 .AIF ~ .AENDI 之间有 .END。
请删除 .END。

665 (E) EXPANSION TOO LONG

在 .DEFINE 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

667 (E) SUCCESSFUL CONDITION .AERROR

根据 .AIF 的条件，处理了包含 .AERROR 的语句。
为了不处理 .AERROR，请修正条件。

668 (E) ILLEGAL VALUE IN OPERAND

.AIFDEF 的操作数有误。
请通过 .DEFINE 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、.DEFINE、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) ILLEGAL SYMBOL IN OPERAND

在单片机种类为 SH2A、SH2AFPU 时，给预处理器控制指令指定了预处理器变量以外的符号。
请修正符号。

700 (W) ILLEGAL VALUE IN OPERAND (助记符)

DSP 运算指令的操作数超出了值的范围。
请修正值。

701 (W) MULTIPLE REGISTER IN DESTINATION (助记符 , 助记符)

给 DSP 运算指令的目标操作数指定了多个相同寄存器。
请修正寄存器。

702 (W) ILLEGAL OPERATION SIZE (助记符)

DSP 运算指令或者数据传送指令的操作长度有误。
请修正或者删除操作长度。

703 (W) MULTIPLE REGISTER IN DESTINATION (助记符 , 助记符)

对 DSP 运算指令和数据传送指令指定了相同的目标寄存器。
请修正寄存器的指定。

704 (W) A PRIVILEGED INSTRUCTION " 助记符 " IS USED

在 chkmd 选项的指定中，存在特权模式指令。

705 (W) A LDTLB INSTRUCTION IS USED

在 chktlb 选项的指定中，存在 LDTLB 指令。

706 (W) A CACHE INSTRUCTION " 助记符 " IS USED

在 `chkcach` 选项的指定中, 存在高速缓存关联的指令。

707 (W) A DSP INSTRUCTION " 助记符 " IS USED

在 `chkdsp` 选项的指定中, 存在 DSP 关联的指令。

708 (W) A FPU INSTRUCTION " 助记符 " IS USED

在 `chkfpu` 选项的指定中, 存在 FPU 关联的指令。

800 (W) SYMBOL NAME TOO LONG

预处理器变量超过了 32 个字符。

请修正预处理器变量。

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。

807 (W) ILLEGAL OPERATION SIZE

操作长度有误。

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误。

请修正记述。

汇编程序有可能错误解释整数常数 (程序员不希望的值)。

810 (W) TOO MANY OPERANDS

执行指令的操作数太多或者注释有误。

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

在段的重新开始时指定了不同种类的段, 或者在绝对地址段的重新开始时再次指定了段的起始地址。

重新开始段时, 不能指定不同种类的段和起始地址。

已经开始的段的指定照样有效。

814 (W) ILLEGAL OBJECT CODE SIZE

确保长度有误。
能指定的确保长度只限 :12。

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) ILLEGAL DATA AREA ADDRESS

数据或者数据区的配置不正确。
为了使起始地址为偶数，请确保字单位的数据和数据区。
为了使起始地址为 4 的倍数，请确保长字或者单精度的数据和数据区。
为了使起始地址为 8 的倍数，请确认双精度单位的数据和数据区。
按照数据或者数据区域的指定配置汇编程序。

817 (W) ILLEGAL BOUNDARY VALUE

代码段或者堆栈段的调整数小于 4。
指定有效。但是，当将执行指令、DSP 指令和扩展指令记述为奇数地址时，输出警告 882。
在给代码段或者堆栈段的调整数指定 1 时，需要特别注意。

818 (W) COMMANDLINE OPTION MISMATCH FOR FLOATING DIRECTIVE

在单片机种类为 SH2E 时，指定了 round=nearest 或者 denormalize=on；在单片机种类为 SH2AFPU 时，指定了 denormalize=on。
请更改 Round 选项或者 denormalize 选项的指定。
按照 round 选项或者 denormalize 选项的指定，汇编程序生成目标码。

825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION

在虚设段中记述了执行指令、DSP 指令、扩展指令或者确保数据的汇编程序控制指令。
请删除执行指令、DSP 指令、扩展指令或者确保数据的汇编程序控制指令。
汇编程序忽视执行指令、扩展指令、DSP 指令或者确保数据的汇编程序控制指令的记述。

826 (W) ILLEGAL PRECISION

浮点常数的精度和操作长度指定的精度不同。
请修正操作长度指定的精度或者浮点常数的精度。
汇编程序以操作长度指定的精度为有效进行处理。

832 (W) MULTIPLE 'P' DEFINITIONS

作为默认段名的 P 与其他符号 P 重复。
请不要重复 P。
汇编程序将 P 认为默认段名，其他符号 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) ILLEGAL CONSTANT SIZE

一部分整数常数的记述有误。
请修正记述。
汇编程序有可能错误解释整数常数（程序员不希望的值）。

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、LATIN1 码或者没有指定 sjis 选项、euc 选项、latin1 选项。
请在注释、字符串中指定 shift JIS 码、EUC 码、LATIN1 码或者指定 sjis 选项、euc 选项、latin1 选项。

839 (W) ILLEGAL FIGURE IN OPERAND

在定点数据为字的情况下，指定了 6 位以上的数据；在长字的情况下，指定了 11 位以上的数据。
请删除多余的位。

840 (W) OPERAND OVERFLOW

浮点数据发生了上溢。
请更改值。
在为正值的情况下，为 $+\infty$ ；在为负值的情况下，为 $-\infty$ 。

841 (W) OPERAND UNDERFLOW

浮点数据发生了下溢。
请更改值。
在为正值的情况下，为 $+0$ ；在为负值的情况下，为 -0 。

842 (W) OPERAND DENORMALIZED

给浮点数据指定了非正规化数。
请确认浮点数据。
按照指定，汇编程序生成目标码（设定非正规化数）。

843 (W) INEFFECTIVE FLOATING POINT OPERATION

在常数表达式中进行了 $\infty-\infty$ 、 $0.0/0.0$ 等的无效运算。
假定与表示无效运算结果的非数对应的内部表示值。

844 (W) DIVISION BY FLOATING POINT ZERO

在常数表达式中进行了被零除的运算。
按照符号，假定与 $+\infty$ 或者 $-\infty$ 对应的内部表示值。

845 (W) ILLEGAL IMMEDIATE VALUE

立即值不正确。低 8 位不为 0。
汇编程序将立即值的低 8 位校正到 0。

850 (W) ILLEGAL SYMBOL DEFINITION

给标号字段指定了符号。
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。

854 (W) .AWHILE ABORTED BY .ALIMIT

展开次数达到了 .ALIMIT 设定的上限值而中断展开。
请检查重复展开的条件。

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

位移量的值不正确。
在操作长度为字时，位移量不为偶数；在操作长度为长字时，位移量不为 4 的倍数。
请考虑汇编程序校正位移量。
根据操作数长度，汇编程序校正位移量后生成目标码。在操作长度为字时，为了使位移量变为偶数而舍去低位部分；在操作长度为长字时，为了使位移量变为 4 的倍数而舍去低位部分。

874 (W) CANNOT CHECK DATA AREA BOUNDARY

对 PC 相对数据传送指令，不能检查数据区的边界。在连接时，请充分注意数据区的边界。
在数据传送指令属于相对地址或者在数据区为外部参照符号等时，汇编程序输出此警告。

875 (W) CANNOT CHECK DISPLACEMENT SIZE

对 PC 相对数据传送指令，不能检查位移量的大小。在连接时，请充分注意数据传送指令和数据区的距离。
在数据传送指令属于相对地址或者在数据区为外部参照符号等时，汇编程序输出此警告。

876 (W) ASSEMBLER OUTPUTS BRA INSTRUCTION

汇编程序自动输出 BRA 指令。
请使用 .POOL 等指定文字池的输出位置或者通过汇编程序自动生成的 BRA 指令确认程序是否正常运行。
因为找不到文字池的输出点，所以表示自动输出文字池和跳过该文字池的 BRA 指令。

880 (W) .END NOT FOUND

在程序中没有 .END。
请记述 .END。

881 (W) ILLEGAL DIRECTIVE IN REPEAT LOOP

在重复循环内指定了不正确的汇编程序控制指令。

请删除不正确的汇编程序控制指令。

如果在重复循环内记述了确保数据或者数据区的控制指令 `.ALIGN`、`.ORG`，就将 1 条控制指令作为 1 条指令进行重复指令数的计数。

882 (W) ILLEGAL ADDRESS

将执行指令和扩展指令记述为奇数地址。

请将执行指令和扩展指令记述为偶数地址。

883 (W) MULTIPLE FILE NAMES

多次指定了 `.FILE` 的文件。

忽视第 2 次以后的指定。

884 (W) " 单片机种类 1" IS INTERPRETED AS " 单片机种类 2"

`cpu=< 单片机种类 1>` 无效。

解释为 `cpu=< 单片机种类 2>` 进行汇编。

885 (W) CANNOT SUPPORT " 单片机种类 "

不支持 `.CPU` 指定的单片机种类。

解释为 `cpu` 选项指定的单片机种类。

901 (F) SOURCE FILE INPUT ERROR

在输入源文件时发生了错误。

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了错误。

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了错误。

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。
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。
请分割程序。

933 (F) ILLEGAL ENVIRONMENT VARIABLE

单片机种类有误。
请修正单片机种类。

935 (F) SUBCOMMAND FILE INPUT ERROR

在输入子命令文件时发生了错误。
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。

950 (F) MEMORY OVERFLOW

存储器不足。
请分割源程序。

951 (F) LITERAL POOL OVERFLOW

文字池的内部符号个数超过了 100,000 个。
请分割源程序。

952 (F) LITERAL POOL OVERFLOW

文字池的容量已满。
在文字池发生溢出前，请插入无条件转移。

953 (F) MEMORY OVERFLOW

存储器不足。
请分割源程序。

954 (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了错误。
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。

958 (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）。

可能用 **.ORG** 给定位计数器设定了较大的偏移量或者用 **.DATAB** 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化，所以更改了 offset 值。请确认是否有问题。在想抑止 offset 值的变更时，请在汇编 " 文件 " 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。在要作为模块间的优化对象时, 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" 中的未定义符号的 "符号" 指定了堆栈容量的 "容量"。
- L0103 (I) Multiple stack sizes specified to the symbol "符号"**
给符号的 "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 选项之后没有处理的指定。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 " 单片机种类 "
" 选项 " 只在 " 单片机种类 " 时才有效。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 次指定了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, 作为入口符号, 指定了 entry 选项指定的 " 符号 " 以外的符号。优先指定的选项。
- L1120 (W) Section address is not assigned to " 段 "
没有指定 " 段 " 的地址。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" 地址与调整数发生了矛盾。根据调整数校正段地址。
- 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、调整数不同。作为 "段 2" 的调整数, 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。对于超出指定的 record 格式范围, 用其他的 record 格式输出。
- L1141 (W) Cannot fill unused area from "地址" with the specified value
因为空区的容量不为 space 选项指定的值的倍数, 所以指定的数据不能输出到 "地址" 后面。
- 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
在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
在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
在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
在 "文件" 中有不支持的段。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**
多次输入了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 "**
输入了不同调整数的同名段。指定的最大调整数有效。
- 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 " 单片机种类 1 " in " 文件 " differ from " 单片机种类 2 "**
输入了不同单片机种类的文件。将 **H8SX** 作为单片机种类继续处理。
- 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极限值。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。
- L1402 (W) Parentheses specified in option "start" with optimization**
用 **start** 选项记叙括号 **"()"** 时，不能使用优化功能。
优化功能无效。
- 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，所以不能对编译时 **stack** 选项指定的堆栈容量进行匹配性检查。
为了对编译时 **stack** 选项进行匹配性检查，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、英文字母、\$（不以数字开头）。
- L2016 (E) Invalid alignment value specified in option " 选项 " : " 调整数 "
" 选项 " 指定的 " 调整数 " 是无效值。
必须指定 1、2、4、8、16 或者 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"
不存在 "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 " 不是单片机信息文件。

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)"。

L2212 (E) " 选项 " cannot be specified for inter-module optimization
information in " 文件 "

因为在 " 文件 " 内有模块间的优化信息, 所以不能用 " 选项 " 的选项。在编译和汇编时, 不能使用 `goptimize` 选项。

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。
- L2323 (E) Section "段 1(地址范围)" overlaps with section "段 2(地址范围)" in physical space**
在物理存储器的分配上, "段 1(地址范围)" 和 "段 2(地址范围)" 重叠。
必须重新分配各段的地址。
< 地址范围 >: < 段的起始地址 >-< 段的结束地址 >
- 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。
请确认下述 (1) ~ (3):
(1) 不能在指定 **map** 选项 (编译时) 的前后更改程序。
(2) 根据 **optlnk** 的优化, 指定 **map** 选项 (编译时) 前后的符号排列顺序可能发生了变化。请将编译时的 **map** 选项或者 **optlnk** 优化选项置为无效。

(3) 在使用 `tbr` 选项或者 `#pragma tbr` 时, 根据编译程序的优化, 在指定 `map` 选项 (编译时) 后的符号可能被删除。请将编译时的 `map` 选项、`tbr` 选项或者 `#pragma tbr` 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" 和其他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, 要求生成示例。

对于正在使用 "示例" 的文件, 请确认在建立可再定位文件时是否使用了 `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文件时是否使用了 `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, 请参照各单片机的硬件手册。
- 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 " 单片机种类 " in " 文件 "**
输入了不同单片机种类的文件。
- 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**
段的个数超过了编译限制。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**
符号个数超过了编译限制。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**
模块的个数超过了编译限制。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“(ctrl)+C”键中断。

L3420 (F) Error occurred in " 装入模块 "

在执行 " 装入模块 " 中发生了错误。

P3500 (F) Bad instantiation request file -- instantiation assigned to more than one file

示例的生成指定文件有误。

请重新编译连接对象的文件。

P3501 (F) Instantiation loop

示例的生成处理进入了循环处理。

输入的文件名有可能与其他文件的示例生成要求文件名一致。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 "

不能建立示例的生成指定文件。

请确认目标建立文件夹的存取权是否正确。

P3503 (F) Cannot change to directory " 文件夹 "

不能移动到 " 文件夹 "。

请确认是否存在 " 文件夹 "。

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。

请与销售部门联系，告知信息中的内部错误号、文件、行号、注释的内容。

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` 指定的有优化和无优化的函数。在删除调试信息后进行转换。

G1002 (W) Command parameter specified twice

2 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。请确认是否正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。请确认是否正在存取临时文件。

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。

G3201 (F) Cannot execute compiler

不能启动编译程序。请确认编译程序的路径名。

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。

G3300 (F) Already existent file " 文件 "

文件已经存在。

第 16 章 编译限制

16.1 编译程序的编译限制

编译程序的编译限制如表 16.1 所示。

请在此编译限制的范围内建立源程序。

表 16.1 编译程序的编译限制

分类	项目	编译限制
1 启动	define 选项能指定的宏名总数	无限制（取决于存储器容量）
2	文件名的字符数	无限制（取决于 OS）
3 源程序	1 行的字符数	32768 个字符
4	1 个文件的源程序行数	无限制（取决于存储器容量）
5	能编译的源程序的总行数	无限制（取决于存储器容量）
6 预处理器	#include 语句的嵌套深度	无限制（取决于存储器容量）
7	#define 语句的宏名总数	无限制（取决于存储器容量）
8	宏定义、宏调用的参数的个数	无限制（取决于存储器容量）
9	宏名的再置换数	无限制（取决于存储器容量）
10	条件编译的嵌套层数	无限制（取决于存储器容量）
11	#if、#elif 语句中能指定的运算符、非运算符的合计数	无限制（取决于存储器容量）
12 声明	函数定义的个数	无限制（取决于存储器容量）
13	为外部连接的标识符（外部名）的个数	无限制（取决于存储器容量）
14	在 1 个函数中有效标识符（内部名）的个数	无限制（取决于存储器容量）
15	限定基本型的指针、数组以及函数声明的个数	16 个
16	数组的维数	6 维
17	数组 / 结构体的大小	2147483647 个字节
18 文	复合语句的嵌套深度	无限制（取决于存储器容量）
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	4096 层
20	1 个函数中能指定的 goto 标号的个数	2147483646 个
21	switch 语句的个数	2048 个
22	switch 语句的嵌套深度	2048 层
23	1 个 switch 语句中能指定的 case 标号的个数	2147483646 个
24	for 语句的嵌套深度	2048 层
25 表达式	字符串的字符数	32766 个字符
26	函数定义、函数调用的参数的个数	2147483646 个
27	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	open 函数能同时打开的文件个数	可变 *1
29 段	段名的长度 *2	8192 个字符
30	1 个文件中 #pragma section 能指定的段数	2045 个

【注】 *1 详细内容请参照“9.2.2(4), C/C++ 库函数的初始设定（_INITLIB）”。

*2 是编译程序在生成目标时建立的长度，#pragma section 或者 section 选项能指定的长度小于此长度。

16.2 汇编程序的汇编限制

汇编程序的汇编限制如表 16.2 所示。

表 16.2 汇编程序的汇编限制

项目	汇编限制
1 1 行的字符数	8192 个字符
2 字符常数	4 个字符
3 符号长度	无限制（取决于存储器容量）*1
4 符号个数	无限制（取决于存储器容量）
5 外部参照符号的个数	无限制（取决于存储器容量）
6 外部定义符号的个数	无限制（取决于存储器容量）
7 段的最大容量	H'FFFFFFF 个字节
8 段的个数	H'FEF1 个（有调试）， H'FEFA 个（无调试）
9 文件 include	嵌套 30 层
10 字符串的字符数	255 个字符
11 文件名的字符数	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。

.DEFINE 能指定的置换符号的字符个数没有限制。但是，置换字符串的字符数最多 255 个字符，并且 1 行中能记述的字符个数最多 8192 个字符。

第 17 章 版本升级的注意事项

17.1 版本升级时的注意事项

说明从旧版本（SuperH RISC engine C/C++ 编译程序包 Ver.8.x 及以前的版本）升级后的使用注意事项。

17.1.1 程序的运行保证

在编译程序版本升级后进行程序开发时，程序的运行有可能发生变化。在创建程序时，必须注意以下几点，并且请充分测试用户程序。

(1) 取决于执行时间和时序的程序

语言规格没规定程序的执行时间。因此，根据编译程序版本的不同，程序的执行时间和 I/O 等外围设备的时序有可能出现偏差、或者如中断处理等异步处理的时间出现差异等，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，有可能因编译程序的版本而发生运行变化。

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 的值是 3 时，为 f(3, 4) 或者 f(4, 3)。          */
```

(3) 取决于结果上溢或者不适当运算的程序

如果产生上溢或者进行了不适当运算，就不能保证结果值。因此，有可能因版本变化而发生运行的变化。

例：

```
int a, b;
x = (a * b) / 10;          /* 有可能因 a 和 b 的值的范围而发生上溢          */
```

(4) 变量未被初始化、型的不一致

在变量未被初始化时，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，有可能因编译程序的版本而发生运行变化。

例：

file 1: <pre>int f(double d) { : }</pre>	file 2: <pre>int g(void) { f(1); }</pre>	函数调用侧的参数是int型，而函数定义侧的参数是double，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。

17.1.2 与旧版本的兼容性

以下说明在将旧版本的编译程序（Ver.5.x 以下）、汇编程序（Ver.4.x 以下）以及连接编辑程序（Ver.6.x 以下）输出的目标文件和库文件进行连接时、或者在使用旧版本使用的调试程序时的注意点。

(1) 目标文件格式

目标文件格式从原来的 SYSROF 更改为标准格式的 ELF 格式，调试信息格式也更改为标准格式的 DWARF2 格式。

在将旧版本的编译程序（Ver.5.x 以下）和汇编程序（Ver.4.x 以下）的输出目标文件（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，不能将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进行转换。

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 格式的装入模块转换为 SYSROF 或者 ELF/DWARF1 格式。

(2) include 文件的检索开始文件夹

在以文件夹相对格式检索被指定的 include 文件时，旧版本从编辑程序启动文件夹开始检索，但是新版本更改为从存放源文件的文件夹开始检索。

(3) C++ 程序

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，还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，所以请参照“9.2.2 执行环境的设定”进行修正。

(4) 公共段的废除（汇编程序）

随着目标格式的变更，废除了公共段的支持。

(5) .END 的入口指定（汇编程序）

能用 .END 进行入口指定的符号只有外部定义符号。

(6) 模块间的优化

旧版本的编译程序（Ver.5.x 以下）和汇编程序（Ver.4.x 以下）输出的目标文件不能作为模块间的优化对象。对于需要模块间优化的对象文件，必须重新进行编译和汇编。

(7) 有关优化连接编辑程序支持的目标文件

因版本的不同，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: Ver.7 以下的编译程序和 Ver.5 以下的汇编程序
- 优化连接编辑程序 Ver.8: Ver.8 以下的编译程序和 Ver.6 以下的汇编程序
- 优化连接编辑程序 Ver.9: Ver.9 以下的编译程序和 Ver.7 以下的汇编程序

17.1.3 与旧版本的目标文件的兼容性

为了将 Ver.6 目标文件和 Ver.7 以上的目标文件进行连接，必须指定以下选项：

- gbr = user
- pack = 4 （Ver.8 以上）
- bit_order = left （Ver.8 以上）

在编译或者建立库时，必须选项指定以下相同的选项。如果在建立 Ver.6 的目标文件时指定了以下选项，也必须对 Ver.7 以上的版本指定相同的选项。

- endian = big | little （SH-2、SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP）
- pic = 0 | 1 （SH-1 以外）
- fpu = single | double （SH2A-FPU、SH-4、SH-4A）
- fpscr = safe | aggressive （SH2A-FPU、SH-4、SH-4A）
- round = zero | nearest （SH2A-FPU、SH-4、SH-4A）
- denormalize = on | off （SH-4、SH-4A）
- double = float （SH-4、SH-4A 以外）
- exception | noexception
- rtti = on | off
- rtnext | nortnext
- macsave

如果以“9.3.2 函数调用的规则”的调用规则为基准，就能连接汇编程序。

【注】 1 对于手册中没有记载的内容，不保证版本升级的兼容性。

当记述了取决于寄存器保存 / 恢复顺序等编译程序输出代码的汇编代码时，不能将 Ver.6 目标文件和 Ver.7 以上的目标文件进行连接。

2 有关 OS 和中间件等的连接，请询问营业部门。

17.1.4 命令行接口

(1) 汇编程序、优化连接编辑程序命令行的规则

在文件名和选项之间需要空格，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

废除了会话形式的选项指定的支持，合并了旧版本的模块间优化工具（optlnksh）、连接编辑程序（lnk）、库生成程序（lbr）、目标转换程序（cnvs）。因此，命令行规格有了很大的更改。更改的命令一览表如表 17.1 和表 17.2 所示。

表 17.1 连接命令的变更一览表

No.	命令名	V6	V7	备注
1	start	start= 段 (地址) 缩短为 st	start= 段 / 地址 缩短为 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) 缩短为 re	rename= (更改前 = 更改后) , (更改前 = 更改后) , — 缩短为 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	缩短为 f	缩短为 fo	—
11	output / nooutput	缩短为 o 可指定 nooutput	缩短为 ou 不可指定 nooutput	只能指定 output
12	cpu	缩短为 c	缩短为 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	总是抑止
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 V7 的 list 选项

No.	命令名	V6	V7	备注
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 17.2 库生成程序命令的变更一览表

No.	命令名	V2	V7	备注
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	缩短为 d	缩短为 del	—
6	create	create (s u)	output form=library(s u)	—
7	output	output (s u) 缩短为 o	output form=library(s u) 缩短为 ou	—
8	replace	缩短为 r	缩短为 rep	—
9	abort	abort	废除	废除了会话格式
10	exit	不可省略	可省略	—

17.1.5 提供的内容

在“SuperH RISC engine C/C++ 编译程序包”提供的内容中，更改了以下文件：

(1) 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。

(2) 头文件

追加了作为标准库提供的初始设定程序 _INITSET 函数、_CALL_INIT 函数、_CALL_END 函数的头文件 _h_c_lib.h

追加了定义有关定点数内部表示的各种极限值的头文件 fixed.h。

17.1.6 列表文件规格

(1) 优化连接编辑程序

更新了原来的连接映像列表和库列表的格式。

17.2 追加和改善的内容

17.2.1 共同的追加和改善（程序包 Ver.6）

(1) 极限值的放宽

大幅度放宽了源程序和命令行的限制。

- 文件名的字符数：251 个字节 → 无限制
- 符号长度：251 个字节 → 无限制
- 符号数：32,767 个 → 无限制
- 源程序行数：C/C++:32767 行、ASM:65535 行 → 无限制
- C 程序字符串长度：512 个字符 → 32766 个字符
- 汇编程序行长度：255 个字符 → 8192 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

(2) 文件夹名和文件名的连字符（-）

能给文件夹名和文件名指定连字符（-）。

(3) 版权显示抑止

能通过 logo/nologo 选项指定是否显示版权。

(4) 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。

17.2.2 编译程序的追加和改善

(1) Ver.7 主要追加和改善的功能

(a) 外部变量的存取优化功能（支持 map 选项）

优化以连接时的变量或者函数的分配地址为基准的外部变量存取指令和函数转移指令。使用第一次连接时优化连接编辑程序输出（指定 map 选项）的外部符号分配信息文件进行重新编译，实施优化。

(b) GBR 相对存取代码的自动生成（支持 gbr 选项）

如果指定 gbr=auto，编译程序就进行 GBR 的设定并且自动生成 GBR 相对存取代码。保证函数调用前后的 GBR 值，但是不能使用 GBR 关联的内部函数。

(c) speed/size 选项的强化

追加了 speed/size 选项（shift、blockcopy、division、approxdiv 选项），能更精确地调整 size/speed。

(d) 内部函数等的强化

- 内部函数的追加
双精度乘法、SWAP 指令、LDTLB 指令、NOP 指令命令
- #pragma 的追加和变更

#pragma entry	指定入口函数、设定 SP
#pragma stacksize	指定堆栈大小
#pragma interrupt	支持 sp=< 变量 >+< 常数 >、sp=&< 变量 >+< 常数 >
- 段运算符的支持
支持能用 C 语言记述段地址和大小参照的功能。
- 放宽了地址转换的错误
放宽了初始化外部变量时的地址初始化转换表达式的错误。

(e) 库的改善

- 支持可重入库
在通过库生成工具指定 reent 选项时，生成可重入库。
- malloc 确保长度单位和输入 / 输出文件数的可变性
在初始化 C/C++ 库函数时，能通过 _sbrk_size 指定 malloc 确保长度，并且通过 _nfiles 指定输入 / 输出文件数，从而能节约 RAM 容量。如果省略指定，malloc 确保长度为 1024，而输出 / 输出文件数为 20。
- 简易 I/O 的支持
当通过库生成工具指定 nofloat 选项时，不支持浮点转换，生成小容量的 I/O 程序。

(2) Ver.7.1 主要追加和改善的功能

(a) 优化控制选项的追加

追加了以下选项，能进行比优化级更详细的调整。

- global_volatile
- opt_range
- del_vacant_loop
- max_unroll
- infinite_loop
- global_alloc
- struct_alloc
- const_var_propagate
- const_load
- schedule

(3) Ver.8 主要追加和改善的功能

(a) 新单片机的支持

支持 SH-4A 和 SH4AL-DSP。

(b) 语言规格的扩展和变更

- 支持 DSP-C 语言。
- 支持 long long 型和 unsigned long long 型。

(c) 内部函数等的强化

- 追加了面向 DSP 的内部函数
 - 绝对值、MSB 检测、算术移位、舍入运算、位模式的复制
 - 模寻址设定、模寻址解除、CS 位的设定
- 追加了面向 SH-4A 和 SH4AL-DSP 的内部函数
 - 正弦和余弦的计算、平方根的倒数、指令高速缓存块的无效化、
 - 指令高速缓存块的预取、数据操作的同步
- #pragma 的追加和变更
 - #pragma ifunc 抑止浮点寄存器的保存和恢复
 - #pragma bit_order 指定位域的排列顺序
 - #pragma pack 指定结构体、联合体、类的调整数

(d) 枚举型长度的自动选择（支持 auto_enum 选项）

作为收纳枚举型的最小型，进行枚举型的处理。

(e) 指定结构体、联合体、类成员的调整数（支持 pack 选项）

能指定结构体、联合体、类成员的调整数。

(f) 位域排列顺序的指定（支持 bit_order 选项）

能指定位域成员的排列顺序。

(g) 错误级的变更（支持 change_message 选项）

能分别更改信息级消息和警告级消息的错误级。

(h) 极限值的放宽

将 switch 语句的个数扩展到 2048 个。

(i) 定点型 DSP 库的支持

支持定点型的 DSP 库。

(4) Ver.9 主要追加和改善的功能

(a) 新单片机的支持

支持 SH-2A 和 SH2A-FPU。

另外，还追加了活用 SH-2A 和 SH2A-FPU 的 TBR 的功能（tbr 选项、#pragma tbr）。

(b) 语言规格的扩展和变更

- 使以下项目符合了 ANSI 规格：

- 数组的索引

```
int iarray[10], i=3;
```

```
i[iarray] = 0; /* 和 iarray[i] = 0; 相同 */
```

- 能指定 union 的位域

```
union u {
```

```
    int a:3;
```

```
};
```

- 常数运算

```
static int i=1||2/0; /* 将被零除的运算从 Error 更改为 Warning */
```

- 库、宏的追加

```
strtoul,FOPEN_MAX
```

- 通过指定 strict_ansi 选项，使以下项目符合了 ANSI 规格。因此，有可能产生和 Ver.8 以下版本的不同结果。

- unsigned int 和 long 型运算

- 浮点运算的结合规则

- 通过指定 enable_register 选项，优先将指定 register 存储类的变量分配到寄存器。

(c) 内部函数的强化

- 追加了面向 SH-2A 和 SH2A-FPU 内部函数

饱和运算、TBR 的设定和参照

- 对不能用 C 语言记述的指令，追加了内部函数

T 位的参照和设定、连结寄存器的中央抽出、带进位加法、带借位减法、符号取反、1 位除法、循环、移位

(d) 极限值的放宽

放宽了以下极限值：

- 根据重复语句（while 语句、do 语句、for 语句）和选择语句（if 语句、switch 语句）的组合产生的嵌套深度：32 层 → 4096 层
- 1 个函数中能指定的 goto 标号数：511 个 → 2147483646 个
- switch 语句的嵌套深度：16 层 → 2048 层
- 1 个 switch 语句中能指定的 case 标号数：511 个 → 2147483646 个
- 函数定义或者函数调用的参数的个数：63 个 → 2147483646 个
- 段名的长度：31 个字节 → 8192 个字节
- 1 个文件中 #pragma section 能指定的段数：64 个 → 2045 个

(e) 存储空间分配指定的扩展

能更详细地指定存储空间分配。

- abs16/abs20/abs28/abs32 选项
- #pragma abs16/abs20/abs28/abs32

(f) 变量绝对地址的指定（支持 #pragma address）

能将外部变量的地址指定为绝对地址。

(g) 外部变量存取优化功能的扩展（支持 smap 选项）

对定义在编译对象文件内的外部变量，进行外部变量的存取优化。不需要如 map 选项的重新编译。

(h) 数学函数库精度的提高

提高了数学函数库的运算精度。因此，有可能产生和 Ver.8 以下版本的不同结果。

(5) Ver.9.01 主要追加和改善的功能

(a) 调试信息输出模式的追加（支持 optimize=debug_only 选项）

通过指定 optimize=debug_only 选项，能随时参照局部变量的信息。

(b) 中断规格的追加（SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP）

追加了以下的 #pragma interrupt 的中断规格和内部函数，能记述执行效率高的中断函数。

- 中断规格
 - #pragma interrupt sr_rts 寄存器组的切换，RTS 指令返回的指定
 - #pragma interrupt bank 中断处理函数的指定
 - #pragma interrupt rts RTS 指令返回的指定
- 内部函数
 - sr_jsr() 多重中断控制

(c) 浮点数 - 整数转换时的范围检查省略功能的追加（支持 simple_float_conv 选项：SH-2E、SH2A-FPU、SH-4、SH-4A）

通过指定 simple_float_conv 选项，对于无符号整型和浮点型之间的型转换，能生成省略检查转换对象值范围的代码。

(d) 已有功能的规格追加和变更

- 能对 SH-2A 和 SH2A-FPU 指定 division=cpu=inline | runtime 选项。
- SH-4 也能使用高速缓冲块操作的内部函数 ocbi()、ocbp() 和 ocbwb()。
- 与 inline 选项无关，能对指定 #pragma inline 的函数进行 inline 展开。
- 在指定 subcommand 选项和 listfile 选项时，能将 subcommand 选项指定的文件内容输出到编译列表。因此，在使用瑞萨综合开发环境输出列表文件时，能在列表文件中显示选项。

(e) 数学函数程序库的改善（SH-1、SH-2、SH2-DSP、SH-2A、SH-3、SH3-DSP、SH4AL-DSP）

对于浮点数用的数学函数 sinf、cosf、tanf、expf、logf、sqrtf 和 atanf，减小了目标大小，提高了运算速度和运算精度。因此，这些数学函数的运算结果有可能与 Ver.9.00 不同。

17.2.3 汇编程序的追加和改善的功能

(1) Ver.7 主要追加和改善的功能

(a) 新单片机的支持

支持 SH-2A 和 SH2A-FPU。

(b) 极限值的放宽

将 define 选项和 .DEFINE 控制指令的极限值从 32 个字符扩展为无限制。

(c) .STACK 控制指令的支持

支持 .STACK 控制指令。因此，能将通过汇编程序记述的函数堆栈大小反映到堆栈解析工具。

17.2.4 优化连接编辑程序追加和改善的功能

(1) Ver.7 主要追加和改善的功能

(a) 通配符的支持

能用输入文件或者 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

能分别更改信息级、警告级和错误级消息的错误级、及其输出的有无。

(e) 二进制和 Intel (扩展) HEX 的支持

能输入 / 输出二进制文件。另外，也能选择 Intel(扩展)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) 调试信息的删除功能

能通过 strip 选项，只删除装入模块文件或者库文件内的调试信息。

(h) 调试信息的压缩功能

能通过指定 compress 选项压缩调试信息。

(2) Ver.7.1 主要追加和改善的功能

(a) 外部符号分配信息文件的输出（支持 map 选项）

当指定 map 选项时，生成编译程序在外部变量存取优化时使用的外部符号分配信息文件。

(3) Ver.8 主要追加和改善的功能

(a) 空区域的输出指定

能通过指定 space 选项，将指定值填入到空区域。

(b) 存储器使用量的指定

能通过指定 memory 选项，指定内部存储器的使用量。

(c) 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 中为 Fatal，而在 Ver.8 中改为 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。

(4) Ver.9 主要追加和改善的功能

(a) 新单片机的支持

支持单片机种类为 SH-2A 或者 SH2A-FPU 时的目标文件的输入。

(b) 对 binary 选项输入段的调整数的指定

能对 binary 选项指定的段，指定调整数。

(c) 对照表的信息输出

能通过指定 show=xreference 选项，将对照表信息输出到连接列表内。因此，可得知变量 / 函数的参照位置。

(d) 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。

(5) Ver.9.01 主要追加和改善的内容

(a) 段单位的优化抑止功能

追加新选项（section_forbid）。

通过使用 section_forbid 选项，能以段为单位抑止模块间的优化。

(b) 覆盖功能的强化

导入了给 start 选项使用括号 "()" 的记述方法。

与旧版相比，能记述更复杂的覆盖分配。

(c) 库内的同名符号通知

在连接时，如果使用的库文件内有多个同名符号，就用以下信息发出警告。

** L1320 (W) Duplicate Symbol " 符号 " in " 库 (模块)"

(6) Ver.9.02 主要追加和改善的内容

(a) 物理空间分配时的重叠检测功能

追加了新选项 (ps_check)。

通过使用 ps_check 选项，能检测物理空间内的目标重叠。

(b) 数据记录的字节数指定功能

追加了新选项 (byte_count)。

通过使用 byte_count 选项，能改变 Intel HEX 格式文件的数据记录的最大字节数。

(c) 空区域的随机数填充功能

通过使用 space=random 选项，能将随机数填入空区域。

(d) 建立库时的存储器使用量减小功能

在建立库文件时也能使用存储器使用量减小功能 (memory=low)。

17.3 格式转换程序的操作方法

17.3.1 目标文件的格式

目标文件的格式符合标准格式的 ELF 格式，调试信息的格式符合标准格式的 DWARF2 格式。

17.3.2 与旧版本的兼容性

(1) 目标文件和库文件

在将旧版本的编译程序（Ver.5.x 以下）和汇编程序（Ver.4.x 以下）输出的目标文件和库文件输入到优化连接编辑程序时，必须用格式转换程序将它们转换为 ELF 格式。但是，在转换时调试信息被删除。

另外，不能将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进行转换。

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的编译程序（Ver.5.x 以下）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旧版本的连接编辑程序（Ver.6.0 以下）的输出格式。可转换的目标文件格式如表 17.3 所示。不支持 SH-2A、SH2A-FPU、SH-4A、SH4AL-DSP 的 ELF 格式的装入模块文件。

表 17.3 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 的指定选项	目标文件格式		
				目标	调试信息	可否转换
1	Ver.4.x	以下	debug	SYSROF	SYSROF	○
2	Ver.3.x		sdebug	SYSROF	SYSROF	×
3			sysrof	debug	SYSROF	○
4	Ver.5.x		sdebug	SYSROF	DWARF1	×
5	Ver.4.x		elf	debug	ELF	○
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。

不能将旧版本的连接编辑程序（Ver.6.0 以下）输出的装入模块文件转换为 ELF 格式。

当使用本版本的编译程序、汇编程序或者优化连接编辑程序新追加的功能时，不能进行转换。

当使用 Ver.6 以上版本的编译程序、汇编程序或者优化连接编辑程序新追加的功能时，不能从 ELF 格式转换为旧格式。

17.3.3 选项指定规则

命令行的格式如下：

helcfnv[△ < 选项 >…] △ < 文件名 >[…] [△ < 选项 >…]

< 选项 >: -< 选项 >[= < 子选项 >]

< 文件名 >: 也能指定通配符（*、?）。

17.3.4 选项解说

在命令行格式的情况下，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。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 [项目] 表示对应的对话菜单。

格式转换程序自动判断转换对象文件的种类（目标文件、库文件、装入模块）。

(1) 目标文件和库文件的转换

将旧版本的编译程序（Ver.5.x 以下）和汇编程序（Ver.4.x 以下）建立的可再定位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 17.4 目标文件、库文件转换的选项一览表

项目	选项	对话菜单	指定内容
1 地址空间的指定 *1	Address_space=< 大小 > < 大小 >:20 <u>24</u> 28 32	—	地址空间的指定
2 fpu	Fpu	—	有 FPU
3 dsp	Dsp	—	有 DSP
4 endian 的指定	Endian = { <u>B</u> ig Little }	—	endian 的指定

【注】 *1 用于 H8S、H8/300 系列的选项。对 SuperH 无效。

有 FPU 的指定

Fpu	
格 式	Fpu
说 明	在单片机为 SH-2E 或者 SH-4 时指定。
例	helfcnv -fpu *.obj *.lib ； 将文件夹中的全部 *.obj, *.lib 转换为 elf 格式

有 DSP 的指定

Dsp

格 式	Dsp
说 明	在单片机为 SH2-DSP 或者 SH3-DSP 时指定。
例	helfcnv -dsp *.obj ; 将文件夹中的全部 *.obj 转换为 elf 格式

Endian 指定

Endian

格 式	Endian = { <u>Big</u> Little }
说 明	指定目标和库的 Endian。 当对旧版本的编译程序（Ver.4.x 以上）和汇编程序（Ver.3.x 以前）创建的小端法目标文件进行转换时，必须指定 endian=little。
例	helfcnv -endian=little *.obj ; 将文件夹中的全部 *.obj 转换为小端法

(2) 装入模块文件的转换

将 ELF 格式装入模块转换为旧版本的连接编辑程序（Ver.6.0 以下）输出的目标文件格式。当装入模块文件中包含调试信息时，也转换调试信息。

表 17.5 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	<u>Sysrof</u>	Output	转换为 SYSROF 格式
	Dwarf1	[Type of output file:]	转换为 ELF/DWARF1 格式

转换格式的指定

Sysrof
Dwarf1

Output[Type of output file:]

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

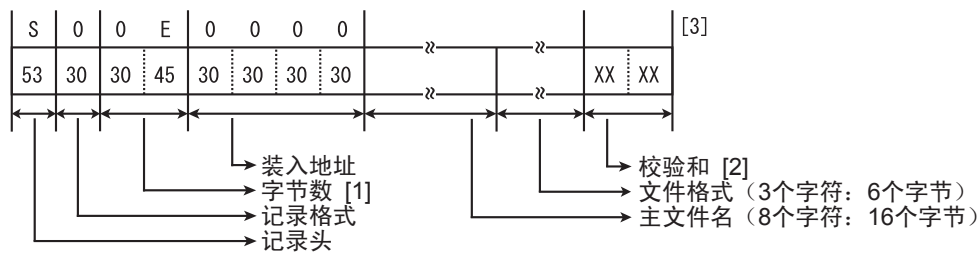
第 18 章 附录

18.1 Motorola S 格式、Intel HEX 格式文件

本节说明根据连接编辑程序输出的 Motorola S 格式文件和 Intel HEX 格式文件。

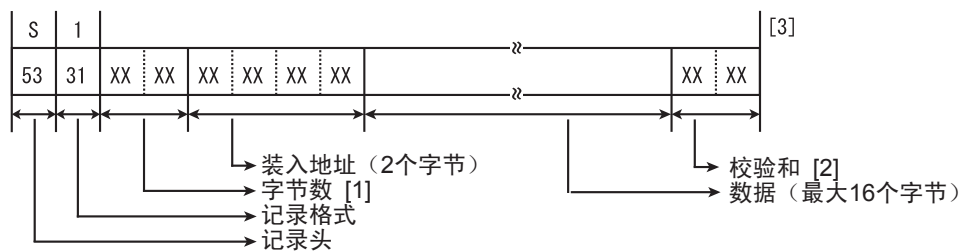
18.1.1 Motorola S 格式文件

(a) 头记录 (S0 记录)

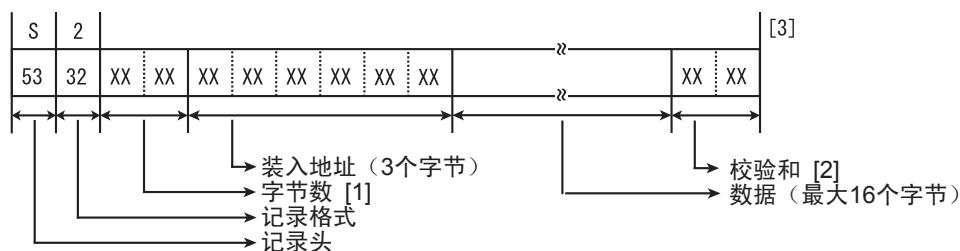


(b) 数据记录 (S1、S2、S3 记录)

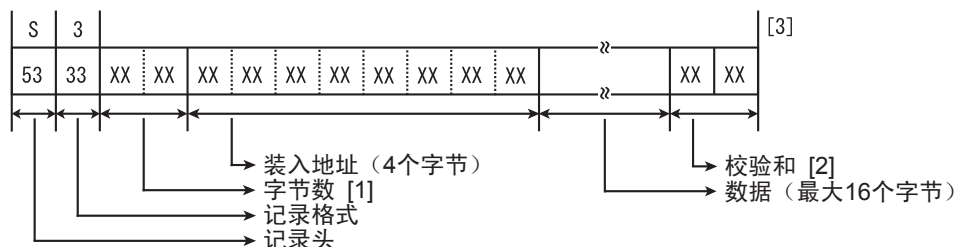
(i) 装入地址为 0 ~ FFFF 时



(ii) 装入地址为 10000 ~ FFFFFF 时

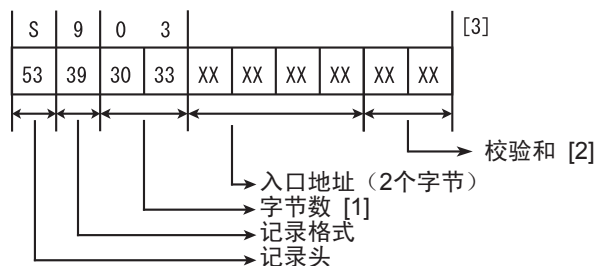


(iii) 装入地址为 1000000 ~ FFFFFFFF 时

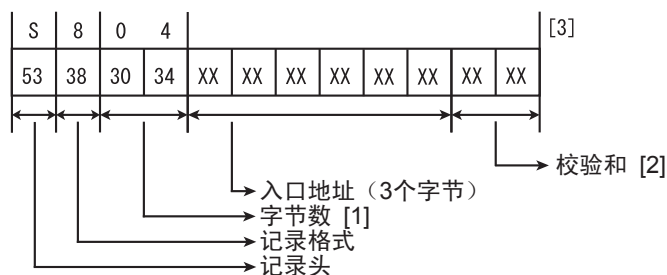


(c) 结束记录 (S9、S8、S7 记录)

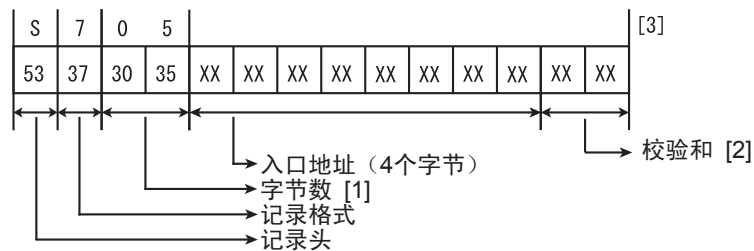
(i) 入口地址为 0 ~ FFFF 时



(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) 到校验和的字节数
 - [2] 从字节数到校验和前的数据值的总和 (字节单位) 的1的补数
 - [3] 在校验和的后面加换行码

18.1.2 Intel HEX 格式文件

各数据记录的执行地址按以下方法计算：

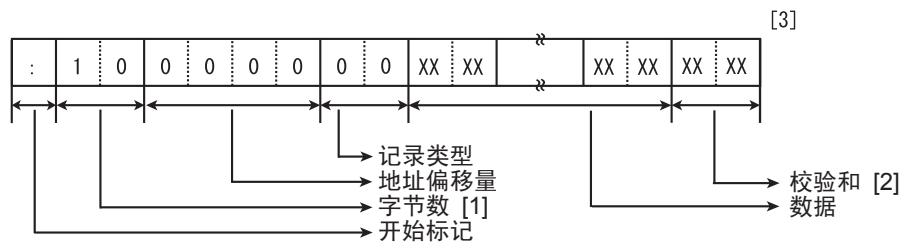
(1) 段地址

(段基地址 $\ll 4$) + (数据记录的地址偏移量)

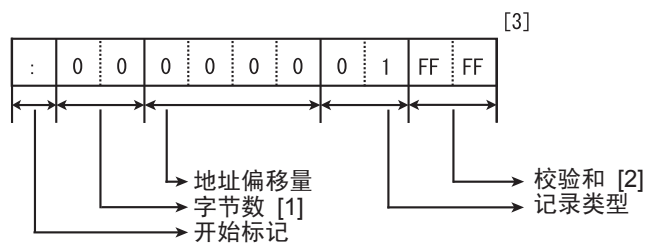
(2) 线性地址

(线性基地址 $\ll 16$) + (数据记录的地址偏移量)

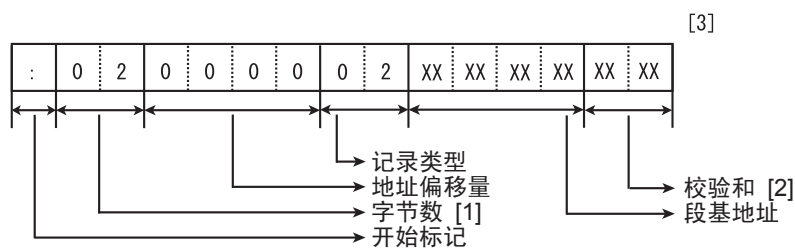
(a) 数据记录 (00 记录)



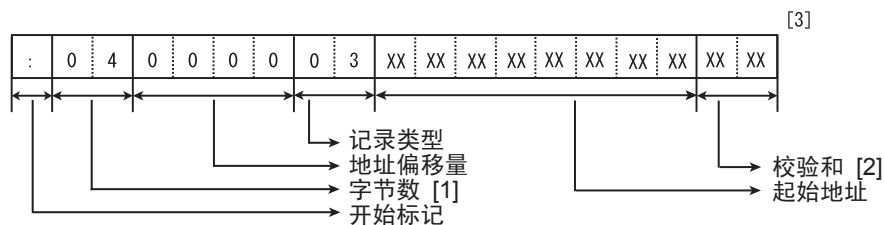
(b) 结束记录 (01 记录)



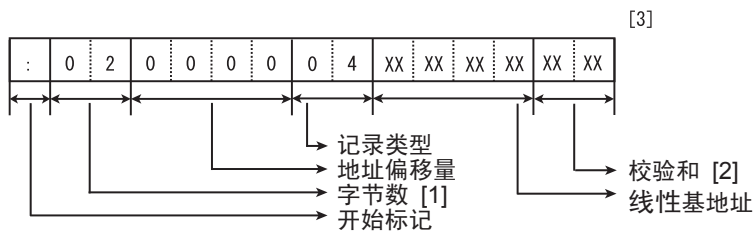
(c) 扩展段记录 (02 记录)



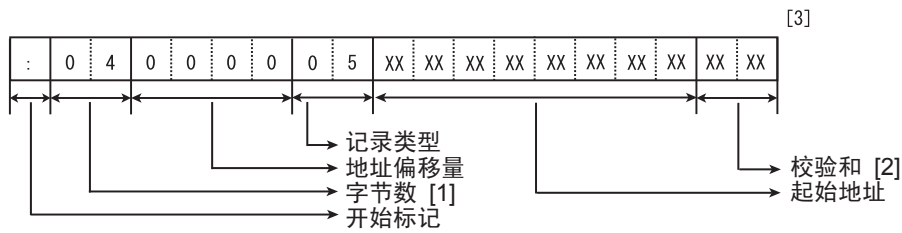
(d) 起始地址记录 (03 记录)



(e) 扩展线性地址纪录（04 记录）



(f) 32 位起始线性地址记录（05 记录）



- 【注】 [1] 从记录类型的下一个数据到校验和的前一个数据的字节数
[2] 从字节数到校验和前的数据值的总和（16进制数）的2的补数
[3] 在校验和的后面加换行码

18.2 ASCII 码一览表

表 18.1 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

索引

Symbols

#pragma.....	239	.CPU.....	616
#pragma abs16.....	242	.DATAB.....	627
#pragma abs20.....	242	.DATA.....	626
#pragma abs28.....	242	.DEBUG.....	645
#pragma abs32.....	242	.DEFINE.....	670
#pragma address.....	262	.END.....	659
#pragma bit_order.....	259	.ENDIAN.....	646
#pragma entry.....	253	.ENDM.....	681
#pragma gbr_base.....	258	.EQU.....	622
#pragma gbr_base1.....	258	.EXITM.....	676
#pragma global_register.....	257	.EXPORT.....	640
#pragma ifunc.....	254	.FDATA.....	632
#pragma inline_asm.....	250	.FDATAB.....	633
#pragma inline.....	249	.FORM.....	653
#pragma interrupt.....	244	.FREG.....	625
#pragma noregalloc.....	252	.FRES.....	639
#pragma noregsave.....	252	.GLOBAL.....	642
#pragma pack.....	260	.HEADING.....	654
#pragma regsave.....	252	.IMPORT.....	641
#pragma section.....	240	.INCLUDE.....	661
#pragma stacksize.....	243	.INSTR.....	689
#pragma tbr.....	254	.LEN.....	688
#pragma unpack.....	260	.LINE.....	648
\$EVEN.....	577	.LIST.....	650
\$EVEN2.....	577	.MACRO.....	681
\$G0.....	153	.ORG.....	620
\$G1.....	153	.OUTPUT.....	643
\$ODD.....	577	.PAGE.....	655
\$ODD2.....	577	.PRINT.....	649
\$TBR.....	153	.PROGRAM.....	657
\$XB.....	153, 233	.RADIX.....	658
\$XC.....	152, 233	.REG.....	624
\$XD.....	152, 233	.RES.....	635
\$YB.....	153, 233	.SDATA.....	628
\$YC.....	152, 233	.SDATAB.....	629
\$YD.....	152, 233	.SDATAC.....	630
* 指定.....	383, 385	.SDATAZ.....	631
.AELIF.....	671	.SECTION.....	617
.AELSE.....	671, 673	.SPACE.....	656
.AENDI.....	671, 673	.SRES.....	636
.AENDR.....	674	.SRESC.....	637
.AENDW.....	675	.SRESZ.....	638
.AERROR.....	677	.STACK.....	659
.AIF.....	671	.SUBSTR.....	690
.AIFDEF.....	673	.XDATA.....	634
.ALIGN.....	621	__accum.....	233
.ALIMIT.....	677	__circ.....	234
.AREPEAT.....	674	__DATE_.....	217
.ASSIGN.....	623	__ec2p_os.....	469
.ASSIGNA.....	667	__fixed.....	233
.ASSIGNC.....	669	__FLT_.....	135
.AWHILE.....	675	__HITACHI_.....	135
		__HITACHI_VERSION_.....	135
		__RENESAS_.....	135
		__RENESAS_VERSION_.....	135
		__sat.....	234
		__secend.....	265

_secksize	265
_sectop	265
TIME	217
_X	233
_Y	233
_B_cnt_ptr	455
_B_len_ptr	455
_BIG	135
_CLOSEALL	178
_DON	135
_DSPC	135
_ec2p_new_handler	478
_file_Ptr	476
_FLT	135
_fmtmask	447
_FPD	135
_FPS	135
_FXD	135
_h_c_lib.h	819
_im	480, 487
_Init	476
_INIT_IOLIB	176
_INIT_LOWLEVEL	176
_INIT_OTHERLIB	176
_INITLIB	176
_IOFBF	373
_IOLBF	373
_IONBF	373
_LIT	135
_PIC	135
_re	480, 487
_REENTRANT	128
_RON	135
_SH	135
_SH1	135
_SH2	135
_SH2A	135
_SH2AFPU	135
_SH2DSP	135
_SH2E	135
_SH3	135
_SH3DSP	135
_SH4	135
_SH4A	135
_SH4ALDSP	135
_statemask	447
~mystrbuf	476

Numerics

10 进制数字	327
16 进制数字	327
8 位绝对区域地址值的指定	123

A

abort	84, 196, 513
abs	136, 425, 485, 492
abs16	14
abs20	14
abs28	14
abs3	14
absolute	95
absolute_forbid	109
ACCUM_BIT	339
ACCUM_EPSILON	339
ACCUM_MAX	339
ACCUM_MIN	339
acos	342
acosf	356
add4	293
addc	306
addv	307
adjustfield	446
aggressive	46
ALIGN	617
align16	16
align32	16
all	13, 14, 33, 127
app	447
approxdiv	45
arg	485, 492
ASCII 码	836
asctime	513
asin	342
asinf	356
asmcode	10
assert	326
assert.h	326
assigna	65
assignc	66
atan	343
atan2	344
atan2f	358
atanf	357
ate	447
atexit	195, 513
atof	407
atoi	407
atol	408
atolaccum	409
atolfixed	409
atoll	408
auto	13, 28
auto_enum	47
auto_literal	76

B

B	152
B_beg_pptr	455

B_beg_ptr	455	cachesize	108
B_end_ptr	455	cal	136
B_next_pptr	455	CALL	650
B_next_ptr	455	calloc	418
badbit	447	calloc_X	420
bank	244	calloc_Y	422
basefield	447	calls	73
beg	447	case	29
BIG	646	cc	136
big	55, 82, 830	ceil	352
bin	136	ceilf	366
binary	92, 95, 447	change_message	7, 120
bit_order	58	char	218
blockcopy	30	CHAR_BIT	337
bls	136	CHAR_MAX	337
bool	218	CHAR_MIN	337
boolalpha	446, 452	chcount	462
branch	68, 106	chkalign8	80
bsearch	424	chkcache	78
bss	11, 14, 18	chkdsp	79
bss_order	17	chkfpu	79
BTBL	176	chkmd	77
BUFSIZ	373	chktlb	78
Byte_count	102	class 型	219
版权	124	clearerr	404
饱和限定词	234	clipsb	316
保护位	324	clipsw	316
比较型条件汇编	663	clipub	317
编程	152	clipuw	317
调整数	152	clock	513
变量存取优化对象的符号信息	147	close	182, 476
编译程序	2	clr_circ	300
编译列表	137	clr	304
表达式	573	CODE	617, 650
标号	561	code	10, 73, 152
标量型	218	COL	653
标志	380	columns	87
标识符	213	comment	42
标准 C 库	320	complex	127, 480
标准错误输出文件	323	compress	117
标准输出文件	323	COND	650
标准输入 / 输出文件	323	conditionals	73
标准输入文件	323	conj	485, 493
并行运算指令	605	const	11, 12, 14, 18
不支持的库	513	const_iterator	495
		const_load	37
		const_var_propagate	36
C		ConvComplete	544
C	152	ConvCyclic	545
c	60, 136	ConvPartial	546
C\$INIT	153	CopyFromX	552
C\$VTBL	153	CopyFromY	552
C++ 初始处理 / 后处理的数据区	153	CopyToX	551
C++ 虚拟函数表区	153	CopyToY	551
C/C++ 库	320	CopyXtoY	550
C/C++ 库函数的初始设定	176	CopyYtoX	550
C_flg_ptr	455	CorrCyclic	548

Correlate	547	DBL_MAX	335
cos	345, 485, 493	DBL_MAX_10_EXP	335
cosf	359	DBL_MAX_EXP	335
cosh	346, 486, 493	DBL_MIN	335
coshf	360	DBL_MIN_10_EXP	335
cp	136	DBL_MIN_EXP	335
cpp	60, 136	DBL_NEG_EPS	336
cppstring	127	DBL_NEG_EPS_EXP	336
cpu	15, 54, 81, 113	DBL_POS_EPS	336
CPU 选项卡	53, 81	DBL_POS_EPS_EXP	336
CREF	649	DCF	606
cross_reference	74	DCT	606
ctime	513	debug	11, 67, 97
ctype	127	debug_only	24
ctype.h	327	dec	446, 454
cur	447	declaration	17
参数的长度指定	381	DEF	650
参数的分配	204	define	6, 64, 92
参数的传递方法	202	definition	17
参数分配区	203	definitions	73
参照	218	del_vacant_loop	34
操作长度	581, 608, 611	delete	118
操作码	561	denormalize	56, 83
操作数	561	difftime	513
常数	235, 565	Dlir	536
常数符号	563	Dlir1	537
常数区	152	dispsize	69
长字的定点常数	571	div	426
程序的开发步骤	1	div_t	406
程序结构	152	div0s	311
程序区	152	div0u	312
初始化的数据区	152	div1	311
初始设定	175	division	15
初始设定程序的建立	159	dmuls_h	284
初始值	152	dmuls_l	284
窗函数	528	dmulu_h	285
存储器管理库	478	dmulu_l	285
存储器使用量的削减指定	117	double	55, 218
存储器型限定词	233	double = float	57
存储区的分配	159	double_complex	
错误	708, 779, 798, 811	double_complex	487
错误格式	708, 779, 798, 811	imag	487
错误级	708, 779, 798, 811	operator*=	488
错误信息	145, 150, 708, 779, 798, 811	operator+=	488
错误指示符	324	operator/=	488
		operator-=	488
		operator=	488
		real	487
D		double_complex 成员外函数	489
D	152	double_complex 类	487
DATA	617	double 型	229
data	11, 12, 14, 18, 152	dsp	830
data_stuff	101	dspc	42
DBG	643	DSP-C 规格	233
dbg	136	DSP 寄存器直接	608, 611
DBL_DIG	336	DSP 库	514
DBL_EXP_DIG	336	DSP 运算操作	605, 606
DBL_MANT_DIG	336		

DSP 运算指令	608
DSP 指令	605, 608
DTBL	176
dwarf1	831
DUMMY	617
大小选择模式	692
大小指定模式	692
打印字符	327
带变址的 GBR 间接	582
带变址的寄存器间接	582
带变址的寄存器间接 / 后增	611
带条件的 DSP 运算指令	609
带位移量的 GBR 间接	582
带位移量的寄存器间接	582
带位移量的 PC 相对	582
带位移量的 TBR 双重间接	582
单片机选项	123
单数据的传送操作	605
低级接口例程	179
递减	582
递增	582
地址符号	563
地址一致性的检查	113
调试信息	97
调试信息的删除	120
调试信息的压缩	117
定标	517
定点常数	570
定点常数的数据格式	571
定点型	233
定点转换说明符	237
定位计数器	572
定义型条件汇编	663
定义域错误	341, 355
段	152
段初始化表	176
段的连接	156
段地址	110
段地址运算符	265
段名	563
段内数据的紧凑分配	101
段属性	152
段信息列表	143
段选项	110
堆区	153
堆栈区	153
堆栈信息文件	116
堆栈转换的指定	244
对照表	142
对照表信息	148

E

EACCU MO	777
EACCU MU	777
EBADF	777
EC++ 类库	444
ECBASE	776
ecpp	41
ECSPEC	777
EDBLO	776
EDBLU	776
EDOM	340, 354, 776
EDSP_BAD_ARG	514
EDSP_NO_HEAP	514
EDSP_OK	514
EEXP	776
EEXP N	776
EFFTALLSCALE	517
EFFTMIDSCALE	517
EFFTNOSCALE	517
EFIXEDO	777
EFIXEDU	777
EFLOATO	776
EFLOATU	776
EIOBFRSM	778
EIOBRESM	193, 778
ELACCU MO	777
ELACCU MU	777
ELDBLO	777
ELDBLU	777
ELFIXEDO	777
ELFIXEDU	777
EMALFRSM	193, 777
EMALRESM	193, 777
enable_register	47
end	125, 447
end_cnvl	287
endian	55, 82, 830
endl	473
ends	473
ensigdsp.h	514
entry	93
enum	218
EOF	322, 373
eof	455
eofbit	447
EQ	671, 675
ERANGE	338, 340, 354, 776
EREPEAT	705
errno	325, 338
errno.h	338
errno_addr	185
error	7, 84, 120
ESTRN	776
ETLN	776
ETOKFRSM	193, 778
ETOKRESM	193, 778
euc	61, 62, 86
exception	59
exclude	77
exit	125, 195, 513
EXP	650
exp	136, 348, 486, 493

expand	68
expansion	21
expansions	73
expf	362
extract	119
二进制文件	92, 323

F

fabs	352
fabsf	366
failbit	447
fclose	376
FCNVDS	600
FCNVSD	600
fdiv	48
feof	404
ferror	405
fflush	376
FftComplex	518
FftInComplex	522
FftInReal	523
FFT 结构	517
FftReal	519
fgetc	392
fgetpos	513
fgets	393
FILE	373
file_inline	31
file_inline_path	7
FILE 结构体	322
FILE 型	178
fillch	446
filt_ws.h	514
fipr	289
Fir	532
Fir1	533
fixed	447, 454
fixed.h	339
FIXED_BIT	339
fixed_const	49
FIXED_EPSILON	339
FIXED_MAX	339
fixed_max	49
FIXED_MIN	339
fixed_noround	50
FLOAT	600
float	218
float.h	335
float_complex	
float_complex	480
imag	481
operator*=	481
operator+=	481
operator/=	481
operator-=	481
operator=	481

real	480
float_complex 成员外函数	482
float_complex 类	480
floatfield	447
float 型	228
floor	353
floorf	367
FLT_DIG	335
FLT_EXP_DIG	336
FLT_GUARD	335
FLT_MANT_DIG	336
FLT_MAX	335
FLT_MAX_10_EXP	335
FLT_MAX_EXP	335
FLT_MIN	335
FLT_MIN_10_EXP	335
FLT_MIN_EXP	335
FLT_NEG_EPS	336
FLT_NEG_EPS_EXP	336
FLT_NORMALIZE	335
FLT_POS_EPS	336
FLT_POS_EPS_EXP	336
FLT_RADIX	335
FLT_ROUNDS	335
flush	473
fmod	353
fmodf	367
fmtfl	446
fmtflags	446
fopen	377
form	95
fprintf	380
fpscr	46
fpu	55, 829
fputc	394
fputs	395
fread	400
free	418
free_X	420
free_Y	422
FreeDir	543
FreeFft	527
FreeFir	542
FreeIir	542
FreeLms	543
freopen	378
frexp	348
frexpf	362
fsca	301
fscanf	384
FSCHG	600
fseek	402
fsetpos	513
fsrra	301
fsy	136
fsymbol	112
ftell	403

FTRC	600
ftv	290
ftvadd	291
ftvsub	292
function_call	106
function_forbid	109
fwrite	401
返回码	323
返回值的设定位置	205
范围错误	341, 355
方式种类	152
非数值	228, 229, 230, 567
非规范化数	227, 228, 229, 567
浮点	214
浮点常数	566
浮点数据格式	568
浮点型	323
浮点型的规格	227
浮点型的内部表示	227
符号	227, 563, 568, 571
符号地址文件	112
符号定义	92
符号名的变更	118
符号名的删除	118
符号删除优化信息	147
符号信息	146
符号指定的 PC 相对	582
复数计算类库	480
复数数据的数组格式	516

G

gbr	28
gbr_and_byte	277
gbr_or_byte	277
gbr_read_byte	274
gbr_read_long	275
gbr_read_word	274
gbr_tst_byte	278
gbr_write_byte	275
gbr_write_long	276
gbr_write_word	276
gbr_xor_byte	278
GBR 段	153
GE	671, 675
GenBlackman	528
GenGNoise	553
GenHamming	529
GenHanning	529
GenTriangle	530
get_cr	270
get_fpscr	289
get_gbr	273
get_imask	271
get_tbr	318
get_vbr	272
getc	396

getchar	396
getenv	513
getline	510
gets	397
global_alloc	35
global_volatile	32
gmtime	513
goodbit	447
goptimize	25
GT	671, 675
高速傅立叶转换	516
高速缓存的容量	108
格式转换程序	3, 828
共用码长度	107
工作区	531
关键字	563
关键字指定	686

H

h	136
H16	97
H20	97
H32	97
head	127
hex	136, 447, 454
hexadecimal	95
HEX 文件格式	834
hide	121
HIGH 运算	576
HLNK_DIR	134
HLNK_LIBRARY1	133
HLNK_LIBRARY2	133
HLNK_LIBRARY3	133
HLNK_TMP	133
HUGE_VAL	340
HUGE_VALF	354
HWORD 运算	577
函数	322
函数存取优化对象的符号信息	147
宏	322
宏处理除外	684
宏调用	679, 686
宏定义	679
宏功能	678
宏内注释	685
宏生成号	684
宏体	683
后增寄存器间接	582, 611
环境	213
环境变量	132
汇编程序的语言规格	561
汇编程序控制指令	615
汇编列表	141

I		
icbi	302	
lfftComplex	520	
lfftInComplex	524	
lfftInReal	525	
lfftReal	521	
ifthen	29	
ifunc	15	
ii	136	
lir	534	
lir1	535	
imag	485, 492	
in	447	
include	5, 21, 64	
infinite_loop	35	
information	7, 120	
init_cnt	445	
InitDlir	541	
InitFft	527	
InitFir	540	
Initlir	540	
InitLms	541	
inline	15, 29, 30, 31, 37	
in-place FFT	517	
input	91	
int	218	
INT_MAX	337	
INT_MIN	337	
int_type	445	
internal	446, 453	
iomanip	444	
ios	127, 444	
ios		
~ios	449	
bad	450	
clear	450	
copyfmt	451	
eof	450	
fail	450	
good	450	
init	449	
ios	449	
operator	450	
operator void	449	
rdbuf	451	
rdstate	450	
setstate	450	
tie	450, 451	
ios_base		
_ec2p_copy_base	447	
_ec2p_init_base	447	
~Init	445	
~ios_base	448	
fill	448	
flags	448	
fmtflags	446	
Init	445	
Init 类	445	
ios_base	447	
iostate	447	
openmode	447	
precision	448	
seekdir	447	
setf	448	
width	448	
unsetf	448	
ios_base 类	446	
ios 类	449	
ios 类操纵符	452	
iostate	446	
iostream	444	
isalnum	328	
isalpha	329	
iscntrl	329	
isdigit	330	
isgraph	330	
islower	331	
isprint	331	
ispunct	332	
isspace	332	
istream	444	
istream		
_ec2p_getistr	463	
~istream	463	
~sentry	461	
gcount	464	
get	464, 465	
getline	465	
ignore	465	
istream	463	
operator bool	461	
operator>>	464	
peek	465	
putback	466	
read	466	
readsome	466	
seekg	466	
sentry	461	
sentry 类	461	
sync	466	
tellg	466	
unget	466	
istream 成员外函数	468	
istream 类	462	
istream 类操纵符	467	
isupper	333	
isxdigit	333	
iterator	495	
J		
jmp_buf	368	
jump	68	
基本型	218	

寄存器	216
寄存器别名	563
寄存器间接	582, 611
寄存器直接	582
记录长度的统一	97
基数	323
继续处理	125
结构体	216, 219
结构体型	219
结束处理	125
结束处理例程	195
结束码	116
精度	381
警告	708, 779, 798, 811
局部标号	579
局部符号名的隐含指定	121
卷积	544
绝对地址变量区	153
绝对寻址方式	152

K

可重入库	511
空格类字符	327
空区域输出的指定	99
控制字符	327
空字符	322
库列表	149
库内模块、段、符号信息	151
库文件	91
库信息	150
库	237
括号	574
扩展功能	239
扩展重复循环指令的自动生成功能	704

L

L_tmpnam	373
labs	426
LACCUM_BIT	339
LACCUM_EPSILON	339
LACCUM_MAX	339
LACCUM_MIN	339
lang	60
latin1	61, 85
lbp	136
ld_ext	296
LDBL_DIG	336
LDBL_EXP_DIG	336
LDBL_MANT_DIG	336
LDBL_MAX	335
LDBL_MAX_10_EXP	335
LDBL_MAX_EXP	335
LDBL_MIN	335
LDBL_MIN_10_EXP	335

LDBL_MIN_EXP	335
LDBL_NEG_EPS	336
LDBL_NEG_EPS_EXP	336
LDBL_POS_EPS	336
LDBL_POS_EPS_EXP	336
ldexp	349
ldexpf	363
ldiv	427
ldiv_t	406
ldtlb	283
LE	671, 675
left	58, 259, 446, 453
length	21
lfixed_as_long	298
LFIXED_BIT	339
LFIXED_EPSILON	339
LFIXED_MAX	339
LFIXED_MIN	339
lib	136
library	91, 95
Limit	549
limits.h	337
LIN	653
lines	87
lis	136
LIST	649
list	71, 103
listfile	20
literal	37, 68
LITTLE	646
little	55, 82, 830
llabs	427
lldiv	428
lldiv_t	406
LLONG_MAX	337
LLONG_MIN	337
Lms	538
Lms1	539
locale.h	513
localeconv	513
localtime	513
LOCATE	617
log	349, 486, 493
log10	350, 486, 493
log10f	364
logf	363
logic_gbr	39
LogMagnitude	526
logo	61, 88, 124
long	218
long _accum	233
long _fixed	233
long double	218
long double 型	229
long long	218
long_as_lfixed	298
LONG_MAX	337

LONG_MIN	337
longjmp	369
loop	44
LOW 运算	577
lpp	136
lseek	183
lst	136
LT	671, 675
LWORD 运算	577
类	216
立即数	582, 609
联合体	216, 219
联合体型	219
连接列表	144
连接映像信息	145
列表种类	20, 70
列表内容	104
列表文件	103
列表选项	103
零	227, 229, 230, 567
流输入 / 输出类库	444
流输入 / 输出	322
滤波器	531

memset	442
message	6, 100
Minl	559
mktime	513
modf	350
modff	364
mot	136
movt	304
msg_unused	100
MsPower	556
mtrx4mul	294
mtrx4muladd	295
mtrx4mulsub	295
myfptr	476
mystdbuf	476
没被参照的定义符号的通知	100
枚举型	216
命令指定的信息	140
模块的抽出	119
模块的置换	119
目标种类	8, 67
目标文件的格式	828
目标信息	139

M

machine.h	269
machinecode	10
macl	288
macll	288
macsave	43
macw	287
macwl	287
malloc	419
malloc_X	421
malloc_Y	423
map	26, 99, 136
math	127
math.h	340
mathf	127
mathf.h	354
MatrixMult	554
max_unroll	34
Maxl	559
mblen	513
mbstowcs	513
mbtowc	513
Mean	557
memchr	437
memcmp	435
memcpy	431
memcpy_X_X	431
memcpy_X_Y	432
memcpy_Y_X	432
memcpy_Y_Y	433
memmove	443
memory	117

N

NE	671, 675
nearest	56, 82
negc	310
nest	42
new	127, 478
new_handler	478
noalign	16
noblock	33
noboolalpha	452
NOCALL	650
NOCODE	650
nocompress	117
NOCOND	650
NOCREF	649
nocross_reference	74
NODBG	643
nodebug	11, 67, 97
NODEF	650
noexception	59
noexclude	77
NOEXP	650
noexpansion	21
nofloat	128
noinclude	21
noinline	31
noline	10
NOLIST	649
nolist	71
nolistfile	20
nologo	61, 88, 124
noloop	33, 44

nomask	15
nomessage	6, 100
none	13
nonest	42
NOOBJ	643
noobject	21, 69
nooptimize	106
nop	283
noprelink	93
norm	485, 493
nortnext	44
noscope	38
NOSCT	649
nosection	74
noshow	73
noshowbase	452
noshowpoint	453
noshowpos	453
noskipws	453
nosource	21, 72
nospeed	25
NOSRC	649
nostatistics	21
nostuff	18
not-in-place FFT	517
NOTOPN	777
nouppercase	453
npos	495
NULL	322, 325
内部错误	708, 798
内部函数	266

O

O_APPEND	181
O_CREAT	181
O_RDONLY	181
O_RDWR	181
O_TRUNC	181
O_WRONLY	181
OBJ	643
obj	136
object	21, 69, 95
objectfile	12
ocbi	302
ocbp	302
ocwbw	302
oct	446, 454
OFF	645, 650
off	56, 59, 83
off_type	445
offsetof	325
ok_	461, 469
ON	645, 650
on	56, 59, 83
open	181, 476
openmode	446

operator-	484, 491
operator delete	479
operator new	478, 479
operator!=	485, 492, 509
operator*	484, 491
operator+	484, 491, 509
operator/	484, 492
operator==	484, 492, 509
operator>	509
operator>=	510
operator>>	468, 485, 492, 510
operator1"	509
operator22"	474, 485, 492, 510
operator3"	509
opt_range	33
optimize	24, 106
ostream	444
ostream	
~ostream	470
~sentry	469
flush	471
operator bool	469
operator22"	471
ostream	470
putc	471
seekp	472
sentry	469
sentry 类	469
tellp	471
write	471
ostream 成员外函数	474
ostream 类	470
ostream 类操纵符	473
overflow	476
ovf_addc	307
ovf_addv	308
out	447
outcode	62, 86
output	98, 127

P

P	152
p	136
pabs_la	297
pabs_lf	297
pack	58
patch	45
path	132
pbackfail	476
pdmsb_la	297
pdmsb_lf	297
Peak1	560
peripheral	15
perror	405
pi	136
pic	57

polar	485, 493
pool	68
pos_type	445
pow	351, 486, 493
PowerON_Reset	175
powf	365
pp	136
prec	446
prefetch	282
prefi	303
preinclude	5
preprocessor	10
printf	387
pro	136
PROfile	108
profile	108
program	11, 14
PS_check	114
psha_la	298
psha_if	298
ptrdiff_t	325
PTRERR	776
putc	397
putchar	398
puts	398
配置信息	108
偏移种类	402

Q

qNAN	567
qsort	425
奇偶运算	577
其他种类	40
其它种类	75
其它选项	115

R

raise	513
rand	417
RAND_MAX	406
read	182
real	485, 492
realloc	419
realloc_X	421
realloc_Y	423
record	97
reent	128
reference	104
register	106
rel	136
relocate	95
remove	513
rename	118, 513
REPEAT	700

repeat	50
replace	119
resbank	244
resetiosflags	475
return	68
rewind	403
right	58, 259, 446, 453
rndtoa	299
rndtof	299
rom	98
ROM 化支持	98
rotcl	313
rotcr	314
rotl	312
rotr	313
round	56, 82
rti	136
rtnext	44
rts	244
rtti	59
run	14
runtime	15, 29, 30, 127

S

S	153
s_len	495
s_ptr	495
s_res	495
S1	97
S2	97
S3	97
S9	116
safe	46, 106
same_code	106
samecode_forbid	109
samesize	107
save_cont_reg	43
sb	449
sbr	123
sbrk	184
sbrk_X	184
sbrk_Y	185
scanf	387
SCHAR_MAX	337
SCHAR_MIN	337
schedule	37
scientific	447, 454
scope	38
SCT	649
sdebug	97
section	11, 74, 104
section_forbid	109
SEEK_CUR	373, 402
SEEK_END	373, 402
SEEK_SET	373, 402
seekdir	446

seekoff	476	SHC_TMP	133
seekpos	476	SHCPU	132
set_circ_x	299	SHDSP	616
set_circ_y	299	shdsp	81
set_cr	270	shdsplib.lib	514
set_cs	300	shdsppic.lib	514
set_fpscr	288	shift	29
set_gbr	273	shll	314
set_imask	271	shlr	315
set_new_handler	479	short	218
set_tbr	318	short_format	106
set_vbr	272	show	21, 73, 104
setbase	475	showbase	446, 452
setbuf	378, 476	showmanyc	476
setfill	475	showpoint	446, 452
setjmp	369	showpos	446, 453
setjmp.h	368	SHRT_MAX	337
setiosflags	475	SHRT_MIN	337
setlocale	513	signal	513
setprecision	475	signal.h	513
sett	305	signal_sem	186
setw	475	signed char	218
setvbuf	379	SIMple_float_conv	51
S 格式文件	832	sin	345, 486, 493
SH1	132, 616	sinf	359
sh1	54, 81	single	55
SH2	132, 616	sinh	347, 486, 494
sh2	54, 81	sinhf	361
SH2A	132, 616	size	25
sh2a	54, 81	size_t	325
SH2AFPU	132, 616	SIZEOF 运算	575
sh2afpu	54, 81	sjis	61, 62, 85, 86
SH2DSP	132	skipws	446, 453
sh2dsp	54	sleep	280
SH2E	132, 616	smachine.h	269
sh2e	54, 81	smanip 类操纵符	475
SH3	132, 616	smap	27
sh3	54, 81	sNAN	567
SH3DSP	132, 616	sni	136
sh3dsp	54, 81	softpipe	38
sh3dspnb.lib	514	source	21, 72
sh3dspnl.lib	514	sp	244, 253
sh3dsppb.lib	514	space	99
sh3dsppl.lib	514	speed	25, 106
SH4	132, 616	sprintf	388
sh4	54, 81	sqrt	351, 486, 494
SH4A	132, 616	sqrtf	365
sh4a	54, 81	sr_jsr	319
SH4ALDSP	132, 616	sr_rts	244
sh4aldsp	54, 81	srand	417
sh4aldspnb.lib	514	SRC	649
sh4aldspnl.lib	514	src	136
sh4aldsppb.lib	514	sscanf	388
sh4aldsppl.lib	514	st_ext	296
shar	315	STACK	617
SHC_INC	133	stack	116, 152
SHC_LIB	132	start	110

STARTOF 运算	575	xspn	460
state	449	streambuf 类	455
static	13	streamoff	445
statistics	21	streamsize	445
stdarg	127	strerror	442
stdarg.h	370	strtime	513
stddef.h	325	strict_ansi	48
stderr	323, 373	string	12, 127, 495
stdin	323, 373	string	
stdio	127	~string	500
stdio.h	373, 513	append	502
stdlib	127	assign	503
stdlib.h	406, 513	at	502
stdout	323, 373	begin	501
strcat	434	c_str	505
strchr	437	capacity	501
strcmp	436	clear	501
strcoll	513	compare	507
strcpy	433	copy	505
strcspn	438	empty	502
streambuf	444	end	501
streambuf		erase	504
~streambuf	456	find	505
eback	458	find_first_not_of	506
egptr	458	find_first_of	505, 506
epptr	459	find_last_not_of	506
gbump	458	find_last_of	506
gptr	458	insert	503
in_avail	457	length	501
overflow	460	max_size	501
pbackfail	460	operator	502
pbase	459	operator+=	502
pbump	459	operator=	500
pptr	459	replace	504
pubseekoff	457	reserve	501
pubseekpos	457	resize	501
pubsetbuf	457	rfind	505
pubsync	457	size	501
sbumpc	457	string	500
seekoff	459	swap	505
seekpos	459	substr	506
setbuf	459	string.h	429, 513
setg	459	string_unify	106
setp	459	string 类	495
sgetc	457	string 类操纵符	508
sgetn	458	strip	120
showmanyc	460	strlen	443
snextc	457	strncat	435
sputbackc	458	strncmp	436
sputc	458	strncpy	434
sputn	458	strpbrk	438
streambuf	456	strchr	439
sungetc	458	strspn	439
sync	460	strstr	440
uflow	460	strtod	410
underflow	460	strtok	441
xsgetn	460	strtol	411

strtolaccum	416
strtolfixed	415
strtoll	413
strtoul	412
strtoull	414
struct_alloc	36
strxfrm	513
stuff	18
stype	95
sub4	293
subc	308
subcommand	62, 88, 122
subv	309
swap	510
swapb	286
swapw	286
symbol	104
symbol_delete	106
symbol_forbid	109
sync	476
synco	303
SYS_OPEN	373
sysrof	831
system	513
上溢	230
舍入	323
声明	217
剩余选项	124
实数数据的数组格式	517
实现性定义	324
输出格式	95
输出文件	98
输出选项	94
数据的内部表示	218
数据格式	515
数据纪录的字节数指定	102
数据传送指令	606, 611
输入文件	91
输入选项	90
数组	215
数组型	219
算术转换	236

T

TAB	653
tab	21, 73
table	29
tan	346, 486, 494
tanf	360
tanh	347, 486, 494
tanhf	361
tas	280
tbr	16
TBR 表的区域	153
td	136
template	13

ti	136
tiestr	449
time	513
time.h	513
TMP_MAX	373
tmpfile	513
tmpnam	513
tn	244
tolower	334
toupper	334
tpdir	136
trace	282
trapa	281
trapa_svc	281
trunc	447
特殊字符	327
条件	605
条件汇编	663
条件汇编功能	662
条件重复展开	664
统计信息	140

U

UCHAR_MAX	337
UINT_MAX	337
ULLONG_MAX	337
ULONG_MAX	337
umachine.h	269
unaligned	30
underflow	476
unf_subc	309
unf_subv	310
ungetc	399
unitbuf	446
unsigned char	218
unsigned int	218
unsigned long	218
unsigned long long	218
unsigned short	218
uppercase	446, 453
used	13
user	28
USHRT_MAX	337

V

va_arg	371
va_end	372
va_list	370
va_start	371
value_type	480, 487
variable_access	106
variable_forbid	109
Variance	558
VEC_TBL	167

VectorMult.....	555
vfprintf.....	389
volatile_loop.....	46
vprintf.....	390
vsprintf.....	391

W

wait_sem.....	186
warning.....	7, 84, 120
wcstombs.....	513
wctomb.....	513
wide.....	446
width.....	21
write.....	183
ws.....	467
外部符号分配信息文件的输出.....	99
未初始化的数据区.....	152
尾数.....	227
位移量.....	582
位域.....	216, 223
位置指定.....	686
位置指示符.....	324
文本文件.....	323
文件包含功能.....	660
文件存取模式.....	324
文件的关闭.....	178
文件规格.....	136
文件结束指示符.....	324
文件扩展名.....	136
文件指针.....	322
文字.....	691
文字池.....	691
文字池的自动生成功能.....	691
物理空间的重叠检查.....	114
无穷大.....	227, 229, 230, 567
无效运算.....	231

X

X 存储器常数区.....	152
X 存储器初始化数据区.....	152
X 存储器未初始化数据区.....	153
xreference.....	104
X 数据的传送操作.....	605
xtrct.....	305
系数的定标.....	531
下溢.....	230
先减寄存器间接.....	582, 611
陷阱指令返回的指定.....	244
限制.....	813
项.....	573
相对寻址方式.....	152
相互关系.....	544
向量表的设定.....	167
效率.....	516

小数.....	568, 571
消息级.....	120
写操作.....	152
信息.....	708, 798
信息消息.....	100
型限定词.....	217, 233
型转换.....	235
型转换的规则.....	202
选项的指定规则.....	89
选项信息.....	144, 149
循环限定词.....	234
寻址方式.....	581, 608, 611

Y

Y 存储器常数区.....	152
Y 存储器初始化数据区.....	152
Y 存储器未初始化数据区.....	153
Y 数据传送操作.....	605
验证选项.....	113
英文大写字母.....	327
英文小写字母.....	327
英文字母.....	327
优化.....	106
优化种类.....	22
优化的部分抑止.....	109
优化连接编辑程序.....	2
优化选项.....	105
预处理器.....	217
预处理器变量.....	662
预定义宏.....	135
语句.....	217
预连接程序.....	3, 93
语言规格.....	213
源种类.....	4, 63
源列表信息.....	137, 141
源语句.....	561, 605
运算符.....	237, 573
运算符的评价顺序.....	232

Z

zero.....	56, 82
栈解析工具.....	3, 131
正规化.....	323
正规化数.....	227, 228, 229, 567
整数.....	214
整数常数.....	565
整型预处理器变量.....	662
置换符号.....	662
致命错误.....	708, 779, 798, 811
指数.....	227, 568
指向函数成员的指针.....	218
指向数据成员的指针.....	218
执行环境的设定.....	166
执行开始的地址.....	93

执行指令	581
指针	215, 218
指针转换	237
重复循环指令的自动生成功能	699
重复展开	664
助记符	581, 611
注释	561
转换字符	382, 385
字的定点常数	570
字段宽度	381, 385
字符	213
字符常数	566
字符种类	327
字符串	579
字符串操作函数	688
字符串操作类库	495
字符列操作函数	685
字符型预处理器变量	662
子命令文件	122
子命令文件选项	122

瑞萨单片机开发环境系统

用户手册

SuperH™ RISC engine C/C++ 编译程序、汇编程序、
优化连接编辑程序

Publication Date: 1st Edition, Jun. 20, 2007

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, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.

Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

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

SuperH™ RISC engine 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

RCJ10J0045-0100