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.

Keep safety first in your circuit designs!

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

Notes regarding these materials

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

SuperH RISC Engine C/C++编译程序应用笔记

前言

前言

Renesas Technology SuperH RISC engine 家族的下一代单片微型计算机可在合并各种外围设备的同时提供高性能的处理，并且专为嵌入式应用程序和在低功耗下操作而精心设计。

这些应用笔记说明如何使用 SuperH RISC engine C/C++ 编译程序封装 9.00 版本有效创建可利用 Renesas Technology SuperH RISC engine 家族的功能与性能之应用程序的方法。

要获取有关 C/C++ 编译程序的详细规格，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)。

这些应用笔记的组织

这些应用笔记包含下列十个章节和一个附录。

第 1 节 提供概述并叙述安装方法以及编程开发的步骤。

第 2 节 举例阐释调试过程以及说明如何使用 C 语言创建程序的方法。

第 3 节 提供在结合 C 语言程序和汇编语言程序时，以及在配合使用交叉软件和利用 C/C++ 编译程序所创建的目标文件时需要注意的警告，此外，也说明 SuperH RISC engine C/C++ 编译程序的扩展功能，以及嵌入式设备的软件之特殊步骤。

第 4 节 说明 HEW 选项。

第 5 和第 6 节 说明如何创建特别设计为利用 Renesas Technology SuperH RISC engine 家族微型计算机的性能之 C 语言应用程序的方法。

第 7 节 阐释使用 HEW 的实用方法。

第 8 节 阐释有效的 C++ 编程技术。

第 9 节 说明有用的选项，以及连接期间在模块间执行支线优化的功能。

第 10 节提供用户常见问题的解答。

附录说明每个 SuperH RISC engine C/C++ 编译程序版本中的变更。

相关手册

以下列出相关的手册。

- Renesas Technology SuperH RISC engine 家族，微型计算机硬件手册 (Renesas Technology SuperH RISC engine Family, Microcomputer Hardware Manuals)
- 高性能嵌入式工作区用户手册 (High-performance Embedded Workshop User's Manual)
- HEW 创建程序用户手册
- HEW 调试程序用户手册
- SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)

- SuperH RISC engine 模拟程序/调试程序用户手册 (SuperH RISC engine Simulator/Debugger User's Manual)
- SuperH RISC engine 高性能嵌入式工作区教程 (SuperH RISC engine High-prformance Embedded Workshop Tutorial)

交叉软件版本

为了能够使用 SuperH RISC engine C/C++ 编译程序 9.00 版本，必须使用下列交叉软件版本。

交叉软件名称	版本
SH 系列交叉汇编程序	7.00
H 系列优化连接编辑程序	9.00
SH 系列程序库生成程序	3.00

本应用笔记所使用的符号以及惯例。

[ ]: 表示可省略格子内的项目。

(RET): 表示需要按下回车 (Enter) 键。

Δ: 表示一个或多个空格或制表符。

abc: 加粗项目必须由用户输入。

<>: 须指定括号内的项目。

… : 表示紧邻在前的项目被指定一次或多次。

H': 前面标有 H' 的整数常数为十六进制。

0x: 前面标有 0x 的整数常数为十六进制。

UNIX 是在美国及其他国家（地区）的注册商标，通过 X/Open Company limited 独家授权。

MS-DOS® 是 Microsoft Corporation 在美国及其他国家（地区）的注册商标。

Microsoft® WindowsNT® 操作系统、Microsoft®、Windows®98 以及 Windows 2000 操作系统、Microsoft® WindowsMe® 操作系统、Microsoft® WindowsXp® 操作系统是 Microsoft Corporation 在美国及其他国家（地区）的注册商标。

IBM PC 是 International Business Machines Corporation 的注册商标。

建议依照下列方式阅读这些应用笔记。

编号	情况	这些应用笔记的使用
1	初次使用 SuperH RISC engine C/C++ 编译程序 (1) 您要了解如何使用编译程序来创建加载模块，以及如何使用交叉软件。 (2) 您要创建可在 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 上运行的程序。	(1) 启动编译程序的步骤在第 1.4 节“执行方法”中说明。 在第 1.5 节中，说明程序开发的步骤、使用完成加载模块所需交叉软件之操作。 (2) 在第 2.2 节和 2.3 节“样品程序简介”中包含一些程序。 所提供的这些程序将说明嵌入式设备需要具备的编译程序功能之绝对最小值。 创建简单程序时请参考这些数值，然后使用模拟程序、调试程序和其他工具来确认其运算。 其他编译程序功能在第 3 节“编译程序”中描述。 如果您在创建加载模块时遇到问题，请参考第 3.15 节“交叉软件的相关事项”。
2	需要创建应用于嵌入式设备的程序 。 (1) 有一个程序与将会套接的其他微型计算机配合使用。 (2) 新的程序将会创建。	(1) 阅读第 2.2 节至 2.3 节“ 样品程序简介”，以及第 3 节“编译程序”，了解您可以使用的功能，然后思考汇编语言代码是否不能以 C 语言重写。 有关结合汇编语言程序与 C 程序的信息，请参考第 3.15.1 节“汇编语言程序的相关事项”。 (2) 先阅读第 2.2 节至 2.3 节“样品程序简介”，以获得有关创建程序的摘要信息。 接着阅读第 3 节“编译程序”，了解有关 SuperH RISC engine C/C++ 编译程序的扩展功能。 创建程序时，请参考第 5 节“有效的编程技术”，以确保您的程序在刚开始时就已成功。
3	需要增进执行速度，或缩减程序大小。	请参考第 5 节“有效的编程技术”来增进性能。
4	程序未如预期般运行。	检查每个相关项目所附的警告信息，以及第 10 节“常见问题集”中的项目，以确定是否有任何相关信息。

SuperH RISC Engine C/C++编译程序应用笔记

概述

第 1 节　概述

1.1　摘要

SuperH RISC engine C/C++ 编译程序汲取 Renesas Technology SuperH RISC engine 家族应用于嵌入式应用程序的单片微型计算机功能与性能，使其 C 程序语言的创建更具效率。

本文档将说明使用此 C/C++ 编译程序创建应用程序的步骤。

1.2　功能

SuperH RISC engine C/C++ 编译程序具备下列功能：

(1) 丰富的功能

下列功能可用来创建 Renesas Technology SuperH RISC engine 家族的有效应用程序。

- 以 C 语言表示 Renesas Technology SuperH RISC engine 家族的中断功能和特殊指令。
- 生成位置无关代码（除了 SH-1）
- 快速浮点运算
- 可选择编译程序设定以提供执行速度的优先级或存储器的使用效率

(2) 强大的优化功能

下列优化类型将会执行从而通过其 RISC（精简指令系统计算机）类型的指令系统来利用 Renesas Technology SuperH RISC engine 家族的性能。

- 自动/优化分配本地变量到寄存器
- 缓和处理强度
- 流水线优化
- 常数卷积
- 字符串共享
- 删除公用表达式/循环不变量
- 删除不必要的文本
- 尾递归的优化
- 在模块之间优化

因此，可以在不需要明确地考虑 Renesas Technology SuperH RISC engine 家族的体系结构的情形下编程。

1.3 安装方法

1.3.1 PC 版本

用于 Windows 98/Me/2000/XP/NT 的 SuperH RISC engine C/C++ 编译程序的操作环境，以及在 Windows 98/Me/2000/XP/NT 上的安装步骤说明如下。

(1) 操作环境

- 主机计算机： IBM-PC 兼容型
- （可以运行 Windows 98/Me/2000/XP/NT 的 CPU）
- OS： Windows 98/Me/2000/XP/NT（日文或英文）
- 存储器大小：最低 128 MB、建议 256 MB 或以上
- 硬盘空间：需要 120 MB 或以上的磁盘空间（供完全安装）
- 显示：SVGA 或以上
- I/O 设备：CD-ROM 驱动器
- 其他：鼠标或其他指针设备

(2) 安装方法

要在 PC 上安装集成开发环境，请在 [控制面板] 中的 [添加/删除程序 applet] 中单击“设置”按钮，然后按照画面上的说明执行。

(3) 从 DOS 提示使用编译程序

在 Windows 下从 DOS 命令行使用编译程序时，必须设定某些环境变量。

环境变量的说明

(a) 环境变量 SHC_LIB

表示保存 SuperH RISC engine C/C++ 编译程序之主要文件的地点。

(b) 环境变量 SHC_TMP

指定创建 C/C++ 编译程序使用的临时文件的路径。 此设定不能省略。

(c) 环境变量 SHC_INC

此环境变量在从指定路径读取 C/C++ 编译程序的标准标题文件时设定。 可在路径之间使用逗号（“,”）隔开来指定多个路径。 如果不设定此环境变量，标准标题文件将会从 SHC_LIB 读取。

首先，使用下列内容创建批文件，这是从 DOS 提示开始的必要步骤。 如果此批文件已经存在，应该添加下列项目。下述实例用于将集成环境安装到硬盘驱动器 C。

要设定路径，请在 MS-DOS 提示时先使用“SET”命令确定当前的路径设定，然后在必要时添加。

下列是批文件的语法实例。

```
PATH C:\Hew3\Tools\Renesas\Sh\9_0_0\bin; %PATH%

SET SHC_LIB=C:\Hew3\Tools\Renesas\Sh\9_0_0\bin

SET SHC_TMP=C:\tmp

SET SHC_INC=C:\Hew3\Tools\Renesas\Sh\9_0_0\include
```

接着，输入下列批文件的路径，作为 DOS 提示属性对话框中 [程序(Program)] 标签上的“批文件”(Batch file) 路径。

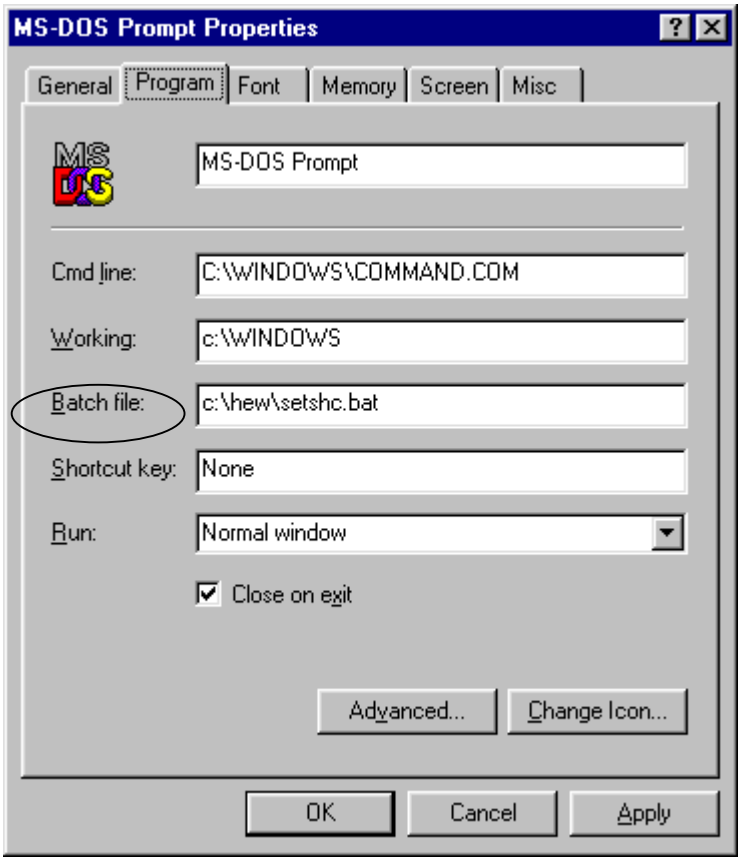


图 1.1 “MS-DOS 提示属性 (1)”(MS-DOS Prompt Properties (1))

完成以上设定后，重新开始 MS-DOS 提示会话。

注意： 如果在运行批文件时，显示“不足空间供环境变量使用”（Insufficient space for environment variables）信息，请进行下列更改。

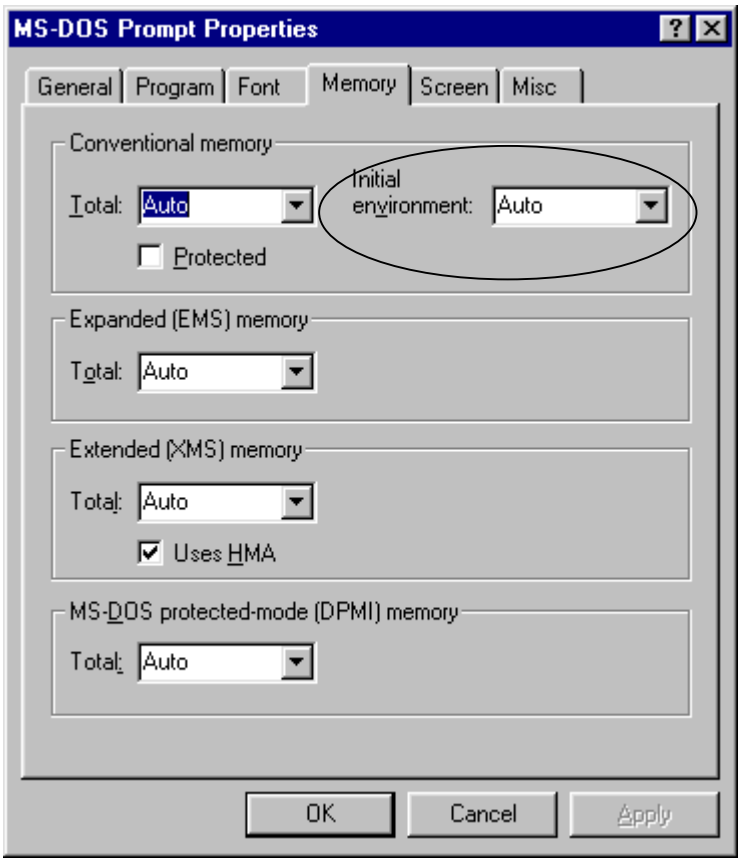


图 1.2 “MS-DOS 提示属性 (2)”(MS-DOS Prompt Properties (2))

在 DOS 提示属性对话框的 [存储器(Memory)] 标签上，将“初始环境”（Initial environment）从“自动”（Automatic）更改为 1024。

更改此设定后，MS-DOS 提示会话必须重新开始。

1.3.2 UNIX 版本

在 UNIX 系统上安装 C/C++ 编译程序的步骤如下所述：

注意： 不要在安装目录的名称中使用日文字符或空格。

(1) 安装媒介

编译程序以单张 CD-ROM 的形式分配。

(2) 安装方法

请使用下列步骤以安装编译程序。 在说明中出现 (RET) 的地方，请按下回车 (Enter) 键。

(a) 安装编译程序/模拟程序

安装编译程序/模拟程序的步骤如下：

(i) 为编译程序/模拟程序创建路径

使用任何任意名称，创建一个用于存储编译程序文件的路径。

% mkdirΔ<编译程序和模拟程序路径名称> (RET)

(ii) 安装 CD-ROM

如下所示安装 CD-ROM。 如果安装自动执行，则不需要执行下列命令。

[在 Solaris]

% mountΔrΔFΔhsfsΔdev/dsk/c0t6d0s2Δ/cdrom (RET)

[在 HP-UX]

% mountΔ/dev/dsk/c201d2s0Δ/cdrom (RET)

(iii) 复制编译程序/模拟程序

移到新建的路径，然后从 CD-ROM 将 SuperH RISC engine C/C++ 编译程序/模拟程序的文件解压缩至上述步骤（i）所创建的路径。

[在 Solaris]

% cdΔ<compiler and simulator pathname> (RET)

% tarΔ-xvfΔ/cdrom/sh c sim pack sparc/Program.tar (RET)

[在 HP-UX]

% cdΔ<compiler and simulator pathname> (RET)

% tarΔ-xvfΔ/cdrom/Program.tar (RET)

(iv) 更改环境设定

如下所述设定环境变量与路径名称。（两个星号 ** 表示应该指定的适当值。） 要获取有关环境变量的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)。

% setenvΔSHC_LIBΔ<compiler and simulator pathname> (RET)

% setenvΔSHC_INCΔ<compiler and simulator pathname> (RET)

% setenvΔSHC_TMPΔusr/tmp (RET)

% setenvΔSHCCPUΔSH (RET)**

% setenvΔHLNK_TMPΔ/usr/tmp (RET)

% setenvΔHLNK_LIBRARY1Δ<compiler and simulator pathname>/****.lib (RET)**

% setenvΔHLNK_LIBRARY2Δ<compiler and simulator pathname>/****.lib (RET)**

(v) 卸装 CD-ROM

% umountΔ/cdrom (RET)

(b) 安装模拟程序

安装比版本 8 更早的 UNIX 模拟程序版本的步骤如下所述。

(i) 安装 CD-ROM

请参考 README.TXT 文件以了解安装 CD-ROM 的步骤。

使用 README.TXT 文件中所述的方法复制交叉软件时，请移到要将它复制的目录，然后按照 (iii)“启动安装程序”中的说明执行。

(ii) 从 CD-ROM 上的 tar file 装入安装程序

（下列实例假设 CD-ROM 驱动器设备名称为 /cdrom。）

tarΔxvfΔcdrom/program.tarΔcas_install (RET)

(iii) 启动安装程序

cas install (RET)

(iv) 安装程序启动后立即显示

下列信息会在安装程序启动后立即显示。

开始安装循环准确模拟程序。 根据信息内容输入参数。

(v) 选择 CPU

选择要使用的模拟程序 CPU。（在此实例中，“10”选择“SH4 CPU”。）

如果选择 CPU SH4 或 SH2DSP，请按照 (vi)“选择联合验证工具”中的说明执行。

如果选择 SH4 或 SH2DSP 以外的 CPU，请按照 (vii)“输入用于安装定义文件的目录名称”中的说明执行。

```
Target
CPU(1:SH1,2:SH2,3:SH3,4:SH3E,5:SHDSP,6:SH2E,7:SH4BSC,8:SH3DSP,9:SHDSPC,10:SH4,11:SH2DSP): 10
```

(vi) 选择联合验证工具

如果在步骤 (v) 中选择了 CPU SH4 或 SH2DSP，请选择要使用的模拟程序联合验证工具的名称。（在此实例中，“1”选择“Seamless”。）

不使用联合验证工具时，请选择“否 (No)”。

如果在步骤 (v) 中选择了 CPU SH2DSP，将不能选取“Eaglei”。

```
SH-4:Please select Co-Velification Tool(1:Seamless,2:Eaglei,3:No): 1
```

(vii) 输入用于安装定义文件的目录名称

输入用于安装定义文件的目录名称。 括号 ( ) 内的目录为默认目录。 默认目录根据下列规则生成。

<当前目录> + "/df_CSDSH"

如果默认目录名称可接受，只需输入 (RET)。 可将目录名称输入为绝对路径或相对路径。 （在此实例中，输入了 (RET)。）

```
Directory name for the definition files(/export/home1/cas/cassh3sim/df_CSDSH): (RET)
```

(viii) 输入具有 CD-ROM 驱动器的宿主机名称

输入具有 CD-ROM 驱动器的宿主机名称。 启动主机的名称为默认名称。 从启动主机的 CD-ROM 驱动器安装时，请按下 (RET)。

从网络上其他主机的 CD-ROM 驱动器安装时，请输入该主机的名称。 但是，在此情形下，将假设可以从远程外壳登录（/etc/hosts.equiv 和 $HOME/.rhosts 文件已设定）。 要获取有关远程外壳的环境设定和相关事项的详细资料，请参考该启动机器的手册。

从启动主机的 CD-ROM 驱动器安装时，请按照 (x) “输入 tar file 名称” 中的说明执行。

从网络上其他主机的 CD-ROM 驱动器安装时，请按照 (ix) “输入具有 CD-ROM 驱动器的宿主机的登录名称” 中的说明执行。 （在下列实例中，具有 CD-ROM 驱动器的主机名称是 sp3。）

```
Host name connected to a tape driver(sparc2): sp3 (RET)
```

(ix) 输入具有 CD-ROM 驱动器的宿主机的登录名称

输入具有 CD-ROM 驱动器的宿主机的登录名称。 从其他主机的 CD-ROM 驱动器安装时，将会显示此信息。（在此实例中，具有 CD-ROM 驱动器的宿主机的登录名称是 “远程”（remote）。）

```
Login name of host connected to a tape driver:remote (RET)
```

(x) 输入 tar file 名称

输入 tar file 名称。 HP9000 的默认名称是 /dev/rmt/0m，而 SPARC 的默认名称是 /dev/rmt/0。 （以下实例假设 CD-ROM 驱动器设备名称为 /cdrom。）

```
Tape driver name(/dev/rmt/0):  /cdrom/simulator.tar (RET)
```

(xi) 安装定义文件前输入 (RET)

确定具有定义文件的 CD-ROM 已安装到 CD-ROM 驱动器，然后输入 (RET)。

```
Input return, after setting the tape including the definition files to the tape driver.
(RET)
```

(xii) 选择是否安装主要文件

选择是否要安装界面软件的主要文件。

要安装主要文件请输入 “y” （是），否则请输入 “n” （否）。

如果输入“n”，请按照 (xv) “选择是否安装设置文件”中的说明执行。

（在此实例中，输入了“y”。）

```
Do you install the main files?(y/n):y (RET)
```

(xiii) 输入主要文件的目录名称

输入用于安装界面软件主要文件的目录名称。

默认目录根据下列规则生成。

<当前目录> + "/main"

可将目录名称输入为绝对路径或相对路径。

（在下列目录中，输入了 (RET)。）

```
Directory name for the main files(/export/home1/cas/cassh3sim/main): (RET)
```

(xiv) 安装主要文件前输入 (RET)

确定具有界面软件主要文件的 CD-ROM 已安装到 CD-ROM 驱动器，然后输入 (RET)。

```
Input return,after setting the tape including the main files to the tape driver. (RET)
```

(xv) 选择是否安装设置文件

选择是否要复制设置样品文件。 要复制设置文件请输入“y”（是），否则请输入“n”（否）。

（在此实例中，输入了“y”；之后将会显示安装文件名。）

```
Do you copy the setup files to current directory?(y/n): y (RET)
```

(xvi) 选择路径和环境变量设定

选择是否要将路径和环境变量设定添加到外壳脚本。

如果输入“y”，安装程序将从“SHELL”环境变量确定登录外壳类型，在“HOME”环境变量所指定的目录下备份任意外壳脚本文件（请参考表 1.1），然后设定路径和环境变量信息。

设定符合下列规格。

如果尚未安装主要文件（请参考 (xii) 选择是否安装主要文件），路径设定将不会改变。

如果在 (vii) “输入用于安装定义文件的目录名称”或 (xiii) “输入主要文件的目录名称”步骤中指定了相对路径，路径和环境变量信息将使用输入路径信息设定，而且除了启动安装程序的目录外，不能在其他目录中执行。

如果输入“n”，安装程序将移至 (xviii) “安装完成信息”步骤，且安装程序将会终止。

表 1.1 用于不同外壳的文件名

编号	外壳名称	脚本文件名	备份文件名
1	Bourne 外壳 (sh)	.profile	.profile.bak
2	C 外壳 (csh)	.cshrc	.cshrc.bak
3	Korn 外壳 (ksh)	.profile	.profile.bak

（在此实例中，输入了“y”；之后将会显示外壳脚本文件名。）

```
Do you append the path list and the environment variables in shell script?(y/n): y(RET)

/export/home1/cas/.cshrc
```

(xvii) 选择是否覆盖备份文件

备份外壳脚本时，如果存在与备份文件之名称相同的文件时，将会显示此信息。 选择是否要覆盖该文件。

（在此实例中，登录外壳是 C 外壳。）

```
Do you overwrite the backup file(.cshrc.bak)?(y/n): y(RET)
```

(xviii) 安装完成信息

完成安装的所有步骤时，下列信息将会显示且安装程序将会终止。

```
Installation of the cycle-accurate simulator completed.
```

(c) 安装 Acrobat® Reader

手册可从 Windows 中查看。 用来查看手册的软件 (Acrobat® Reader) 必须安装在运行 Windows 98/Me/2000/XP/NT 的计算机上。

Acrobat® Reader 版权所有 © 1987-2001 Adobe Systems Incorporated。 保留所有权利。

Adobe 和 Acrobat 是 Adobe Systems 在特定的管辖区内注册的商标。

执行安装使用下列步骤。 在开始安装步骤前，确保已关闭所有应用程序。

(i) 将随附的 CD-ROM 放入 CD-ROM 驱动器。（这里将 CD-ROM 驱动器假设为驱动器 D。）

(ii) 从 Windows® 的“开始”菜单，单击 [运行…]。

(iii) 在 [运行…] 对话框中指定 CD-ROM 上 [PDF_READ\Japanese] 目录中的 Acrobreader51_jpn.exe（日文）或 [PDF_read\English] 目录中的 Acrobreader51_eng.exe（英文）（例如：D:\PDF_Read\Japanese\Acrobreader51_jpn.exe），然后单击 [确定(OK)]。

(iv) 按照画面上的安装说明执行。

1.4 执行方法

1.4.1 启动嵌入式工作区

完成安装后，“嵌入式工作区”(Embedded Workshop) 的安装程序将在 Windows“开始”菜单的“程序”文件夹中创建一个名为“嵌入式工作区 2”(Embedded Workshop 2) 的文件夹，并在这个文件夹内放置“嵌入式工作区”和其他文件的可执行程序的快捷方式。

“开始”菜单的内容将取决于所安装的工具而有所不同。

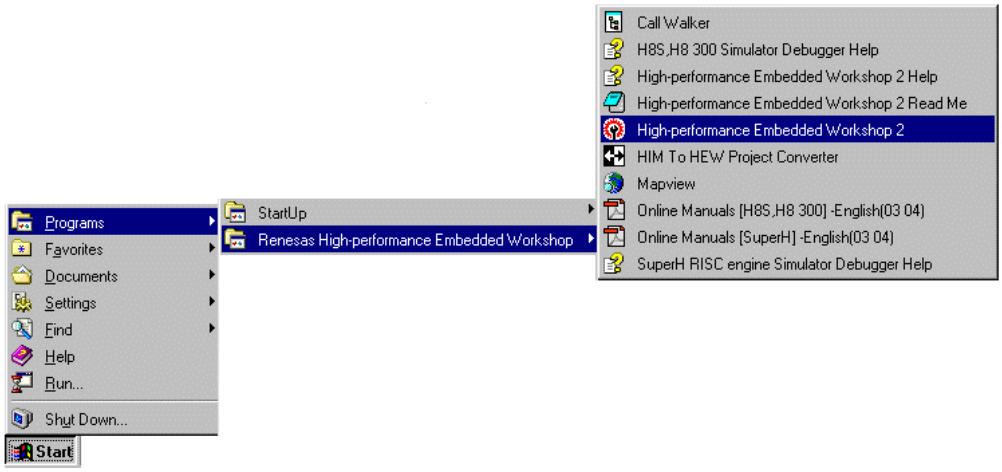


图 1.3 从“开始”菜单启动“嵌入式工作区”

在“开始”菜单中单击“嵌入式工作区”(Embedded Workshop) 时，将会显示一则启动信息，然后显示“欢迎！”(Welcome!) 对话框（图 1.4）。

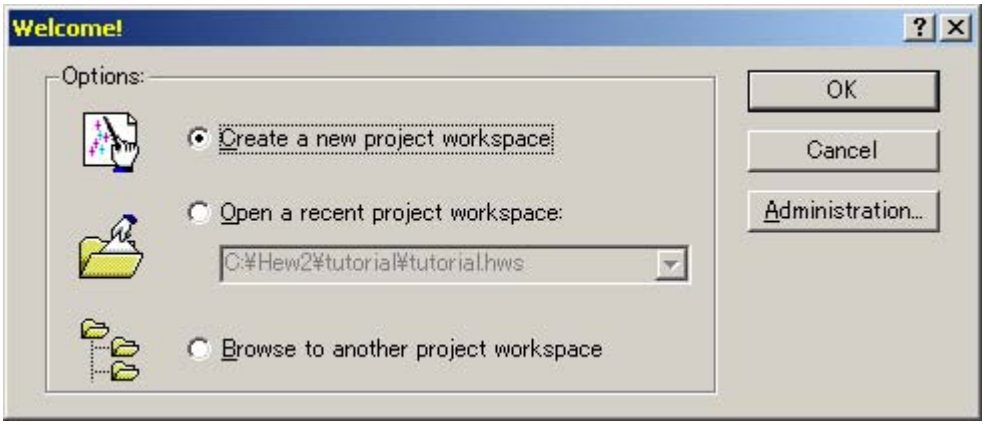


图 1.4 “欢迎！”(Welcome!) 对话框

如果是第一次使用“嵌入式工作区”，或是开始进行新的工程时，请选取 [创建新的工程 (Create a New Project)]，然后单击 [确定 (OK)]。 若要在已创建的工程中继续工作，请选取 [打开现有工程 (Open an Existing Project)] 或 [浏览至另一个工程工作空间 (Browse to another project workspace)]，然后单击 [确定 (OK)]。 不论选取哪一个选项，单击 [退出(Exit)] 将会终止“嵌入式工作区”。 通过单击 [管理... (Administration...)]，可以注册和删除与“嵌入式工作区”一起使用的系统工具。

1.4.2 启动编译程序

在本小节内，将举例说明执行 SuperH RISC engine C/C++ 编译程序的方法。 要获取有关编译程序选项的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)。 使用 PC 版本时，请参考《操作手册》。

表 1.2 编译条件表

命令	选项	编译文件的扩展名	编译条件
shcpp	任何	任何	C++-编译
shc	-lang=c	任何	C-编译
	-lang=cpp		C++-编译
	未指定 lang 选项	*.c	C-编译
		.cpp、.cc、*.cp、*.CC	C++-编译

命令 shc C-编译* 或 C++-编译 C 程序和 C++ 程序，将分别根据 lang 选项或程序文件名的执行。 命令 shcpp C++-编译所有程序，不论它们是 C 程序或 C++ 程序。 编译条件在表 1.2 中描述。

注意： *C-编译是指根据 C 语言的语法来编译程序；C++-编译则指根据 C++ 语言的语法来编译程序。

下面说明使用编译程序的基本步骤。

(1) 程序编译

要编译“test.c”C 源程序：

shcΔtest.c (RET)

要编译“test.cpp”C++ 源程序：

shcΔtest.cpp(RET)

shcppΔtest.cpp(RET)

(2) 显示命令输入格式和编译程序选项

本命令在标准输出画面上显示命令输入格式以及编译程序选项的列表。

shc (RET)

shcpp(RET)

(3) 指定选项

选项（调试 (debug)、列表文件 (listfile)、显示 (show)，等等）以一个连字号 (-) 为前缀，多个选项由空格（Δ）区隔。在 PC 版本中，可在 DOS 提示时使用正斜杠 (/) 来取代连字号。

当指定多个子选项时，必须以逗号（，）区隔。

shcΔ-debugΔ-listfileΔ-show=noobject,expansionΔtest.c(RET)

在 PC 版本中，也可以使用圆括号将子选项括起来。

shcΔ/debugΔ/listfileΔ/show=(noobject,expansion)Δtest.c(RET)

(4) 编译多个 C/C++ 程序

多个 C/C++ 程序可同时被编译。 以下是用于编译 C 源程序的命令实例。

实例 1： 指定要编译的多个程序

shcΔtest1.cΔtest2.c (RET)

实例 2： 指定选项（选项指定给所有 C 源程序）

shcΔ-listfileΔtest1.cΔtest2.c (RET)

“列表文件”(listfile) 选项对 test1.c 和 test2.c 都有效。

实例 3： 指定选项（选项个别指定给每个程序）

shcΔtest1.cΔtest2.cΔ-listfile(RET)

在这里，“列表文件”选项只对 test2.c 有效。为个别程序指定选项先于为所有源程序指定选项。

注意：

(1) 如果在安装后无法运行编译程序，请检查下列事项。

- 确定 PATH 环境变量包含含有 C/C++ 编译程序的目录。
- 确定 SHC_LIB 环境变量设定为含有主要 C/C++ 编译程序文件的目录。

SHC_LIB 环境变量用于设定含有主要 C/C++ 编译程序文件的目录。 因此，如果没有将整套 C/C++ 编译程序文件放到相同的目录中，编译程序将无法运行。

(2) 编译程序会在编译时根据所使用的是 shc 或 shcpp 命令，决定要使用的语法；但是即使使用的是 shc 命令，它将取决于文件扩展名和选项来执行 C++-编译。

1.5 程序开发的步骤

图 1.5 显示开发一个 C/C++ 语言程序所使用的步骤。 阴影部分显示作为 SuperH RISC engine C/C++ 编译程序封装提供的软件。

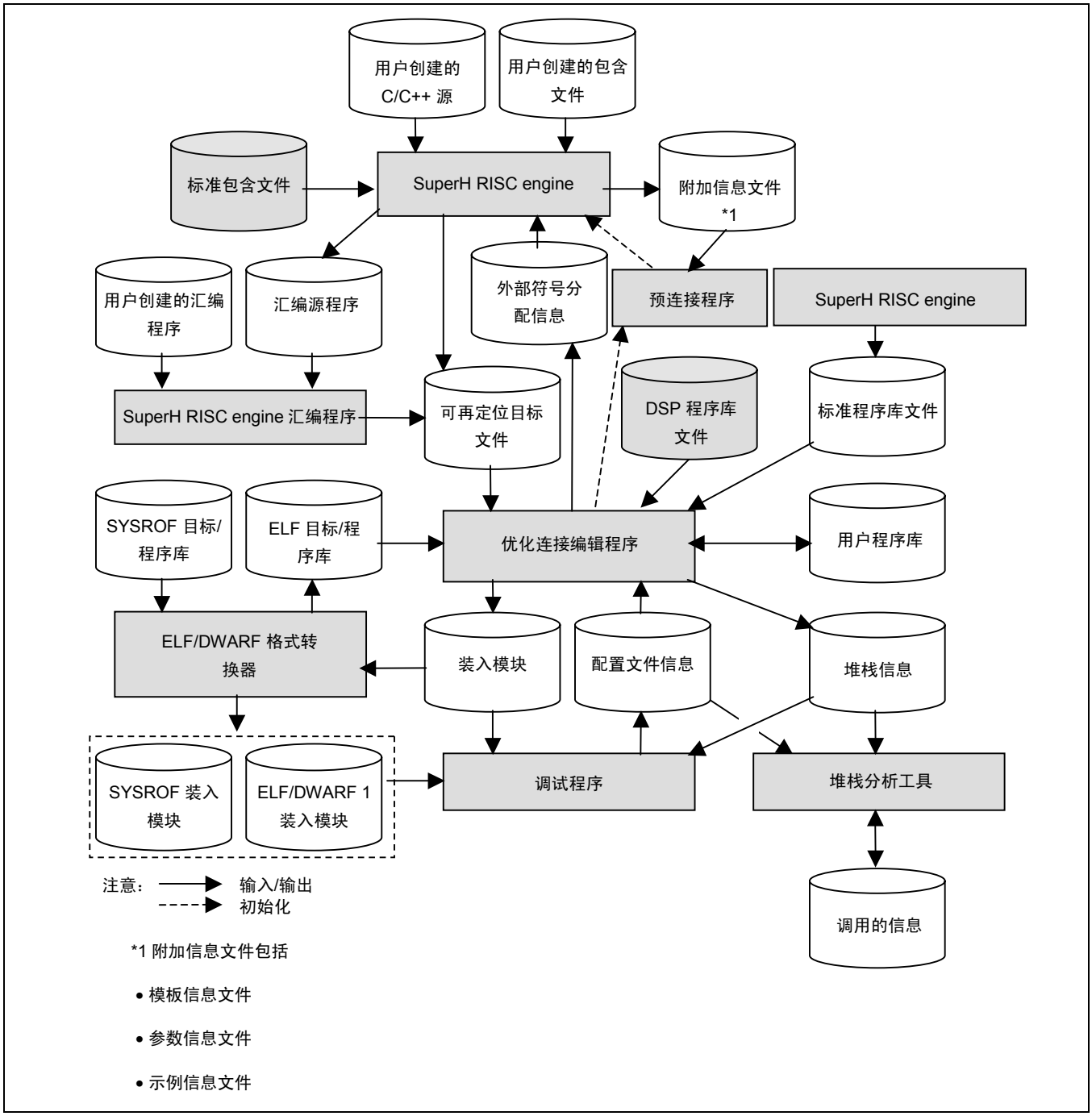


图 1.5 程序开发的步骤

以下说明源文件实例 on_motor.c 的程序开发步骤。要获取有关使用交叉软件的详细资料，请参考交叉软件封装的用户手册。

(1) 创建源文件

使用编辑程序创建源文件。

(2) 生成可再定位目标文件

启动编译程序，然后编译源文件。

shcΔon_motor.c (RET)

将会生成已经优化但不含调试信息的可再定位目标文件（称为“on_motor.obj”）。 要生成列表文件，请指定“列表文件”选项。

(3) 生成装入模块文件

在包括程序库文件 sensor.lib 和启动连接编辑程序时，将会生成具有 on_motor.abs 名称的可执行装入模块文件。

optlnkΔ-nooptΔ-subcommand = link.sub (RET)

lnk.sub 的内容如下。

```
Sdebug
input on_motor
library sensor.lib
Exit
```

即使可再定位目标文件包含调试信息，但如果在连接时省略调试选项，调试信息将不会输出到装入模块文件。

(4) S 类型格式文件输出

为了能够使用 ROM 编程器写入 EPROM，lnk.sub 应该编码如下。

```
Form=stype
Sdebug
input on_motor
library sensor.lib
Exit
```

将生成具有 on_motor.mot 名称的 S 类型格式装入模块文件。

SuperH RISC Engine C/C++编译程序应用笔记

创建和调试程序的步骤

第 2 节　创建和调试程序的步骤

2.1　创建工程

2.1.1　创建模拟程序调试程序的工程

(1) 指定工程

当您在 [欢迎！(Welcome!)] 对话框中选取 [创建新的工程工作空间 (Create a new project workspace)] 单选按钮和单击 [确定 (OK)] 后，用于创建新的工作空间和工程的 [新的工程工作空间 (New Project Workspace)] 对话框（图 2.1）将会启动。您可以在这个对话框中指定工作空间名称（创建新的工作空间时，工程名称在默认情形下将会一样）、CPU 家族和工程类型等等。

例如，当您在 [工作空间名称 (Workspace Name)] 字段中输入“tutorial”，那么 [工程名称 (Project Name)] 字段将显示为“tutorial”且 [目录 (Directory)] 字段也将显示为“c:\hew2\tutorial”。 如果您要更改工程名称，请在 [工程名称 (Project Name)] 字段中人工输入一个新的工程名称。 如果您要更改用作新的工作空间的目录，可单击 [浏览… (Browse…)] 按钮指定一个目录，或直接在 [目录 (Directory)] 字段中人工输入目录路径。

在这里，指定 [演示 (Demonstration)] 为左边的 [工程 (Project)] 类型。

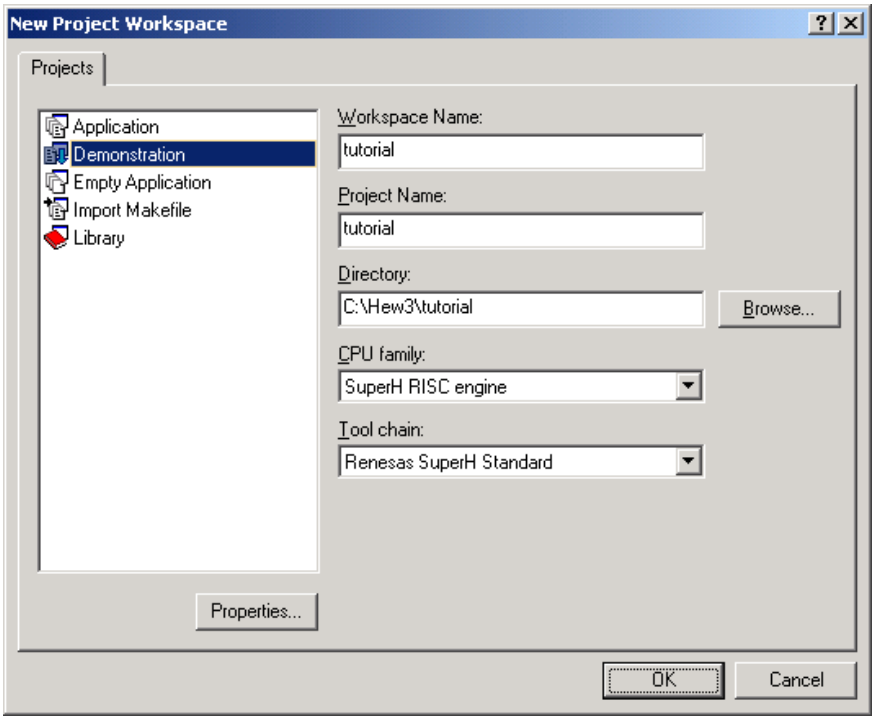


图 2.1 “新的工程工作空间”(New Project Workspace) 对话框

(2) 选择目标 CPU

当您在 [新的工程工作空间 (New Project Workspace)] 对话框中单击 [确定 (OK)] 时，工程生成程序将会调用。首先选择您要使用的 CPU。 [CPU 类型 (CPU Type)] 列表中显示的 CPU 类型将会分类到 [CPU 系列 (CPU Series)] 列表中显示的 CPU 系列。在 [CPU 系列：(CPU Series:)] 列表框和 [CPU 类型：(CPU Type:)] 列表框 中选定的项目指定将生成的文件。 选择要开发的程序的 CPU 类型。 如果您要选择的 CPU 类型没有在 [CPU 类型：(CPU Type:)] 列表中显示， 请选择具有相似硬件规格的 CPU 类型，或选择 [其他 (Other)]。

- 单击 [下一步> (Next>)] 移到下一个画面。
- 单击 [<上一步 (<Back)] 移到前一个画面或对话框。
- 单击 [完成 (Finish)] 以打开 [摘要 (Summary)] 对话框。
- 单击 [取消 (Cancel)] 以返回显示 [新的工程工作空间 (New Project Workspace)] 对话框。

[<上一步 (<Back)] 、[下一步> (Next>)] 、[完成 (Finish)] 和 [取消 (Cancel)] 是所有向导对话框中常用的按钮。

在此教程中，在 [CPU 系列 (CPU Series)] 列表中选择 [SH-1]（图 2.2）。 然后单击 [下一步> (Next>)]。

如果您选择了“demonstration”（演示），您将不能选取“CPU 类型 (CPU Type)”。

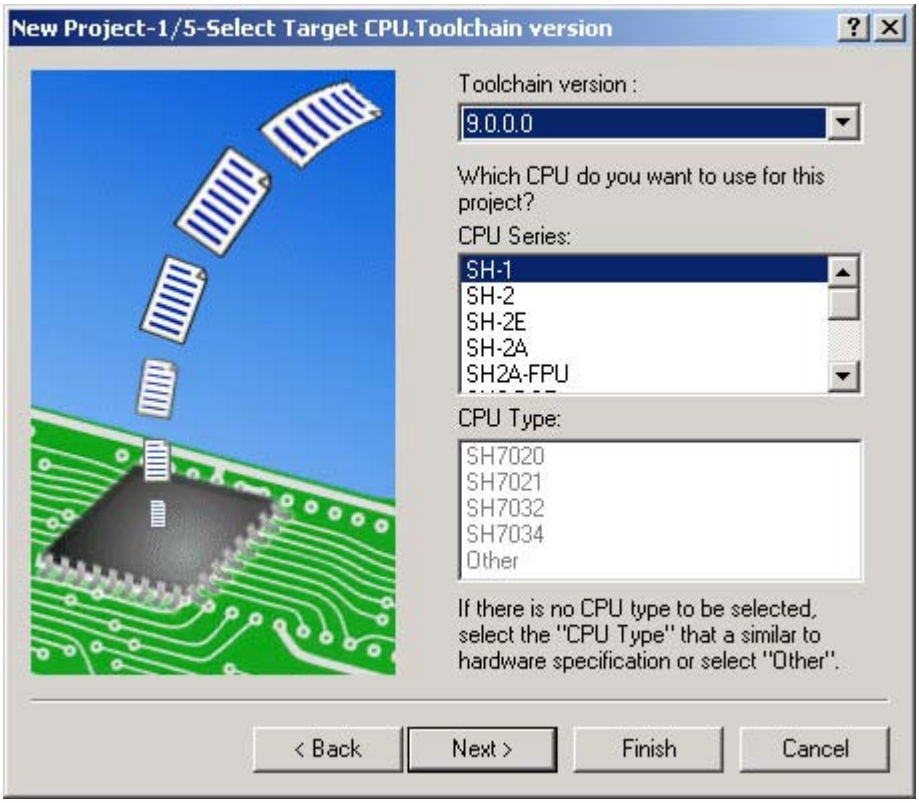


图 2.2 “新的工程步骤 1”(New Project Step 1) 对话框

(3) 选项设定

在“步骤 1”(Step-1) 画面上单击 [下一步> (Next>)] 按钮将打开图 2.3 中显示的对话框。 在这个画面上，您可以指定所有工程文件的普遍选项。 这些选项的设定会对应步骤 1 画面中所选取的 CPU 系列而修改。 要在创建工程后更改选项设定，请从 HEW 窗口的 [选项 (Options)->SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain)] 菜单项选择 CPU 标签。

单击 [下一步> (Next>)] 按钮而不更改设定。“步骤 3”（Step-3）画面将会显示。

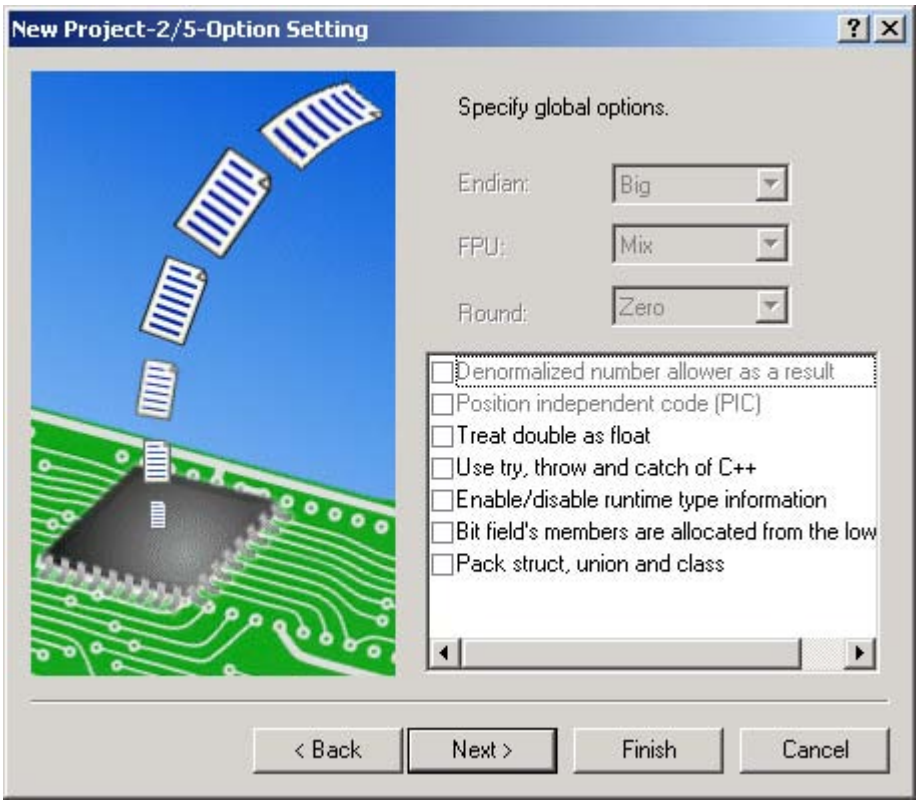


图 2.3 “新的工程步骤 2”(New Project Step 2) 对话框

(4) 设定调试的目标系统

在“步骤 2”(Step-2) 画面上单击 [下一步> (Next>)] 按钮时，图 2.4 中显示的画面将会出现。 此画面用于指定调试的目标系统。 从 [目标：(Target:)] 下的列表选取（核选）调试的目标。 可以不选择目标或选择多个目标。

在此教程中，选择 [SH-1 模拟程序 (SH-1 Simulator)] 然后单击 [下一步> (Next>)] 按钮。

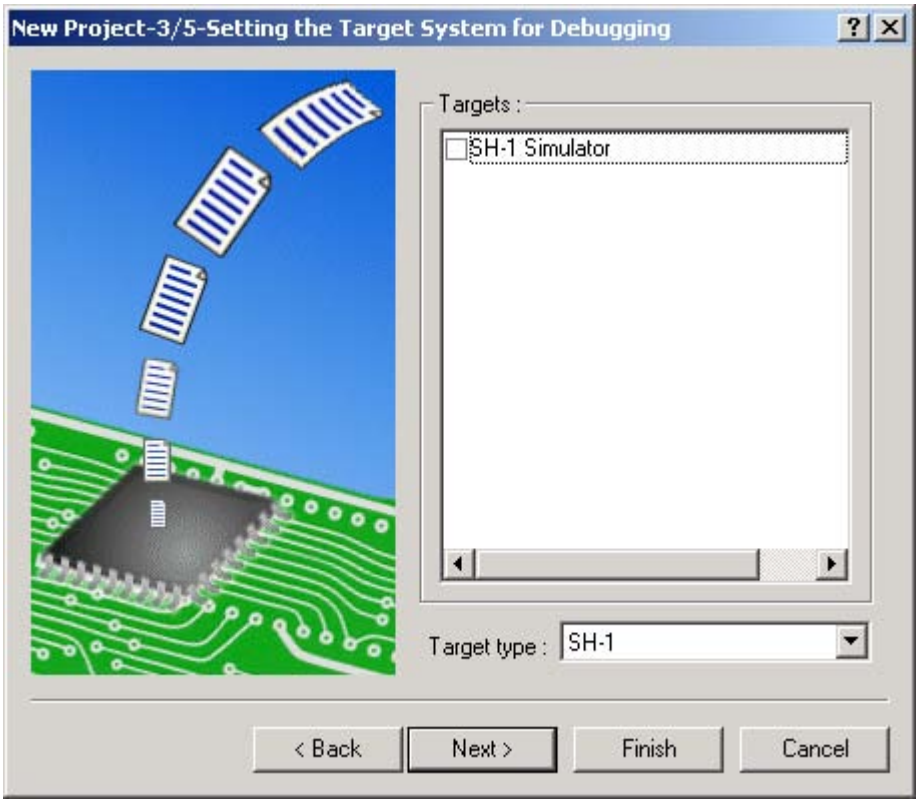


图 2.4 “新的工程步骤 3”(New Project Step 3) 对话框

(5) 设定调试程序选项

在“步骤 3”(Step-3) 画面上单击 [下一步> (Next>)] 按钮时，图 2.5 中显示的画面将会出现。 此画面用于指定所选取的调试目标的可选设定。

在默认情况下，HEW 将创建两个配置，[发布 (Release)] 和 [调试 (Debug)] 。 选取了用于调试的目标后，HEW 将创建另一个配置（包含目标的名称）。

配置的名称可在 [配置名称：(Configuration name:)] 中修改。调试目标的选项将显示在 [资料选项：(Detail options:)] 之下。要更改设定，请选取 [项目 (Item)] 然后单击 [修改 (Modify)]。 当无法修改的项目被选取时，虽选取了 [项目 (Item)] 但 [修改 (Modify)] 将保持灰色显示。

在此教程中，单击 [下一步> (Next>)] 按钮而不更改设定。

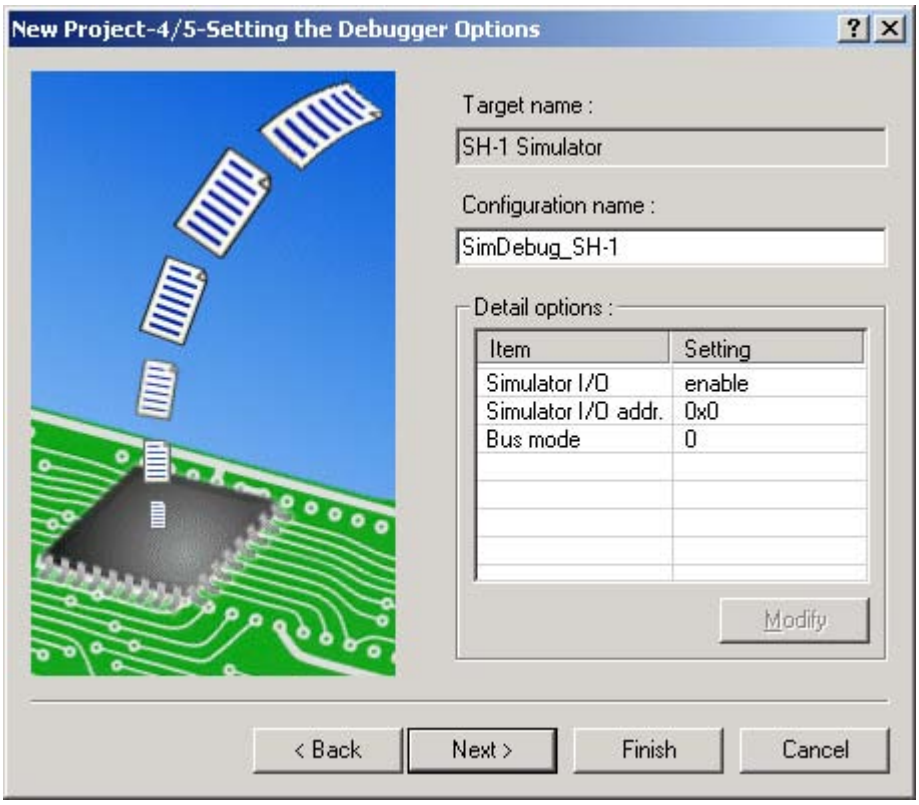


图 2.5 “新的工程步骤 4”(New Project Step 4) 对话框

(6) 确认设定（“摘要”对话框）

在“步骤 4”(Step-4) 画面上单击 [下一步> (Next>)] 按钮将显示图 2.6 中显示的画面。 在这个画面上，显示所要创建的工程的源文件信息。 确认后，单击 [完成 (Finish)]。

在“步骤 5”(Step-5) 画面上单击 [完成 (Finish)]，工程生成程序将会在 [摘要 (Summary)] 对话框中显示所生成文件的列表（图 2.7）。 确认该对话框的内容然后单击 [确定 (OK)]。

核选 [在工程目录内生成 Readme.txt 为摘要文件 (Generate Readme.txt as a summary file in the project directory)] 复选框时，[摘要 (Summary)] 对话框中所显示的工程信息将会以“Readme.txt”的文本文件名存储在工程目录内。

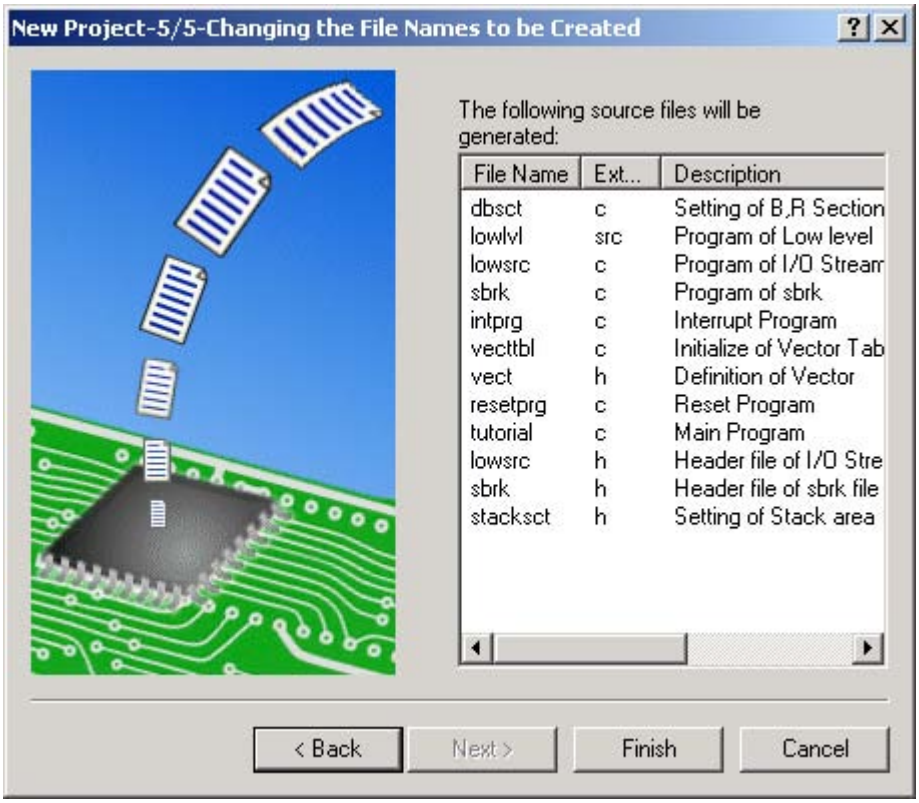


图 2.6 “新的工程步骤 5”(New Project Step 5) 对话框

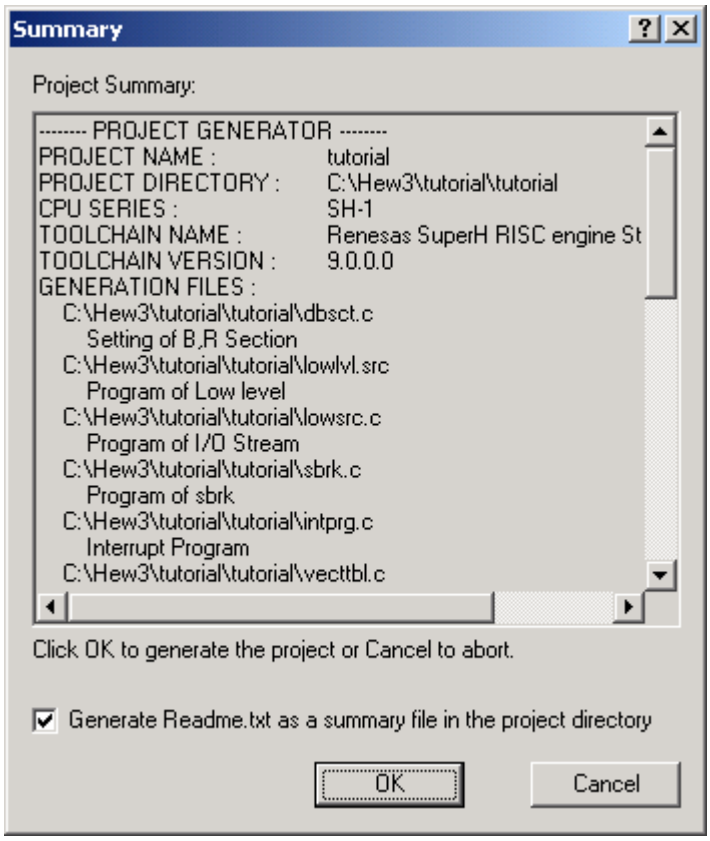


图 2.7 “摘要” (Summary) 对话框

(7) 其他

如果从“工程类型”(Project Type) 选取了“演示”(demonstration)，在模拟程序调试时可使用的低层程序库样品将会被嵌入。 将会嵌入的文件如下：

- lowlvl.src（标准 I/O 样品汇编程序列表）
- lowsrc.c（低层程序库源文件）
- lowsrc.h（低层程序库标题文件）

2.2 样品程序简介 （SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP）

在本小节中，使用具有 2.8 图中所示结构的样品程序来说明创建程序的实际步骤。 开发环境在表 2.1 中描述。

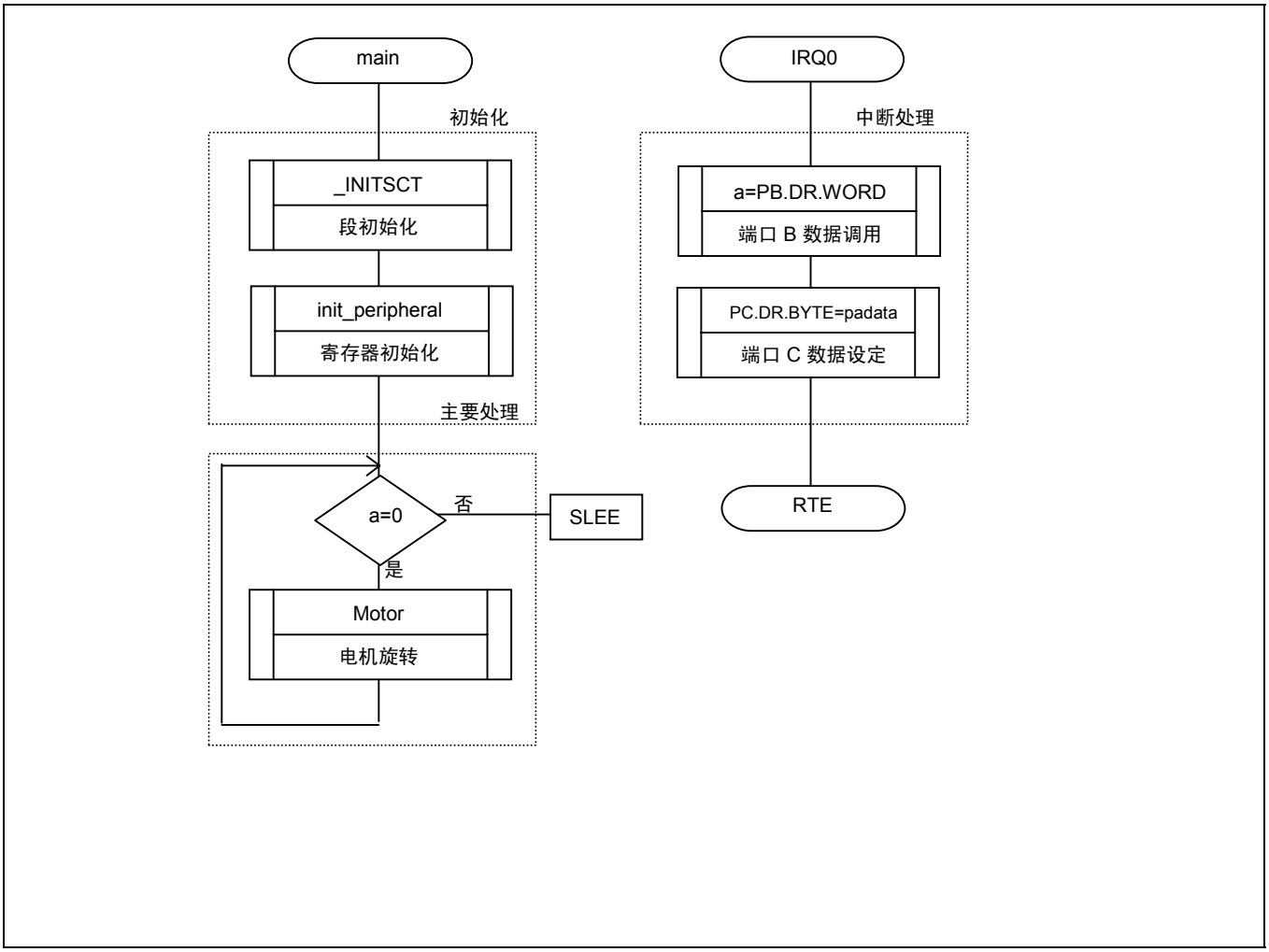


图 2.8 样品程序结构

表 2.1 样品程序开发环境

OS	UNIX
CPU	SH-1

2.2.1 创建向量表

向量表创建程序在图 2.9 中显示。 要获取创建向量表的详细资料，请参考第 3.1.3 节“创建向量表”。

图 2.9 显示与图 2.10 中以汇编语言编写的相同程序。

```
/************************************************************/
/*                          文件名 "vect.c"                 */
/************************************************************/
       extern void main(void);
       extern void inv_inst(void);
       extern void IRQ0(void);

   void (* const vec_table[])(void)={
          main,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          inv_inst, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          IRQ0
       };
```

图 2.9 向量表创建程序（C 语言版本）

SH-1 的向量表在表 2.2 中显示。

加电复位时，函数 main 将会启动。 此时堆栈指针 将会设定为 0。

函数 inv_inst 的起始地址将会设定为向量号 32，而函数 IRQ0 的起始地址将会设定为向量号 64。这些向量号分别为用户向量和外部中断的起始向量号。

表 2.2 异常处理向量表

异常源	向量号	向量表地址偏移	
加电复位	PC	0	H'00000000 至 H'00000003
	SP	1	H'00000004 至 H'00000007
人工复位	PC	2	H'00000008 至 H'0000000B
	SP	3	H'0000000C 至 H'0000000F
:		:	:
陷阱指令（用户向量）		32	H'00000080 至 H'00000083
		:	:
		63	H'000000FC 至 H'000000FF
中断	IRQ0	64	H'00000100 至 H'00000103
	:	:	:

```
.SECTION VECT,DATA,ALIGN=4
.IMPORT      _main
.IMPORT      _inv_inst
.IMPORT      _IRQ0
.DATA.L      _main                ;_main 起始地址设定为向量号 0
.DATA.L      H'0000000             ;SP 起始地址设定为向量号 1
.ORG         H'0080
.DATA.L      _inv_inst            ;_inv_inst 起始地址设定为向量号 32
.ORG         H'0100
.DATA.L      _IRQ0                ;_IRQ0 起始地址设定为向量号 64
.END
```

图 2.10 向量表创建程序（汇编语言版本）

在汇编语言程序中，C 语言程序的外部名称的前面将会放置一个下划线“_”。

2.2.2 创建标题文件

图 2.11 显示所有样品程序普遍使用的标题文件。通过定义 IPRA 和其他 I/O 端口，即可如同变量般以名称存取 I/O 端口。

```
/************************************************************************/
/*                      文件名 "7032.h" （提取）                        */
/************************************************************************/
/************************************************************************/
/*                          定义 I/O 寄存器                             */
/************************************************************************/
struct st_intc { /* struct INTC      */
                 union {     /*  IPRA       */
                             unsigned short WORD;/*字存取          */
                 struct {    /*  位存取      */
                         unsigned char UU:4;        /*  IRQ0       */
                         unsigned char UL:4;        /*  IRQ1       */
                         unsigned char LU:4;        /*  IRQ2       */
                         unsigned char LL:4;        /*  IRQ3       */
                         } BIT;      /*             */
                  } IPRA;    /*             */
                 union {     /*  IPRB       */
                        unsigned short WORD;        /*  字存取      */
                        struct {     /*  位存取      */
                         unsigned char UU:4;        /*  IRQ4       */
                         unsigned char UL:4;        /*  IRQ5       */
                         unsigned char LU:4;        /*  IRQ6       */
                         unsigned char LL:4;        /*  IRQ7       */
                         } BIT;      /*             */
                  }  IPRB;   /*             */
}; /*     */
```

```
#define INTC  (*(volatile struct st_intc  *)0x5FFFF84)
/* INTC 地址  */
/************************************************************************/
/*                              定时器寄存器                            */
/************************************************************************/
struct st_itu0 { /* struct ITU0       */
                  union {     /* TCR  */
               unsigned char BYTE;    /* 字节存取       */
               struct { /* 位存取              */
                         unsigned char  :1; /*      */
                         unsigned char CCLR :2;     /*  CCLR        */
                         unsigned char CKEG :2;     /*  CKEG        */
                         unsigned char TPSC :3;     /*  TPSC        */
                       }   BIT;      /*      */
                  }    TCR;   /*      */
};  /*  */
 #define ITU0  (*(volatile struct st_itu0  *)0x5FFFF04)
   /* ITU0 地址       */
/************************************************************************/
/*                              PORT 寄存器                             */
/************************************************************************/
struct st_pa {   /*  struct PA        */
               union {  /*  PADR       */
                  unsigned short  WORD;       /*  字存取        */
                      struct {        /*  字节存取      */
                             unsigned char H;        /*  高   */
                             unsigned char L;        /*  低   */
                  }   BYTE;    /*      */
                             struct {     /*  位存取        */
                                  unsigned char B15 :1;   /*  位 15*/
                                  unsigned char B14 :1;   /*  位 14*/
                                  unsigned char B13 :1;   /*  位 13*/
                                  unsigned char B12 :1;   /*  位 12*/
                                  unsigned char B11 :1;   /*  位 11*/
                                  unsigned char B10 :1;   /*  位 10*/
                                  unsigned char B9  :1;   /*  位 9*/
                                  unsigned char B8  :1;   /*  位 8*/
                                  unsigned char B7  :1;   /*  位 7*/
                                  unsigned char B6  :1;   /*  位 6*/
                                  unsigned char B5  :1;   /*  位 5*/
                                  unsigned char B4  :1;   /*  位 4*/
                                  unsigned char B3  :1;   /*  位 3*/
                                  unsigned char B3  :1;   /*  位 2*/
                                  unsigned char B3  :1;   /*  位 1*/
                                  unsigned char B3  :1;   /*  位 0*/
```

```
                                    } BIT; /*      */
                            }    DR;    /*      */
};  /*  */
#define PB    (*(volatile struct st_pa  *)0x5FFFFC2)
    /*  PB  地址  */
struct st_pc {   /*  struct PC        */
             union { /*  PCDR */
                    unsigned char  BYTE;      /*  字节存取      */
                    struct {   /*  位存取       */
                                    unsigned char B7 :1;   /*  位 7        */
                                    unsigned char B6 :1;   /*  位 6        */
                                    unsigned char B5 :1;   /*  位 5        */
                                    unsigned char B4 :1;   /*  位 4        */
                                    unsigned char B3 :1;   /*  位 3        */
                                    unsigned char B2 :1;   /*  位 2        */
                                    unsigned char B1 :1;   /*  位 1        */
                                    unsigned char B0 :1;   /*  位 0        */
                            } BIT;    /*      */
                  }   DR;     /*      */
};  /*  */
#define PC  (*(volatile struct st_pc *)0x5FFFFD1)
    /* PC  地址    */
/************************************************************************/
/*                              文件名 "sample.h"                       */
/************************************************************************/
/************************************************************************/
/*                              定时器寄存器                            */
/************************************************************************/
 struct tcsr {   /* */
                 short OVF  :1;       /*TCSR struct OVF 位                */
                 short WTIT :1;       /* WTIT 位                          */
                 short   :3;  /* 工作区域                                */
                 short CKS2 :1;       /* CKS2 位                          */
                 short CKS1 :1;       /* CKS1 位                          */
                 short  :9;   /* 工作区域       */
};  /*  */
#define TCSR_FRT (*(volatile unsigned short *)0x5FFFFB8)
    /*  */
#define TCSR__FRT (*(volatile struct tcsr *)0x5FFFFB8)
    /*  */
     extern void motor( void );        /* motor 模块    */
     extern void _INITSCT( void );
    /* 段初始化模块                          */
     extern void init_peripheral(void);
    /* 外围初始化模块                       */
```

图 2.11 标题文件

2.2.3 创建 Main 处理程序

main 处理程序在图 2.12 中显示。此处，在加电复位时启动的函数 main 以及直到中断发生时持续调用的函数 motor 将会定义。

```
/*********************************************************************/
/*                              文件名 "sample.c"                    */
/*********************************************************************/
 #include "7032.h"
 #include "sample.h"
 #include <machine.h>          /* 定义嵌入式函数 sleep                */
const short padata=0x3;        /* C 段                                */
 short a=0;                    /* D 段                                */
 int work;                             /* B 段                                */
/*********************************************************************/
/*                              main 模块                            */
/*********************************************************************/
 void main( void )
 {
       _INITSCT();             /* 初始化每一个段                      */
        init_peripheral();
        while(!a) motor();
        sleep();
 }
/*********************************************************************/
/*                              motor 模块                           */
/*********************************************************************/
 void motor( void )            /* 调用直到中断发生                    */
 {
            :
            :
        return;
 }
```

图 2.12 Main 处理程序

在函数 main 中，_INITSCT 和 init_peripheral 将会调用以执行段初始化和内部寄存器初始化。 接着，程序将会等待全局变量 a 的值更改。在这段期间，函数 motor 将会持续被调用。如果 a 不为零，将会进入低功耗状态。

2.2.4 创建初始化元件

图 2.13 显示设定段初始化中使用的外部名称的值的汇编程序语言；
图 2.14 显示执行段初始化和寄存器初始化的 C 语言程序。

```
;*********************************************************************
;                              文件名 "sct.src"                       *
;*********************************************************************
                                    .SECTION B,DATA,ALIGN=4
                                    .SECTION R,DATA,ALIGN=4
                                    .SECTION D,DATA,ALIGN=4
          ; 要添加的所有段将会在此处列出

                                    .SECTION C,DATA,ALIGN=4
          __B_BGN:                  .DATA.L (STARTOF B)
          __B_END:                  .DATA.L (STARTOF B)+(SIZEOF B)
          __D_BGN:                  .DATA.L (STARTOF R)
          __D_END:                  .DATA.L (STARTOF R)+(SIZEOF R)
          __D_ROM:                  .DATA.L (STARTOF D)

                                    .EXPORT __B_BGN
                                    .EXPORT __B_END
                                    .EXPORT __D_BGN
                                    .EXPORT __D_END
                                    .EXPORT __D_ROM
                                    .END
```

图 2.13 初始化程序（汇编语言部分）

B 段和 D 段的起始与终止地址将会定义。

在编译时，如果没有使用“段”（section） 选项来指定段名称，C/C++ 编译程序将自动分配下列名称。

程序段：	P
常数段：	C
初始化数据段：	D
未初始化数据段：	B

R 段显示的 RAM 区域用来存放通过连接编辑程序的 ROM 支持函数从 ROM 上复制来的初始化数据区域。要获取有关连接编辑程序的 ROM 支持函数的详细信息，请参考第 3.15.2 (1) 节“ROM 支持函数”。

STARTOF 是一个使用“STARTOF <section name>”（STARTOF <段名称>）格式来决定段的起始地址的运算符。

SIZEOF 是一个使用“SIZEOF <section name>”（SIZEOF <段名称>）格式来决定段的大小（以字节单元）的运算符。

```
/*********************************************************************/
/*                                  文件名“init.c”            */
/*********************************************************************/
   #include "7032.h"
   #include "sample.h"
/*********************************************************************/
/*                            段初始化模块                           */
/*********************************************************************/
   extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;
   void _INITSCT(void)
   {
          register int *p,*q;
          for (p=_B_BGN; p<_B_END; p++)
                *p=0;
          for (p=_D_BGN; q=_D_ROM, p<_D_END; p++,q++)
                *p=*q;
 }
/********************************************************************* */
/*                            外围初始化模块                       * */
/********************************************************************* */
   void init_peripheral(void)
   {
          INTC.IPRA.WORD = 0x3000;                  /* 初始化 IPRA */
          ITU0.TCR.BYTE = 0x02;                     /* 初始化 TCR0 */
          TCSR_FRT = 0x5A01;                        /* 初始化 TCSR */
          PB.DR.WORD = 0x80;                        /* 初始化 PORT */
   }
```

图 2.14 初始化程序（C 语言部分）

在段初始化模块 _INITSCT 中，B 段会零清除以及根据在 sct.src 中指定的段地址将 ROM 初始化数据复制到 RAM。int 类型的说明符将会使用，但如果大小在 4n 字节以外，则会使用 char。

在内部寄存器初始化模块 init_peripheral 中，将会执行下列设定。

- 在中断优先级寄存器 A 中，IRQ0 中断优先级将会设定为 3。
- 在定时器控制寄存器 0 中，将会禁止清除 16 位集成定时器脉冲元件的定时器计数器 0，在上升沿计数，而内部时钟将设定为以 ϕ/4 计数。
- 监视定时器的定时器计数器将设定为 0x01。
- 端口 B 将设定为 0x80。

2.2.5 创建中断函数

图 2.15 显示中断函数。 外部中断处理程序函数 IRQ0 和陷阱指令函数 inv_inst 将会定义。

```
/************************************************************************/
/*                              文件名 "int.c"                          */
/************************************************************************/
   #include "7032.h"
   #include "sample.h"
   extern const short padata;                /*      C 段            */
   extern short a;                           /*      D 段            */
   extern int work;                          /*      B 段            */
   #pragma interrupt(IRQ0, inv_inst)
/************************************************************************/
/*                              中断模块 IRQ0                           */
/************************************************************************/
   void IRQ0(void)
   {
          a = PB.DR.WORD;
          PC.DR.BYTE = padata;
   }
/************************************************************************/
/*                              中断模块 inv_inst                       */
/************************************************************************/
   void inv_inst(void)
   {
          return;
   }
```

图 2.15 中断函数

当 IRQ0 外部中断发生时，函数 IRQ0 会将全局变量 a 设定为 PB.DR.WORD (0x80)。 由此，意味着 CPU 将会置入低功耗状态。

2.2.6 为装入模块创建批文件

图 2.16 显示用于创建 S 类型装入模块 (sample.mot) 的批文件。

```
shcΔ-debugΔsample.cΔinit.cΔint.c
#编译 C 程序
asmshΔsct.srcΔ–debug
#汇编汇编程序
shcΔ-debugΔ-section=c=VECTΔvect.c
#编译向量表创建程序
optlnkΔ-nooptΔ-subcommand=rom.sub
#使用子命令文件连接
rmΔ*.obj
#移除目标模块文件
```

图 2.16 创建装入模块的批文件

在这里，vect.c 被编译到独立文件中，而选项 section=VECT 用于使它成为其他初始化数据元件以外的段。 在连接时，它将会分配到从 0 起始的地址。

2.2.7 创建连接编辑程序子命令文件

图 2.17 显示在创建装入模块时使用的连接编辑程序的子命令文件（文件名 rom.sub）。

```
Sdebug
input sample,init,int,vect,sct
                    ; 指定输入文件
library /user/unix/SHCV5.0/shclib.lib
                    ; 指定标准程序库
output sample.abs    sample.abs ; 指定输出文件名
rom D=R             ; 指定 ROM 支持选项
start VECT/0,P,C,D/0400,R,B/F0000000
                    ; 指定每个段的起始地址
                    ; 分配从地址 0 起始的 VECT 段
                    ; 将 P、C、D 段分配到从地址 H'400 起始的区域
                    ; 将 R、B 段分配到从地址 H'F0000000 起始的区域
form s              ;指定 S 类型格式
list  sample.map    ; 指定存储器映像信息输出
Exit
```

图 2.17 连接编辑程序的子命令文件

2.3 样品程序简介 （SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP）

在本小节中，将介绍 SH7708 例子的样品程序。 这里介绍的样品程序执行从复位直到传递至 main() 函数的执行之处理。这是 CPU 启动时所需的最小程序实例。

2.3.1 创建中断处理程序

与 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU 和 SH2-DSP 不同，在 SH-3、SH3-DSP、SH-4、SH-4A 和 SH4 AL-DSP 例子中，中断出现时的向量控制必须在软件中指定。

SH-3 中的中断之固定地址将在三个不同原因下设定为 PC（程序计数器）： 复位、异常，以及中断。 因此，必须在每一个该类地址上编写用于确定中断因数并转移至相应处理例程的中断处理函数。

个别处理程序详述如下。 此处实例中的向量基址寄存器 (VBR) 固定于 H'00000000，且未使用存储器管理单元 (MMU) 。

(1) 复位处理程序（地址 H'00000000）

在加电或人工复位时，PC 将设定为 H'a0000000。 由于地址 H'00000000 和 H'a0000000 与一个常用物理地址相应，因此程序将放置在 H'00000000。此时将会执行下列步骤：

- 异常 judgment 由 EXEVT 执行。
- 从向量表调用处理例程。

处理过程在图 2.18 中显示。

```
;*********************************************************************
;           文件名 "reset.src"
;*********************************************************************
;   SH708 复位处理程序例程
            .IMPORT       _vecttbl
            .IMPORT       _stacktbl
            .SECTION   VECT,CODE,LOCATE=H'0
__reset:
;************************************************************
;           将堆栈指针设定为“R15”前，
;           您应该通过 BSC 初始化堆栈 RAM 区域。
;************************************************************
; 实例 ） 区域1 （CS1） -> 堆栈 RAM
; 区域1
; 总线大小 ->16位
; D23-D16 ->不是 PORT
; 等待 3 状态
; BCR2>> PORTEN:A1SZ0:A1SZ0
; 0: 1 :0
; >> BCR2=0x3fff8
            MOV.L      BSCR2,R0
            MOV.L      #H'3fff8,R1
            MOV.W   R1,@R0
; WCR2>> A1-2W1:A1-2W0
```

```
; 1: 1
; >> WCR2=0xffff
           MOV.L   WCR2,R0
           MOV.L   #H'ffff,R1
           MOV.W  R1,@R0
;************************************************************
           MOV.L  VECTadr,R1
           MOV.L  STACKadr,R2
           MOV.L      EXPEVT,R0
           MOV.L      @R0,R0
           CMP/EQ     #0,R0       ; 加电复位
           BT      PON_RESET
           CMP/EQ     #H'20,R0
           BT         MANUAL_RESET
           ; if( EXPEVT != RESET)
           ; while(1);
LOOP
    BRA        LOOP
           NOP
PON_RESET
           MOV.L      @(0,R1),R1      ; 设定函数
           MOV.L      @(0,R2),R15 ; 设定堆栈指针
           JMP        @R1
           NOP
MANUAL_RESET
           MOV.L      @(4,R1),R1      ; 设定函数
           MOV.L      @(4,R2),R15 ; 设定堆栈指针
           JMP        @R1
           NOP
;
           .ALIGN        4
VECTadr    .DATA.L       _vecttbl
STACKadr   .DATA.L       _stacktbl
EXPEVT     .DATA.L       H'ffffffd4
BSCR2      .DATA.L          H'ffffff62
WCR2       .DATA.L       H'ffffff66
           .END
```

图 2.18 复位处理程序

(2) 通用异常处理处理程序 (VBR+H'100)

- 异常因数代码从 EXPEVT 读取。
- 此因数的处理函数（向量函数）从向量表读取。
- 终止例程 将会设定。执行跳转到向量函数。

在此例子中，将使用 RTE 指令以跳转到向量函数。 这里的 PR 寄存器值将会在跳转至向量函数前立即更改，由此，从向量函数返回时，控制将传递至终止例程。 由于 PR 在向量函数处理过程中是终止例程，因此需要通过执行 RTS 返回向量函数。 所以，在定义向量函数时，不要使用“#pragma interrrupt”。

(3) VBR+H'400 TLB 遗漏异常处理程序

由于未使用 MMU，因此将不被包括。

(4) VBR+H'600 中断处理程序

- 中断因数代码从 INTEVT 读取。
- 此因数的处理函数（向量函数）从向量表读取。
- 此因数的中断屏蔽级别从中断屏蔽表设定。
- 终止例程将会设定。
- 执行跳转到向量函数。

此处理基本上与通用异常处理程序的一样；从向量函数返回时，控制将传递至终止例程。

```
;*********************************************************************
; 文件：vhandler.src
;*********************************************************************
       .include  "env.inc"
       .include  "vect.inc"

IMASKclr: .equ   H'FFFFFF0F
RBBLclr:  .equ   H'CFFFFFFF
MDRBBLset:    .equ   H'70000000
       .import       _RESET_Vectors
       .import       _INT_Vectors
       .import       _INT_MASK
;*********************************************************************
;* 宏定义
;*********************************************************************
.macro PUSH_EXP_BASE_REG
stc.l ssr,@-r15 ; 保存 ssr
stc.l spc,@-r15 ; 保存 spc
sts.l pr,@-r15 ; 保存上下文寄存器
stc.l r7_bank,@-r15
stc.l r6_bank,@-r15
stc.l r5_bank,@-r15
stc.l r4_bank,@-r15
stc.l r3_bank,@-r15
stc.l r2_bank,@-r15
stc.l r1_bank,@-r15
stc.l r0_bank,@-r15
.endm
;
.macro POP_EXP_BASE_REG
```

```
ldc.l @r15+,r0_bank ; 恢复寄存器
ldc.l @r15+,r1_bank
ldc.l @r15+,r2_bank
ldc.l @r15+,r3_bank
ldc.l @r15+,r4_bank
ldc.l @r15+,r5_bank
ldc.l @r15+,r6_bank
ldc.l @r15+,r7_bank
lds.l @r15+,pr
ldc.l @r15+,spc
ldc.l @r15+,ssr
.endm
;***********************************************************************
;       复位
;***********************************************************************
        .section  RSTHandler,code
_ResetHandler:
              mov.l  #EXPEVT,r0
              mov.l  @r0,r0
              shlr2  r0
              shlr   r0
              mov.l  #_RESET_Vectors,r1
              add    r1,r0
              mov.l  @r0,r0
              jmp    @r0
              nop
;***********************************************************************
;       异常中断
;***********************************************************************
        .section  INTHandler,code
        .export   INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
              PUSH_EXP_BASE_REG
;
              mov.l  #EXPEVT,r0        ; 设定事件地址
              mov.l  @r0,r1            ; 设定异常码
              mov.l  #_INT_Vectors,r0  ; 设定向量表地址
              add #-(h'40),r1          ; 异常码 - h'40
              shlr2  r1
              shlr   r1
              mov.l  @(r0,r1),r3       ; 设定中断函数 addr
;
              mov.l  #_INT_MASK,r0     ; 中断屏蔽表 addr
```

```
                shlr2   r1
                mov.b   @(r0,r1),r1             ; 中断屏蔽
                extu.b r1,r1
;
                stc sr,r0                       ; 保存 sr
                mov.l   #(RBBLclr&IMASKclr),r2  ; RB,BL,屏蔽清除数据
                and r2,r0                       ; 清除屏蔽数据
                or      r1,r0                   ; 设定中断屏蔽
                ldc r0,ssr                      ; 设定当前状态
;
                ldc.l   r3,spc
                mov.l   #__int_term,r0          ; 设定中断终止
                lds r0,pr
;
                rte
                nop
;
                .pool
;
;*********************************************************************
;       中断终止
;*********************************************************************
                .align 4
__int_term:
                mov.l #MDRBBLset,r0             ; 设定 MD,BL,RB
                ldc.l   r0,sr                   ;
                POP_EXP_BASE_REG
                rte                             ; 返回
                nop
;
                .pool
;
;*********************************************************************
;       TLB 遗漏中断
;*********************************************************************
        .org    H'300
_TLBmissHandler:
                PUSH_EXP_BASE_REG
;
                mov.l   #EXPEVT,r0              ; 设定事件地址
                mov.l   @r0,r1                  ; 设定异常码
                mov.l   #_INT_Vectors,r0        ; 设定向量表地址
                add #-(h'40),r1                 ; 异常码 - h'40
```

```
                shlr2   r1
                shlr    r1
                mov.l   @(r0,r1),r3             ; 设定中断函数 addr
;
                mov.l   #_INT_MASK,r0           ; 中断屏蔽表 addr
                shlr2   r1
                mov.b   @(r0,r1),r1             ; 中断屏蔽
                extu.b  r1,r1
;
                stc sr,r0                       ; 保存 sr
                mov.l   #(RBBLclr&IMASKclr),r2  ; RB,BL,屏蔽清除数据
                and r2,r0                       ; 清除屏蔽数据
                or      r1,r0                   ; 设定中断屏蔽
                ldc r0,ssr                      ; 设定当前状态
;
                ldc.l   r3,spc
                mov.l   #__int_term,r0          ; 设定中断终止
                lds     r0,pr
;
                rte
                nop
;
                .pool
;
;*********************************************************************
;       IRQ
;*********************************************************************
        .org    H'500
_IRQHandler:
                PUSH_EXP_BASE_REG
;
                mov.l   #INTEVT,r0              ; 设定事件地址
                mov.l   @r0,r1                  ; 设定异常码
                mov.l   #_INT_Vectors,r0        ; 设定向量表地址
                add #-(h'40),r1                 ; 异常码 - h'40
                shlr2   r1
                shlr    r1
                mov.l   @(r0,r1),r3             ; 设定中断函数 addr
;
                mov.l   #_INT_MASK,r0           ; 中断屏蔽表 addr
                shlr2   r1
                mov.b   @(r0,r1),r1             ; 中断屏蔽
                extu.b  r1,r1
;
```

```
                stc sr,r0                           ; 保存 sr
                mov.l  #(RBBLclr&IMASKclr),r2       ; RB,BL, 屏蔽清除数据
                and r2,r0                           ; 清除屏蔽数据
                or     r1,r0                        ; 设定中断屏蔽
                ldc r0,ssr                          ; 设定当前状态
;
                ldc.l  r3,spc
                mov.l  #__int_term,r0               ; 设定中断终止
                lds r0,pr
;
                rte
                nop
;
                .pool
                .end
```

图 2.19 中断处理程序

注意： 包含文件“env.inc”和“vect.inc”会在创建 SH3 工程时由 HEW 自动生成。

2.3.2 创建向量表

(1) 向量表设定 <vect.c>

本节描述向量表、中断优先级表以及 TRAPA 函数表。 每个因数的名称会在此表中注册，用户所创建函数的实际名称会在 vect7708.h 标题文件中提供。

```
/************************************************************/
/* 文件名 "vect.c"                                          */
/************************************************************/
#include "vect7708.h"

/************************************************************/
/* 分配堆栈区域                                             */
/************************************************************/
#pragma section STK /* 段名称 "BSTK" */
long stack[STACK_SIZE];
#pragma section
/************************************************************/
/* 分配定义表                                               */
/************************************************************/
const void *stacktbl[]={
    STACK_PON,
    STACK_MANUAL
};
/************************************************************/
/* 分配向量表(EXPEVT 或 INTEVT CODE H'000-H'5a0)            */
```

```
/************************************************************/
void (*const vecttbl[])(void) = {
                            /* EVT KIND CODE REG
    */
    RESET_PON,                      /* PON 复位             H'000   EXPEVT
    */
    RESET_MANUAL,           /* 人工复位                H'020   EXPEVT          */
    TLB_MISS_READ,          /* TLB MISS(R)             H'040   EXPEVT          */
    TLB_MISS_WRITE,         /* TLB MISS(W)             H'060                   */
    TLB_1ST_PAGE,           /*                         H'080                   */
    TLB_PROTECT_READ,       /*                         H'0a0                   */
    TLB_PROTECT_WRITE,      /*                         H'0c0                   */
    ADR_ERROR_WRITE,        /*                         H'0e0                   */
    ADR_ERROR_WRITE,        /*                         H'100                   */
    RESERVED,               /*                         H'120   ------          */
    RESERVED,               /*                         H'140   ------          */
    TRAP,                   /*                         H'160   (使用 TRA)      */
    ILLEGAL_INST,           /*                         H'180   EXPEVT          */
    ILLEGAL_SLOT,           /*                         H'1a0   EXPEVT          */
    NMI,                    /*                         H'1c0   INTEVT          */
    USER_BREAK,             /*                         H'1e0   EXPEVT          */
    IRQ15,                  /*                         H'200   INTEVT          */
    IRQ14,                  /*                         H'220   INTEVT          */
    IRQ13,                  /*                         H'240   INTEVT          */
    IRQ12,                  /*                         H'260   INTEVT          */
    IRQ11,                  /*                         H'280   INTEVT          */
    IRQ10,                  /*                         H'2a0   INTEVT          */
    IRQ9,                   /*                         H'2c0   INTEVT          */
    IRQ8,                   /*                         H'2e0   INTEVT          */
    IRQ7,                   /*                         H'300   INTEVT          */
    IRQ6,                   /*                         H'320   INTEVT          */
    IRQ5,                   /*                         H'340   INTEVT          */
    IRQ4,                   /*                         H'360   INTEVT          */
    IRQ3,                   /*                         H'380   INTEVT          */
    IRQ2,                   /*                         H'3a0   INTEVT          */
    IRQ1,                   /*                         H'3c0   INTEVT          */
    RESERVED,               /*                         H'3e0   ------          */
    TMU0_TUNI0,             /*                         H'400   INTEVT          */
    TMU1_TUNI1,             /*                         H'420   INTEVT          */
    TMU2_TUNI2,             /*                         H'440   INTEVT          */
    TMU2_TICPI2,            /*                         H'460   INTEVT          */
    RTC_ATI,                /*                         H'480   INTEVT          */
    RTC_PRI,                /*                         H'4a0   INTEVT          */
    RTC_CUI,                /*                         H'4c0   INTEVT          */
    SCI_ERI,                /*                         H'4e0   INTRVT          */
    SCI_RXI,                /*                         H'500   INTRVT          */
    SCI_TXI,                /*                         H'520   INTRVT          */
```

```
    SCI_TEI,                    /*                      H'540   INTRVT          */
    WDT_ITI,                    /*                      H'560   INTEVT          */
    REF_RCMI,                   /*                      H'580   INTEVT          */
    DEF_RPVI,                   /*                      H'5a0   INTEVT          */
    RESERVED
};
/*************************************************************/
/* 分配中断优先级表 H'1c0-H'5a0                                */
/*************************************************************/
const char imasktbl[]={
    15<<4,                      /*      NMI 级 16(IMASK=0-15)                  */
    IP_RESERVED,                /* ---------------
    */
    15<<4,                      /*      IRQ15 (IRL0000)                        */
    14<<4,                      /*      IRQ14 (IRL0001)                        */
    13<<4,                      /*      IRQ13 (IRL0010)                        */
    12<<4,                      /*      IRQ12 (IRL0011)                        */
    11<<4,                      /*      IRQ11 (IRL0100)                        */
    10<<4,                      /*      IRQ10 (IRL0101)                        */
     9<<4,                      /*      IRQ9 (IRL0110)                         */
     8<<4,                      /*      IRQ8 (IRL0111)                         */
     7<<4,                      /*      IRQ7 (IRL1000)                         */
     6<<4,                      /*      IRQ6 (IRL1001)                         */
     5<<4,                      /*      IRQ5 (IRL1010)                         */
     4<<4,                      /*      IRQ4 (IRL1011)                         */
     3<<4,                      /*      IRQ3 (IRL1100)                         */
     2<<4,                      /*      IRQ2 (IRL1101)                         */
     1<<4,                      /*      IRQ1 (IRL1110)                         */
    IP_RESERVED,                /*      ---------------                        */
    IP_TMU0,                    /*      TMU0 TUNI0                             */
    IP_TMU1,                    /*      TMU1 TUNI1                             */
    IP_TMU2,                    /*      TNU2 TUNI2                             */
    IP_TMU2,                    /*       TICPI2                                */
    IP_RTC,                     /*      RTC ATI                                */
    IP_RTC,
    IP_RTC,
    IP_SCI,                     /*      SCI      ERI                           */
    IP_SCI,
    IP_SCI,
    IP_SCI,
    IP_WDT,                     /*      WDT ITI                                */
    IP_REF,                     /*      REF RCMI                               */
    IP_REF,                     /*      REF ROVI                               */
    IP_RESERVED
};
void (*const trap tbl[])(void)={
```

```
    TRAPA_0,
    TRAPA_1,
    TRAPA_2,
    TRAPA_3,
    TRAPA_4,
    TRAPA_5,
    TRAPA_6,
    TRAPA_7,
    TRAPA_8,
    TRAPA_9,
    TRAPA_10,
    TRAPA_11,
    TRAPA_12,
    TRAPA_13,
    TRAPA_14,
    TRAPA_15
};
```

图 2.20 向量表

(2) 向量函数注册 <vect7708.h>

用户所定义函数的实际名称和其他参数将会设定。 如果已添加中断处理函数，此区域将会更改。

此处的处理包括：

- 定义堆栈大小
- 定义每个因数的向量函数名称
- 设定中断优先级 （值设定为 IPRA 和 IPRB）

名为“halt”的函数将会在此处为未使用的向量定义。用户函数本身必须使用 #pragma 中断声明定义为中断函数。此外，如果函数已注册，该函数的 extern 声明必须显示在此文件中。

```
/************************************************************/
/* 文件名 "vect7708.h"                                      */
/************************************************************/


/************************************************************/
/* 堆栈大小定义                                             */
/************************************************************/
#define STACK_SIZE            (0x4096/4)      /* 4096 字节 */
#define STACK_PON             (&stack[STACK_SIZE])
#define STACK_MANUAL                  (&stack[STACK_SIZE])
extern long stack[];

/************************************************************/
/* 复位函数定义                                             */
/************************************************************/
```

```
#define    RESET_PON            init /* PON 复位      H'000   EXPEVT  */
#define    RESET_MANUAL   init       /* 人工复位      H'020   EXPEVT  */
/************************************************************/
/* 中断函数定义                                                */
/************************************************************/
#define   TLB_MISS_READ      halt    /* TLB MISS(R)  H'040   EXPEVT  */
#define   TLB_MISS_WRITE     halt    /* TLB MISS(W)  H'060   EXPEVT  */
#define   TLB_1ST_PAGE       halt    /*      H'080           EXPEVT  */
#define   TLB_PROTECT_READ   halt    /*      H'0a0           EXPEVT  */
#define   TLB_PROTECT_WRITE  halt    /*      H'0c0           EXPEVT  */
#define   ADR_ERROR_WRITE    halt    /*      H'0e0           EXPEVT  */
#define   ADR_ERROR_WRITE    halt    /*      H'100           EXPEVT  */
/*#define RESERVED           halt */ /*      H'120           ------  */
/*#define RESERVED           halt */ /*      H'140           ------  */
#define   TRAP               trap    /*      H'160 (使用 TRA)        */
#define   ILLEGAL_INST       halt    /*      H'180           EXPEVT  */
#define   ILLEGAL_SLOT       halt    /*      H'1a0           EXPEVT  */
#define   NMI                halt    /*      H'1c0           INTEVT  */
#define   USER_BREAK                 halt    /*      H'1e0           EXPEVT  */
#define   IRQ15              irq15   /*      H'200           INTEVT  */
#define   IRQ14              halt    /*      H'220           INTEVT  */
#define   IRQ13              halt    /*      H'240           INTEVT  */
#define   IRQ12              halt    /*      H'260           INTEVT  */
#define   IRQ11              halt    /*      H'280           INTEVT  */
#define   IRQ10              halt    /*      H'2a0           INTEVT  */
#define   IRQ9               halt    /*      H'2c0           INTEVT  */
#define   IRQ8               halt    /*      H'2e0           INTEVT  */
#define   IRQ7               halt    /*      H'300           INTEVT  */
#define   IRQ6               halt    /*      H'320           INTEVT  */
#define   IRQ5               halt    /*      H'340           INTEVT  */
#define   IRQ4               halt    /*      H'360           INTEVT  */
#define   IRQ3               halt    /*      H'380           INTEVT  */
#define   IRQ2               halt    /*      H'3a0           INTEVT  */
#define   IRQ1               halt    /*      H'3c0           INTEVT  */
/*#define RESERVED           halt    *//*    H'3e0           ------  */
#define   TMU0_TUNI0                 halt    /*      H'400           INTEVT  */
#define   TMU1_TUNI1                 halt    /*      H'420           INTEVT  */
#define   TMU2_TUNI2                 halt    /*      H'440           INTEVT  */
#define   TMU2_TICPI2        halt    /*      H'460           INTEVT  */
#define   RTC_ATI            halt    /*      H'480           INTEVT  */
#define   RTC_PRI            halt    /*      H'4a0           INTEVT  */
#define   RTC_CUI            halt    /*      H'4c0           INTEVT  */
#define   SCI_ERI            halt    /*      H'4e0           INTRVT  */
#define   SCI_RXI            halt    /*      H'500           INTRVT  */
```

```
#define   SCI_TXI             halt    /*      H'520           INTRVT  */
#define   SCI_TEI             halt    /*      H'540           INTRVT  */
#define   WDT_ITI             halt    /*      H'560           INTEVT  */
#define   REF_RCMI            halt    /*      H'580           INTEVT  */
#define   DEF_RPVI            halt    /*      H'5a0           INTEVT  */
#define   RESERVED            halt
extern void init(void);
extern void halt(void);
extern void _trap(void);
extern void irq15(void);

/*********************************************************************/
/* 中断屏蔽定义                                                      */
/*********************************************************************/
#define IP_TMU0     (0<<4)
#define IP_TMU1     (0<<4)
#define IP_TMU2     (0<<4)
#define IP_RTC      (0<<4)
#define IP_SCI      (0<<4)
#define IP_WDT      (0<<4)
#define IP_REF      (0<<4)
#define IP_RESERVED    (15<<4)
/*********************************************************************/
/* IPRA,IPRB 定义                                                    */
/*********************************************************************/
#define WORD_IPRA   ((IP_TMU0<<12)|(IP_TMU1<<8)|(IP_TMU2<<4)|IP_RTC)
#define WORD_IPRB   ((IP_WDT<<12)|(IP_REF<<8)|(IP_SCI<<4)|0)
extern void set_ip(void);
extern long stack[];

/*********************************************************************/
/* TRAPA 系统调用定义                                                */
/*********************************************************************/
#define TRAPA_0     halt
#define TRAPA_1     halt
#define TRAPA_2     halt
#define TRAPA_3     halt
#define TRAPA_4     halt
#define TRAPA_5     halt
#define TRAPA_6     halt
#define TRAPA_7     halt
#define TRAPA_8     halt
#define TRAPA_9     halt
#define TRAPA_10    halt
```

```
#define TRAPA_11    halt
#define TRAPA_12    halt
#define TRAPA_13    halt
#define TRAPA_14    halt
#define TRAPA_15    halt /*#15(#0F) 应该是异常例程（非法使用）                    */
```

图 2.21 向量函数名称定义

2.3.3 创建标题文件

样品程序的常用标题文件如下所示。

```
/*********************************************************************/
/*                    文件名 "7700s.h"       （提取）                 */
/*********************************************************************/
struct st_intc {                                    /*  struct INTC */
               union {                              /*    ICR       */
                     unsigned short WORD;           /*  字节存取      */
                     struct {                       /*  位存取        */
                            unsigned short NMIL:1;  /*    NMIL      */
                            unsigned short     :6;  /*              */
                            unsigned short NMIE:1;  /*    NMIE      */
                            }       BIT;            /*              */
                     }          ICR;                /*              */
               union {                              /*    IPRA      */
                     unsigned short WORD;           /*  字存取        */
                     struct {                       /*  位存取        */
                            unsigned short UU:4;    /*    IRQ0      */
                            unsigned short UL:4;    /*    IRQ1      */
                            unsigned short LU:4;    /*    IRQ2      */
                            unsigned short LL:4;    /*    IRQ3      */
                            }       BIT;            /*              */
                     }          IPRA;               /*              */
               union {                              /*    IPRB      */
                     unsigned short WORD;           /*  字存取        */
                     struct {                       /*  位存取        */
                            unsigned short UU:4;    /*    IRQ4      */
                            unsigned short UL:4;    /*    IRQ5      */
                            unsigned short LU:4;    /*    IRQ6      */
                            unsigned short LL:4;    /*    IRQ7      */
                            }       BIT;            /*              */
                     }          IPRB;               /*              */
               char             wk1[234];           /*              */
               unsigned int   TRA;                  /*    TRA       */
               unsigned int   EXPEVT;               /*    EXPEVT    */
               unsigned int   INTEVT;               /*    INTEVT    */
```

```
};                                                      /*              */

union un_ccr {                                          /*  联合 CCR    */
             unsigned int       LONG;                   /*  Long 存取   */
             struct {                                   /*  位存取      */
                    unsigned int              :26;      /*              */
                    unsigned int RA   :1;               /*    RA        */
                    unsigned int      :1;               /*    0         */
                    unsigned int CF   :1;               /*    CF        */
                    unsigned int CB   :1;               /*    CB        */
                    unsigned int WT   :1;               /*    WT        */
                    unsigned int CE   :1;               /*    CE        */
                    }           BIT;                    /*              */
};                                                      /*              */
#define SCI    (*(volatile struct st_sci  *)0xFFFFFE80) /* SCI  地址    */
#define TMU    (*(volatile struct st_tmu  *)0xFFFFFE90) /* TMU  地址    */
#define TMU0   (*(volatile struct st_tmu0 *)0xFFFFFE94) /* TMU0 地址    */
#define TMU1   (*(volatile struct st_tmu0 *)0xFFFFFEA0) /* TMU1 地址    */
#define TMU2   (*(volatile struct st_tmu2 *)0xFFFFFEAC) /* TMU2 地址    */
#define RTC    (*(volatile struct st_rtc  *)0xFFFFFEC0) /* RTC  地址    */
#define INTC   (*(volatile struct st_intc *)0xFFFFFEE0) /* INTC 地址    */
#define BSC    (*(volatile struct st_bsc  *)0xFFFFFF60) /* BSC  地址    */
#define CPG    (*(volatile struct st_cpg  *)0xFFFFFF80) /* CPG  地址    */
#define UBC    (*(volatile struct st_ubc  *)0xFFFFFF90) /* UBC  地址    */
#define MMU    (*(volatile struct st_mmu  *)0xFFFFFFE0) /* MMU  地址    */
#define CCR    (*(volatile union  un_ccr  *)0xFFFFFFEC) /* CCR  地址    */
```

图 2.22 标题文件

2.3.4 创建初始化部分

复位后，BSC 和指针将会设定，而控制将会传递至初始化函数。

初始化函数会设定中断优先级和初始化段，然后将控制传递至用户函数的起始。

(1) 初始化函数 <init.c、cntrl.h>

- 设定中断优先级
- 清除高速缓存
- 打开高速缓存
- 初始化段
- 设定中断屏蔽
- 转移到用户函数

```
/*****************************************************************/
/* 文件名 "cntrl.h" */
/*****************************************************************/
#include <machine.h>
#include "7700s.h"
/*****************************************************************/
/* 控制 BL ,MD 位 */
/*****************************************************************/
#define BLoff()     set_cr((get_cr()&0xefffffff))
#define BLon()      set_cr((get_cr()|0x10000000))
#define USRmode()   set_cr((get_cr()|0x40000000))
/*****************************************************************/
/* 高速缓存控制 */
/*****************************************************************/
#define CacheON()           (CCR.BIT.CE=1)
#define CacheOFF()          (CCR.BIT.CE=0)
#define CacheFLASH()              (CCR.BIT.CF=1)
```

图 2.23 宏定义程序

```
/*****************************************************************/
/* 文件名 "init.c" */
/*****************************************************************/
#include <machine.h>
#include "cntrl.h"
void init(void)
{
    set_ip();
    CacheOFF();
    CacheFLASH();
    CacheON();
    BLoff();          /* 块位关闭                */
    _INITSCT();       /* 段初始化                */
    set_imask(0);     /* 中断优先级 0            */
    main();           /* 用户 main() 例程        */
    halt();           /* halt()                  */
}
```

图 2. 24 初始化程序 (1)

(a) 设定中断优先级<ipr.c>

IPRA 和 IPRB 用于设定在 vect7708.h 中定义的每个中断因数的中断优先级。

```
/************************************************************/
/* 文件名 "ipr.c" */
/************************************************************/
#include "7700s.h"
#include "vect7708.h"
void set_ip(void)
{
    INTC.IPRA.WORD=WORD_IPRA;
    INTC.IPRB.WORD=WORD_IPRB;
}
```

图 2.25 用于设定中断优先级的程序

(b) 段初始化 <sect.src、initsct.c>

将会执行分配给 RAM 的段初始化。

未初始化的数据段 B 将会清除为 0。已初始化的数据项目将会从 ROM 中的段 D 复制到 RAM 中的段 R。 (initsct.c)

此外，需要使用汇编语言代码来取得段起始地址和大小。 (sct.src)

```
;********************************************************************
; 文件名 "sct.src"
;********************************************************************
          .SECTION  B,DATA,ALIGN=4
          .SECTION  R,DATA,ALIGN=4
          .SECTION  D,DATA,ALIGN=4
;   如果存在其他段，请在此插入“.SECTION XXX”,
          .SECTION  C,DATA,ALIGN=4
__B_BGN:  .DATA.L       (STARTOF B)
__B_END:  .DATA.L       (STARTOF B)+(SIZEOF B)
__D_BGN:  .DATA.L       (STARTOF R)
__D_END:  .DATA.L       (STARTOF R)+(SIZEOF R)
__D_ROM:  .DATA.L       (STARTOF D)
          .EXPORT       __B_BGN
          .EXPORT       __B_END
          .EXPORT       __D_BGN
          .EXPORT       __D_END
          .EXPORT       __D_ROM
          .END
```

图 2.26 段定义程序

```
/************************************************************/
/* 文件名 "initsct.c"                                       */
/************************************************************/
extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;

void _INITSCT(void)
{
    register int *p, *q;
    for (p=_B_BGN; p<_B_END; p++){
        *p=0;
    }
    for (p=_D_BGN, q=_D_ROM; p<_D_END; p++, q++){
        *p=*q;
    }
}
```

图 2.27 段初始化程序

2.3.5 创建主要处理部分和中断处理部分

创建函数 main()、halt() 和 irq15() 之后，上述程序将会连接。

```
void main(void)
{
/* 用户程序描述 */
}
#pragma interrupt(halt, irq15)

void halt(void)
{
    while(1);                    /* 错误处理的例程                    */
                            /* 此处保留为无穷循环              */
}
void irq15(void)
{
                            /* IRQ15 处理程序                  */
}
```

图 2.28 Main 处理程序

2.3.6 为装入模块创建批文件

图 2.29 显示用于创建 S 类型装入模块 (sample.mot) 的批文件。

```
shcΔ-debugΔ-cpu=sh3Δvect.cΔinit.cΔipr.cΔinitsct.cΔmain.c
                                                    #编译 C 程序
asmshΔsct.srcΔ-debugΔ-cpu=sh3
asmshΔintr.srcΔ-debugΔ-cpu=sh3
asmshΔreset.srcΔ-debugΔ-cpu=sh3
                                                    #汇编汇编程序
optlnkΔ-nooptΔ-subcommand=lnk.sub
                                                    #使用子命令文件连接
rmΔ*.obj
                                                    #移除目标模块文件
```

图 2.29 用于创建装入模块的批文件

2.3.7 创建连接编辑程序子命令文件

图 2.30 显示在创建装入模块时使用的连接编辑程序的子命令文件（文件名 lnk.sub）。

```
Sdebug
input     vect, init, ipr, initsct, main, intr, sct, reset
                        ; 指定输入文件
Library   /user/unix/SHCV50/shc3npb.lib
                        ; 指定标准程序库
output    sample.abs    sample.abs ; 指定输出文件
rom D=R                 ; 指定 ROM 支持选项
start     P,C,D/10000,R,B,BSTK/04000000
                        ; 指定每个段的起始地址。
                        ; 不要为段 VECT 指定地址，
                        ; 因为段 VECT 已分配到绝对地址段
                        ;（已分配到地址 0）。
                        ; 将 P、C 和 D 段分配到
                        ; 从地址 H'04000000 起始的区域。
                        ; 将 R 和 B 段分配到
                        ; 从地址 H'04000000 起始的区域。
form      s             ; 指定 S 类型格式
list      sample.map    ; 指定存储器映像信息输出
Exit
```

图 2.30 连接编辑程序的子命令文件

2.4 使用模拟程序调试程序进行调试

2.4.1 设定配置

使用在第 2.1.1 节中创建的工程来执行模拟程序调试程序。

从 [选项 (Option)] 菜单选取 [创建配置 (Build Configurations) …] 以显示可用的环境。 在图 2.32 中显示的画面上，选取您使用的环境。 在此例子中，选取 [SimDebug SH-1]。

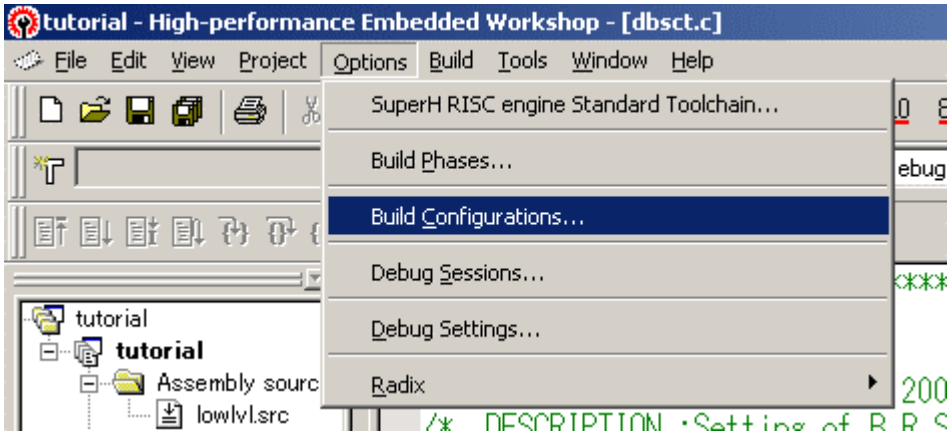


图 2.31 选项菜单

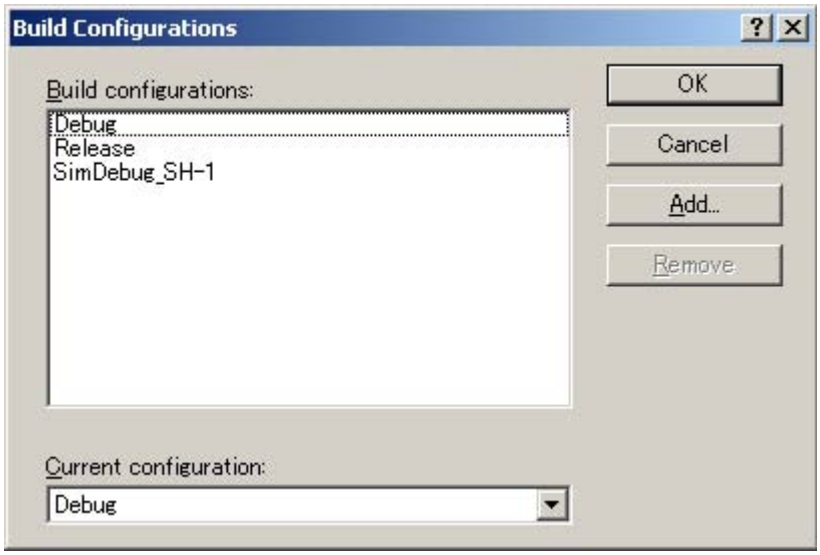


图 2.32 “创建配置”(Build Configurations) 对话框

2.4.2 分配存储器资源

若要运行正在开发的应用程序，分配存储器资源是必要的。 使用演示工程时，存储器资源将会自动分配，因此请检查设定值。

- 从 [选项 (Option)] 菜单选取 [模拟程序 (Simulator) -> 存储器资源 (Memory Resource) …] 以显示当前的存储器资源分配。

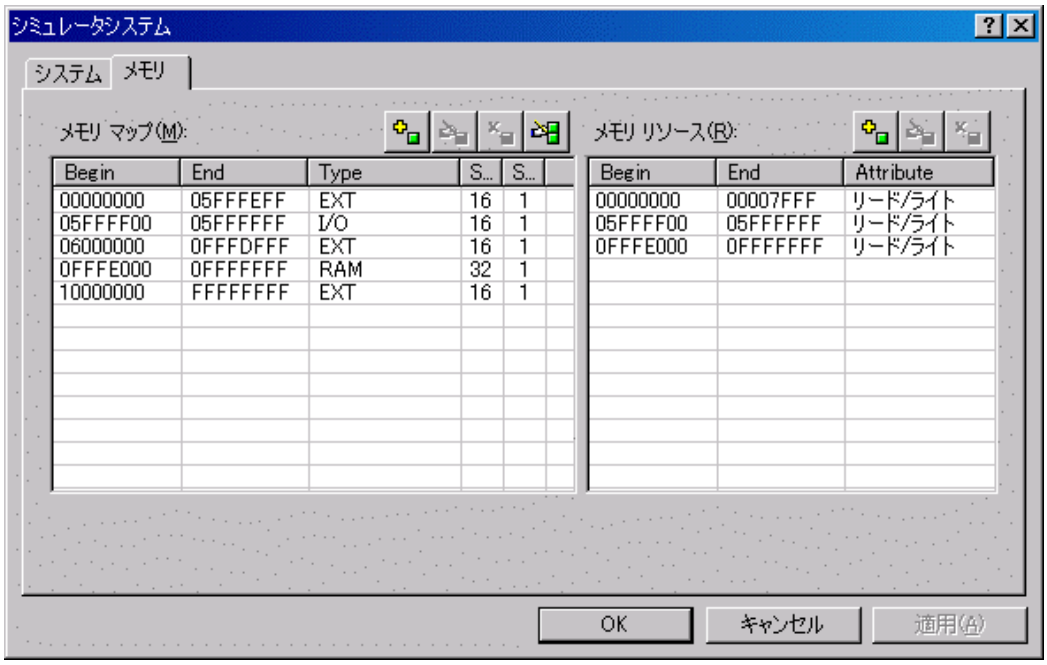


图 2.33 “模拟程序 -> 模拟程序系统”（Simulator -> Simulator System）对话框

程序区域将分配到地址 H'00000000 至 H'00007FFF。 堆栈区域将分配到地址 H'0FFFE000 至 H'0FFFFFFF，可以从该地址读取或写入。

- 通过单击 [确定（OK）] 关闭对话框。

存储器资源也可通过使用 [SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain)] 对话框上的 [模拟程序 (Simulator)] 标签参考或修改。 在任何一个对话框中所作的更改将会反映出来。

2.4.3 下载样品程序

使用演示工程时，样品程序的下载将会自动设定，因此请检查设定值。

- 在 [选项 (Options)] 菜单上选取 [调试设定(Debug Settings)...] 以打开 [调试设定(Debug Settings)] 对话框。

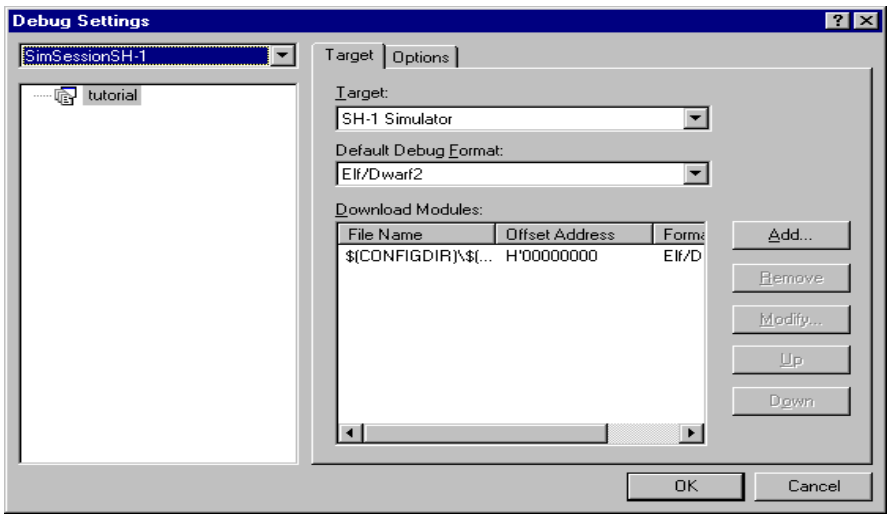


图 2.34“调试设定”(Debug Settings) 对话框

- 将被下载的文件会在 [下载模块 (Download Modules)] 中设定。
- 通过单击 [确定（OK）] 按钮关闭 [调试设定(Debug Settings)] 对话框。
- 从 [调试 (Debug)] 菜单选取 [下载模块 (Download Modules) -> 全部下载模块 (All Download Modules)] 以下载样品程序。

2.4.4 设定模拟的 I/O

使用演示工程时，模拟的 I/O 将会自动设定，因此请检查设定值。

- 从 [选项 (Options)] 菜单选取 [模拟程序->系统 (Simulator->System)] 以打开 [模拟程序系统 (Simulator System)] 对话框。

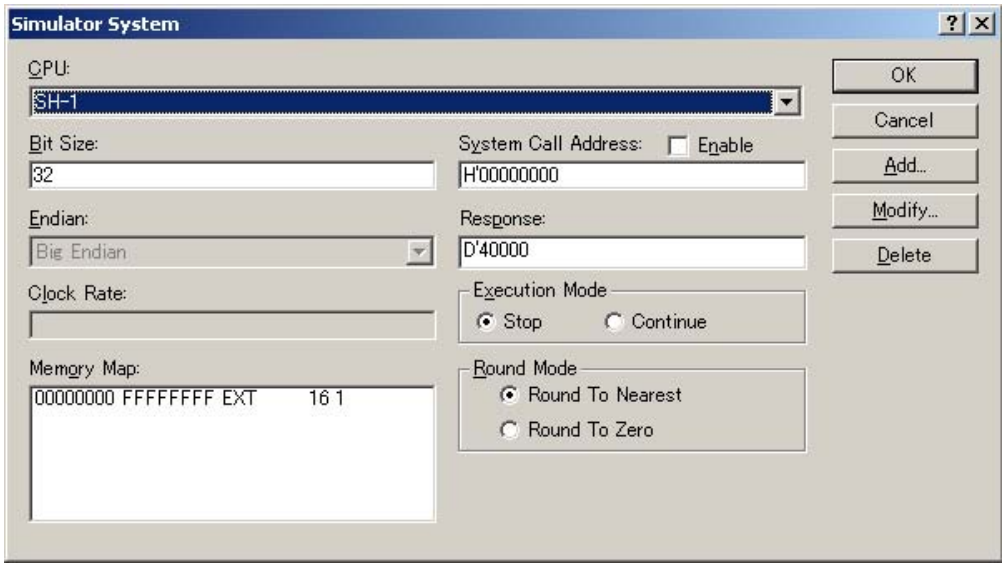


图 2.35 “模拟程序系统”（Simulator System）对话框

- 确定已在 [系统调用地址 (System Call Address)] 中核选 [允许 (Enable)]。

- 单击 [确定 (OK)] 按钮以允许模拟的 I/O。
- 从 [视图 (View)] 菜单选取 [CPU->模拟的 I/O (CPU->Simulated I/O)] 以打开 [模拟的 I/O (Simulated I/O)] 窗口。

如果 [模拟的 I/O (Simulated I/O)] 窗口没有打开，“模拟的 I/O”(Simulated I/O) 将不会被允许。

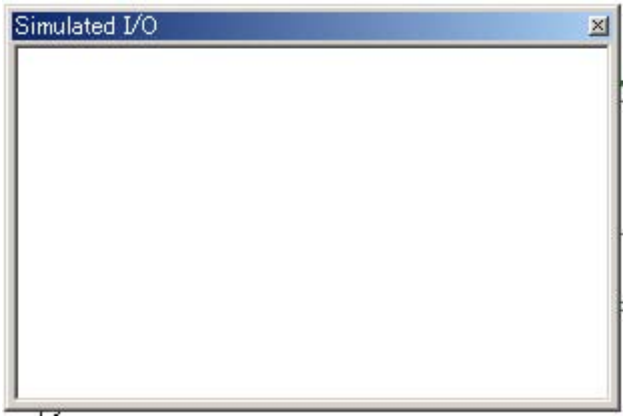


图 2.36 “模拟的 I/O”(Simulated I/O) 窗口

2.4.5 设定跟踪信息采集条件

- 从 [视图 (View)] 菜单选取 [代码 (Code) -> 跟踪 (Trace)] 以打开 [跟踪 (Trace)] 窗口。用鼠标右键单击 [跟踪 (Trace)] 窗口以打开弹出式菜单，然后从弹出式菜单中选取 [采集 (Acquisition)...]。

以下的 [跟踪采集 (Trace Acquisition)] 对话框将会显示。

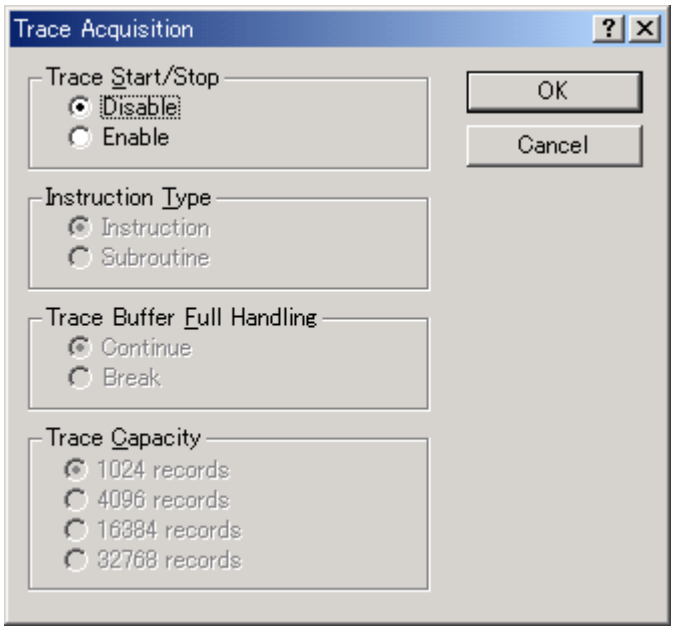


图 2.37 “跟踪采集”(Trace Acquisition) 对话框

- 在 [跟踪采集 (Trace Acquisition)] 对话框中将 [跟踪开始/停止 (Trace start/Stop)] 设定为 [允许 (Enable)]，然后单击 [确定 (OK)] 按钮以允许跟踪信息的采集。

2.4.6 状态窗口

终止的原因可以在 [状态 (Status)] 窗口中显示。

- 从 [视图 (View)] 菜单选取 [CPU -> 状态 (Status)] 以打开 [状态 (Status)] 窗口，然后选取 [状态 (Status)] 窗口中的 [平台 (Platform)] 页。

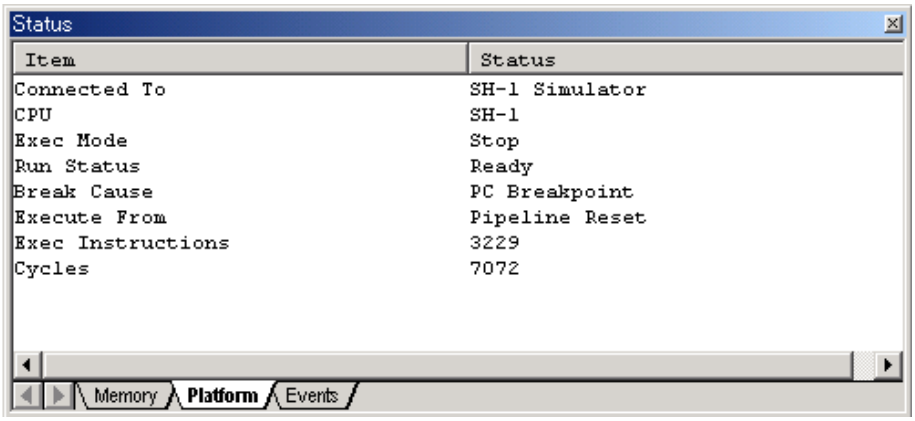


图 2.38“视图->CPU->状态”（View->CPU->Status）窗口

2.4.7 寄存器窗口

寄存器值可以在 [寄存器 (Register)] 窗口中检查。

- 从 [视图 (View)] 菜单选取 [CPU -> 寄存器 (Registers)]。

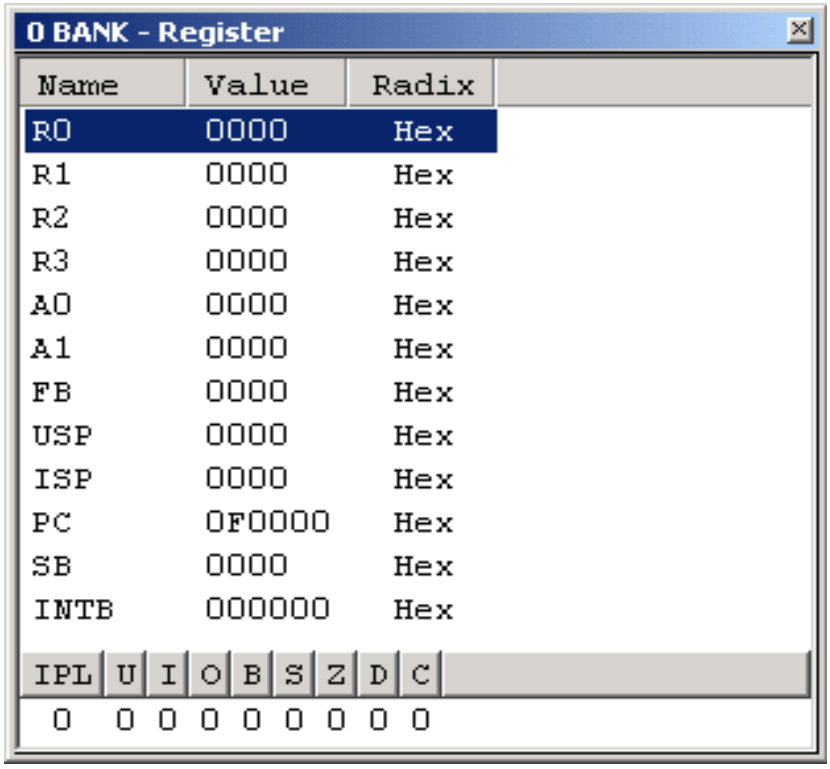


图 2.39“视图->CPU->寄存器”（View->CPU->Register）窗口

2.4.8 跟踪

(1) 跟踪缓冲器

跟踪缓冲器用于阐明指令的执行履历。

- 从 [视图 (View)] 菜单选取 [代码 (Code) -> 跟踪 (Trace)] 以打开 [跟踪 (Trace)] 窗口。 向上滚动至窗口的最上层。

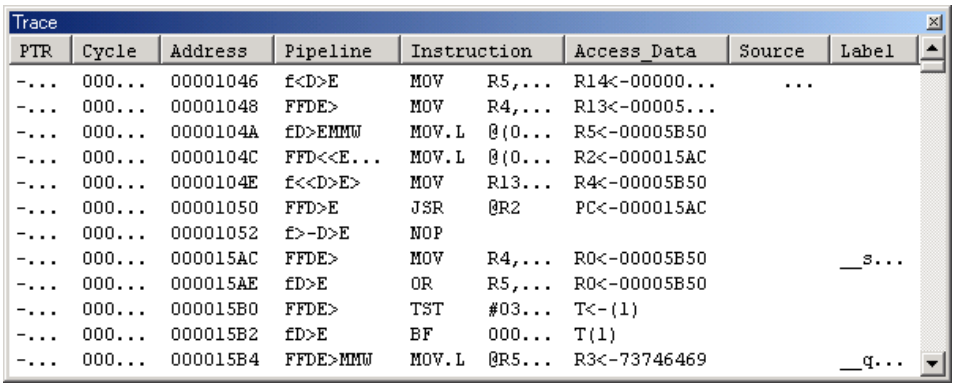


图 2.40“跟踪”(Trace) 窗口（跟踪信息显示）

(2) 跟踪搜索

在 [跟踪 (Trace)] 窗口上单击鼠标右键以启动弹出式菜单，然后选取 [查找 (Find) …] 以打开 [跟踪搜索 (Trace Search)] 对话框。

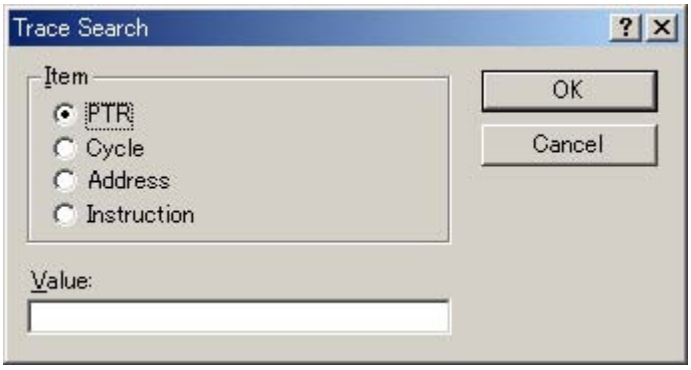


图 2.41“跟踪搜索”(Trace Search) 对话框

在 [项目 (Item)] 中设定要搜索的项目，以及在 [值 (Value)] 中设定要搜索的内容，然后单击 [确定 (OK)] 按钮开始跟踪搜索。 找到搜索的项目时，将会突出显示第一行。 若要继续搜索相同的内容 [值 (Value)]，请在 [跟踪 (Trace)] 窗口上单击鼠标右键以显示弹出式菜单，然后从弹出式菜单选取 [查找下一个 (Find Next)]。 下一个搜索到的行将会突出显示。

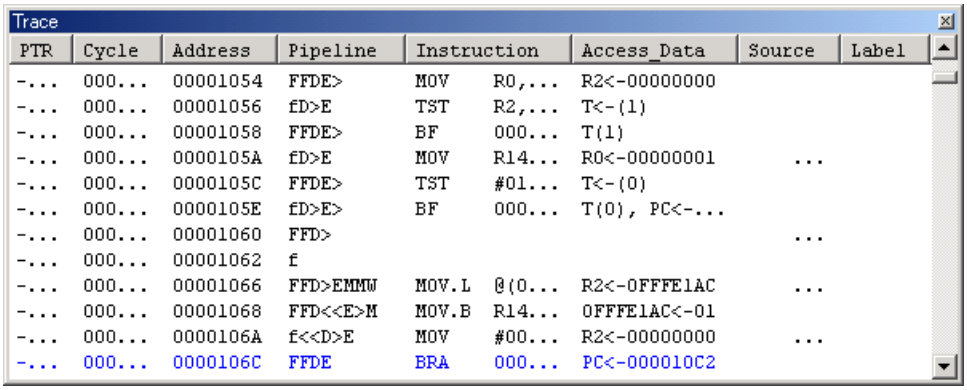


图 2.42 “跟踪”(Trace) 窗口

2.4.9 显示断点

在程序中设定的所有断点列表可以在 [事件点 (Eventpoint)] 窗口中检查。

- 从 [视图 (View)] 菜单选取 [代码->事件点 (Code->Eventpoint)]。

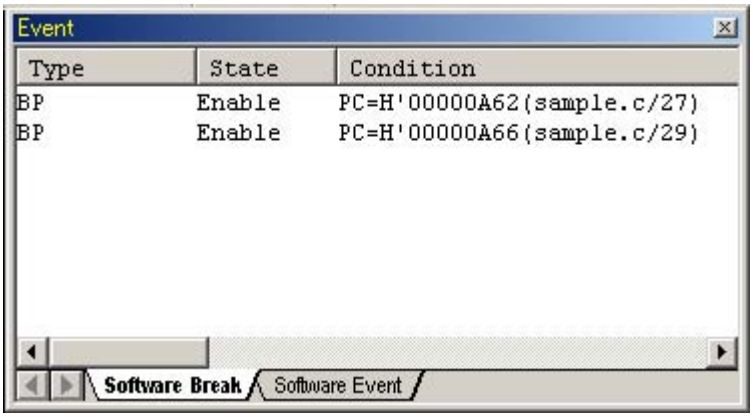


图 2.43 “事件点”(Eventpoint) 窗口

[事件点 (Eventpoint)] 窗口可以用来设定断点、定义新的断点，以及删除断点。

- 关闭 [事件点 (Eventpoint)] 窗口。

2.4.10 显示存储器内容

存储块的内容可以在“存储器”(Memory) 窗口上显示。 例如，以字节大小显示主列的存储器的步骤如下。

从 [视图 (View)] 菜单选取 [CPU -> 存储器 (Memory)] 以在 [起始 (Begin)] 字段输入存储器区域的起始地址及在 [终止 (End)] 字段输入终止地址。

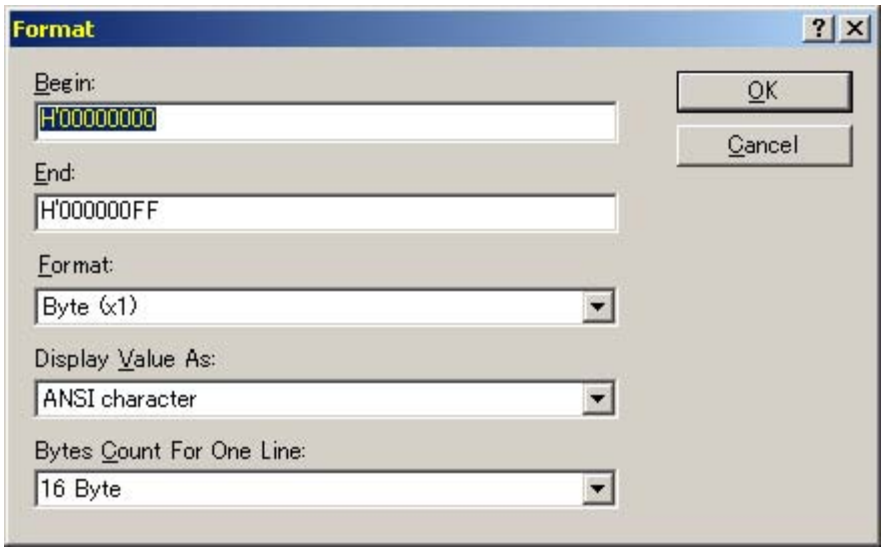


图 2.44“设定地址”(Set Address) 对话框

单击 [确定 (OK)] 按钮以打开“存储器”(Memory) 窗口，它将显示指定的存储器区域。

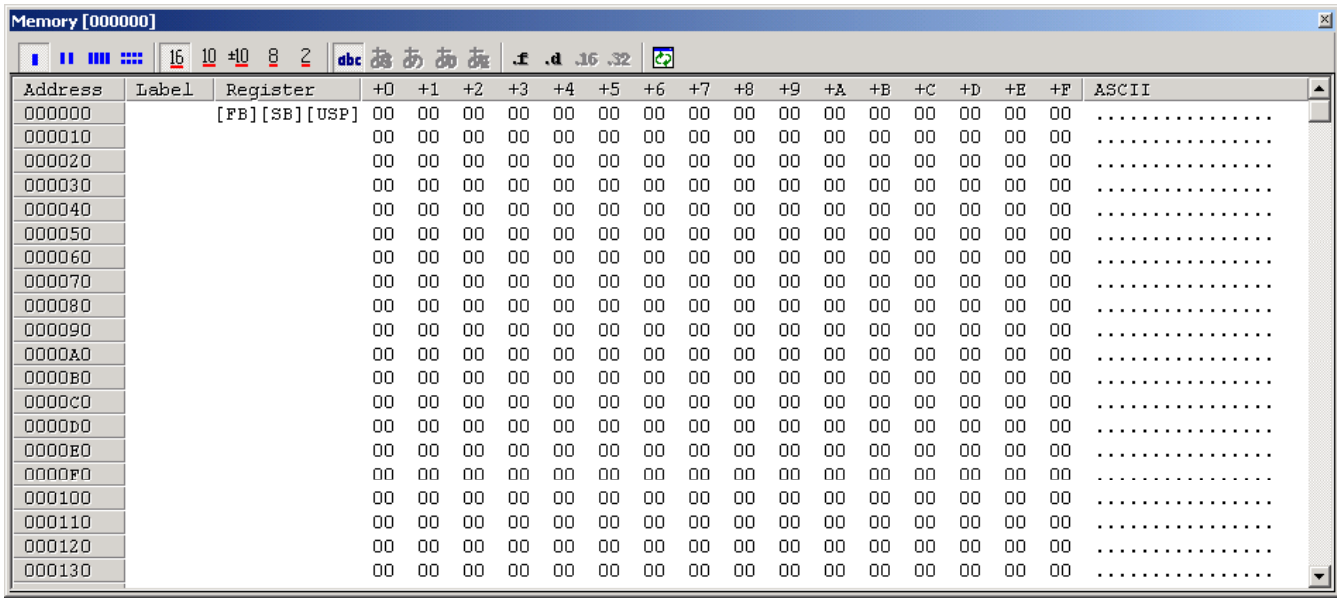


图 2.45 “存储器”(Memory) 窗口

2.5 模拟程序/调试程序中的标准 I/O 和文件 I/O 处理

模拟程序/调试程序可以让用户从要调试的程序执行标准 I/O 和文件 I/O 处理。执行 I/O 处理时，[模拟 I/O (Simulation I/O)] 窗口必须打开。 支持的 I/O 处理如下：

表 2.3 I/O 函数

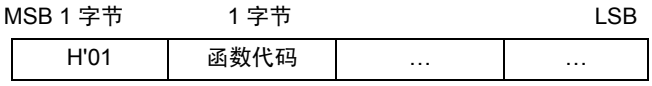
编号	函数代码	函数名称	描述
1	H'21	GETC	从标准输入设备输入一个字节。
2	H'22	PUTC	输出一个字节到标准输出设备。
3	H'23	GETS	从标准输入设备输入一行。
4	H'24	PUTS	输出一行到标准输出设备。
5	H'25	FOPEN	打开文件。
6	H'06	FCLOSE	关闭文件。
7	H'27	FGETC	从文件输入一个字节。
8	H'28	FPUTC	输出一个字节到文件。
9	H'29	FGETS	从文件输入一行。
10	H'2A	FPUTS	输出一行到文件。
11	H'0B	FEOF	检查文件的结束。
12	H'0C	FSEEK	移动文件指针。
13	H'0D	FTELL	返回文件指针的当前位置。

若要执行 I/O 处理，首先，在“模拟程序系统”(Simulator System) 对话框中，指定 [系统调用地址(System Call Address)] 中的 I/O 位置，核选 [允许(Enable)] ，然后执行要调试的程序。

侦测子例程调用指令（BSR、JSR 或 BSRF）时，也就是在用户程序执行期间系统调用到专用地址时，模拟程序/调试程序会通过将 R0 和 R1 值用作参数来执行 I/O 处理。

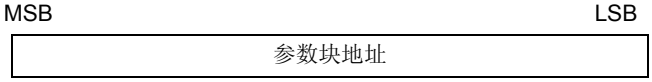
所以，发出系统调用前，请在用户程序中设定如下：

- 将函数代码（表 2.3）设定为 R0 寄存器



- 将参数块地址设定为 R1 寄存器

（对于参数块，参考每个函数的描述。）



- 预留参数块和 I/O 缓冲器区域

在存取参数块之每一个参数的情形下，在参数大小中执行 I/O 处理后，模拟程序/调试程序将会从紧随系统调用指令之后的指令继续模拟。

注意： 如果 JSR、BSR 或 BSRF 指令在系统调用时执行，紧随 JSR、BSR 或 BSRF 之后的指令将作为普通指令，而不是时隙指令。因此，任何作为普通指令时与作为时隙指令时运行结果不同的指令不能紧跟在 JSR，BSR 或 BSRF 之后使用。

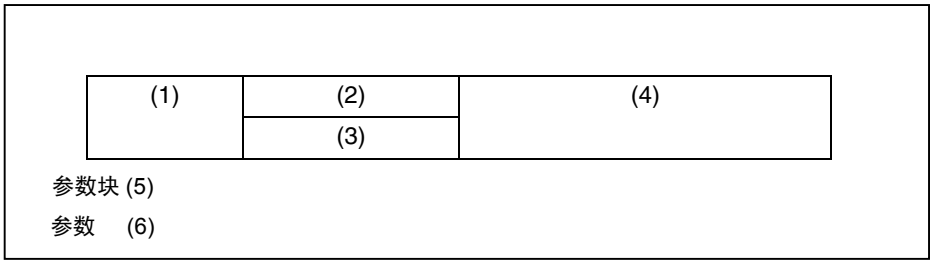


图 2.46 I/O 函数格式的描述

项目的内容如下：

(1) 相应于表 2.3 的数字

(2) 函数名称

(3) 函数代码

(4) I/O 概述

(5) I/O 参数块

(6) I/O 参数

1	GETC H'21	从标准输入设备输入一个字节。

参数块　　一个字节　　一个字节

+0 +2	输入缓冲器起始地址

参数　　• 输入缓冲器起始地址（输入）
写入输入数据的缓冲器的起始地址

2	PUTC H'22	输出一个字节到标准输出设备。

参数块　　一个字节　　一个字节

+0 +2	输出缓冲器起始地址

参数　　• 输出缓冲器起始地址（输入）
存储输出数据的缓冲器的起始地址

3	GETS H'23	从标准输入设备输入一行。

参数块

	一个字节	一个字节
+0 +2	输入缓冲器起始地址	

参数　　• 输入缓冲器起始地址（输入）
写入输入数据的缓冲器的起始地址

4	PUTS H'24	输出一行到标准输出设备。

参数块

	一个字节	一个字节
+0 +2	输出缓冲器起始地址	

参数　　• 输出缓冲器起始地址（输入）
存储输出数据的缓冲器的起始地址

5	FOPEN H'25	打开文件。

FOPEN 将会打开文件并返回文件编号。 进行此处理后，输入、输出或关闭文件时必须使用返回的文件编号。 最多可同时打开 256 个文件。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	打开模式	未用
+4 +6	文件名的起始地址	

参数　　• 返回值（输出）
0: 正常完成
-1: 错误
• 文件编号（输出）
打开后用于所有文件存取的编号
打开模式（输入）
H'00 "r"
H'01 "w"

H'02 "a"
H'03 "r+"
H'04 "w+"
H'05 "a+"
H'10 "rb"
H'11 "wb"
H'12 "ab"
H'13 "r+b"
H'14 "w+b"
H'15 "a+b"

这些模式说明如下：

“r”： 打开以读取。
“w”： 打开空的文件以写入。
“a”： 打开以附加（在文件结束处开始写入）。
“r+”： 打开以读取和写入。
“w+”： 打开空的文件以读取和写入。
“a+”： 打开以读取和附加。
“b”： 在二进制模式中打开。

- 文件名的起始地址（输入）

用于存储文件名的区域的起始地址

6	FCLOSE H'06	关闭文件。

参数块

	一个字节	一个字节
+0	返回值	文件编号

参数

- 返回值（输出）

0: 正常完成
-1: 错误

- 文件编号（输入）

文件打开时返回的编号

7	FGETC H'27	从文件输入一个字节。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	未用	
+4	输入缓冲器的起始地址	
+6		

参数

- 返回值（输出）

0: 正常完成
-1: 错误

- 文件编号（输入）

文件打开时返回的编号

• 输入缓冲器的起始地址（输入）

用于存储输入数据的缓冲器的起始地址

8	FPUTC H'28	输出一个字节到文件。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	未用	
+4	输出缓冲器的起始地址	
+6		

参数

• 返回值（输出）

0: 正常完成

-1: 错误

• 文件编号（输入）

文件打开时返回的编号

• 输出缓冲器的起始地址（输入）

用于存储输出数据的缓冲器的起始地址

9	FGETS H'29	从文件读取字符串数据。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	缓冲器大小	
+4	输入缓冲器的起始地址	
+6		

参数

• 返回值（输出）

0: 正常完成

-1: 侦测到 EOF

• 文件编号（输入）

文件打开时返回的编号

• 缓冲器大小（输入）

用于存储读取数据的区域的大小

（最多可存储 256 字节）

• 输入缓冲器的起始地址（输入）

用于存储输入数据的缓冲器的起始地址

10	FPUTS H'2A	将字符串数据写入文件。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	未用	
+4	输出缓冲器的起始地址	
+6		

参数

- 返回值（输出）

  0: 正常完成

  -1: 错误

- 文件编号（输入）

  文件打开时返回的编号

- 输出缓冲器的起始地址（输入）

  用于存储输出数据的缓冲器的起始地址

11	FEOF	检查文件的结束。
	H'0B	

参数块

	一个字节	一个字节
+0	返回值	文件编号

参数

- 返回值（输出）

  0: 文件指针不在 EOF

  -1: 侦测到 EOF

- 文件编号（输入）

  文件打开时返回的编号

12	FSEEK	将文件指针移到指定的位置。
	H'0C	

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	方向	未用
+4	偏移	
+6		

参数

- 返回值（输出）

  0: 正常完成

  -1: 错误

- 文件编号（输入）

  文件打开时返回的编号

- 方向（输入）

  0:“偏移”确定指定地址距文件起始地址的字节数

  1:“偏移”确定指定地址距当前文件指针的字节数

2:“偏移”确定指定地址距文件末尾的字节数

- 偏移（输入）

  指定地址距由方向参数指定的地址的字节数

13	FTELL H'0D	返回文件指针的当前位置。

参数块

	一个字节	一个字节
+0	返回值	文件编号
+2	未用	
+4	偏移	
+6		

参数

- 返回值（输出）

  0: 正常完成

  -1: 错误

- 文件编号（输入）

  文件打开时返回的编号

- 偏移（输出）

  文件指针的当前位置

  （从文件起始算起的字节）

以下显示标准输入和输出一个字符的实例（好像键盘输入）。

```
;------------------------------------------------------------------------
;
;    文件         : lowlvl.src
;    日期         : 2002 年 3 月 5 日，星期二
;    描述：低层程序
;    CPU 类型     :
;
;    本文件由“瑞萨工程生成程序”（版本.3.0）生成。
;
;------------------------------------------------------------------------

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                                lowlvl.src
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;              SH 系列模拟程序调试程序界面例程
;                              -输入/输出一个字符-
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      .EXPORT     _charput
      .EXPORT     _charget
SIM_IO:    .EQU      H'0000          ;指定 TRAP_ADDRESS
```

```
     .SECTION    P, CODE, ALIGN=4

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: 一个字符输出
;           C 程序界面:  charput(char)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
          MOV.L       O_PAR,R0          ; 设定输出缓冲器地址为 R0
          MOV.B       R4,@R0            ; 设定输出字符为缓冲器
          MOV.L       #O_PAR,R1         ; 设定参数块地址为 R1
          MOV.L       #H'01220000,R0    ; 指定函数代码 (PUTC)
          MOV.W       #SIM_IO,R2        ; 设定系统调用地址为 R2
          JSR         @R2
     NOP
     RTS
     NOP

          .ALIGN      4
O_PAR:                                  ; 参数块
          .DATA.L     OUT_BUF

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; _charget:一个字符输入
;       C 程序界面:  char charget(void)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .ALIGN      4
_charget:
          MOV.L       #I_PAR,R1         ; 设定参数块地址为 R1
          MOV.L       #H'01210000,R0    ; 指定函数代码 (GETC)
          MOV.W       #SIM_IO,R2        ; 设定系统调用地址为 R2
          JSR         @R2
     NOP
          MOV.L       I_PAR,R0          ; 设定输入缓冲器地址为 R0
          MOV.B       @R0,R0            ; 返回输入数据
     RTS
     NOP

          .ALIGN      4
I_PAR:                                  ; 参数块
          .DATA.L     IN_BUF
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;             I/O 缓冲器定义
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     .SECTION    B,DATA,ALIGN=4

OUT_BUF:
          .RES.L      1               ; 输出缓冲器
IN_BUF:
          .RES.L      1               ; 输入缓冲器

     .END
```

SuperH RISC Engine C/C++编译程序应用笔记

编译程序

第 3 节　编译程序

3.1　中断函数

3.1.1　中断函数的定义（无选项）

描述：

预处理程序指令 (#pragma) 可用来在 C 语言中创建中断函数。 使用“#pragma interrupt”来声明的函数会在函数处理前后保存/恢复用于函数内的所有寄存器（除了全局基址寄存器 GBR 和向量基址寄存器 VBR）。 因此，被中断的函数不需制定处理中断的规定。

格式：

#pragma interrupt（<函数名称>，[<函数名称>…]）

使用的实例：

声明了中断函数 handler1。 此函数从被中断的函数接收堆栈并加以使用，且在完成处理后，以 RTE 指令返回。

<未保存/恢复 GBR、VBR 时的情况>

C 语言代码

```
#pragma interrupt(handler1)    /* 声明中断函数 */
void handler1(void)
    {
        :                      /* 中断函数处理 */
        :
    }
```

扩展为汇编语言代码

```
                .EXPORT    _handler1
                .SECTION   P,CODE,ALIGN=4
_handler1:                          ; 函数：handler1
                    :           ; 保存工作寄存器
                    :           ; 中断函数处理
                    :           ; 恢复工作寄存器
                RTE
                NOP
                .END
```

<当存储或恢复 GBR 及 VBR 时>

C 语言代码

```
#pragma interrupt(handler1)
void handler1(void)
{
    void** save_vbr;              /* 定义 VBR 存储区域  */
    void* save_gbr;               /* 定义 GBR 存储区域  */
    save_vbr = get_vbr();         /* 保存 VBR           */
    save_gbr = get_gbr();         /* 保存 GBR           */
        :                         /* 中断函数处理       */
        :
    set_vbr(save_vbr);            /* 恢复 VBR           */
    set_gbr(save_gbr);            /* 恢复 GBR           */
}
```

扩展为汇编语言代码

```
          .EXPORT     handler1
          .SECTION    P,CODE,ALIGN=4
handler1:                       ; 函数:  handler1
                                ; 帧大小=8
           MOV.L      R5,@-R15
           STC        GBR,R5     ; 保存 GBR
           MOV.L      R4,@-R15
           STC        VBR,R4     ; 保存 VBR
               :                 ; 保存工作寄存器
               :                 ; 中断函数处理
               :                 ; 恢复工作寄存器
           LDC        R4,VBR     ; 恢复 VBR
           LDC        R5,GBR     ; 恢复 GBR
           MOV.L      @R15+,R4
           MOV.L      @R15+,R4
           RTE
           NOP
           .END
```

重要信息：

(1) 只有 void 数据类型可由中断函数返回。

实例：

```
#pragma interrupt(f1, f2)     /* 声明中断函数   */
void  f1(void){...}           /* 定义中断函数 f1 */
int   f2(void){...}           /* 定义中断函数 f2 */
```

中断函数 f1 的定义正确，但中断函数 f2 的定义产生错误。

(2) 唯一可在中断函数的定义中指定的存储器类说明符是 extern。 即使指定了 static，它将被处理为 extern。

(3) 被声明为中断函数的函数不可作为普通函数调用。 若被声明为中断函数的函数作为普通函数被调用，运行时运作将不被保证。

实例：

- test1.c 文件内容

```
#pragma interrupt(f1)       /* 声明中断函数       */
void f1(void){...}          /* 声明中断函数 f1  */
int  f2(){f1();}
```

- test2.c 文件内容

```
f3(){ f1(); }
```

在文件 test1.c 中，函数 f2 发生一项错误。在文件 test2.c 中，函数 f3 未发生错误，但是函数 f1 被解释为 extern int f1()，且运行时操作变得不稳定。

(4) 在中断发生时，SH-3、SH3-DSP、SH-4A 和 SH4AL-DSP 的操作和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU 和 SH2-DSP 不同，且需要中断处理程序。 下面显示中断处理程序的一则实例。

SH-3 中断处理程序的实例

```
;*********************************************************************
;   文件    :vhandler.src
;*********************************************************************
       .include  "env.inc"
       .include  "vect.inc"

IMASKclr:     .equ   H'FFFFFF0F
RBBLclr:      .equ   H'CFFFFFFF
MDRBBLset:        .equ   H'70000000
       .import       _RESET_Vectors
       .import       _INT_Vectors
       .import       _INT_MASK
;*********************************************************************
;*     宏定义
;*********************************************************************
                .macro PUSH_EXP_BASE_REG
        stc.l   ssr,@-r15               ; 保存 ssr
        stc.l   spc,@-r15               ; 保存 spc
        sts.l   pr,@-r15                ; 保存上下文寄存器
        stc.l   r7_bank,@-r15
        stc.l   r6_bank,@-r15
        stc.l   r5_bank,@-r15
```

```
        stc.l   r4_bank,@-r15
        stc.l   r3_bank,@-r15
        stc.l   r2_bank,@-r15
        stc.l   r1_bank,@-r15
        stc.l   r0_bank,@-r15
                .endm
;
                .macro POP_EXP_BASE_REG
        ldc.l   @r15+,r0_bank           ;恢复寄存器
        ldc.l   @r15+,r1_bank
        ldc.l   @r15+,r2_bank
        ldc.l   @r15+,r3_bank
        ldc.l   @r15+,r4_bank
        ldc.l   @r15+,r5_bank
        ldc.l   @r15+,r6_bank
        ldc.l   @r15+,r7_bank
        lds.l   @r15+,pr
        ldc.l   @r15+,spc
        ldc.l   @r15+,ssr
                .endm
;*********************************************************************
;       复位
;*********************************************************************
       .section  RSTHandler,code
_ResetHandler:
              mov.l  #EXPEVT,r0
              mov.l  @r0,r0
              shlr2  r0
              shlr   r0
              mov.l  #_RESET_Vectors,r1
              add    r1,r0
              mov.l  @r0,r0
              jmp    @r0
              nop
;*********************************************************************
;       异常中断
;*********************************************************************
       .section  INTHandler,code
       .export   INTHandlerPRG
```

```
INTHandlerPRG:
_ExpHandler:
              PUSH_EXP_BASE_REG
;
              mov.l  #EXPEVT,r0                ; 设定事件地址
              mov.l  @r0,r1                    ; 设定异常码
              mov.l  #_INT_Vectors,r0          ; 设定向量表地址
              add    #-(h'40),r1               ; 异常码 - h'40
              shlr2  r1
              shlr   r1
              mov.l  @(r0,r1),r3               ; 设定中断函数 addr
;
              mov.l  #_INT_MASK,r0             ; 中断屏蔽表 addr
              shlr2  r1
              mov.b  @(r0,r1),r1               ; 中断屏蔽
              extu.b r1,r1
;
              stc    sr,r0                     ; 保存 sr
              mov.l  #(RBBLclr&IMASKclr),r2    ; RB、BL、屏蔽 清除数据
              and    r2,r0                     ; 清除屏蔽数据
              or     r1,r0                     ; 设定中断屏蔽
              ldc    r0,ssr                    ; 设定当前状态
;
              ldc.l  r3,spc
              mov.l  #__int_term,r0            ; 设定中断终止
              lds    r0,pr
;
              rte
              nop
;
              .pool
;
;*********************************************************************
;       中断终止
;*********************************************************************
              .align 4
__int_term:
              mov.l   #MDRBBLset,r0            ; 设定 MD、BL、RB
              ldc.l  r0,sr                     ;
              POP_EXP_BASE_REG
              rte                              ; 返回
              nop
;
              .pool
;
```

```
;*********************************************************************
;       TLB 遗漏中断
;*********************************************************************
        .org    H'300
_TLBmissHandler:
                PUSH_EXP_BASE_REG
;
                mov.l   #EXPEVT,r0              ; 设定事件地址
                mov.l   @r0,r1                  ; 设定异常码
                mov.l   #_INT_Vectors,r0        ; 设定向量表地址
                add     #-(h'40),r1             ; 异常码 - h'40
                shlr2   r1
                shlr    r1
                mov.l   @(r0,r1),r3             ; 设定中断函数 addr
;
                mov.l   #_INT_MASK,r0           ; 中断屏蔽表 addr
                shlr2   r1
                mov.b   @(r0,r1),r1             ; 中断屏蔽
                extu.b  r1,r1
;
                stc     sr,r0                   ; 保存 sr
                mov.l   #(RBBLclr&IMASKclr),r2  ; RB、BL、屏蔽 清除数据
                and     r2,r0                   ; 清除屏蔽数据
                or      r1,r0                   ; 设定中断屏蔽
                ldc     r0,ssr                  ; 设定当前状态
;
                ldc.l   r3,spc
                mov.l   #__int_term,r0          ; 设定中断终止
                lds     r0,pr
;
                rte
                nop
;
                .pool
;
;*********************************************************************
;       IRQ
;*********************************************************************
        .org    H'500
_IRQHandler:
                PUSH_EXP_BASE_REG
;
                mov.l   #INTEVT,r0              ; 设定事件地址
                mov.l   @r0,r1                  ; 设定异常码
```

```
                mov.l   #_INT_Vectors,r0            ; 设定向量表地址
                add     #-(h'40),r1                 ; 异常码 - h'40
                shlr2   r1
                shlr    r1
                mov.l   @(r0,r1),r3                 ; 设定中断函数 addr
;
                mov.l   #_INT_MASK,r0               ; 中断屏蔽表 addr
                shlr2   r1
                mov.b   @(r0,r1),r1                 ; 中断屏蔽
                extu.b  r1,r1
;
                stc     sr,r0                       ; 保存 sr
                mov.l   #(RBBLclr&IMASKclr),r2      ; RB、BL、屏蔽 清除数据
                and     r2,r0                       ; 清除屏蔽数据
                or      r1,r0                       ; 设定中断屏蔽
                ldc     r0,ssr                      ; 设定当前状态
;
                ldc.l   r3,spc
                mov.l   #__int_term,r0              ; 设定中断终止
                lds     r0,pr
;
                rte
                nop
;
                .pool
                .end
```

注意： 在 SH-3 中断处理程序实例的表部分中，没有相应地址的地方应被放空。
在此例子中，将使用 RTE 指令以跳转到向量函数。 另外，在从向量函数返回时，控制被传递给终止例程，因此需要通过执行 RTS 来返回向量函数。
所以，在定义向量函数时，不要使用“#pragma interrrupt”。
包含文件“env.inc”和“vect.inc”会在创建 SH3 工程时由 HEW 自动生成。
在上表所述的 PUSH_EXP_BASE_REG 和 POP_EXP_BASE_REG 宏中，只有 R0-R7 库的寄存器由“stc.l rn_bank, @-R15”指令保存，及由“ldc.l @+R15, rn_bank”指令恢复。
可被“MOV”指令存取的一般寄存器未被保存或恢复。
若使用 SH-3、SH3-DSP、SH-4、SH-4A 或 SH4AL-DSP，当中断被接受时，SR 寄存器中的 RB 位被设定为 1。
所以若您在中断处理程序前后使用这些宏，只有 R0_BANK0 至 R7_BANK0 的寄存器被保存，而 R0_BANK1 至 R7_BANK1 的寄存器不会被保存。
因此即使您在中断处理程序的开头使用这些宏，只要程序运行的 RB 保持为 1，R0_BANK1 至 R7_BANK1 的寄存器将不会被保存，而这些寄存器的值将被损坏。
若使用这些宏，您必须在 RB=0 中断前运行程序，或修改这些宏以保存/恢复 R0_BANK1 至 R7_BANK1 的寄存器。

3.1.2 中断函数的定义（具有选项）

描述：

在中断函数定义中可用的选项为“指定堆栈切换”和“指定陷阱指令返回”，以及“指定寄存器库”。
通过在外部中断发生时使用“指定堆栈切换”，堆栈指针被切换到指定的地址，而堆栈被用于执行中断函数。在返回时，堆栈指针被返回到中断时的位置（图 3.1）。 在使用此选项时，必须为被中断函数的堆栈保留足够富余，以供中断函数使用。
通过使用“指定陷阱指令返回”选项，TRAPA 指令被用于返回。 若此选项未被指定，RTE 指令将被用于返回。
SH-2A 和 SH2A-FPU 具有内建寄存器库，用于在中断处理期间快速保存及恢复寄存器。
通过使用“指定寄存器库”选项，编译程序会生成用于保存和恢复寄存器的代码，假设寄存器库可以使用。
尤其，库的目标寄存器（R0 到 R14、GBR、MACH、MACL 和 PR）会在中断异常发生时被自动保存。 这抑制了保存代码在中断函数中的生成。
RESBANK 指令被用于恢复库的目标寄存器。

格式：

#pragma interrupt（<函数名称>[（<中断指定>）][,<函数名称>[（<中断指定>）]…]）

表 3.1 中断指定

编号	项目	格式	选项	指定
1	堆栈 切换指定	sp=	{ <变量> | &<变量> | <常数> | <变量>+<常数> | &<变量>+<常数> }	使用变量或常数指定新堆栈的地址 <变量>： 变量（指针类型） &<变量>： 变量地址（目标类型） <常数>： 常数
2	TRAP 指令 返回指定	tn=	<常数>	指定 TRAPA 指令的结束。 <常数>： 常数值（TRAP 向量号）
3	寄存器库指定	resbank	无	抑制下列寄存器的寄存器保存代码输出： R0 至 R14、GBR、MACH、MACL、PR 若“tn”未被指定，RESBANK 指令将在 RTE 指令前被生成。

使用的实例：

实例 1：

声明了中断函数 handler2。 此函数使用数组 STK 作为堆栈（图 3.1），并在处理完成时通过 TRAPA#63 指令返回控制。

C 语言代码

```
extern  int  STK[100];
int  *ptr  = STK + 100;
#pragma interrupt(handler2(sp=ptr, tn=63))
                                        /* 声明中断函数            */
void  handler2(void)
{
          :                             /* 描述中断函数处理  */
          :


}
```

扩展为汇编语言代码

```
          .IMPORT     _STK
          .EXPORT     _ptr
          .EXPORT     _handler2
          .SECTION    P,CODE,ALIGN=4
handler2:                               ; 函数： handler2
                                        ; 帧大小=4
          MOV.L       R0,@-R15
          MOV.L       L217,R0           ; _ptr
          MOV.L       @R0, R0
          MOV.L       R15,@-R0
          MOV      R0,R15
           :                            ; 保存工作寄存器
           :                            ; 中断函数处理
           :                            ; 恢复工作寄存器
          MOV.L       @R15+,R15
          MOV.L       @R15+,R0
          TRAPA       #63
L217 :
          .DATA.L     _ptr
          .SECTION    D,DATA,ALIGN=4
_ptr:                                   ; static: ptr
          .DATA.L     H'00000190+ STK
          .END
```

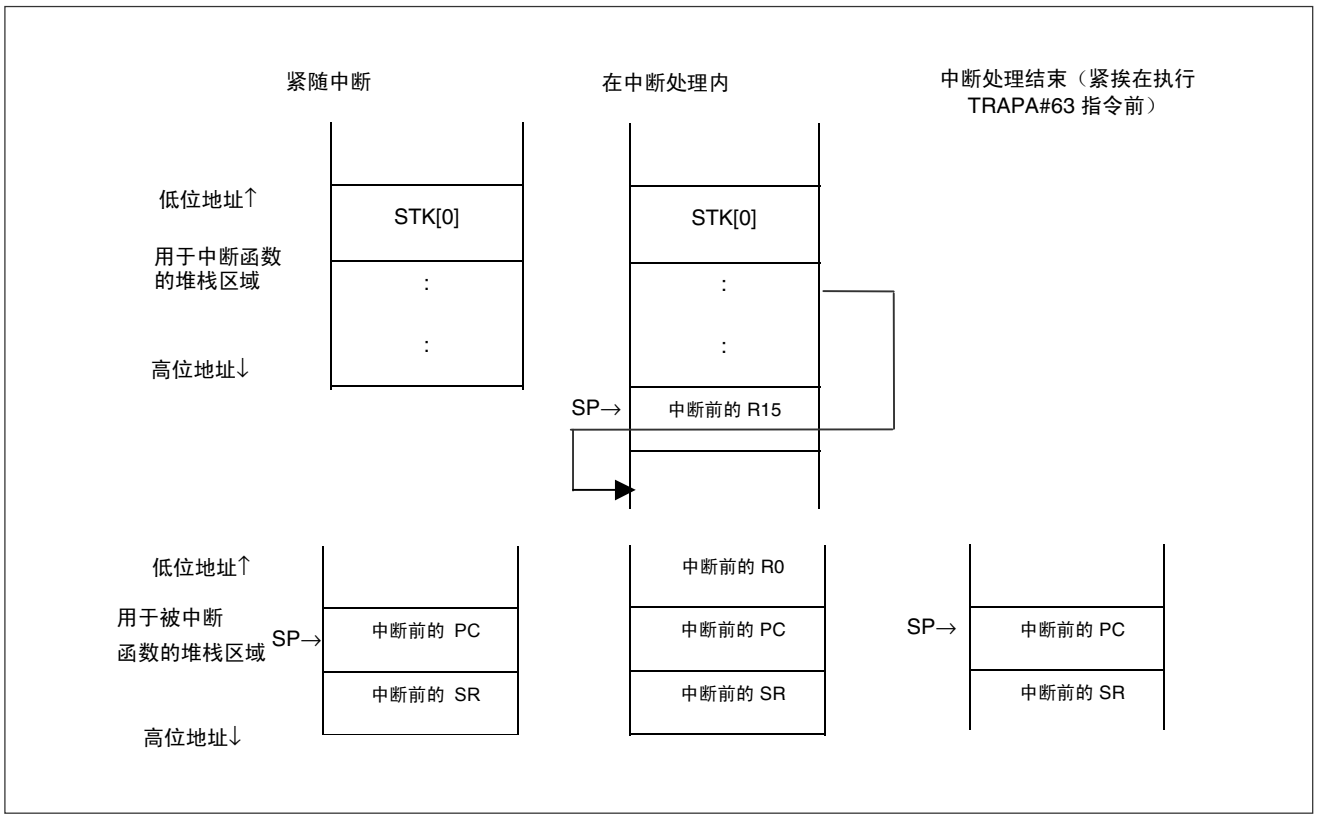


图 3.1 中断函数使用堆栈的实例

实例 2：

C 语言代码

```
#pragma interrupt(handler(resbank))
double flag1;
int  flag2;
void handler()
{
            flag1 = 1;
            flag2 = 1;
}
```

扩展为汇编语言代码（“-cpu=sh2afpu”被指定）

```
_handler:
                                        ; 不为库的目标工作寄存器输出
                                        ; 保存代码。
        FMOV.S      FR8,@-R15           ; 保存除了库的目标寄存器以外的工作
        FMOV.S      FR9,@-R15           ; 寄存器。
        MOVA        L11,R0
        MOV         #1,R4
        FMOV.S      @R0+,FR8
        MOV.L       L11+8,R1
        FMOV.S      @R0,FR9
        MOV.L       L11+12,R6
        FMOV.S      FR9,@-R1
        FMOV.S      FR8,@-R1
        MOV.L       R4,@R6
        FMOV.S      @R15+,FR9           ; 恢复除了库的目标寄存器以外的
        FMOV.S      @R15+,FR8           ; 工作寄存器。
        RESBANK                         ; 恢复库的目标寄存器。
        RTE
        NOP
```

重要信息：

(1) “resbank”指定在“cpu=sh2a|sh2afpu”被指定时有效。

(2) 寄存器库必须在中断发生于“resbank”被指定的函数中前被启用。

(3) 若您指定“resbank”和“tn”，寄存器保存代码或 RESBANK 指令都不会被生成。在这种情形下，指定 RESBANK 指令在陷阱例程中被生成。

(4) 当寄存器从指定了“resbank”的函数被恢复时，“#pragma global_register”中指定的变量值返回到中断前的值，即使它们在中断处理期间被更改。

3.1.3 创建向量表

描述：

向量表可通过下列设定以 C 语言创建。

(1) 预备了与向量表配合使用的数组，并为数组中各个元素指定了异常处理函数指针。
(2) 在编译文件后，向量表的起始地址被指定连接文件。

使用的实例：

C 语言代码： vect_table.c

```
extern void reset(void);            /* 加电复位处理函数   */
extern void warm reset(void);       /* 手动复位处理函数   */
extern void irq0(void);             /* IRQ0 中断处理函数  */
extern void irq1(void);             /* IRQ1 中断处理函数  */
          :
          :
void(* const vect_table[])(void) = {
              reset,                /* 加电复位的起始地址 */
              0,                    /* 加电复位的堆栈指针 */
              warm reset,           /* 手动复位的起始地址 */
              0,                    /* 手动复位的堆栈指针 */
               :
               :
              irq0,                 /* 向量号 64 */
              irq1,                 /* 向量号 65 */
               :
               :
};
```

批文件

```
shcΔ-section=c=VECTΔvect_table
shcΔresetΔwarm_resetΔirq0Δirq1 …
optlnkΔ-nooptΔ-subcommand=link.sub
```

link.sub 的内容如下。

```
sdebug
input vect_table
input reset
input warm_reset
input irq0,irq1
…
output sample.abs
start VECT/0,P,C,D/0400,B/0F000000
exit
```

通过编译 vect_table.c，可再定位的目标文件 vect table.obj 仅为初始化数据段 (VECT) 生成。
VECT 段被设定到起始地址 H'0 并和其他文件一起连接，以获取加载模块文件 sample.abs。

扩展为汇编语言代码： vect_table.src

```
            .IMPORT     reset
            .IMPORT     warm reset
            .IMPORT     irq0
            .IMPORT     irq1
            .EXPORT     vect table
            .SECTION    VECT,DATA,ALIGN=4
vect table:                                 ; static: vect table
            .DATA.L     reset
            .DATA.L     H'00000000
            .DATA.L     warm reset
            .DATA.L     H'00000000
                :
                :
            .DATA.L     irq0, irq1
                :
                :
            .END
```

重要信息：

(1) 在中断发生时，SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 的操作和 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU 和 SH2-DSP 不同，且需要中断处理程序。

(2) 向量表必须分配到定义的绝对地址，以便它在这里被创建为独立文件，但通过使用段切换函数，它可被包含在具有其他模块的文件中。 有关详情，请参考第 3.7 节“段名称指定”。

3.2 内建函数

表 3.2 中显示的内建函数是为启用以 C 表示 SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 的固有指令而提供。 当使用这些内建函数时，必须包含标准标头文件“machine.h”、“smachine.h”或“umachine.h”。
<machine.h> 的内容根据 SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 的执行模式划分如下：

(1) <machine.h>: 所有内建函数
(2) <smachine.h>: 仅能在有权限的模式中使用的内建函数
(3) <umachine.h>: (2) 所包含以外的内建函数

表 3.2 内建函数列表 (1)

编号	项目	函数	可用的 CPU	可用的执行模式	参阅节
1	状态寄存器 (SR)	设定 SR。	所有 CPU	仅限有权限的模式	3.2.1
2		参考 SR。			
3		设定中断屏蔽。			
4		参考中断屏蔽。			
5	向量基址寄存器 (VBR)	设定 VBR。	所有 CPU	仅限有权限的模式	3.2.2
6		参考 VBR。			
7	全局基址寄存器 (GBR)	设定 GBR。	所有 CPU	所有执行模式	3.2.3
8		参考 GBR。			3.2.4
9		在由 GBR 和偏移指定的地址参考字节数据。			
10		在由 GBR 和偏移指定的地址参考字数据。			
11		在由 GBR 和偏移指定的地址参考长字数据。			
12		在由 GBR 和偏移指定的地址设定字节数据。			
13		在由 GBR 和偏移指定的地址设定字数据。			

表 3.2 内建函数列表 (2)

编号	项目	函数	可用的 CPU	可用的执行模式	参阅节
14		在由 GBR 和偏移指定的地址设定长字数据。			
15		在由 GBR 指定的地址 AND 字节数据。			
16		在由 GBR 指定的地址 OR 字节数据。			
17		在由 GBR 指定的地址 XOR 字节数据。			
18		在由 GBR 指定的地址测试字节数据。			
19	系统控制	SLEEP 指令	所有 CPU	仅限有权限的模式	3.2.5
20		TAS 指令		所有执行模式	
21		TRAPA 指令			
22	乘法累加运算	字单位中的乘法和累加	所有 CPU	所有执行模式	3.2.6
23		长字单位中的乘法和累加	不包括 cpu=sh1		
24		字单位中环形缓冲兼容的乘法和累加	所有 CPU	所有执行模式	3.2.7
25		长字单位中环形缓冲兼容的乘法和累加	不包括 cpu=sh1		
26	64 位乘法	带符号 64 位数据乘法的高 32 位	不包括 cpu=sh1	所有执行模式	3.2.8
27		带符号 64 位数据乘法的低 32 位			
28		无符号 64 位数据乘法的高 32 位	不包括 cpu=sh1	所有执行模式	3.2.9
29		无符号 64 位数据乘法的低 32 位			

表 3.2　内建函数列表 (3)

编号	项目	函数	可用的 CPU	可用的执行模式	参阅节
30	交换高及低数据位	SWAP.B 指令	所有 CPU	所有执行模式	3.2.10
31		SWAP.W 指令			
32		交换 4 字节数据的高及低位。			
33	系统调用	执行系统调用	所有 CPU	所有执行模式	3.2.11
34	预取指令	预取指令	仅指定了 cpu=sh2a、sh2afpu、sh3、sh3dsp、sh4、sh4a、sh4aldsp 时	所有执行模式	3.2.12
35	LDTLB 指令	扩展 LDTLB。	仅在指定了 cpu=sh3、sh3dsp、sh4、sh4a、sh4aldsp 时	仅限有权限的模式	3.2.13
36	NOP 指令	扩展 NOP。	所有 CPU	所有执行模式	3.2.14
37	浮点	设定 FPSCR。	仅在指定了 cpu=sh2e、sh2afpu、sh4、sh4a 时	所有执行模式	3.2.15
38	单位	参考 FPSCR。			
39	单精度浮点	FIPR 指令	仅在指定了 cpu=sh4、sh4a 时	所有执行模式	
40	向量运算	FTRV 指令			
41		4 维向量转换为 4 x 4 矩阵和 4 维向量加法。			
42		4 维向量转换为 4 x 4 矩阵和 4 维向量减法。			
43		执行 4 维向量加法。	仅在指定了 cpu=sh2afpu、sh4、sh4a 时		
44		执行 4 维向量减法。			
45		执行 4x4 矩阵乘法。	仅在指定了 cpu=sh4、sh4a 时		
46		执行 4x4 矩阵乘法及加法。			
47		执行 4x4 矩阵乘法及减法。			
48	扩展寄存器存取	加载数据到扩展寄存器。	仅在指定了 cpu=sh4、sh4a 时	所有执行模式	3.2.16
49		从扩展寄存器恢复数据。			

表 3.2 内建函数列表 (4)

编号	函数	描述	可用的 CPU	可用的执行模式	参阅节
50	DSP 指令	绝对值	在指定了 sh2dsp、sh3dsp、sh4aldsp 及 dspc 时	所有执行模式	3.2.17
51		MSB 侦测			
52		算术移位			
53		舍入运算			
54		位模式复制			
55		取模寻址设定		仅限有权限的模式	
56		取模寻址取消			
57		CS 位设定（DSR 寄存器）		所有执行模式	
58	正弦及余弦	正弦及余弦计算	仅在指定了 cpu=sh4a 时	所有执行模式	3.2.18
59	平方根的倒数	平方根的倒数	仅在指定了 cpu=sh4a 时	所有执行模式	3.2.19
60	指令高速缓存失效	指令高速缓存块失效	仅在指定了 cpu=sh4a、sh4aldsp 时	所有执行模式	3.2.20
61	高速缓存块运算	高速缓存块失效	仅在指定了 cpu=sh4a、sh4aldsp 时	所有执行模式	3.2.21
62		高速缓存块清除			
63		高速缓存块回写			
64	指令高速缓存预取	指令高速缓存块的预取	仅在指定了 cpu=sh4a、sh4aldsp 时	所有执行模式	3.2.22
65	系统同步化	同步化数据运算。	仅在指定了 cpu=sh4a、sh4aldsp 时	所有执行模式	3.2.23

表 3.2 内建函数列表 (5)

编号	函数	描述	可用的 CPU	可用的执行模式	参阅节
66	参考及设定 T 位	参考 T 位。	所有 CPU	所有执行模式	3.2.24
67		清除 T 位。			
68		设定 T 位。			
69	切除连接的寄存器的中部	从连接的 64 位数据切除中间 32 位。	所有 CPU	所有执行模式	3.2.25
70	使用进位的加法	相加两个数据项目及 T 位，并将进位应用到 T 位。	所有 CPU	所有执行模式	3.2.26
71		相加两个数据项目及 T 位，并参考进位。			
72		相加两个数据项目，并将进位应用到 T 位。			
73		相加两个数据项目，并参考进位。			
74	具有借位的减法	从 data1 减去 data2 及 T 位，并将借位应用到 T 位。	所有 CPU	所有执行模式	3.2.27
75		从 data1 减去 data2 及 T 位，并参考借位。			
76		从 data1 减去 data2，并将借位应用到 T 位。			
77		从 data1 减去 data2，并参考借位。			
78	符号倒置	从 0 减去 data 及 T 位，并将借位应用到 T 位。	所有 CPU	所有执行模式	3.2.28
79	一位除法	执行 data2 对 data1 的一位除法，并将结果应用到 T 位。	所有 CPU	所有执行模式	3.2.29
80		执行 data2 对 data1 的带符号除法初始化，并参考 T 位。			
81		执行无符号除法初始化。			

表 3.2 内建函数列表 (6)

编号	函数	描述	可用的 CPU	可用的执行模式	参阅节
82	旋转	将数据向左旋转一位，然后将移到操作数外的位应用到 T 位。	所有 CPU	所有执行模式	3.2.30
83		将数据向右旋转一位，然后将移到操作数外的位应用到 T 位。			
84		将数据向左旋转一位，包括 T 位，然后将移到操作数外的位应用到 T 位。			
85		将数据向右旋转一位，包括 T 位，然后将移到操作数外的位应用到 T 位。			
86	移位	将数据向左移一位，然后将移到操作数外的位应用到 T 位。	所有 CPU	所有执行模式	3.2.31
87		将数据逻辑右移一位，然后将移到操作数外的位应用到 T 位。			
88		将数据算术右移一位，然后将移到操作数外的位应用到 T 位。			
89	饱和运算	带符号一字节数据的饱和运算	仅在指定了 cpu=sh2a、sh2afpu 时	所有执行模式	3.2.32
90		带符号二字节数据的饱和运算			
91		无符号一字节数据的饱和运算			
92		无符号二字节数据的饱和运算			
93	参考和设定 TBR	设定 TBR 中的数据。	仅在指定了 cpu=sh2a、sh2afpu 时	所有执行模式	3.2.33
94		参考 TBR 的值。			

3.2.1 设定和参考状态寄存器

描述：

表 3.3 中所显示的函数是为用于状态寄存器的设定及参考而提供。

表 3.3 状态寄存器的内建函数列表

编号	函数	格式	描述
1	状态寄存器设定	void set_cr(int cr)	设定状态寄存器中的 cr（32 位）
2	状态寄存器参考	int get_cr(void)	参考状态寄存器。
3	设定中断屏蔽	void set_imask(int mask)	设定中断屏蔽（4 位）中的屏蔽（4 位）。
4	参考中断屏蔽	int get_imask(void)	参考中断屏蔽状态（4 位）

使用的实例：

通过将中断屏蔽设定到最高值（15），函数 func1 将在处理期间禁用外部中断。 在处理完成后，将恢复原始中断屏蔽级别。

C 语言代码

```
#include <machine.h>

void  func1(void)
{
    int   mask;             /* 为中断屏蔽级别定义存储区。   */

    mask = get_imask();     /* 保存中断屏蔽级别             */
    set_imask(15);          /* 将中断屏蔽级别设定到 15      */
        :                      /* 执行禁用中断的处理              */
        :
    set_imask(mask);           /* 恢复中断屏蔽级别            */
}
```

扩展为汇编语言代码

```
            .EXPORT     func1
            .SECTION    P,CODE,ALIGN=4
func1:                                  ; 函数：func1
                                        ; 帧大小=0
            MOV.W       L216,R3         ; H'FF0F
            STC         SR,R0
            SHLR2       R0
            SHLR2       R0
            AND         #15,R0
            MOV         R0,R4
            STC         SR,R0
            AND         R3,R0
            OR          #240,R0
            LDC         R0,SR
             :
             :
            MOV         R4,R0
            AND         #15,R0
            SHLL2       R0
            SHLL2       R0
            STC         SR,R2
            MOV         R3,R1
            AND         R1,R2
            OR          R2,R0
            LDC         R0,SR
            RTS
            NOP
L216:
            .DATA.W     H'FF0F
            .END
```

3.2.2 设定及参考向量基址寄存器

描述：

表 3.4 显示为设定和读取向量基址寄存器所提供的函数。

表 3.4 向量基址寄存器的内建函数列表

编号	函数	格式	描述
1	向量基址寄存器设定	void set_vbr(void **base)	设定向量基址寄存器中的 **base （32 位）。
2	向量基址寄存器参考	void **get_vbr(void)	参考向量基址寄存器。

使用的实例：

复位后，向量基址寄存器 (VBR) 被初始化为 0。若向量表在 0 以外的地址开始，接下来的函数应以手动复位设定到起始地址 (H'00000008)，以便当在系统启动执行了手动复位时，异常处理可以使用向量表来执行。

C 语言代码

```
#include <machine.h>
#define VBR  0x0000FC00          /* 定义向量表的起始地址     */

void  warm reset(void)
{
    set_vbr((void **)VBR);
                                 /* 设定向量基址寄存器到向量表的  */
                                 /* 起始地址                      */
}
```

扩展为汇编语言代码

```
        .EXPORT       warm_reset
        .SECTION      P,CODE,ALIGN=4
warm reset:                             ; 函数：warm reset
                                        ; 帧大小=0
        MOV.L         L215,R3           ; H'0000FC00
        LDC           R3,VBR
        RTS
        NOP
L215:
        .DATA.L       H'0000FC00
        .END
```

重要信息：

向量基址寄存器应仅在设定向量表后更改。 若顺序相反，同时外部中断在设定向量表时发生，将会造成系统崩溃。

3.2.3 存取 I/O 寄存器(1)

描述：

表 3.5 中显示的函数是为用于操控全局基址寄存器 (GBR) 以存取 I/O 寄存器而提供。

表 3.5 与全局基址寄存器一同使用的内建函数列表

编号	函数	格式	描述
1	全局基址寄存器设定	void set_gbr(void **base)	设定全局基址寄存器中的 *base（32 位）
2	全局基址寄存器参考	int *get_gbr(void)	参考全局基址寄存器
3	在由 GBR 和偏移指定的地址参考字节数据。	unsigned char gbr_read_byte(int offset)	在由相加 GBR 和偏移指定的地址参考字节数据（8 位）
4	在由 GBR 和偏移指定的地址参考字数据。	unsigned short gbr_read_word(int offset)	在由相加 GBR 和偏移指定的地址参考字数据（16 位）
5	在由 GBR 和偏移指定的地址参考长字数据。	unsigned long gbr_read_long(int offset)	在由相加 GBR 和偏移指定的地址参考长字数据（32 位）
6	在由 GBR 和偏移指定的地址设定字节数据。	void gbr_write_byte(int offset,unsigned char data)	在由相加 GBR 和偏移指定的地址指定字数据（8 位）
7	在由 GBR 和偏移指定的地址设定字数据。	void gbr_write_word(int offset,unsigned short data)	在由相加 GBR 和偏移指定的地址指定字数据（16 位）
8	在由 GBR 和偏移指定的地址设定长字数据。	void gbr_write_long(int offset,unsigned long data)	在由相加 GBR 和偏移指定的地址指定字数据（32 位）
9	在由 GBR 和偏移指定的地址进行 AND 字节数据运算。	void gbr_and_byte(int offset,unsigned char mask)	在由 GBR 和偏移指定的地址 AND 屏蔽和字节数据，然后以偏移设定结果。
10	在由 GBR 和偏移指定的地址进行 OR 字节数据运算。	void gbr_or_byte(int offset,unsigned char mask)	在由 GBR 和偏移指定的地址 OR 屏蔽和字节数据，然后以偏移设定结果。
11	在由 GBR 和偏移指定的地址进行 XOR 字节数据运算。	void gbr_xor_byte(int offset,unsigned char mask)	在由 GBR 和偏移指定的地址 XOR 屏蔽和字节数据，然后以偏移设定结果。
12	在由 GBR 和偏移指定的地址测试字节数据。	void gbr_tst_byte(int offset,unsigned char mask)	在由 GBR 和偏移指定的地址 AND 屏蔽和字节数据，将结果和 0 比较，然后根据比较结果来设定或清除 T 位。

注意： (1) 当存取大小是字时，基址应设为 2 的倍数，当存取大小是长字时，它应被设定为 4 的倍数。
(2) 对于表 3.5 中编号 3 到 8，偏移必须是常数。 当存取大小是字节时，将可以指定最大 +255 字节的偏移，当存取大小是字时，则可以指定最大 +510 字节的偏移。此外，当存取大小是长字时，将可以指定最大 +1020 字节的偏移。
(3) 屏蔽必须是常数。 屏蔽可以在 0 到 +255 的范围内指定。
(4) 全局基址寄存器是控制寄存器，因此 C/C++ 编译程序不在函数入口和出口保存及恢复值。当更改全局基址寄存器值时，用户必须在函数入口及出口保存和恢复寄存器值。
(5) 这项函数在指定了 gbr=auto 时无效。

使用的实例：

以下是使用 SH-1 内部 16 位集成定时器脉冲元件的定时器驱动程序实例。

C 语言代码

```
#include <machine.h>
#define IOBASE 0x05fffec0           /* 定义 I/O 基址地址              */
#define TSR   (0x05ffff07 - IOBASE)
                                    /* 清除寄存器的比较匹配           */
                                    /* 标志                           */
#define TSRCLR (unsigned char)0xf8
                                    /* 清除定时器状态标志寄存器的     */
                                    /* 比较匹配标志                   */
void  tmrhdr(void)
{
    void    *gbrsave;               /* 为全局基址寄存器定义堆栈       */
                                    /* 区域                           */
    gbrsave = get gbr();            /* 保存全局基址寄存器             */
    set gbr((void *)IOBASE);
                                    /* 在全局寄存器中指定 I/O 基址    */
                                    /* 寄存器                         */
    gbr read byte(TSR);             /* 读取定时器状态标志寄存器以     */
                                    /* 将它清除                  */
    gbr and byte(TSR, TSRCLR);
                                    /* 清除定时器状态标志寄存器的     */
                                    /* 比较匹配标志                   */
    set gbr(gbrsave);               /* 恢复全局基址寄存器             */
}
```

扩展为汇编语言代码

```
.EXPORT     _tmrhdr
            .SECTION    P,CODE,ALIGN=4
_tmrhdr:                                ; 函数:  tmrhdr
                                        ; 帧大小=4
            ADD         #-4,R15
            STC         GBR,R1
            MOV.L       L11,R5          ; H'05FFFEC0
            MOV.L       R1,@R15
            LDC         R5,GBR
            MOV.B       @(71,GBR),R0
            MOV         #71,R0          ; H'00000047
            AND.B       #248,@(R0,GBR)
            MOV.L       @R15,R2
            LDC         R2,GBR
            RTS
            ADD         #4,R15
L11:
            .DATA.L     H'05FFFEC0
```

3.2.4 存取 I/O 寄存器(2)

使用的实例：

通过使用标准程序库 offsetof，消除了预先计算全局基址寄存器相对偏移的需要。

C 语言代码

```
#include <stddef.h>
#include <machine.h>
struct IOTBL{
   char  cdata1;                /* 偏移 0   */
   char  cdata2;                /* 偏移 1   */
   char  cdata3;                /* 偏移 2   */
   short sdata1;                /* 偏移 4   */
   int   idata1;                /* 偏移 8   */
   int   idata2;                /* 偏移 12  */
} table;
void f (void)
{
     void      *gbrsave;        /* 为全局基址寄存器定义堆栈       */
                                /* 区域                           */
     gbrsave = get gbr();       /* 保存全局基址寄存器             */
     set gbr(&table);           /* 设定全局基址寄存器中的 table   */
                                /* 起始地址                       */
               :
               :
     gbr and byte(offsetof(struct IOTBL, cdata2), 0x10);
                                /* AND table.cdata2 和 0x10，然后 */
                                /* 在 table.cdata2 中保存结果     */
               :
               :
     set gbr(gbrsave);          /* 恢复全局基址寄存器             */
}
```

扩展为汇编语言代码

```
      .EXPORT     table
      .EXPORT     f
      .SECTION    P,CODE,ALIGN=4
f:                                          ; 函数: f
                                            ; 帧大小=0
      MOV.L          L217+2,R3                ; table
      MOV         #1,R0
      STC         GBR,R4
      LDC         R3,GBR
         :
         :
      AND.B       #16,@(R0,GBR)
         :
         :
      RTS
      LDC         R4,GBR
L217 :
      .RES.W      1
      .DATA.L     table
      .SECTION    B,DATA,ALIGN=4
table:                                      ; static: table
      .RES.L      4
      .END
```

3.2.5 系统控制

描述：

表 3.6 显示作为 Renesas Technology SuerH RISC engine 家族的特殊指令提供的函数。

表 3.6 特殊指令的内建函数列表

编号	函数	格式	描述
1	SLEEP 指令	void sleep(void)	扩展为 SLEEP 指令
2	TAS 指令	int tas(char *addr)	扩展为 TAS.B @addr
3	TRAPA 指令	int trapa(int trap_no)	扩展为 TRAPA #trap_no
4	OS 系统调用	-	参考第 3.2.11 节
5	PREF 指令	-	参考第 3.2.12 节

注意： (1) 在表中，trap_no 必须是常数。
(2) 内建函数 trapa 从 C 语言程序启动中断函数。 创建调用目标函数作为中断函数。

使用的实例：

SLEEP 指令被发出以使 CPU 进入低功耗模式。 在低功耗模式中，CPU 的内部状态被保存，紧接着执行的指令被停止，同时系统等待中断请求。 当中断发生时，CPU 将退出低功耗模式。

C 语言代码

```
#include <machine.h>
void  func(void)
{
        :
        :
    sleep();                          /* 发出 SLEEP 指令 */
        :
        :
}
```

扩展为汇编语言代码

```
        .EXPORT     func
        .SECTION    P,CODE,ALIGN=4
func:                                 ; 函数：func
                                      ; 帧大小=0
        SLEEP
        RTS
        NOP
        .END
```

3.2.6 乘法累加运算 (1)

描述：

表 3.7 显示为乘法累加运算提供的函数。

表 3.7 乘法累加运算的内建函数列表

编号	函数	格式	描述
1	字单位的乘法和累加	int macw(short *ptr1, short *ptr2,unsigned int count)	按计数指定的次数在字数据 *ptr1 （16 位）和字数据 *ptr2 （16 位）之间执行乘法和累加
2	长字单位的乘法和累加	int macl(int *ptr1, int *ptr2,unsigned int count)	按计数指定的次数在长字数据 *ptr1 （32 位）和长字数据 *ptr2 （32 位）之间执行乘法和累加

字的乘法累加函数 macw 在所有的 CPU SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 上被支持。 长字乘法累加函数 mac1 在除了 SH-1 以外的 CPU 上被支持。
内建的乘法累加函数不执行参数检查。对于字运算，乘法累加运算的数据表应由二字节对齐，长字运算则由四字节对齐。

使用的实例：

一项乘法累加运算被执行。 若执行次数在 32 或以下，它们将通过重复 MAC 指令来执行，若执行次数在 33 或以上，或执行次数是变量，MAC 指令将被循环。

C 语言代码

```
#include <machine.h>
short a[SIZE];
short b[SIZE];

Void  func(void)
{
      int x;
            :
            :
      x = macw(a,b,SIZE);
            :
            :
}
```

```
a[0]          *    b[0]          +
a[1]          *    b[1]          +
a[2]          *    b[2]          +
  :                    :          +
a[SIZE-2]     *    b[SIZE-2]     +
a[SIZE-1]     *    b[SIZE-1]
```

扩展为汇编语言代码

- 大小为 32 的情形： 通过重复执行 MAC 指令来达到

```
        .EXPORT  func
        .SECTION P,CODE,ALIGN=4
func:                                   ; 函数：func
                                        ; 帧大小=8
        STS.L    MACH,@-R15
        STS.L    MACL,@-R15
        MOV.L    L217+2,R3              ; b
        CLRMAC
        MOV.L    L218+6,R2              ; a
        MAC.W    @R2+,@R3+
          :                             ; 为大小重复
          :
        STS      MACL,R0
        LDS.L    @R15+,MACL
        RTS
        LDS.L    @R15+,MACH
L218 :
       .RES.W    1
       .DATA.L   b
       .DATA.L   a
```

- 大小 > 32 或变量的情形：通过 MAC 指令的循环函数来达到

```
        .EXPORT     func
        .SECTION    P,CODE,ALIGN=4
func:                                       ; 函数：func
                                            ; 帧大小=8
        STS.L       MACH,@-R15
        MOV         #33,R3
        STS.L       MACL,@-R15
        TST         R3,R3
        CLRMAC
        BT          L218
        MOV.L       L220+2,R2               ; b
        SHLL        R3
        MOV.L       L220+6,R1               ; a
        ADD         R1,R3
L219 :
        MAC.W       @R1+,@R2+
        CMP/HI      R1,R3
        BT          L219
L218 :
        STS         MACL,R0
        LDS.L       @R15+,MACL
        RTS
        LDS.L       @R15+,MACH
L220 :
        .RES.W      1
        .DATA.L     b
        .DATA.L     a
```

3.2.7 乘法累加运算 (2)

描述：

表 3.8 显示为环形缓冲兼容的乘法累加运算而提供的函数。

表 3.8 环形缓冲兼容之乘法累加运算的内建函数列表

编号	函数	格式	描述
1	字单位中环形缓冲兼容的乘法和累加	int macwl(short *ptr1, short *ptr2,unsigned int count, unsigned int mask)	按计数指定的次数在字数据 *ptr1（16 位）和字数据 *ptr2 （16 位）之间执行乘法累加
2	长字单位中环形缓冲兼容的乘法和累加	int macll(int *ptr1, int *ptr2,unsigned int count, unsigned int mask)	按计数指定的次数在长字数据 *ptr1（32 位）和长字数据 *ptr2 （32 位）之间执行乘法和累加

环形缓冲兼容的字乘法累加函数 macw1 在所有的 CPU SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A，和 SH4AL-DSP，在 SH-2、SH-2E、SH-3，和 SH-4 中被支持。环形缓冲兼容的长字乘法累加函数 mac11 在 SH-1 以外的 CPU 中被支持。
内建的环形缓冲兼容乘法累加函数不执行参数检查。若是字乘法累加运算，第一个参数应由二字节对齐，对于长字运算应为四字节，第二个参数应以环形缓冲屏蔽大小的两倍对齐。

使用的实例：

一项环形缓冲兼容的乘法累加运算被执行。 第二个参数必须以环形缓冲大小的两倍对齐，因此使用个别的文件。

C 语言代码：macwl.c

```
#include <machine.h>

short a[SIZE];
extern short b[];

void  func(void)
{
     int x;
         :
         :
     x = macwl(a,b,SIZE,~0x10);

         :
         :

}
```

```
        a[0]          *     b[0]        +
        a[1]          *     b[1]        +
          :                   :         +
        a[7]          *     b[7]        +
        a[8]          *     b[0]        +
        a[9]          *     b[1]        +
          :                   :         +
      a[15]           *     b[7]        +
          :                   :         +
    a[SIZE-8]         *     b[0]        +
    a[SIZE-7]         *     b[1]        +
          :                   :         +
    a[SIZE-1]         *     b[7]
```

扩展为汇编语言代码：macwl.src

```
            .IMPORT  b
            .EXPORT  a
            .EXPORT  func
            .SECTION P,CODE,ALIGN=4
func:                                       ; 函数：func
                                            ; 帧大小=8
            STS.L        MACH,@-R15
            MOV          #33,R3
            STS.L        MACL,@-R15
            TST          R3,R3
            CLRMAC
            BT           L218
            MOV.L        L220+6,R1          ; b
            SHLL         R3
            MOV.L        L220+6,R4          ; a
            MOV          #-17,R2
            ADD          R4,R3
L219 :
            MAC.W        @R4+,@R1+
            AND          R2,R1
            CMP/HI       R4,R3
            BT           L219
L218 :
            STS          MACL,R0
            LDS.L        @R15+,MACL
            RTS
            LDS.L        @R15+,MACH
L220 :
            .RES.W       1
            .DATA.L      b
            .DATA.L      a
            .SECTION     B,DATA,ALIGN=4
a :                                         ; static: a
            .RES.W       33
            .END
```

3.2.8 64 位乘法 (1)

表 3.9 显示为带符号 64 位乘法提供的函数。

表 3.9 带符号 64 位乘法的内建函数列表

编号	函数	格式	描述
1	带符号 64 位数据乘法的高 32 位	long dmuls_h(long data1, long data2)	带符号的 32 位 x 带符号的 32 位执行带符号 64 数据的乘法，并返回一个高 32 位的结果
2	带符号 64 位数据乘法的低 32 位	long dmuls_l(long data1, long data2)	带符号的 32 位 x 带符号的 32 位执行带符号 64 数据的乘法，并返回一个低 32 位的结果

带符号 64 位数据的乘法被 SH-1 以外的所有 CPU 支持。

使用的实例：

C 语言代码

```
#include <machine.h>
extern long data1,data2;
extern long result;
void main( void )
{
    result = dmuls h(data1,data2);
                                        /* 执行带符号 64 位数据的乘法 */
}
```

扩展为汇编语言代码

```
            .IMPORT     _result
            .IMPORT     _data1
            .IMPORT     _data2
            .EXPORT     _main
            .SECTION     P,CODE,ALIGN=4
_main:                                  ; 函数：main
                                        ; 帧大小=8
            STS.L       MACL,@-R15
            STS.L       MACH,@-R15
            MOV.L       L11+2,R2        ; _data1
            MOV.L       L11+6,R5        ; _data2
            MOV.L       @R2,R6
            MOV.L       @R5,R2
            DMULS.L     R6,R2
            MOV.L       L11+10,R6       ; _result
            STS         MACH,R2
```

```
MOV.L       R2,@R6
            LDS.L       @R15+,MACH
            RTS
            LDS.L       @R15+,MACL
L11:
            .RES.W      1
            .DATA.L     _data1
            .DATA.L     _data2
            .DATA.L     _result
            .END
```

3.2.9 64 位乘法 (2)

表 3.10 显示为无符号 64 位乘法提供的函数。

表 3.10 无符号 64 位乘法的内建函数列表

编号	函数	格式	描述
1	无符号 64 位数据乘法的高 32 位	long dmulu_h(long data1, long data2)	无符号的 32 位 x 无符号的 32 位执行无符号 64 数据的乘法，并返回一个高 32 位的结果
2	无符号 64 位数据乘法的低 32 位	long dmulu_l(long data1, long data2)	无符号的 32 位 x 无符号的 32 位执行无符号 64 数据的乘法，并返回一个低 32 位的结果

无符号 64 位数据的乘法被 SH-1 以外的所有 CPU 支持。

使用的实例：

C 语言代码

```
#include <machine.h>
extern long data1,data2;
extern long result;
void main( void )
{
  result = dmulu h(data1,data2);
                                        /* 执行带符号 64 位数据的乘法 */
}
```

扩展为汇编语言代码

```
            .IMPORT     result
            .IMPORT     data1
            .IMPORT     data2
            .EXPORT     main
            .SECTION    P,CODE,ALIGN=4
_main:                                  ; 函数：main
                                        ; 帧大小=8
            STS.L       MACL,@-R15
            STS.L       MACH,@-R15
            MOV.L       L11+2,R2        ; _data1
            MOV.L       L11+6,R5        ; _data2
            MOV.L       @R2,R6
            MOV.L       @R5,R2
            DMULU.L     R6,R2
            MOV.L       L11+10,R6       ; _result
            STS         MACH,R2
```

```
            MOV.L       R2,@R6
            LDS.L       @R15+,MACH
            RTS
            LDS.L       @R15+,MACL
L11:
            .RES.W      1
            .DATA.L     data1
            .DATA.L     data2
            .DATA.L     result
            .END
```

3.2.10 交换高位及低位数据

表 3.11 显示为交换高位及低位数据所提供的函数。

表 3.11 环形缓冲兼容之乘法累加运算的内建函数列表

编号	函数	格式	描述
1	SWAP.B 指令	unsigned short swapb(unsigned short data)	交换 2 字节数据的高位及低位 1 字节
2	SWAP.W 指令	unsigned long swapw(unsigned long data)	交换 4 字节数据的高位及低位 2 字节
3	交换 4 字节数据的高及低位	unsigned long end_cnvl(unsigned long data)	以高低位顺序相反的单一字节来安排 4 字节数据

使用的实例：

C 语言代码

```
#include <machine.h>
extern unsigned short data;
extern unsigned short result;
void main( void )
{
      result = swapb(data);
                                        /* 若 data = 0x1234, result = 0x3412。*/
}
```

扩展为汇编语言代码

```
            .IMPORT     _result
            .IMPORT     _data
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                                  ; 函数：main
                                        ; 帧大小=0
            MOV.L       L11,R6          ; _data
            MOV.W       @R6,R2
            SWAP.B      R2,R6
            MOV.L       L11+4,R2        ; _result
            RTS
            MOV.W       R6,@R2
L11:
            .DATA.L     _data
            .DATA.L     result
            .END
```

3.2.11 系统调用

描述：

下面显示能够从 C 语言程序发出系统调用的内建函数格式。 系统调用的参数数量是 0 至 4 之间的变量。
然而，TRAPA 无法直接调用具有 RTE 返回的 C 函数。
在实践中，trapa_svc（一项 C 内建函数）应被用于注册处理程序函数（应以汇编语言编写），同时 R0 函数代码应被测试以调用各个例程。
从此例程以汇编语言编写的返回应通过 RTE。

格式：

```
ret=trapa svc(int trap no, int code,
    [type1 p1[, type2 p2[, type3 p3[, type4 p4]]]])
    trap no          : trap 编号（由常数指定）
    code             : 函数代码，分配到 R0
    p1               : 第一个参数，分配到 R4
    p2               : 第二个参数，分配到 R5
    p3               : 第三个参数，分配到 R6
    p4               : 第四个参数，分配到 R7
    type1 to type4   参数类型是整数类型 ([unsigned]char,
                         [unsigned]short,[unsigned]int,
                         [unsigned]long)，或指针类型
```

使用的实例：

使用此函数，将发出可使用 trap 编号 #63 来指定的 OS 系统调用。

C 语言代码

```
#include <machine.h>
#define  SIG SEM  0xffc8
void  main(void)
{
       :
       :
    trapa svc(63, SIG SEM, 0x05);
       :
       :
}
```

扩展为汇编语言代码

```
          .EXPORT      main
          .SECTION     P,CODE,ALIGN=4
main:     :                              ; 函数：main
          :                              ; 帧大小=0
```

```
          MOV.L       L215+2,R0       ; H'0000FFC8
          MOV         #5,R4
          TRAPA       #63
           :
           :
          RTS
          NOP
L215:
          .RES.W      1
          .DATA.L     H'0000FFC8
         .END
```

向量表定义

```
void(*const vect[])(void)={
      :
      :
   HDR,.......                    /* 在 trap 63 vector 中定义 HDR */
      :
} ;
```

处理程序（以汇编语言编写）

```
         .IMPORT func
HDR:
                                  ; 保存 PR 和 R1 至 R7 寄存器
                                  ; 选择将根据 R0 函数代码被调用的函数
;          :
          MOV.L    label+2,R0
          JSR       @R0
                                  ; ->若 R4 至 R7 寄存器的内容未
                                  ; 损坏，将传递正确的参数。
          NOP
;          :
                                  ; 恢复 PR 和 R1 至 R7 寄存器
          RTE                     ; ->从异常处理返回
                                  ; func 的返回值 R0 被用作
                                  ; trapa svc 的返回值
          NOP
label:
         .RES.W  1
         .DATA.L  func
         .END
```

3.2.12 预取指令

描述：

下面显示在 SH-2A、SH2A-FPU、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 中执行高速缓存预取的内建函数格式。此内建函数仅在指定了“-cpu=sh2a”、“-cpu=sh2afpu”、“-cpu=sh3”、“-cpu=sh3dsp”、“-cpu=sh4”、“-cpu=sh4a”或“ -cpu=sh4aldsp”时有效。

格式：

```
void prefetch(void *p1)
p1 : prefetch address
```

使用的实例：

C 语言代码

```
#include <umachine.h>
int a[1200];
f()
{
    int *pa = a;
        :
        :
    prefetch(pa+8);
        :
        :
}
```

扩展为汇编语言代码

```
f:                                  ; 函数：f
            :
            :
        ADD     #32,R6
        PREF    @R6
            :
            :
```

3.2.13 LDTLB 指令

描述：

下面显示在 SH-2A、SH2A-FPU、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 中执行 LDTLB 扩展的内建函数格式。此内建函数仅在指定了“-cpu=sh2a”、“-cpu=sh2afpu”、“-cpu=sh3”、“-cpu=sh3dsp”、“-cpu=sh4”、“-cpu=sh4a”或“ -cpu=sh4aldsp”时有效。

格式：

```
void ldtlb (void)
```

使用的实例：

C 语言代码

```
#include <machine.h>
void main(void)
{
       ldtlb();
}
```

扩展为汇编语言代码

```
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                                ; 函数：main
                                      ; 帧大小=0
          LDTLB
          NOP
          NOP
          RTS
          NOP
            .END
```

3.2.14 NOP 指令

描述：

下面显示扩展为 NOP 指令的内建函数格式。

格式：

```
void nop (void)
```

使用的实例：

C 语言代码

```
#include <machine.h>
void  main(void)
{
int a;
    if (a){
        nop();
    }
}
```

扩展为汇编语言代码

```
            .EXPORT     main
            .SECTION    P,CODE,ALIGN=4
main:                                   ; 函数：main
                                        ; 帧大小=0
            TST         R2,R2
            BT          L12
            NOP
            RTS
            NOP
L12:
            RTS
            NOP
            .END
```

3.2.15 单精度浮点运算

描述：

用于单精度浮点运算的内建函数从 SH-4 开始添加。表 3.12 列出可用的运算。 浮点运算单位的内建函数仅在指定了 -cpu=sh2e、-cpu=sh2afpu、-cpu=sh4 或 -cpu=sh4a 时有效。 单精度浮点向量运算的内建函数仅在指定了 -cpu=sh4 或 -cpu=sh4a 时有用。然而，add4() 和 sub4() 也在 cpu=sh2afpu 指定时有效。

表 3.12 单精度浮点运算列表（1）

编号	函数	格式	描述
1	浮点 运算单位	void set_fpscr(int cr)	设定 FPSCR 中的 cr （32 位）
2		int get_fpscr()	参考 FPSCR。
3	单精度浮点向量运算	float fipr(float vect1[4], float vect2[4])	获取两个向量的内部积。
4		float ftrv(float vec1[4], float vec2[4])	通过使用 ld_ext( ) 加载的 tbl (4x4 matrix) 转换 vec1(vector)，然后将结果保存在 vec2 (vector)。
5		void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	通过使用 ld_ext( ) 函数加载的 tbl (4x4 matrix) 转换 vec1(vector)，然后将结果与 vec2 (vector) 相加，再将相加结果保存在 vec3 (vector)。
6		void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	通过使用 ld_ext( ) 函数加载的 tbl (4x4 matrix) 转换 vec1(vector)，然后从结果减去 vec2 (vector)，再将相减结果保存在 vec3 (vector)。
7		void add4(float vec1[4], float vec2[4], float vec3[4])	相加 vec1 (vector) 和 vec2 (vector)，然后将结果保存在 vec3 (vector)。

表 3.12　单精度浮点运算列表 (2)

编号	函数	格式	描述
8	单精度浮点向量运算	void sub4(float vec1[4], float vec2[4], float vec3[4])	从 vec1 (vector) 减去 vec2 (vector)，然后将结果保存在 vec3 (vector)。
9		void mtrx4mul(float mat1[4][4], float mat2[4][4])	通过使用 ld_ext() 函数加载的 tbl (4x4 matrix) 转换 mat1 (4x4 matrix)，然后将结果保存在 mat2。
10		void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	通过使用 ld_ext() 函数加载的 tbl (4x4 matrix) 转换 mat1 (4x4 matrix)，然后将结果与 mat2 (4x4 matrix) 相加，再将相加结果保存在 mat3 (4x4 matrix)。
11		void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	通过使用 ld_ext() 函数加载的 tbl (4x4 matrix) 转换 mat1 (4x4 matrix)，然后从结果减去 mat2 (4x4 matrix)，再将相减结果保存在 mat3 (4x4 matrix)。

使用的实例：

执行 4x4 矩阵的乘法。
其中一个矩阵必须使用 ld_ext 函数预先加载。

C 语言代码

```
#include<machine.h>
float table[4][4] ={{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},
                    {0.0,0.0,1.0,0.0},{0.0,0.0,0.0,1.0}} ;
float data1[4][4] ={{11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0},
                    {11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0}} ;
float data2[4][4] ={{0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0},
                    {0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0}} ;

void main()
{
  ld_ext(table) ;
  mtrx4mul(data1,data2) ;
}
```

扩展为汇编语言代码

```
            .EXPORT     _table
```

```
        .EXPORT    _data1
        .EXPORT    _data2
        .EXPORT    _main
        .SECTION   P,CODE,ALIGN=4
main:                           ; 函数: main
                                ; 帧大小=0
        MOV.L      L11+2,R2     ; _table
        MOV.L      L11+6,R6     ; _data2
        FRCHG
        FMOV.S     @R2+,FR0
        FMOV.S     @R2+,FR1
        FMOV.S     @R2+,FR2
        FMOV.S     @R2+,FR3
        FMOV.S     @R2+,FR4
        FMOV.S     @R2+,FR5
        FMOV.S     @R2+,FR6
        FMOV.S     @R2+,FR7
        FMOV.S     @R2+,FR8
        FMOV.S     @R2+,FR9
        FMOV.S     @R2+,FR10
        FMOV.S     @R2+,FR11
        FMOV.S     @R2+,FR12
        FMOV.S     @R2+,FR13
        FMOV.S     @R2+,FR14
        FMOV.S     @R2+,FR15
        FRCHG
        ADD        #-64,R2
        MOV.L      L11+10,R2    ; _data1
        ADD        #16,R6
        FMOV.S     @R2+,FR0
        FMOV.S     @R2+,FR1
        FMOV.S     @R2+,FR2
        FMOV.S     @R2+,FR3
        FTRV       XMTRX,FV0
        FMOV.S     FR3,@-R6
        FMOV.S     FR2,@-R6
        FMOV.S     FR1,@-R6
        FMOV.S     FR0,@-R6
        FMOV.S     @R2+,FR0
        ADD        #32,R6
        FMOV.S     @R2+,FR1
        FMOV.S     @R2+,FR2
        FMOV.S     @R2+,FR3
        FTRV       XMTRX,FV0
```

```
            FMOV.S      FR3,@-R6
            FMOV.S      FR2,@-R6
            FMOV.S      FR1,@-R6
            FMOV.S      FR0,@-R6
            FMOV.S      @R2+,FR0
            ADD         #32,R6
            FMOV.S      @R2+,FR1
            FMOV.S      @R2+,FR2
            FMOV.S      @R2+,FR3
            FTRV        XMTRX,FV0
            FMOV.S      FR3,@-R6
            FMOV.S      FR2,@-R6
            FMOV.S      FR1,@-R6
            FMOV.S      FR0,@-R6
            FMOV.S      @R2+,FR0
            ADD         #32,R6
            FMOV.S      @R2+,FR1
            FMOV.S      @R2+,FR2
            FMOV.S      @R2+,FR3
            FTRV        XMTRX,FV0
            FMOV.S      FR3,@-R6
            FMOV.S      FR2,@-R6
            FMOV.S      FR1,@-R6
            RTS
            FMOV.S      FR0,@-R6
L11:
            .RES.W      1
            .DATA.L     _table
            .DATA.L     _data2
            .DATA.L     _data1
            .SECTION    D,DATA,ALIGN=4
_table:                             ; static: table
            .DATA.L     H'3F800000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'3F800000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'3F800000
            .DATA.L     H'00000000
```

```
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'3F800000
_data1:                             ; static: data1
            .DATA.L     H'41300000
            .DATA.L     H'41400000
            .DATA.L     H'41500000
            .DATA.L     H'41600000
            .DATA.L     H'41700000
            .DATA.L     H'41800000
            .DATA.L     H'41880000
            .DATA.L     H'41900000
            .DATA.L     H'41300000
            .DATA.L     H'41400000
            .DATA.L     H'41500000
            .DATA.L     H'41600000
            .DATA.L     H'41700000
            .DATA.L     H'41800000
            .DATA.L     H'41880000
            .DATA.L     H'41900000
_data2:                             ; static: data2
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .END
```

使用的实例：

执行向量和矩阵的乘法。 矩阵必须使用 ld_ext 函数预先加载。

C 语言代码

```
#include<machine.h>
float table[4][4]={{1.0,2.0,3.0,4.0},{5.0,6.0,7.0,8.0},
                   {8.0,7.0,6.0,5.0},{4.0,3.0,2.0,1.0}} ;
float data1[4]   = {11.0,12.0,13.0,14.0} ;
float data2[4]   = {0.0,0.0,0.0,0.0} ;

void main()
{
  ld_ext(table) ;
  ftrv(data1,data2) ;
}
```

扩展为汇编语言代码

```
            .EXPORT     _table
            .EXPORT     _data1
            .EXPORT     _data2
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数：main
                                    ; 帧大小=0
            MOV.L       L11+2,R2    ; _table
            FRCHG
            FMOV.S      @R2+,FR0
            FMOV.S      @R2+,FR1
            FMOV.S      @R2+,FR2
            FMOV.S      @R2+,FR3
            FMOV.S      @R2+,FR4
            FMOV.S      @R2+,FR5
            FMOV.S      @R2+,FR6
            FMOV.S      @R2+,FR7
            FMOV.S      @R2+,FR8
            FMOV.S      @R2+,FR9
            FMOV.S      @R2+,FR10
            FMOV.S      @R2+,FR11
            FMOV.S      @R2+,FR12
            FMOV.S      @R2+,FR13
            FMOV.S      @R2+,FR14
            FMOV.S      @R2+,FR15
            FRCHG
            ADD         #-64,R2
            MOV.L       L11+6,R2    ; _data1
            FMOV.S      @R2+,FR0
            FMOV.S      @R2+,FR1
```

```
            FMOV.S      @R2+,FR2
            FMOV.S      @R2+,FR3
            MOV.L       L11+10,R2  ; _data2
            FTRV        XMTRX,FV0
            ADD         #16,R2
            FMOV.S      FR3,@-R2
            FMOV.S      FR2,@-R2
            FMOV.S      FR1,@-R2
            RTS
            FMOV.S      FR0,@-R2
L11:
            .RES.W      1
            .DATA.L     _table
            .DATA.L     _data1
            .DATA.L     _data2
            .SECTION    D,DATA,ALIGN=4
_table:                             ; static: table
            .DATA.L     H'3F800000
            .DATA.L     H'40000000
            .DATA.L     H'40400000
            .DATA.L     H'40800000
            .DATA.L     H'40A00000
            .DATA.L     H'40C00000
            .DATA.L     H'40E00000
            .DATA.L     H'41000000
            .DATA.L     H'41000000
            .DATA.L     H'40E00000
            .DATA.L     H'40C00000
            .DATA.L     H'40A00000
            .DATA.L     H'40800000
            .DATA.L     H'40400000
            .DATA.L     H'40000000
            .DATA.L     H'3F800000
_data1:                             ; static: data1
            .DATA.L     H'41300000
            .DATA.L     H'41400000
            .DATA.L     H'41500000
            .DATA.L     H'41600000
_data2:                             ; static: data2
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .DATA.L     H'00000000
            .END
```

- 获取两个向量的内部积。

C 语言代码

```
#include<machine.h>
float ret = 0;
float data1[]={1.0,2.0,3.0,4.0} ;
float data2[]={11.0,12.0,13.0,14.0} ;

void main()
{
   ret = fipr (data1,data2) ;
}
```

扩展为汇编语言代码

```
            .EXPORT    _ret
            .EXPORT    _data1
            .EXPORT    _data2
            .EXPORT    _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数: main
                                    ; 帧大小=0
            MOV.L       L11,R2      ; _data1
            FMOV.S      @R2+,FR0
            FMOV.S      @R2+,FR1
            FMOV.S      @R2+,FR2
            FMOV.S      @R2+,FR3
            MOV.L       L11+4,R2    ; _data2
            FMOV.S      @R2+,FR4
            FMOV.S      @R2+,FR5
            FMOV.S      @R2+,FR6
            FMOV.S      @R2+,FR7
            MOV.L       L11+8,R2    ; _ret
            FIPR        FV4,FV0
            RTS
            FMOV.S      FR3,@R2
L11:
            .DATA.L     _data1
            .DATA.L     _data2
            .DATA.L     _ret
            .SECTION    D,DATA,ALIGN=4
_ret:                               ; static: ret
            .DATA.L     H'00000000
```

```
_data1:                          ; static: data1
          .DATA.L     H'3F800000
          .DATA.L     H'40000000
          .DATA.L     H'40400000
          .DATA.L     H'40800000
_data2:                          ; static: data2
          .DATA.L     H'41300000
          .DATA.L     H'41400000
          .DATA.L     H'41500000
          .DATA.L     H'41600000
          .END
```

重要信息：

(1) 用于单精度浮点向量运算的内建函数仅在 SH-4 和 SH-4A 中有效（add4() 和 sub4() 也在 SH2A-FPU 中有效）。

(2) 当在中断函数中使用向量运算的内建函数时，应注意下列事项。内建函数 ld_ext(float[4][4]) 和 st_ext(float[4][4]) 修改浮点状态控制寄存器 (FPSCR) 的浮点寄存器库位 (FR) 以存取扩展寄存器，因此若在中断函数中使用内建函数 ld_ext(float[4][4]) 或 st_ext(float[4][4])，中断屏蔽应在使用内建向量运算函数前后被更改。 一个实例在下面提供。

实例：

```
#pragma interrupt (intfunc)
void intfunc(){
     ...
     ld ext();
     ...
}
void normfunc(){
     ...
     int maskdata=get imask();   /*保存中断屏蔽     */
     set imask(15);              /*指定中断屏蔽     */
     ld ext(mat1);
     ftrv(vec1,vec2);
     set imask(maskdata);        /*恢复中断屏蔽     */
    ...
}
```

(3) 内建函数 mtrx4mul、mtrx4muladd、mtrx4mulsub 是对 4x4 矩阵的运算，因此 matrix A x matrix B 的结果可能不与 matrix B x matrix A 的结果一致。

实例：

```
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 = matBxmatA        */
      ld ext(matB);
      mtrx4mul(matA,data2);          /* data2 = matAxmatB        */
      ..../* 此例中，data1[ ][ ] 可能和 data2[ ][ ] 不一致                */
}
```

3.2.16 存取扩展寄存器

描述：

表 3.13 显示为存取扩展寄存器而提供的函数。

表 3.13 存取扩展寄存器的内建函数列表

编号	函数	格式	描述
1	扩展寄存器存取	void ld_ext(float mat[4][4])	将 tbl (4x4 matrix) 加载到扩展寄存器 实例： extern float tbl[4][4]; 在此情况下 在此情况下，ld_ext(tbl) 将 tbl 加载到扩展寄存器。
2		void st_ext(float mat[4][4])	将扩展寄存器的内容保存到 tbl (4x4 matrix)。 实例： extern float tbl[4][4]; 在此情况下 在此情况下，st_ext(tbl) 将扩展寄存器的内容保存到 tbl。

注意： (1) 存取扩展寄存器的内建函数仅在 SH-4 和 SH-4A 中有效。
(2) 当这些函数在中断函数中使用时，中断屏蔽必须被更改。 有关详情，请参考第 3.2.15 节“单精度浮点运算”的重要信息。

3.2.17 DSP 指令

描述：

表 3.14 显示为 DSP 指令提供的函数。

表 3.14 DSP 指令的内建函数列表

编号	函数	格式	描述
1	绝对值	long __fixed pabs_lf (long __fixed data) long __accum pabs_la (long __accum data)	获取绝对值。 若所获取的绝对值无法以返回值的相同类型表示，将无法保证正确的运算（即，用于 pabs_lf() 的 long __fixed 类型或用于 pabs_la() is 的 long __accum 类型）。
2	MSB 侦测	__fixed pdmsb_lf (long __fixed data) __fixed pdmsb_la (long __accum data)	侦测 MSB（获取正常化数据所需的移位量）。
3	算术移位	long __fixed psha_lf (long __fixed data, int count) long __accum psha_la (long __accum data, int count)	执行算术移位。 可为 count 指定的值是从 –32 到 +32。 指定正值时数据左移。 指定负值时数据右移其绝对值的长度。 若指定了有效范围外的值，将无法保证正确运算。
4	舍入错误	__accum rndtoa (long __accum data) __fixed rndtof (long __fixed data)	处理舍入错误。
5	位模式复制	long __fixed long_as_lfixed (long data) long lfixed_as_long (long __fixed data)	从一般寄存器复制位模式到 DSP 寄存器，或相反。
6	取模寻址设定	void set_circ_x (__X__circ __fixed array[], size_t size) void set_circ_y (__Y__circ __fixed array[], size_t size)	设定取模寻址。
7	取模寻址取消	void clr_circ(void)	取消取模寻址（即，将 SR 右边第 10 和 11 位清除至零）。
8	CS 位设定（DSR 寄存器）	void set_cs (unsigned int mode)	设定 CS 位。 mode=0: 进位或借位模式 mode=1: 负值模式 mode=2: 零值模式 mode=3: 溢出模式 mode=4: 带符号的“大于”模式 mode=5: 带符号的“等于或大于”模式

使用的实例：

C 语言代码

```
#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)); /* 设定取模寻址 */
   for(i=0; i < 8; ii++){
   output[i] = input[i];
     }
     clr_circ();                  /* 取消取模寻址 */
}
```

扩展为汇编语言代码

```
            .EXPORT     _output
            .EXPORT     _input
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数：main
                                    ; 帧大小=0
            MOV.L       L13+4,R5    ; _input
            EXTU.W      R5,R2
            MOV         R5,R6
            ADD         #6,R6
            SHLL16      R6
            ADD         R6,R2
            LDC         R2,MOD
            STC         SR,R2
            MOV.W       L13,R4      ; H'F3FF
            AND         R4,R2
            MOV         R2,R6
            MOV         #4,R2       ; H'00000004
            SHLL8       R2
            OR          R2,R6
            LDC         R6,SR
            MOV         #8,R2       ; H'00000008
            MOV.L       L13+8,R6    ; _output
L11:
            MOVX.W      @R5+,X1
            DT          R2
```

```
            PCOPY       X1,A0
            BF/S        L11
            MOVY.W      A0,@R6+
            STC         SR,R2
            AND         R4,R2
            LDC         R2,SR
            RTS
            NOP
L13:
            .DATA.W     H'F3FF
            .RES.W      1
            .DATA.L     _input
            .DATA.L     _output
            .SECTION    $XD,DATA,ALIGN=4
_input:                             ; static: input
            .DATA.W     H'0000,H'2000,H'4000,H'2000
            .SECTION    $YB,DATA,ALIGN=4
_output:                            ; static: output
            .RES.W      8
            .END
```

3.2.18 正弦及余弦

描述：

函数从为 angle 指定的不同角度获取正弦和余弦的近似值，然后将结果设定到以 sinv 及 cosv 表示的区域。

格式：

```
void fsca(long angle, float * sinv, float * cosv)
```

使用的实例：

C 语言代码

```
#include <machine.h>
long angle = (45<<16)/360;  /* 45 degrees */
float  * sinv;
float  * cosv;
void  main(void)
{
      fsca(angle, sinv, cosv);
}
```

扩展为汇编语言代码

```
          .EXPORT     _sinv
          .EXPORT     _cosv
          .EXPORT     _angle
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数：main
                                    ; 帧大小=0
          MOV.L       L11+2,R6      ; _angle
          MOV.L       @R6,R2
          MOV.L       L11+6,R6      ; _sinv
          LDS         R2,FPUL
          FSCA        FPUL,DR0
          MOV.L       @R6,R2
          MOV.L       L11+10,R6     ; _cosv
          FMOV.S      FR0,@R2
          MOV.L       @R6,R2
          RTS
          FMOV.S      FR1,@R2
L11:
          .RES.W      1
          .DATA.L     _angle
          .DATA.L     _sinv
```

```
            .DATA.L     _cosv
            .SECTION    D,DATA,ALIGN=4
_angle:                             ; static: angle
            .DATA.L     H'00002000
            .SECTION    B,DATA,ALIGN=4
_sinv:                              ; static: sinv
            .RES.L      1
_cosv:                              ; static: cosv
            .RES.L      1
            .END
```

3.2.19 平方根的倒数

描述：

函数获取平方根倒数的近似值。

格式：

```
float fsrra(float data)
```

使用的实例：

C 语言代码

```
#include <machine.h>
float  data;
float result;
void  main(void)
{
       result=fsrra(data);
}
```

扩展为汇编语言代码

```
            .EXPORT     _data
            .EXPORT     _result
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数： main
                                    ; 帧大小=0
            MOV.L       L11,R2      ; _data
            FMOV.S      @R2,FR9
            MOV.L       L11+4,R2    ; _result
            FSRRA       FR9
            RTS
            FMOV.S      FR9,@R2
L11:
            .DATA.L     _data
            .DATA.L     _result
            .SECTION    B,DATA,ALIGN=4
_data:                              ; static: data
            .RES.L      1
_result:                            ; static: result
            .RES.L      1
            .END
```

3.2.20 指令高速缓存的失效

描述：

函数使指令高速缓存失效。

格式：

```
void icbi(void *p)
```

使用的实例：

C 语言代码

```
#include <machine.h>
extern int *p;
void  main(void)
{
   icbi(p);
}
```

扩展为汇编语言代码

```
            .IMPORT     _p
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数：main
                                    ; 帧大小=0
            MOV.L       L11+2,R6    ; _p
            MOV.L       @R6,R2
            ICBI        @R2
            RTS
            NOP
L11:
            .RES.W      1
            .DATA.L     _p
            .END
```

3.2.21 高速缓存块运算

描述：

函数在高速缓存块上执行运算。

格式：

```
void ocbi(void *p)     高速缓存块失效
void ocbp(void *p)     高速缓存块清除
void ocbwb(void *p)    高速缓存块回写
```

使用的实例：

C 语言代码

```
#include <machine.h>
extern  int *p;
void  main(void)
{
    ocbi(p);
}
```

扩展为汇编语言代码

```
          .IMPORT     _p
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                                ; 函数：main
                                      ; 帧大小=0
          MOV.L       L11+2,R6        ; _p
          MOV.L       @R6,R2
          OCBI        @R2
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _p
          .END
```

3.2.22 指令高速缓存预取

描述：

函数将从 32 字节边界开始的 32 字节指令块读取入指令高速缓存。

格式：

```
void prefi(void *p)
```

使用的实例：

C 语言代码

```
#include <machine.h>
void * pa;
void  main(void)
{
       prefi(pa);
}
```

扩展为汇编语言代码

```
            .EXPORT     _pa
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; 函数：main
                                    ; 帧大小=0
            MOV.L       L11+2,R6    ; _pa
            MOV.L       @R6,R2
            PREFI       @R2
            RTS
            NOP
L11:
            .RES.W      1
            .DATA.L     _pa
            .SECTION    B,DATA,ALIGN=4
_pa:                                ; static: pa
            .RES.L      1
            .END
```

3.2.23 系统同步化

描述：

函数将指令扩展到 SYNCO 指令。

SYNCO 指令同步化数据运算。 执行 SYNCO 指令允许 SYNCO 指令之后的指令可在 SYNCO 指令前的数据运算完成后启动。

格式：

```
void synco(void)
```

使用的实例：

C 语言代码

```
#include <machine.h>
void  main(void)
{
    synco();
}
```

扩展为汇编语言代码

```
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                                ; 函数：main
                                      ; 帧大小=0
          SYNCO
          RTS
          NOP
          .END
```

3.2.24 参考及设定 T 位

描述：

表 3.1.5 显示为设定和参考 T 位而提供的函数。

表 3.15 T 位的内建函数列表

编号	函数	格式	描述
1	T 位参考	int movt(void)	参考 SR 寄存器中 T 位的值。
2	T 位清除	void clrt(void)	清除 SR 寄存器中的 T 位。
3	T 位设定	void sett(void)	设定 SR 寄存器中的 T 位。

使用的实例：

函数允许您参考 SR 寄存器中 T 位的值。 被参考的值将会是 0 或 1。

C 语言代码

```
#include <machine.h>
int sr_t;
void func(void)
{
      sr_t = movt();
}
```

扩展为汇编语言代码

```
_func:
          MOVT        R2
          MOV.L       L11,R6      ; _sr_t
          RTS
          MOV.L       R2,@R6
```

3.2.25 切除连接的寄存器的中部

描述：

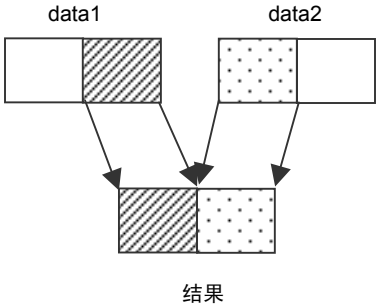
函数从两个连接的 32 位数据项目切除中间 32 位。

格式：

```
unsigned long xtrct(unsigned long data1, unsigned long data2)
```

使用的实例：

在此实例中，data1 和 data2 是连接的，其中间 32 位被切除。



C 语言代码

```
#include <machine.h>
unsigned long result, data1, data2;
void main(void)
{
        result = xtrct(data1,data2);
}
```

扩展为汇编语言代码

```
_main:
          MOV.L         L11,R1      ; _data2
          MOV.L         L11+4,R4    ; _data1
          MOV.L         @R1,R2
          MOV.L         @R4,R5
          MOV.L         L11+8,R6    ; _result
          XTRCT         R5,R2
          RTS
          MOV.L         R2,@R6
```

3.2.26 使用进位的加法

描述：

表 3.16 显示为使用进位的加法提供的函数。

表 3.16 使用进位的加法的内建函数列表

编号	函数	格式	描述
1	使用进位的加法	long addc(long data1, long data2)	相加两个数据项目及 T 位，并将进位应用到 T 位。
2		int ovf_addc(long data1, long data2)	相加两个数据项目及 T 位，并参考进位。
3		long addv(long data1, long data2)	相加两个数据项目，并将进位应用到 T 位。
4		int ovf_addv(long data1, long data2)	相加两个数据项目，并参考进位。

使用的实例：

在此实例中，data1、data2 和 T 位被相加，然后进位被应用到 T 位。

C 语言代码

```
#include <machine.h>
long result, data1, data2;
void main()
{
    result = addc(data1,data2);     /* result = data1 + data2 + T bit */
}
```

扩展为汇编语言代码

```
_main:
          MOV.L       L11,R1      ; _data1
          MOV.L       L11+4,R2    ; _data2
          MOV.L       @R1,R4
          MOV.L       @R2,R2
          MOV.L       L11+8,R5    ; _result
          ADDC        R4,R2
          RTS
          MOV.L       R2,@R5
```

重要信息：

addc 和 ovf_addc 函数参考 T 位的内容。若您恰在这些函数前指定比较或移位，应用到 T 位的计算结果可能造成函数运算不正确。

3.2.27 具有借位的减法

描述：

表 3.17 显示为具有借位的减法提供的函数。

表 3.17 具有借位的减法的内建函数列表

编号	函数	格式	描述
1	具有借位的减法	long subc(long data1, long data2)	从 data1 减去 data2 及 T 位，并将借位应用到 T 位。
2		int unf_subc(long data1, long data2)	从 data1 减去 data2 及 T 位，并参考借位。
3		long subv(long data1, long data2)	从 data1 减去 data2，并将借位应用到 T 位。
4		int unf_subv(long data1, long data2)	从 data1 减去 data2，并参考借位。

使用的实例：

在此实例中，data2 和 T 位被从 data1 减去，然后借位被应用到 T 位。

C 语言代码

```
#include<machine.h>
long result, data1, data2;
void main()
{
    result = subc(data1,data2);      /* result = data1 - data2 - T bit */
}
```

扩展为汇编语言代码

```
_main:
          MOV.L         L11,R1       ; _data1
          MOV.L         L11+4,R2     ; _data2
          MOV.L         @R1,R4
          MOV.L         @R2,R2
          MOV.L         L11+8,R5     ; _result
          SUBC          R4,R2
          RTS
          MOV.L         R2,@R5
```

重要信息：

subc 和 unf_subc 函数参考 T 位的内容。若您恰在这些函数前指定比较或移位，应用到 T 位的计算结果可能造成函数运算不正确。

3.2.28 符号倒置

描述：

函数从 0 减去 data 及 T 位，并将借位应用到 T 位。

格式：

```
long negc(long data)
```

使用的实例：

在此实例中，“data”和 T 位被从 0 减去，然后借位被应用到 T 位。

C 语言代码

```
#include <machine.h>
long result, data;
void main()
{
        result = negc(data);        /* result = 0 - data - T bit */
}
```

扩展为汇编语言代码

```
_main:
          MOV.L         L11,R1      ; _data
          MOV.L         L11+4,R5    ; _result
          MOV.L         @R1,R4
          NEGC          R4,R2
          RTS
          MOV.L         R2,@R5
```

3.2.29 一位除法

描述：

表 3.18 显示为一位除法提供的函数。

表 3.18 一位除法的内建函数列表

编号	函数	格式	描述
1	一位除法	unsigned long div1(unsigned long data1, unsigned long data2)	执行 data2 对 data1 的一位除法，并将结果应用到 T 位。
2		int div0s(long data1, long data2)	执行 data2 对 data1 的带符号除法初始化，并参考 T 位。
3		void div0u(void)	执行无符号除法初始化。

使用的实例：

通过重复执行一位除法，您将可以获得商数。 下面显示一个由 d2 (16 bits) 对 d1 (32 bits) 执行无符号除法的实例，其结果是 "ret" (16 bits)。

C 语言代码

```
#include <machine.h>
unsigned long data1,data2;
unsigned long result;
void main()
{
    unsigned long d1,d2;

    d1 = data1;
    d2 = data2;
    d2 <<= 16;                  /* 使用高 16 位作除数，并将低 16 位设定为 0。    */
    div0u();                    /* 为无符号除法设定初始值                        */
    d1 = div1(d1, d2);          /* 重复 16 次一个步骤的除法                      */
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
```

```
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    result = rotcl(d1);          /* rotcl 是用于生成 ROTCL 指令的内建函数      */
                                 /* 将 T 位作为上一步骤除法的结果应用到商数    */
}
```

扩展为汇编语言代码

```
_main:
          MOV.L     L11,R1       ; _data2
          MOV.L     @R1,R5
          SHLL16    R5
          DIV0U
          MOV.L     L11+4,R2     ; _data1
          MOV.L     @R2,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          MOV       R2,R6
          DIV1      R5,R6
          MOV       R6,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          MOV       R2,R6
          DIV1      R5,R6
          MOV       R6,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          DIV1      R5,R2
          ROTCL     R2
          MOV.L     L11+8,R4     ; _result
          RTS
          MOV.L     R2,@R4
```

重要信息：

(1) 虽然您可通过重复使用 div1 函数来执行除法，但请勿在重复期间更新 M、Q 和 T 位。（比较和移位也会更新 T 位）。
(2) 您必须紧挨在函数前使用 div0s() 或 div0u() 以初始化 M、Q 和 T 位。

3.2.30 旋转

描述：

表 3.19 显示为旋转提供的函数。

表 3.19 旋转的内建函数列表

编号	函数	格式	描述
1	旋转	unsigned long rotl(unsigned long data)	将数据向左旋转一位，然后将移到操作数外的位应用到 T 位。
2		unsigned long rotr(unsigned long data)	将数据向右旋转一位，然后将移到操作数外的位应用到 T 位。
3		unsigned long rotcl(unsigned long data)	将数据向左旋转一位，包括 T 位，然后将移到操作数外的位应用到 T 位。
4		unsigned long rotcr(unsigned long data)	将数据向右旋转一位，包括 T 位，然后将移到操作数外的位应用到 T 位。

使用的实例：

在此实例中，"data" 向左旋转一位，然后移到操作数外的位被应用到 T 位。

C 语言代码

```
#include <machine.h>
unsigned long result, data;
void main()
{
    result = rotl(data);
}
```

扩展为汇编语言代码

```
_main:
          MOV.L       L11,R2      ; _data
          MOV.L       L11+4,R6    ; _result
          MOV.L       @R2,R2
          ROTL        R2          ; 生成 ROTL 指令
          RTS
          MOV.L       R2,@R6
```

重要信息：

rotcl 和 rotcr 函数参考 T 位的内容。若您恰在这些函数前指定比较或移位，应用到 T 位的计算结果可能造成函数运算不正确。

3.2.31 移位

描述：

表 3.20 显示为移位提供的函数。

表 3.20 移位的内建函数列表

编号	函数	格式	描述
1	移位	unsigned long shll(unsigned long data)	将数据向左移一位，然后将移到操作数外的位应用到 T 位。
2		unsigned long shlr(unsigned long data)	将数据逻辑右移一位，然后将移到操作数外的位应用到 T 位。
3		long shar(long data)	将数据算术右移一位，然后将移到操作数外的位应用到 T 位。

使用的实例：

在此实例中，"data" 被左移一位，然后移到操作数外的位被应用到 T 位。

C 语言代码

```
#include <machine.h>
unsigned long result, data;
void main()
{
    result = shll(data);
}
```

扩展为汇编语言代码

```
_main:
          MOV.L       L11,R2     ; _data
          MOV.L       L11+4,R6   ; _result
          MOV.L       @R2,R2
          SHLL        R2         ; 生成 SHLL 指令
          RTS
          MOV.L       R2,@R6
```

3.2.32 饱和运算

描述：

表 3.21 显示为饱和运算提供的函数。

表 3.21 饱和运算的内建函数列表

编号	函数	格式	描述
1	带符号一字节数据的饱和运算	long clipsb(long data)	在数据值介于 –128 到 127 的范围内时进行设定。若数据超出这个范围，函数将设定上或下限。
2	带符号二字节数据的饱和运算	long clipsw(long data)	在数据值介于 -32768 到 32767 的范围内时进行设定。若数据超出这个范围，函数将设定上或下限。
3	无符号一字节数据的饱和运算	unsigned long clipub(unsigned long data)	在数据值介于 0 到 255 的范围内时进行设定。若数据超出这个范围，函数将设定上限。
4	无符号二字节数据的饱和运算	unsigned long clipuw(unsigned long data)	在数据值介于 0 到 65535 的范围内时进行设定。若数据超出这个范围，函数将设定上限。

此内建函数只有在指定了 "-cpu=sh2a" 或 "-cpu=sh2afpu" 时有效。

使用的实例：

在此实例中，数据值在介于 -128 到 127 的范围内时被设定。若数据超出这个范围，上或下限将被设定。

C 语言代码

```
#include <machine.h>
long result, data;
void main()
{
    result = clipsb(data);          /* "result" 的值介于 –128 到 127 的范围内*/
}
```

扩展为汇编语言代码

```
_main:
          MOV.L       L11,R2        ; _data
          MOV.L       @R2,R2
          MOV.L       L11+4,R6      ; _result
          CLIPS.B     R2
          RTS
          MOV.L       R2,@R6
```

3.2.33 参考和设定 TBR

描述：

表 3.22 显示为设定和参考跳转表基址寄存器所提供的函数。

表 3.22 TBR 指令的内建函数列表

编号	函数	格式	描述
1	TBR 设定	void set_tbr(void *data)	设定 TBR 中的数据。
2	TBR 参考	void *get_tbr(void)	参考 TBR 的值。

此内建函数只有在指定了 "-cpu=sh2a" 或 "-cpu=sh2afpu" 时有效。

使用的实例：

在此实例中，数据在 TBR 中被设定。
此函数在 TBR 中用于设定为 TBR 相对函数调用生成的跳转表。

C 语言代码

```
#include <machine.h>
void main(){
      set_tbr(__sectop("$TBR"));    /* 将 $TBR 段的起始设定到 TBR。*/
}
```

扩展为汇编语言代码

```
_main:
          MOV.L       L11,R2      ; STARTOF $TBR
          RTS
          LDC         R2,TBR
L11:
          .DATA.L     STARTOF $TBR
```

3.3 内联扩展

3.3.1 函数内联扩展

描述：

函数的内联扩展为增强程序的执行速度而被使用。通常函数的调用是通过转移到包含一系列运算的代码段来进行，但此功能在函数被调用的点扩展其处理，消除了转移点的指令及增进执行速度。调用目标函数在循环内的扩展可预期为执行速度带来特别杰出的效果。程序的内联扩展有两种，如下所述。

(1) 自动内联扩展

通过在汇编时指定 "-speed" 选项，将可实行自动函数内联扩展功能，而小函数将被自动扩展。 为了对自动函数内联扩展功能施加更深入的控制，可使用 "-inline" 选项来为扩展指定函数大小。 在版本 6 以前，函数大小是以节点数量来指定（变量、运算符及其他元素的数量，不包括声明）。（"-inline" 选项的默认值是 20。） 在版本 7 以后，用户可指定允许使用内联扩展对程序大小造成的增加。

格式：

```
shc -speed [-inline=<numelic value>]...
```

(2) 根据指令的内联扩展

用于内联扩展的函数使用 "#pragma inline" 指令来指定。

格式：

#pragma inline（<函数名称>[，<函数名称>…]）

使用的实例：

在循环内调用的函数被内联扩展。

(1) 自动内联扩展

当使用 "-speed" 选项来编译以下程序时，f 被内联扩展。

C 语言代码

```
extern int *z;
int  f (int p1, int p2)             /* 要被扩展的函数  */
{
        if (p1 > p2)
            return p1;
        else if (p1 < p2)
            return p2;
        else
            return 0;
}
void  g (int *x, int *y, int count)
{
        for ( ; count>0; count--, z++, x++, y++)
              *z = f(*x, *y);
```

```
}
```

(2) 内联扩展

使用 "#pragma inline" 指令指定的函数 f1 和 f2 被扩展。

C 语言代码

```
int  v,w,x,y;
#pragma  inline(f1,f2)              /* 指定要被内联扩展的函数          */
int  f1(int a, int b)               /* 要被扩展的函数              */
{
        return (a+b)/2;
}
int  f2(int  c, int  d)             /* 要被扩展的函数              */
{
        return (c-d)/2;
}
void  g ()
{
        int i;
        for(i=0;i<100;i++){
        if(f1(x,y) == f2(v,w))
        sleep();
        }
}
```

重要信息：

(1) 应将 "#pragma inline" 指令放置在函数前。

(2) 由 "#pragma inline" 指令指定的函数被生成外部定义。 因此当为让文件中的内联扩展被多个文件包含而编写函数时，必须将函数声明为静态。

(3) 下列函数不可被内联扩展。

具有变量参数的函数

参考函数内参数地址的函数

实际与虚设参数的数量及类型不匹配的函数

通过地址调用的函数

(4) 若在 SH-1 以外的其他 CPU 上安装了高速缓存，内联扩展可能造成高速缓存遗漏，因此速度未被增进。

(5) 当使用此功能时，由于代码在函数被调用的点被扩展，因此容易使程序大小增加。 使用此功能前应谨慎考虑程序大小与执行速度之间的平衡。

3.3.2 汇编语言的内联扩展

描述：

有时 CPU 指令不被 C 语言支持，或汇编语言代码比相等的 C 代码更能提供增进的性能。在这些时候，相关的代码可以汇编语言来编写，并和 C 语言程序合并。 SuperH RISH engine C/C++ 编译程序提供内联汇编语言代码的扩展功能，以使内联汇编语言代码能够包含在 C 源程序中。
通过在和 C 语言函数相同的区域中编写汇编语言代码，在函数前放置 "#pragma inline_asm "指令，编译程序在汇编语言代码被调用时对其进行内联扩展。
函数之间的界面应符合 C/C++ 编译程序生成规则。 C/C++ 编译程序生成在寄存器 R4 至 R7 中保存参数值，并在 R0 中放置返回值的代码。对于 SH-2E、SH2A-FPU、SH-4 和 SH-4A，为单精度浮点运算的返回值设定 FR0。对于 SH2A-FPU、SH-4 和 SH-4A，为双精度浮点运算的返回值设定 DR0。

格式：

```
#pragma inline_asm   (< 函数名称 > [,< 函数名称 >...])
```

使用的实例：

当高低位字节频繁交换时，包含了性能瓶颈，可使用汇编语言编写一个字节交换函数并进行内联扩展。

C 语言代码

```
#pragma inline_asm (swap)              /* 指定要被扩展的汇编程序函数 */
short swap(short p1)                      /* 以汇编语言描述要被增进的函数 */
{
          EXTU.W       R4,R0          ; 清除高位字
          SWAP.B       R0,R2          ; 与 R0 低位字交换
          CMP/GT       R2,R0          ; 若 (R2 < R0)
          BT           ?0001          ; 然后转至 ?0001
          NOP                         ;
          MOV          R2,R0          ; 返回 R2
?0001:                                ; 本地标签  本地标签被用作标签。
                                      ;
}
void f (short *x, short *y, int i)
{
      for ( ; i > 0; i--, x++, y++)
         *y = swap(*x);               /* 以函数调用 C 的相同格式来描述 */
}
```

扩展为汇编语言代码（摘录）

```
f:
          MOV.L     R14,@-R15
          MOV       R6,R14
          MOV.L     R13,@-R15
          CMP/PL    R14
          MOV.L     R12,@-R15
          MOV       R5,R13
```

```
            MOV         R4,R12
            BT          L224
            MOV.L       L225,R3             ; L221
            JMP         @R3
            NOP
L224:
L222:
            MOV.W       @R12,R4
            BRA         L223
            NOP
L225:
            .DATA.L     L221
L223:
            EXTU.W      R4,R0
            SWAP.B      R0,R2
            CMP/GT      R2,R0
            BT          ?0001
            NOP
            MOV         R2,R0
?0001:
            .ALIGN      4
            MOV.W       R0,@R13
            ADD         #-1,R14
            ADD         #2,R12
            ADD         #2,R13
            CMP/PL      R14
            BF          L226
            MOV.L       L227+2,R3           ; L222
            JMP         @R3
            NOP
L226:
L221:
            MOV.L       @R15+,R12
            MOV.L       @R15+,R13
            RTS
            MOV.L       @R15+,R14
L227:
            .RES.W      1
            .DATA.L     L222
```

重要信息：

(1) 在对函数进行定义前应先指定 "#pragma inline_asm "。

(2) 由 "#pragma inline_asm " 指令指定的函数被生成外部定义。 因此当为让文件中的内联扩展被多个文件包含而编写函数时，必须将函数声明为静态。

(3) 任何在汇编语言中使用的标签应是本地标签。

(4) 当在以汇编语言编写的函数中使用寄存器 R8 到 R15（若是 SH-2E 也包括 FR12 至 FR15，若是 SH2A-FPU、SH-4 和 SH-4A 则包括 FR12 至 FR15 和 DR12 至 DR14）时，这些寄存器必须在汇编语言代码的起始处和结束处被保存及恢复。 有关详情，请参考第 3.15.1 (2) (c) 节“寄存器相关规则”。

(5) 不应将 RTS 包含在汇编语言函数的结束处。

(6) 当使用此功能时，应在编译时使用指定对象格式 "-code=asmcode" 的选项。

(7) 使用此功能将对 C 源代码级的调试实行限制。

(8) 有关调用 C 语言程序和汇编语言程序间函数的详情，请参考第 3.15.1 (2) 节“函数调用界面”。

(9) 有关结合 C 程序与汇编语言程序的更多信息，请参考第 3.15.1 节“汇编语言程序的相关事项”。

3.3.3 具有内联汇编函数的样品程序

以 C 编写但不具效率的程序，及无法以 C 语言编写的程序，通常会以汇编语言来编写，但通过使用内联汇编函数，这些程序可以主要以 C 编写但具有内联汇编语言代码。

内联汇编函数的优点

内联汇编函数的定义可将它们假设为 C 语言函数。
汇编语言指令可被直接结合，而不具有在调用以汇编语言编写的函数时所通常产生的任何例程调用或返回的内务操作。

内联汇编函数的缺点

在编译时，汇编源程序必须被输出。
因此 C 当地变量无法在调试时被参考。
（在编译时，若指定了 "-code=asmcode" 选项，则必须启动汇编程序。在 C 编译和汇编时，通过指定调试选项，变成可以在 C 源代码级进行逐步执行。）

有效使用内联汇编函数

建议内联汇编函数以下列形式被用作标头文件。

- 函数被声明为静态。
- 仅使用本地标签。
- 没有使汇编程序自动生成文字库的指令被编写。
- 没有在定义结束处编写 RTS（返回）指令。

格式：

```
                                                    /* 内联函数定义        */
                                                    /*  文件：inlasm.h     */
#pragma inline_asm (rev4b)
static  unsigned long rev4b(unsigned long p)
                                                    /* 函数被声明为静态    */
{
        ; 定义中的注解在汇编程序中由分号（；）表示
                SWAP.W    R4,R0
                SWAP.B    R0,R0
        ; RTS 指令未在包含在结束处
}
#pragma inline_asm(ovf)
static  unsigned long ovf()
{
?LABEL001           ; 本地标签在内联汇编函数中被使用
                    ; 本地标签：在 16 字符内，以 "?" 开始
        MOV         R4,R0
            :
        CMP/EQ      #1,R0
        BT          ?LBABEL001
}
#pragma inline_asm (ovfadd)
```

```
#ifdef  NG INLINEASM
                                                    /* 不正确的内联汇编函数定义 */
static  unsigned long ovfadd()
{
          :
       MOV.L     #H'f0000000,R0
       ; 这造成汇编程序自动生成文字库
       ; 指令在此情况下可能无法被正确扩展
       ; 若文字库在此函数的范围外被生成，对齐将被破坏
}
#else
                                                    /* 正确的内联汇编函数定义    */
static  unsigned long ovfadd()
{
             :
       MOV.L     #H'f0000000,R0
           ; 必须在此内联汇编函数定义内包含 .POOL 控制
           ; 指令
           ;指令在此情况下被正确扩展
       .POOL
           ;此 .POOL 指令造成一个文字库在此被扩展
           ; 此程序的实际代码扩展图像如下
           ;       :
           ;       MOV.L   Lxxx,R0
           ;       BRA     Lyyy
           ;       NOP
           ; Lxxx .DATA.L H'f0000000
           ; Lyyy
}
#end
```

在此引进的内联汇编函数的运算如下。 要在版本 8 或以上版本执行 64 位运算，您可以使用版本 8 中支持的加长类型及无符号加长类型。

- 64 位加法
- 64 位减法
- 64 位乘法
- 位旋转
- Endian 转换
- 乘法累加运算
- 溢出检查

为执行 64 位运算，使用了下列标头文件。

```
"longlong.h"
typedef struct{
```

```
    unsigned long H;
    unsigned long L;
  }longlong;
```

(1) 64 位加法

由于 C 语言中的整数数据类型不是 64 位数据，因此以 C 处理变得繁琐。因此下面呈现有效执行 64 位运算的内联汇编语句。

(i) 64 位数据的加法

格式： longlong addll(longlong a,longlong b)

参数： a: 64 位数据

b: 64 位数据

返回值： longlong： 64 位数据

描述： 相加 a 和 b，返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline asm (addll)
static longlong addll(longlong a,longlong b)
{
    MOV     @(0,R15),R0         ; 设定返回值结构 c 的起始地址
    MOV     @(4,R15),R1         ; 设定第一个参数    (a.H)
    MOV     @(8,R15),R2         ;                   (a.L)
    MOV     @(12,R15),R3        ; 设定第二个参数    (b.H)
    MOV     @(16,R15),R4        ;                   (b.L)
    CLRT                        ; 清除 T 位
    ADDC    R4,R2               ; 相加 R4 和 R2 的低 32 位，
                                ;然后根据进位设定 T 位。
    ADDC    R3,R1               ; 使用进位将 R3 和 R1 的高 32 位相加
    MOV     R1,@(0,R0)          ; 设定返回值        (c.H)
    MOV     R2,@(4,R0)          ;                   (c.L)
}
void main( void )
{
      longlong a,b,c;

      a.H=0xefffffff;
      a.L=0xffffffff;
      b.H=0x10000000;
      b.L=0x00000000;
      c=addll(a,b);
      printf("addll = %8X %08X  \n",c.H,c.L);
}
```

(ii) 64 位数据的加法（地址指定）

格式：　　void addllp(longlong *pa,longlong *pb,longlong *pc)
参数：　　pa: 64 位数据的地址
　　pb: 64 位数据的地址
　　pc: 用于存储结果的变量地址
返回的值：　无
描述：　　相加 pa 和 pb，在 pc 中返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline asm(addllp)
static void addllp(longlong *pa,longlong *pb,longlong *pc)
{
    MOV     @(0,R5),R0      ; 将(pa->H) 设定到R0
    MOV     @(4,R5),R1      ; 将(pa->L) 设定到R1
    MOV     @(0,R6),R2      ; 将(pb->H) 设定到R2
    MOV     @(4,R6),R3      ; 将(pb->L) 设定到R3
    CLRT                      ; 清除 T 位
    ADDC    R3,R1           ; 相加 R3 和 R1 的低 32 位，
                            ; 然后根据进位设定 T 位。
    ADDC    R2,R0           ; 使用进位将 R2 和 R0 的高 32 位相加
    MOV     R0,@(0,R4)      ; 将 R0 设定到 (pc->H)
    MOV     R1,@(4,R4)      ; 将 R1 设定到 (pc->L)
}
void main( void )
{
    longlong a,b,c;
    longlong *pa,*pb,*pc;
    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xeffffffff;
    c.L=0xffffffff;

    pa=&a;
    pb=&b;
    pc=&c;
    addllp(pa,pb,pc);
    printf("addllp = %8x %08x \n",pa->H,pa->L);
}
```

(iii)　64 位数据的加法（具有地址指定）
格式：　　void addtoll(longlong *pa,longlong b)
参数：　　*pa:　64 位数据的地址
　　b:　64 位数据

返回的值： 无
描述： 相加 b 及由 pa 指定的数据，然后将结果返回到 pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (addtoll)
static void addtoll(longlong *pa,longlong b)
{
    MOV     @(0,R4),R0      ; 将 (pa->H) 设定到 R0
    MOV     @(4,R4),R1      ; 将 (pa->L) 设定到 R1
    MOV     @(0,R15),R2     ; 将 (b.H) 设定到 R2
    MOV     @(4,R15),R3     ; 将 (b.L) 设定到 R3
    CLRT                    ; 清除 T 位
    ADDC    R3,R1           ; 相加 R3 (b.L) 和 R1 (pa->L)，
                            ; 然后根据进位设定 T 位。
    ADDC    R2,R0           ; 使用进位将 R2 (b.H) 和 R0 (pa->H) 相加
    MOV     R0,@(0,R4)      ; 将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ; 将 R1 设定到 (pa->L)
}
void main( void )
{
    longlong *pa,b,c;

    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xefffffff;
    c.L=0xffffffff;

    pa=&c;
    addtoll(pa,b);
    printf("addtoll = %8x %08x \n",pa->H,pa->L);
}
```

(iv) 64 位数据和 32 位数据的加法
格式： void addtoll32(longlong *pa,long b)
参数： *pa: 64 位数据的地址
b: 32 位数据
返回的值： 无
描述： 相加 b 及由 pa 指定的数据，然后将结果返回到由 pa 指定的地址

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(addtoll32)
static void addtoll32(longlong *pa,long b)
{
    MOV     @(0,R4),R0      ; 将 (pa->H) 设定到 R0
```

```
    MOV     @(4,R4),R1      ; 将 (pa->L) 设定到 R1
    CLRT                    ; 清除 T 位
    ADDC    R5,R1           ; 相加 R1 (pa->L) 和 R5 (b)，
                            ; 然后根据进位设定 T 位
    MOVT    R3              ; 在 R3 中设定进位
    ADD     R3,R0           ; 相加 R0 (pa->H) 和 R3 (carry)
    MOV     R0,@(0,R4)      ; 将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ; 将 R1 设定到 (pa->L)
}

void main( void )
{
    longlong *pa,c;
    long b;
    b=0x00000001;
    c.H=0xeffffffff;
    c.L=0xffffffff;
    pa=&c;
    addtoll32(pa,b);
    printf("addlltoll32 = %8x %08x  \n",pa->H,pa->L);
}
```

(2) 64 位减法

(i) 64 位数据的减法

格式： longlong subll(longlong a,longlong b)
参数： a: 64 位数据
b: 64 位数据
返回的值： longlong：64 位数据
描述： 从 a 减去 b，然后返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(subll)
static longlong subll(longlong a,longlong b)
{
    MOV       @(0,R15),R0       ; 将返回值地址设定到 R0
    MOV       @(4,R15),R1       ; 将 (a.H) 设定到 R1
    MOV       @(8,R15),R2       ; 将 (a.L) 设定到 R2
    MOV       @(12,R15),R3      ; 将 (b.H) 设定到 R3
    MOV       @(16,R15),R4      ; 将 (b.L) 设定到 R4
    CLRT                        ; 清除 T 位
    SUBC      R4,R2             ; 从 R2 (a.L) 减去 R4 (b.L)，
                                ; 并根据借位设定 T 位
    SUBC      R3,R1             ; 使用借位从 R1 (a.H) 减去 R3 (b.H)
    MOV       R1,@(0,R0)        ; 将 R1 设定到 (c.H)
    MOV       R2,@(4,R0)        ; 将 R2 设定到 (c.L)
}
void main( void )
{
    longlong a,b,c;

    a.H=0xffffffff;
    a.L=0xffffffff;
    b.H=0xffffffff;
    b.L=0xffffffff;
    c=subll(a,b);
    printf("subll = %x %08x  \n",c.H,c.L);
}
```

(ii) 64 位数据的减法（具有地址指定）

格式：　　void subtoll(longlong *pa,longlong b)
参数：　　*pa: 64 位数据的地址
　　b:　64 位数据
返回的值：　无
描述：　　从由 pa 指定的数据减去 b，然后将结果返回到 pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline asm(subtoll)
static void subtoll(longlong *pa,longlong b)
{
    MOV       @(0,R4),R0        ; 将 (pa->H) 设定到 R0
    MOV       @(4,R4),R1        ; 将 (pa->L) 设定到 R1
    MOV       @(0,R15),R2       ; 将 (b.H) 设定到 R2
    MOV       @(4,R15),R3       ; 将 (b.L) 设定到 R3
```

```
    CLRT                    ; 清除 T 位
    SUBC    R3,R1           ; 从 R1 (pa.L) 减去 R3 (b.L)，
                            ; 然后根据借位设定 T 位
    SUBC    R2,R0           ; 使用借位从 R0 (pa.H) 减去 R2 (b.H)
    MOV     R0,@(0,R4)      ; 将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ; 将 R1 设定到 (pa->L)
}
void main( void )
{
    longlong *pa,b,c;
    b.H=0xffffffff;
    b.L=0xffffffff;
    c.H=0xffffffff;
    c.L=0xffffffff;
    pa=&c;
    subtoll(pa,b);
    printf("addtoll = %8x %08x \n",pa->H,pa->L);
}
```

(iii) 从 64 位数据减去 32 位数据

格式： void subtoll32(longlong *pa,long b)
参数： *pa: 64 位数据的地址
b: 32 位数据
返回的值： 无
描述： 从由 pa 指定的数据减去 b，然后将结果返回到 pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(subtoll32)
static void subtoll32(longlong *pa,long b)
{
    MOV     @(0,R4),R0      ; 将 (pa->H) 设定到 R0
    MOV     @(4,R4),R1      ; 将 (pa->L) 设定到 R1
    CLRT                    ; 清除 T 位
    SUBC     R5,R1          ; 从 R1 (pa->L) 减去 R5 (b)，
                            ; 然后根据借位设定 T 位
    MOVT    R3              ; 设定一个借位到 R3
    SUB     R3,R0           ; 从 R0 (pa->H) 减去 R3 (borrow)
    MOV     R0,@(0,R4)      ; 将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ; 将 R1 设定到 (pa->L)
}
void main( void )
{
    longlong *pa,c;
    unsigned long b;
```

```
    pa=&c;

    c.H=0xf0000000;
    c.L=0x00000000;

    b=0x00000001;

    subtoll32(pa,b);
    printf("subll = %8x %08x  \n",pa->H,pa->L);
}
```

(3) 64 位乘法

(i) 64 位数据的乘法

格式： longlong mulll(longlong a,longlong b)
参数： a: 64 位数据
b: 64 位数据
返回的值： longlong：64 位数据
描述： 相乘 a 和 b，然后返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(mulll)
static longlong mulll(longlong a,longlong b)
{
    MOV     @(4,R15),R0         ; 将 (a.H) 设定到 R0
    MOV     @(8,R15),R1         ; 将 (a.L) 设定到 R1
    MOV     @(12,R15),R2        ; 将 (b.H) 设定到 R2
    MOV     @(16,R15),R3        ; 将 (b.L) 设定到 R3
    MUL.L   R0,R3               ; 相乘 R0 (a.H) 和 R3 (b.L)
    STS     MACL,R0             ; 代入结果（低 32 位）
    MUL.L   R2,R1               ; 相乘 R1 (a.L) 和 R2 (b.H)
    STS     MACL,R2             ; 代入结果（低 32 位）
    ADD     R2,R0               ;
    DMULU   R1,R3               ; 相乘 R1 (a.L) 和 R3 (b.L)
    STS     MACH,R1             ; 代入结果（高 32 位）
    STS     MACL,R3             ; 代入结果（低 32 位）
    ADD     R1,R0               ;
    MOV     @(0,R15),R4         ;
    MOV     R0,@(0,R4)          ; 将 R0 设定到 (c.H)
    MOV     R3,@(4,R4)          ; 将 R3 设定到 (c.L)
}
void main( void )
{
    longlong a,b,c;
```

```
    a.H=0x7fffffff;
    a.L=0xffffffff;
    b.H=0x00000000;
    b.L=0x00000002;

    c=mulll(a,b);
    printf("mulll = %8x %08x  \n",c.H,c.L);
}
```

(ii) 64 位数据的乘法（指定了地址）

格式：　　void multoll(longlong *pa,longlong b)
参数：　　pa: :64 位数据的地址
　　b: 64 位数据
返回的值：　无
描述：　　相乘 b 及由 pa 指定的数据，然后将结果返回到由 pa 指定的地址

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(multoll)
static void multoll(longlong *pa,longlong b)
{
    MOV     @(0,R4),R0      ;将 (pa->H) 设定到 R0
    MOV     @(4,R4),R5      ;将 (pa->L) 设定到 R5
    MOV     @(4,R15),R1     ;将 (b.L) 设定到 R1
    MUL     R0,R1           ;相乘 R0 (pa->H) 和 R1 (b.L)
    STS     MACL,R3         ;
    DMULU   R5,R1           ;相乘 R5 (pa->L) 和 R1 (b.L)
    STS     MACH,R0         ;代入结果（高 32 位）
    STS     MACL,R1         ;代入结果（低 32 位）
    ADD     R3,R0           ;
    MOV     R0,@(0,R4)      ;将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ;将 R1 设定到 (pa->L)

}
void main( void )
{
    longlong *pa,b,c;
    c.H=0x0000ffff;
    c.L=0xffff0000;
    b.H=0x00000000;
    b.L=0x00010000;

    pa=&c;
    multoll(pa,b);
    printf("multoll = %8x %08x \n",pa->H,pa->L);
```

```
}
```

(iii) 64 位数据与无符号 32 位数据的相乘

格式： void multoll32(longlong *pa,unsigned long b)
参数： *pa: :64 位数据的地址
b: 无符号 32 位数据
返回的值： 无
描述： 相乘 b 及由 pa 指定的数据，然后将结果返回到由 pa 指定的地址

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (multoll32)
static void multoll32(longlong *pa,unsigned long b)
{
    MOV     @(0,R4),R0      ;将 (pa->H) 设定到 R0
    MOV     @(4,R4),R1      ;将 (pa->L) 设定到 R1
    ADDC    R5,R1           ;相加 R1 (pa->L) 和 R5 (b)
                            ;然后根据进位设定 T 位
    MOVT    R3              ;在 R3 中设定进位
    ADD     R3,R0           ;相加 R0 (pa->H) 和 R3 (carry)
    MOV     R0,@(0,R4)      ;将 R0 设定到 (pa->H)
    MOV     R1,@(4,R4)      ;将 R1 设定到 (pa->L)
void main( void )
{
    longlong *pa,c;
    unsigned long b;

b=0xffffff00;
    c.H=0x00000000;
    c.L=0x00000100;
    pa=&c;
    multoll32(pa,b);
    printf("mulltoll32 = %8x %08x \n",pa->H,pa->L);
}
```

(iv) 无符号 32 位数据的乘法

格式： longlong mul64(unsigned long a,unsigned long b)
参数： a: 无符号 32 位数据
b: 无符号 32 位数据
返回的值： longlong：64 位数据
描述： 相乘 a 和 b，然后返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(mul64)
static longlong mul64(unsigned long a,unsigned long b)
{
    MOV    @(0,R15),R0  ;将 c 的地址设定到 R0
    DMULU  R4,R5        ;相乘 R4 (a) 和 R5 (b)
    STS    MACH,R1      ;代入结果（高 32 位）
    MOV    R1,@(0,R0)   ;将 R1 设定到 (c.H)
    STS    MACL,R2      ;代入结果（低 32 位）
    MOV    R2,@(4,R0)   ;将 R2 设定到 (c.L)
}
void main( void )
{
    longlong c;
    unsigned long a,b;

a=0xffffffff;
    b=0x10000000;
    c=mul64(a,b);
    printf("mul64 = %8x %08x  \n",c.H,c.L);
}
```

（v） 带符号 32 位数据的乘法

格式： longlong mul64s(signed long a,signed long b)
参数： a: 32 位数据
b: 32 位数据
返回的值： longlong： 64 位数据
描述： 相乘 a 和 b，然后返回结果

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(mul64s)
static longlong mul64s(signed long a,signed long b)
{
    MOV    @(0,R15),R0   ;将 c 的地址设定到 R0
    DMULS  R4,R5         ;带符号相乘 R4 (a) 和 R5 (b)
    STS    MACH,R1       ;代入结果（高 32 位）
    MOV    R1,@(0,R0)    ;将 R1 设定到 (c.H)
    STS    MACL,R2       ;代入结果（低 32 位）
    MOV    R2,@(4,R0)    ;将 R2 设定到 (c.L)
}
void main( void )
{
    longlong c;
```

```
    signed long a,b;
    a = -1;
    b=1;
    c=mul64s(a,b);
    printf("mul64s = %8x %08x \n",c.H,c.L);
}
```

(4) 位旋转

(i) 将数据的 8 位向左旋转一位

格式： short rot8l(unsigned long a)
参数： a: 无符号 8 位数据
返回的值： short: 8 位数据
描述： 向左旋转一位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot8l)
unsigned char rot8l(unsigned char a)
{
    ROTL    R4          ;左移 1 位
    MOV     R4,R0       ;
    SHLR8   R0          ;右移 8 位
    OR      R4,R0       ;
}
void main( void )
{
    unsigned char a;

   a=0x12;
    a=rot8l(a);
    printf(" rot8l %x \n",a);

}
```

(ii) 将数据的 8 位向左旋转 n 位

格式： short rot8ln(unsigned char a,int n)
参数： a: 无符号 8 位数据
n: 移位量
返回的值： short: 8 位数据
描述： 向左旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot8ln)
unsigned char rot8ln(unsigned char a,int n)
{
    MOV     #0,R1     ;设定计数器寄存器
?LOOP:
    ROTL    R4        ;左移 1 位
    MOV     R4,R2     ;
    SHLR8   R2        ;右移 8 位
    ADD     #1,R1     ;使计数器寄存器增加 1
    CMP/EQ  R1,R5     ;若 R1==R5，将 T 设定到 1
    BF      ?LOOP     ;若 T!=1，转移到 LOOP
    OR      R2,R4     ;在转移前执行
    MOV     R4,R0     ;设定返回值
}
void main( void )
{
    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8ln(a,n);
    printf("b: %x \n",b);
}
```

(iii) 将数据的 8 位向右旋转一位

格式：　　short rot8r(unsigned char a)
参数：　　a: 无符号 8 位数据
返回的值：　short: 8 位数据
描述：　　向右旋转一位，并返回结果

```
#pragma inline_asm(rot8r)
unsigned char rot8r(unsigned char a)
{
    ROTR    R4          ;右移 1 位
    MOV     R4,R0       ;
    SHLR16  R4          ;右移 16 位
    SHLR8   R4          ;右移 8 位
    OR      R4,R0       ;
}
void main( void )
{
    unsigned char a;
```

```
    a=0x12;
    a=rot8r(a);
    printf(" rot8r %x \n",a);

}
```

(iv) 将数据的 8 位向右旋转 n 位

格式： short rot8rn(unsigned char a,int n)
参数： a: 无符号 8 位数据
n: 移位量
返回的值： short: 8 位数据
描述： 向右旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot8rn)
unsigned char rot8rn(unsigned char a,int n)
{
    MOV     #0,R1       设定计数器寄存器
?LOOP:
    ROTR    R4          ;右移 1 位
    MOV     R4,R2       ;
    SHLR16  R2          ;右移 16 位
    SHLR8   R2          ;右移 8 位
    ADD     #1,R1       ;使计数器寄存器增加 1
    CMP/EQ  R1,R5       ;若 R1==R5，将 T 设定到 1
    BF      ?LOOP       ;若 T!=1，转移到 LOOP
    OR      R2,R4       ;在转移前执行
    MOV     R4,R0       ;设定返回值
}
void main( void )
{
    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8rn(a,n);
    printf(" rot8rn %x \n",b);
}
```

（v） 将数据的 16 位向左旋转一位

格式： short rot16l(unsigned short a)
参数： a: 无符号 16 位数据地址
返回的值： short: 16 位数据
描述： 向左旋转一位，并返回结果

```
#pragma inline_asm(rot16l)
unsigned short rot16l(unsigned short a)
{
    ROTL    R4          ;左移 1 位
    MOV     R4,R0       ;
    SHLR16  R0          ;右移 16 位
    OR      R4,R0       ;
}
void main( void )
{
    unsigned short a,b;
    a=0x1234;
    b=rot16l(a);
    printf(" rot16l = %x \n",b);
}
```

(vi) 将数据的 16 位向左旋转 n 位

格式： short rot16ln(unsigned short a,int n)
参数： a: 无符号 16 位数据地址
n: 移位量
返回的值： short: 16 位数据
描述： 向左旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot16ln)
unsigned short rot16ln(unsigned short a,int n)
{
    MOV     #0,R1       ;设定计数器寄存器
?LOOP:
    ROTL    R4          ;左移 1 位
    MOV     R4,R2       ;
    SHLR16  R2          ;右移 16 位
    ADD     #1,R1       ;使计数器寄存器增加 1
    CMP/EQ  R1,R5       ;若 R1==R5，将 T 设定到 1
    BF      ?LOOP       ;若 T!=1，转移到 LOOP
    OR      R2,R4       ;
    MOV     R4,R0       ;设定返回值
}
void main( void )
{
    unsigned short a,b;
    int n;

    a=0x1234;
    n=8;
```

```
    b=rot16ln(a,n);
    printf("rot16ln = %x \n",b);
}
```

(vii) 将数据的 16 位向右旋转一位

格式： short rot16r(unsigned short a)
参数： a: 无符号 16 位数据地址
返回的值： short: 16 位数据
描述： 向右旋转一位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot16r)
unsigned short rot16r(unsigned short a)
{
    ROTR    R4          ;右移 1 位
    MOV     R4,R0       ;
    SHLR16  R0          ;右移 16 位
    OR      R4,R0       ;
}
void main( void )
{
    unsigned short a,b;
    a=0x1234;
    b=rot16r(a);
    printf("rot16r = %x \n",b);
}
```

(viii) 将数据的 16 位向右旋转 n 位

格式： short rot16rn(unsigned short a,int n)
参数： a: 无符号 16 位数据地址
n: 移位量
返回的值： short: 16 位数据
描述： 向右旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot16rn)
unsigned short rot16rn(unsigned short a,int n)
{
    MOV     #0,R1       ;设定计数器寄存器
?LOOP:
    ROTR    R4          ;右移 1 位
    MOV     R4,R2       ;
    SHLR16  R2          ;右移 16 位
    ADD     #1,R1       ;使计数器寄存器增加 1
    CMP/EQ  R1,R5       ;若 R1==R5，将 T 设定到 1
```

```
    BF      ?LOOP       ;若 T!=1，转移到 LOOP
    OR      R2,R4       ;
    MOV     R4,R0       ;设定返回值
}
void main( void )
{
    unsigned short a,b;
    int n;

    a=0x1234;
    n=8;
    b=rot16rn(a,n);
    printf("rot16rn %x \n",b);
}
```

(ix) 将数据的 32 位向左旋转一位

格式：　　short rot32l(unsigned long a)
参数：　　a: 无符号 32 位数据地址
返回的值：　short: 32 位数据
描述：　　向左旋转一位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot32l)
unsigned long rot32l(unsigned long a)
{
    ROTL    R4          ;左移 1 位
    MOV     R4,R0       ;设定返回值
}
void main( void )
{
    unsigned long a;
    a=0x12345678;
    a=rot32l(a);
    printf(" rot32l %8x \n",a);
}
```

(x) 将数据的 32 位向左旋转 n 位

格式：　　short rot32ln(unsigned long a,int b)
参数：　　a: 无符号 32 位数据地址
　　b: 移位量
返回的值：　short: 32 位数据
描述：　　向左旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot32ln)
unsigned long rot32ln(unsigned long a,int b)
{
    MOV     #0,R1         ;设定计数器寄存器
?LOOP:
    ROTL    R4            ;右移 1 位
    ADD     #1,R1         ;使计数器寄存器增加 1
    CMP/EQ  R1,R5         ;若 R1==R5，将 T 设定到 1
    BF      ?LOOP         ;若 T!=1，转移到 LOOP
    MOV     R4,R0         ;设定返回值
}
void main( void )
{
    unsigned long a;
    int    b;
    a=0x12345678;
    b=16;
    a=rot32ln(a,b);
    printf(" rot32ln %8x \n",a);
}
```

(xi) 将数据的 32 位向右旋转一位

格式：　　short rot32r(unsigned long a)
参数：　　a: 无符号 32 位数据地址
返回的值：　short: 32 位数据
描述：　　向右旋转一位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot32r)
unsigned long rot32r(unsigned long a)
{
    ROTR    R4          ;右移 1 位
    MOV     R4,R0       ;设定返回值
}
void main( void )
{
    unsigned long a,b;
    a=0x12345678;
    b=rot32r(a);
    printf(" rot32r %8x \n",b);
}
```

(xii) 将数据的 32 位向右旋转 n 位

格式： short rot32rn(unsigned long a,int b)
参数： a: 无符号 32 位数据地址
b: 移位量
返回的值： short: 32 位数据
描述： 向右旋转 n 位，并返回结果

```
#include <stdio.h>
#pragma inline_asm(rot32rn)
unsigned long rot32rn(unsigned long a,int b)
{
    MOV     #0,R1          ;设定计数器寄存器
?LOOP:
    ROTR    R4             ;右移 1 位
    ADD     #1,R1          ;使计数器寄存器增加 1
    CMP/EQ  R1,R5          ;若 R1==R5，将 T 设定到 1
    BF      ?LOOP          ;若 T!=1，转移到 LOOP
    MOV     R4,R0          ;设定返回值
}
void main( void )
{
    unsigned long a;
    int    b;
    a=0x12345678;
    b=16;
    a=rot32rn(a,b);
    printf("rot32rn %8x \n",b);
}
```

(5) Endian 转换

(i) 交换高 16 位及低 16 位

格式： unsigned long swap(unsigned long a)
参数： a: 无符号 32 位数据
返回的值： unsigned long： 无符号 32 位数据
描述： 交换 a 的高及低 16 位

```
#include <stdio.h>
#pragma inline_asm  (swap)
static unsigned long swap(unsigned long a)
{
    SWAP.W  R4,R0    ;交换 R4 的高 16 位及 R0 的低 16 位
}
void main( void )
{
```

```
    unsigned long a,b;
    a=0xaaaabbbb;
    b=swap(a);
    printf("b: %8x \n",b);
}
```

(ii) 高 16 位及低 16 位的对称交换

格式： unsigned long swapbit(unsigned long a)
参数： a: 无符号 32 位数据
返回的值： unsigned long： 无符号 32 位数据
描述： 个别交换 a 的高及低 16 位

32bit 1bit , 1bit 32bit
31bit 2bit , 2bit 31bit
:
:
18bit 15bit , 15bit 18bit
17bit 16bit , 16bit 17bit

```
#include <stdio.h>

#pragma inline_asm (swapbit)
static unsigned long swapbit(unsigned long a)
{
    MOV     #0,R0       ;设定计数器寄存器
?LOOP:
    ROTCL   R4          ;向左旋转
    ROTCR   R1          ;向右旋转
    ADD     #1,R0       ;使计数器寄存器增加 1
    CMP/EQ  #32,R0      ;若 32==R0，将 T 设定到 1
    BF      ?LOOP       ;若 T!=1，转移到 LOOP
    NOP                 ;
    MOV     R1,R0       ;设定返回值

}
void main( void )
{
    unsigned long a,b;

    a=0x1234;
    b=swapbit(a);

    printf("b: %8x \n",b);

}
```

(iii) Endian 转换

格式： unsigned long swapbyte(unsigned long a)
参数： a: 无符号 32 位数据
返回的值： unsigned long： 无符号 32 位数据
描述： 对 a 进行 Endian 转换，然后返回结果

```
#include <stdio.h>
#pragma inline_asm (swapbyte)
static unsigned long swapbyte(unsigned long a)
{

    SWAP.B   R4,R4          ;交换 R4 的位 0 至 7 及 R4 的位 8 至 15 的数据
    SWAP.W   R4,R4          ;交换 R4 的高 16 位及 R4 的低 16 位
    SWAP.B   R4,R0          ;交换 R4 的位 0 至 7 及 R0 的位 8 至 15 的数据

}
void main( void )
{
    unsigned long a,b;
    a=0xaabbccdd;
    b=swapbyte(a);
    printf("b: %8x \n",b);
}
```

(6) 乘法累加运算

(i) 32 位数据数组的乘法累加运算

格式： long macl32h(long *pa,long *pb,int size)
参数： *pa: :32 位数据数组的起始地址
*pb: :32 位数据数组的起始地址
大小： 数组大小
返回的值： long: 32 位数据
描述： 对数据数组 *pa 及 *pb 执行乘法累加，返回 64 位数据结果的高 32 位

```
#include <stdio.h>
#pragma inline_asm (macl32h)
static long macl32h(long *pa,long *pb,int size)
{
    MOV     #0,R1           ;设定计数器寄存器
    CLRMAC                  ;初始化 MAC
?LOOP:
    MAC.L   @R4+,@R5+       ;乘法累加
    ADD     #1,R1           ;使计数器寄存器增加 1
    CMP/EQ  R1,R6           ;若 R1==R6，将 T 设定到 1
    BF      ?LOOP           ;若 T!=1，转移到 LOOP
    NOP                     ;
```

```
    STS     MACH,R0      ;代入结果
}
void main( void )
{
    int size=3;
    long c;
    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};

    c=macl32h(pa,pb,size);
    printf("macl32h = %8x \n",c);
}
```

(ii) 无符号数据数组的乘法累加运算

格式： longlong macl64(long *pa,long *pb,int size)
参数： *pa: :32 位数据数组的起始地址
*pb: :32 位数据数组的起始地址
大小： 数组大小
返回的值： longlong： 64 位数据
描述： 对 32 位数据数组 *pa 和 *pb 执行乘法累加，并保存结果。

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (macl64)
static longlong macl64(long *pa,long *pb,int size)
{
    MOV     #0,R0              ;设定计数器寄存器
    MOV     @(0,R15),R1        ;设定第一个参数的地址
    CLRMAC                     ;初始化 MAC 寄存器
?LOOP:
    MAC.L   @R4+,@R5+          ;执行乘法累加运算，然后修改地址
    ADD     #1,R0              ;使计数器寄存器增加 1
    CMP/EQ  R0,R6              ;若 R0==R6，将 T 设定到 1
    BF      ?LOOP              ;若 T!=1，转移到 LOOP
    NOP                        ;
    STS     MACH,R2            ;设定结果的高 32 位
    MOV     R2,@(0,R1)         ;
    STS     MACL,R3            ;设定结果的低 32 位
    MOV     R3,@(4,R1)         ;
}
void main( void )
{
    longlong c;
    int size=3;
    long *pa,*pb;
```

```
    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};
    c=macl64(pa,pb,size);
    printf("macl64 = %8X %08X  \n",c.H,c.L);
}
```

(7) 溢出检查

(i) 32 位数据加法的溢出检查

格式： long addovf(long a,long b)
参数： a: 用于加法的 32 位数据
b: 用于加法的 32 位数据
返回的值： long: 32 位数据
描述： 相加 a 和 b，返回结果。

若产生溢出，返回最大值 (7FFFFFFF)。
若产生下溢，返回最小值 (80000000)。
判断是以信号位的更改为根据。

```
#include <stdio.h>
#pragma inline_asm (addovf)
static long addovf(long a,long b)
{
    ADDV    R4,R5           ;执行带符号的加法
                            ;然后根据信号位的更改来设定 T 位
    BF      ?RETURN         ;若 T==0，转移到 OVER
    MOV     #0,R1           ;
    CMP/GT  R4,R1           ;若 R1>R4，设定 T 位
    BF      ?OVER           ;若 T==0，转移到 OVER
    NOP                     ;
    MOV.L   ?DATA+4,R5      ;将 R5 设定到 (H'7FFFFFFF)
    BRA     ?RETURN         ;转移到 RETURN
    NOP                     ;
?OVER:
    MOV.L   ?DATA,R5        ;将 R5 设定到 (H'80000000)
?RETURN:
    MOV     R5,R0           ;将 R5 设定到 R0
    BRA     ?OWARI          ;转移到 OWARI
    NOP                     ;
?DATA:
    .ALIGN  4
    .RES.L   1
    .DATA.L  H'7FFFFFFF
    .DATA.L  H'80000000
?OWARI:
```

```
}
void main( void )
{
    long a,b,c;
    a=0x3000000;
    b=0x2000000;
    c=addovf(a,b);
    printf("c: %x \n",c);
}
```

3.4 寄存器指定

有时需要对频繁存取外部变量的模块增进执行速度。 在这种情况下，指定全局基址变量 (GBR) 的功能将被使用，全局基址寄存器 (GBR) 被用于相对寻址模式中以参考频繁存取的数据。 GBR 所参考的变量被分配到 $G0、$G1 段，并使用保存在 GBR 中 $G0 段起始地址的偏移来参考。相对于需要参考加载地址的代码，这将产生更快及更简练的代码，因此增加了执行速度和 ROM 的使用效率。

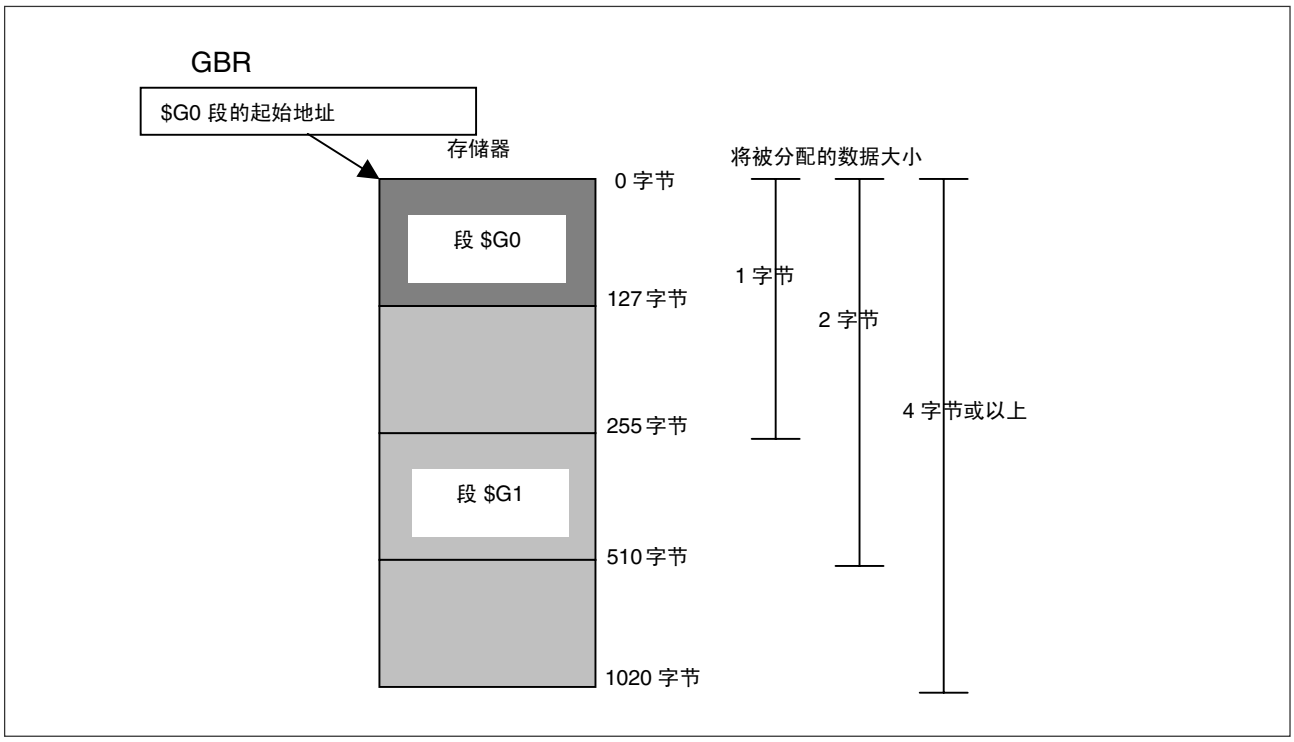


表 3.2 GBR 基址变量参考

3.4.1 GBR 基址变量的指定

描述：

预处理程序指令为实行外部变量的 GBR 基址参考而被使用。
"#pragma gbr_base" 指令指定变量从 GBR 所示地址的偏移是 0 至 127 字节。 在此指定的变量被分配到段 $G1。
"#pragma gbr_base1" 指令指定变量从 GBR 所示地址的偏移是，对于 char 类型及 unsigned char 类型，最多 255 字节；对于 short 及 unsigned short 类型，最多 510 字节； 而对于 int、unsigned int、long、unsigned long、float 和 double 类型，最多 1020 字节。 在此指定的变量被分配到段 $G1。

格式：

```
#pragma gbr_base ( 变量名称 > [,< 变量名称 >...] )
#pragma gbr_base1 ( 变量名称 > [,< 变量名称 >...] )
```

使用的实例：

C 语言代码

```
#pragma gbr_base(a1,b1,c1)
#pragma gbr_base1(a2,b2,c2)
char a1,a2;
short b1,b2;
long c1,c2;
void f()
{
     a1 = a2;
     b1 = b2;
     c1 = c2;
}
```

扩展为汇编语言代码

```
f:
          MOV.B       @(_a2-(STARTOF $G0),GBR),R0
          MOV.B       R0,@(_a1-(STARTOF $G0),GBR)
          MOV.W       @(_b2-(STARTOF $G0),GBR),R0
          MOV.W       R0,@(_b1-(STARTOF $G0),GBR)
          MOV.L       @(_c2-(STARTOF $G0),GBR),R0
          RTS
          MOV.L       R0,@(_c1-(STARTOF $G0),GBR)
```

为使用 GBR 基址变量，需要预先将 $G0 段的起始地址设定为 GBR 寄存器。 下面显示一个实例。

初始化程序（汇编语言部分）

```
                :
               .SECTION   $G0,DATA,ALIGN=4
                :
__G_BGN:       .DATA.L    (STARTOF $G0)        ;$G0 段的起始地址
                :                               ;指定地址
               .EXPORT    __G_BGN
                :
                .END
```

初始化程序（C 语言部分）

```
#include <machine.h>
extern int *_G_BGN;
void  INITSCT()          /* 在 main 函数前执行的函数 */
{
    :
set_gbr( G_BGN);         /* 指定 $G0 段在 GBR 寄存器中的起始地址 */
  :
}
```

重要信息：

在使用此功能时，必须遵守下列规则。

(1) 在程序开始执行处，GBR 应被设定到 $G0 段的起始地址。
(2) 应始终在连接上将 $G1 段紧接着 $G0 段放置。 即使在仅使用 "#pragma gbr_base1" 时，必须始终创建 $G0 段。
(3) 若 $G0 段在连接后的总大小超过 128 字节，或有偏移超出在 "#pragma gbr_base1" 的解释中为不同数据类型所指定偏移的变量，将无法保证运算的正确。
若无法满足上述 2 和 3，将无法保证运算的正确，应检查在连接时输出的映像列表以确认它们获得满足。
(5) 特别频繁存取的数据，以及将执行位运算的数据，应尽可能被分配到段 $G0。当数据被分配到段 $G0 而非段 $G1 时，创建了执行速度和大小更具效率的目标文件。
(6) 被指定 "#pragma gbr_base" 或 "#pragma gbr_base1" 的变量按照变量的声明顺序被分配到段。 应谨记若具有不同大小的变量被交替声明将会使数据大小增加。
(7) 当指定了 gbr=auto 时，#pragma gbr_base 或 #pragma gbr_base1 的指定将无效（版本 7 或以上）。

3.4.2 全局变量的寄存器分配

描述：

由〈变量名称〉指定的全局变量被分配到由〈寄存器名称〉指定的寄存器。

格式：

```
#pragma global_register(< 变量名称 >=< 寄存器名称 >,...)
```

使用的实例：

C 语言代码

```
#pragma global_register(x=R13,y=R14)
      int    x;
      char   *y;
func1()
{
             x++;
}
func2()
{
             *y=0;
}
func(int a)
{
             x = a;
             func1();
             func2();
}
```

扩展为汇编语言代码

```
          .EXPORT      func1
          .EXPORT      func2
          .EXPORT      func
          .SECTION     P,CODE,ALIGN=4
  func1:                              ; 函数：func1
                                      ; 帧大小 = 0
          RTS
          ADD          #1,R13
  func2:                              ; 函数：func2
                                      ; 帧大小 = 0
          MOV          #0,R3
          RTS
          MOV.B        R3,@R14
  func:                               ; 函数：func
```

```
                                ; 帧大小=4
        STS.L       PR,@-R15
        BSR         func1
        MOV         R4,R13
        BRA         func2
        LDS.L       @R15+,PR
        .SECTION    B,DATA,ALIGN=4
        .END
```

重要信息：

(1) 简单类型及指针类型的变量可被用作全局变量。除非指定了 "-double=float" 选项，否则无法指定双精度类型变量（除了 SH2A-FPU、SH-4 和 SH-4A）。
(2) 可被指定的寄存器是 R8 至 R14、FR12 至 FR15（对于 SH-2E、SH2A-FPU、SH-4 和 SH-4A），及 DR12 至 DR14（对于 SH2A-FPU、SH-4 和 SH-4A）。
(3) 无法设定初始值。 同时，地址无法被参考。
(4) 无法保证从连接文件对指定变量的参考。
(5) 可指定静态数据成员，但无法指定非静态数据成员。

可在 FR12 至 FR15 中设定的变量类型

(i) 对于 SH-2E
浮点类型
双精度类型（当指定了 double=float 选项时）
(ii) 对于 SH2A-FPU、SH-4 和 SH-4A
浮点类型（没有 fpu=double 选项）
双精度类型（具有 fpu=single 选项）

可在 DR12 至 DR15 中设定的变量类型

(i) 对于 SH2A-FPU、SH-4 和 SH-4A
浮点类型（具有 fpu=double 选项）
双精度类型（没有 fpu=single 选项）

3.5 寄存器保存/恢复操作的控制

描述：

在被其他函数调用而不执行其他处理的函数中，寄存器有时不需被保存及恢复以进一步增进程序执行速度。在这种情况下，预处理程序指令 "#pragma noregsave"、"#pragma noregalloc" 和 "#pragma regsave" 被用于对寄存器保存/恢复运算进行更全面的控制。

(1) "#pragma noregsave" 指令指定一般用途寄存器在函数的入口及出口点不被保存和恢复。
(2) "#pragma noregalloc" 指令被用于创建不会在函数入口/出口点保存/恢复一般用途寄存器，同时不会为函数调用间的寄存器变量（R8 至 R14）分配寄存器的对象。
(3) "#pragma regsave" 指令被用于创建会在函数入口/出口点保存及恢复 R8 至 R14 的一般用途寄存器，同时不会为寄存器变量（R8 至 R14）分配寄存器的对象。
(4) "#pragma regsave" 和 "#pragma noregalloc" 可同时为相同的函数指定。 这类重叠指定使创建了一个会在函数入口/出口点为寄存器变量（R8 至 R14）保存及恢复所有寄存器，同时不会在函数调用间分配寄存器变量寄存器的对象。

格式：

```
#pragma noregsave(< 函数名称 > [,< 函数名称 >...])
#pragma noregalloc(< 函数名称 > [,< 函数名称 >...])
#pragma regsave(< 函数名称 > [,< 函数名称 >...])
```

使用的实例：

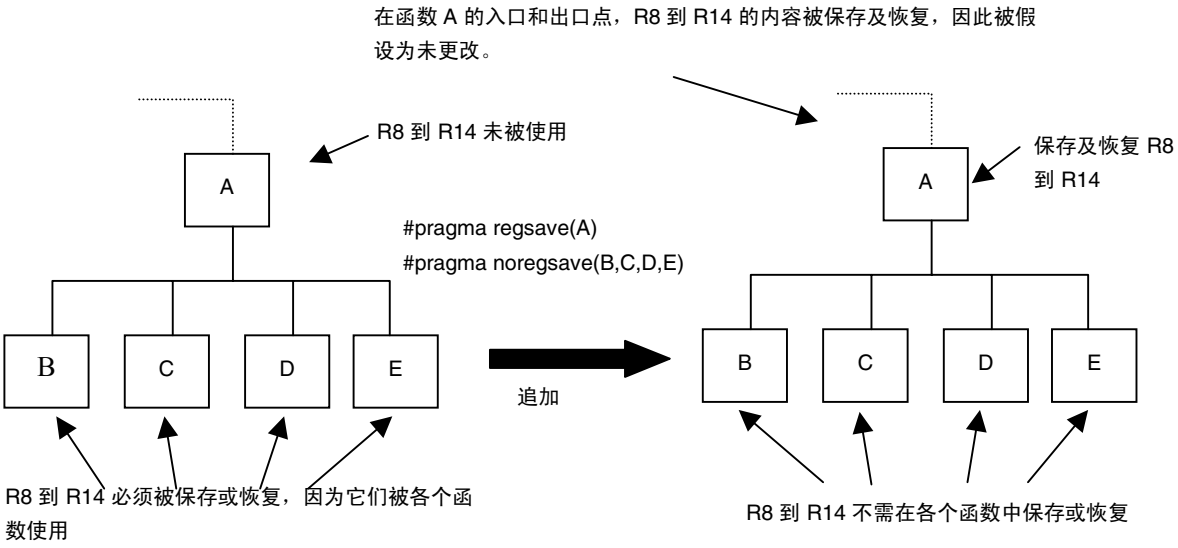
下面的实例显示需要消除寄存器存储/恢复，或创建可将其消除的条件的状况。

实例 1
当在启动时运行的函数中使用了寄存器 R8 至 R14 时，即没有保存及恢复寄存器的需要，而通过指定 "#pragma noregsave" 也使对象大小缩减，同时执行速度获得增进。

实例 2
当在不对调用源函数返回控制而使系统进入低功耗模式的函数中使用寄存器 R1 至 R8 或类似的状况中时，没有需要进行寄存器保存/恢复运算。 因此通过指定 "#pragma noregsave"，对象大小可被缩减而执行速度获得增进。

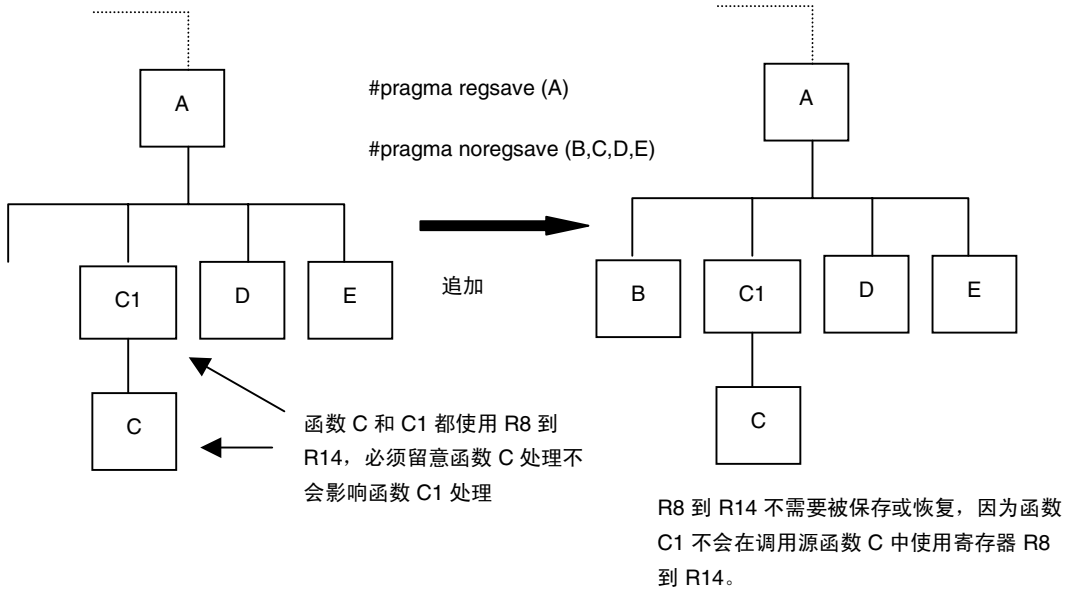
实例 3

当寄存器 R8 至 R14 不由函数 A 分配，但由函数 B、C、D 和 E 分配时，将生成为 R8 至 R14 执行保存/恢复运算的对象。函数 A 不使用 R8 至 R14，因此即使由函数 A 调用的函数没有保存/恢复寄存器也不会带来反效果，但调用函数 A 的函数在有些情况下会使用寄存器。 可以在这种情况下添加指令以便在函数 A 的入口及出口点执行保存/恢复，但不为被函数 A 调用的各个函数执行。



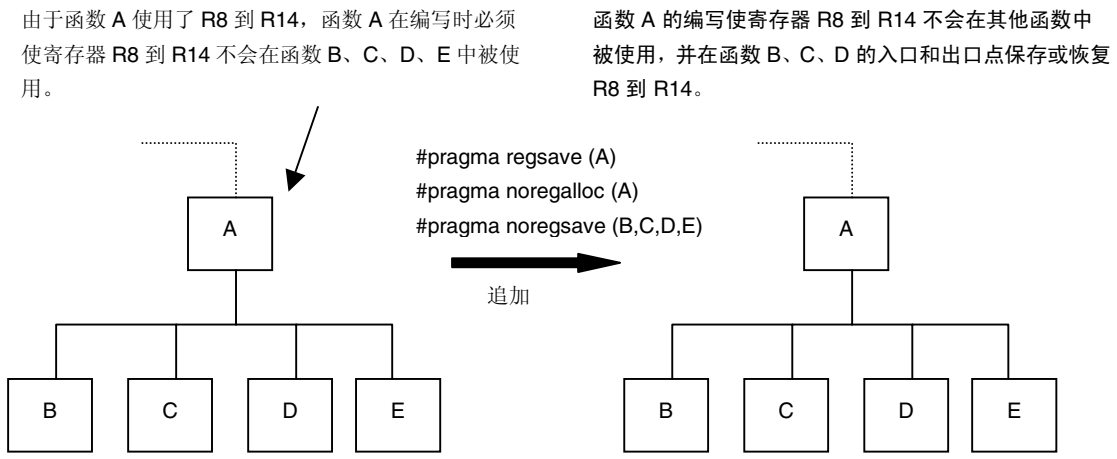
实例 4

当使用与上面实例 3 相同的调用关系，函数 C 及 C1 使用寄存器 R8 至 R14 时，函数 C1 不应在调用源函数 C 中使用寄存器 R8 至 R14。在这种情况下，若将 "#pragma noregalloc" 指令和函数 C1 一同使用，需指定 R8 至 R14 的分配未超出函数调用，然后即可使用 "pragma noregsave" 指令来指定函数 C。



实例 5

当在与上面实例 3 相同的调用关系中，寄存器 R8 至 R14 被用于函数 A 时，函数 A 的编写必须使函数 B、C、D、E 不会使用寄存器 R8 至 R14。在这种情况下，"#pragma regsave" 和 "#pragma noregalloc" 的指定被用于函数 A。通过指定 "#pragma regsave" 和 "#pragma noregalloc"，寄存器 R8 至 R14 在函数入口/出口点被保存和恢复，同时输出了不会对函数调用分配 R8 至 R14 的代码，以便可以为函数 B、C、D、E 指定 "#pragma noregsave"。



重要信息：

通过以下所列以外的方法调用使用 "#pragma noregsave" 所指定函数的结果将不被保证。

(1) 当用作不被任何其他函数调用的第一个启动的函数
(2) 当从使用 "#pragma regsave" 指定的函数进行调用
(3) 当通过使用 "#pragma noregalloc" 指定的函数，从使用 "#pragma regsave" 指定的函数进行调用

3.6 16/20/28/32 位地址区域的指定

描述：

预处理程序指令可被用来向编译程序指定被外部参考的变量及函数地址是 16、20、28 或 32 位。
编译程序假设使用 "#pragma abs16" 声明的标识符可被表示为 16 位地址，同时仅分配 16 位的地址存储空间，而一般上会分配 32 位。 这么一来，对象大小可被缩减，使 ROM 更具使用效率。
此外，若存储器在设计时的分配使被多个函数参考的变量及函数被优先放置在由 16 位表示的地址中，这项功能将可被有效使用。
SuperH RISC engine C/C++ 编译程序从版本 4.1 开始添加了 16 位地址转换选项。此选项也可被用于多重指定。有关详情，请参阅附录 B。
在版本 9 或以上版本中，可在 SH-2A 和 SH2A-FPU 中指定 20/28 位地址区域。此选项也可用于多重指定。

格式：

```
<Preprocessor directive>
    #pragma abs16 (<identifier> [,<identifier>...])
    #pragma abs20 (<identifier> [,<identifier>...])
    #pragma abs28 (<identifier> [,<identifier>...])
    #pragma abs32 (<identifier> [,<identifier>...])
             identifier: variable name | function name
<Options>
    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 }[,...]
             The default is abs32=all.
```

使用的实例：

实例 1：

外部存取的变量及函数被指定为具有 16 位地址。

C 语言代码

```
#pragma abs16 (x,y,z)
extern int x();
int y;
long z;
f ()
{
   z = x() + y;
}
```

扩展为汇编语言代码

```
_f:
            STS.L       PR,@-R15
            MOV.W       L218,R3         ;加载 x 地址
            JSR         @R3
            NOP
            MOV.W       L218+2,R3       ;加载 y 地址
            MOV.L       @R3, R2
            MOV.W       L218+4,R1       ;加载 z 地址
            ADD         R2,R0
            LDS.L       @R15+,PR
            RTS
            MOV.L       R0,@R1
L218 :
            .DATA.W     _x
            .DATA.W     _y
            .DATA.W     _z
```

实例 2：

外部存取的变量及函数被指定为具有 20 位地址。

C 语言代码

```
#pragma abs20 (x,y,z)
extern int x();
int y;
long z;
f()
{
     z = x() + y;
}
```

扩展为汇编语言代码

```
_f:
            STS.L           PR,@-R15
            MOVI20          #_x,R2          ; 加载 x 地址
            JSR/N           @R2
            MOVI20          #_y,R5          ; 加载 y 地址
            MOV.L           @R5,R1
            MOVI20          #_z,R4          ; 加载 z 地址
```

```
            ADD             R1,R0
            LDS.L           @R15+,PR
            RTS
            MOV.L           R0,@R4
_y:
            .RES.L          1
_z:
            .RES.L          1
```

实例 3：

外部存取的变量及函数被指定为具有 28 位地址。

C 语言代码

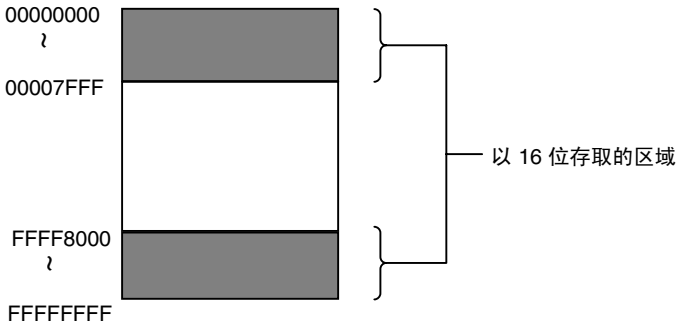
```
#pragma abs28 (x,y,z)
extern int x();
int y;
long z;
f()
{
    z = x() + y;
}
```

扩展为汇编语言代码

```
_f:
            STS.L           PR,@-R15
            MOVI20S         #_x+H'80,R2     ; 加载 x 地址
            ADD             #Low _x,R2
            JSR/N           @R2
            MOVI20S         #_y+H'80,R5     ; 加载 y 地址
            ADD             #Low _y,R5
            MOV.L           @R5,R1
            MOVI20S         #_z+H'80,R4     ; 加载 z 地址
            ADD             #Low _z,R4
            ADD             R1,R0
            LDS.L           @R15+,PR
            RTS
            MOV.L           R0,@R4
_y:
            .RES.L          1
_z:
            .RES.L          1
```

重要信息：

(1) 指定了 "abs16/20/28/32" 选项的变量及函数应使用段切换被放置在不同的段中，段的安排使地址可由在连接上指定的位来表示。若未分配可通过指定的位来表示的地址，错误将在连接时发生。



下表列出各项指定可用的存取范围。

表 3.23　　地址范围

#pragma/选项	地址范围	
	低	高
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*1
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

注意：*请注意地址是 0x07FFFF7F。

(2) 若在编译时指定了位置无关代码的生成，具有指定位数的函数地址将不被生成。

3.7 段名称指定

当需要分配系统中具有相同属性的不同段到不同的地址时（例如，需要分配特定模块到外部 RAM，同时分配另一个模块到内部 RAM），不同的段将被分配名称，同时段在连接时被分配地址。 SuperH RISC engine C/C++ 编译程序提供指定段名称的两种不同方法。 下面显示的是为多个模块个别指定段名称的方法。 在这项解释的实例中，假设模块 f、g、h 及数据 a、b 被划分为 f、h、a 同在一处，及 g、b 同在一处。

段名称指定

SuperH RISC engine C/C++ 编译程序可在编译时通过 "-section" 选项指定对象段名称。 使用此选项，要被分开的模块及数据可被结合到在编译时指定的不同文件、不同段名称中，及在连接时指定的各个不同起始地址。

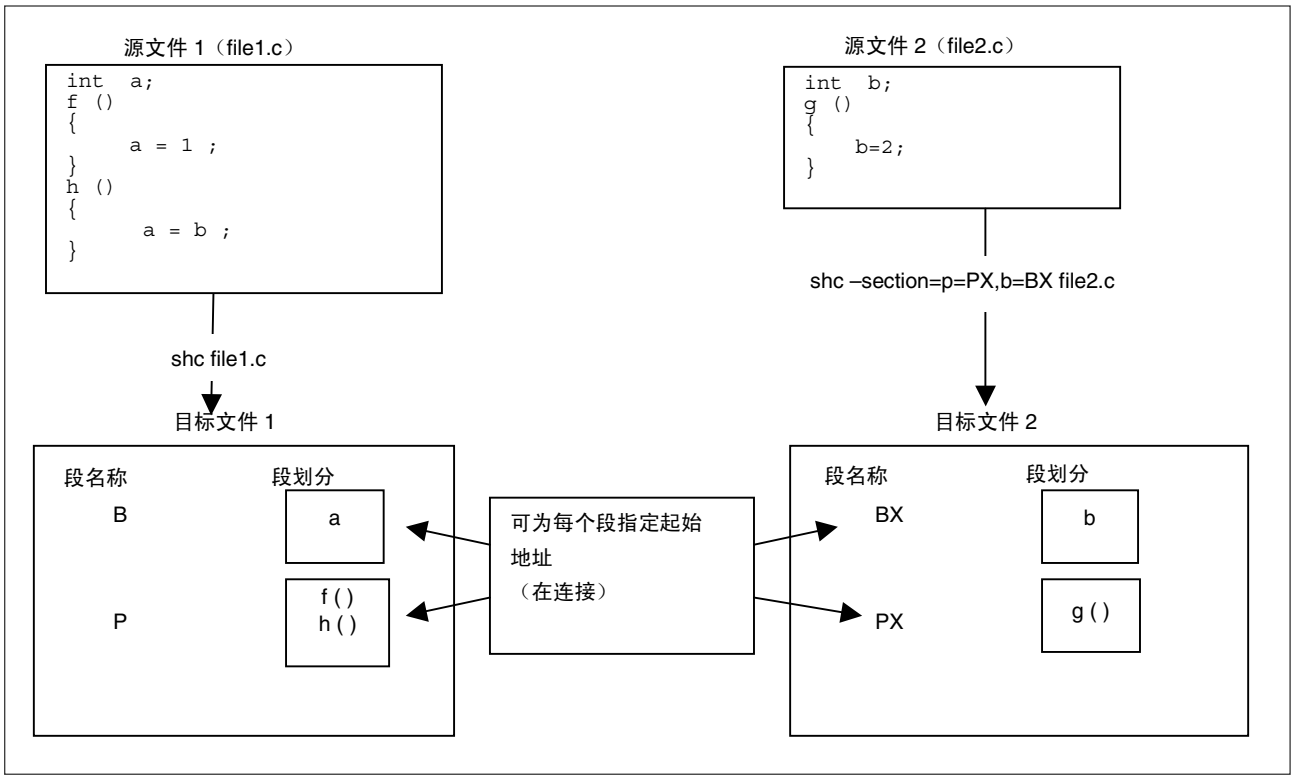


表 3.3 指定段名称的方法

3.7.2 段切换

通过使用 "-section" 选项，可以仅在文件单位中指定段名称，但通过使用 "#pragma section" 指令，具有相同属性的段名称可在单一文件内切换，以对存储器分配具有更佳的控制。 通过使用此功能，第 3.7.1 节中所解释的段划分可在单一文件中描述。 表 3.4 显示一个使用此功能的实例。

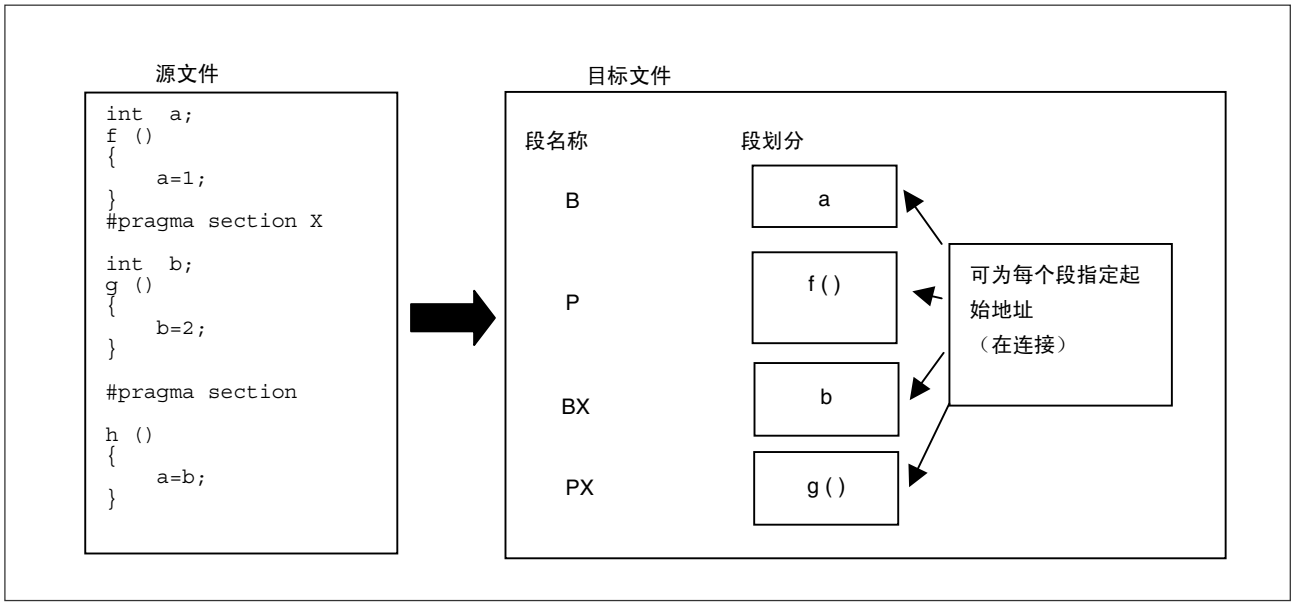


表 3.4 段切换方法

在此表中，通过包含 "#pragma section X" 指令，从这一行到 "#pragma section" 指令的程序段名称变成 "PX"，而未初始化的数据段名称变成 "BX"。
使用 "#pragma section" 指令，将恢复默认段名称。

3.8 入口函数的指定及 SP 设定

描述：

使用<函数名称>指定的函数被处理为入口函数。 寄存器的保存及恢复代码在任何情况下无法由入口函数创建。 若 CPU 是 SH-3、SH3-DSP、SH-4、SH-4A 或 SH4AL-DSP，以及若有 sp=<constant> 的指定或 #pragma stacksize 的声明，堆栈指针初始化代码在函数起始处被输出。

格式：

```
#pragma entry  (< 函数名称 >=<(sp=<常数>))
```

使用的实例：

C 语言代码

实例 1：

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

实例 2：

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

扩展为汇编语言代码

实例 1：

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

实例 2：

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

```
:
L1: .DATA.L STARTOF S + SIZEOF S
:
```

重要信息：

在函数声明之前指定 #pragma 入口。 入口函数最多仅可以指定总共两个装入模块。 对于 <常数>，确保指定 4 的倍数。 若指定了 cpu=sh1|sh2|sh2e|sh2dsp，sp=<常数> 的指定无效。

3.9 位置无关代码

为增进执行速度，ROM 中的代码有时在启动时被复制到 RAM，并从 RAM 运行。 为这样做，程序必须能够将 ROM 代码加载到任意地址内。 这类代码被称为位置无关代码。
通过在使用 SuperH RISC engine C/C++ 编译程序进行编译时指定为命令行选项 "pic=1"，将可生成位置无关代码。

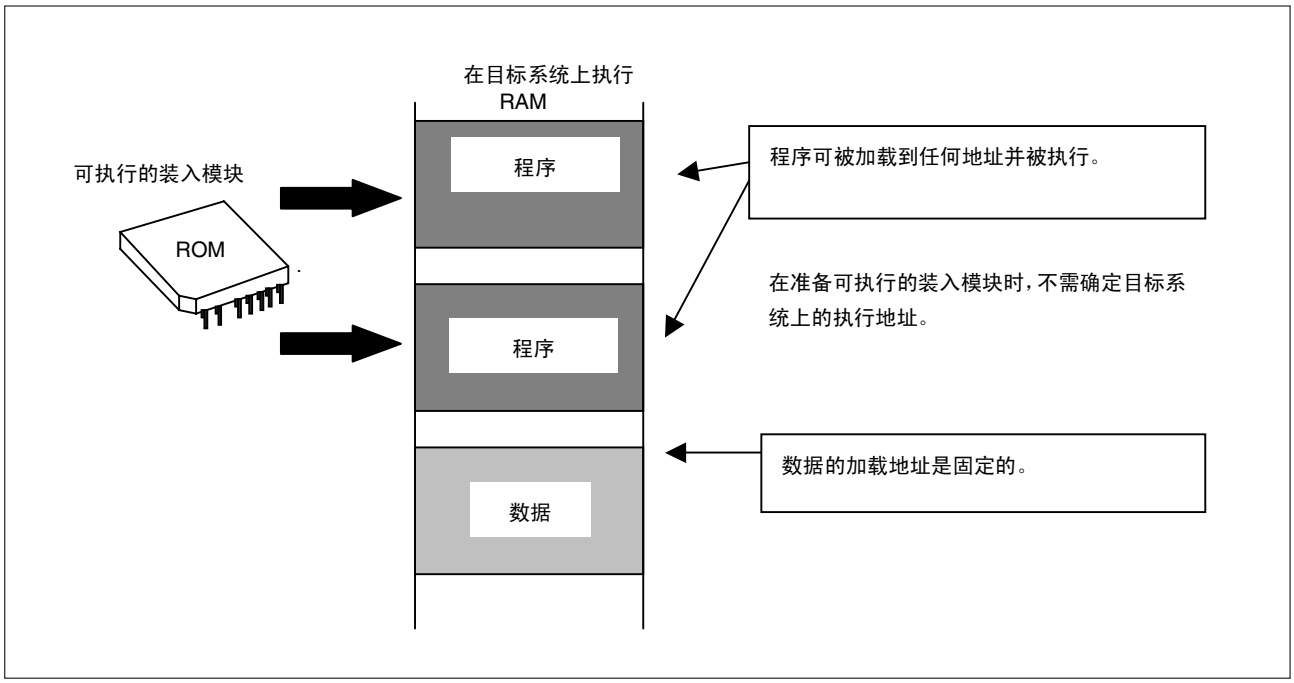


表 3.5 位置无关代码

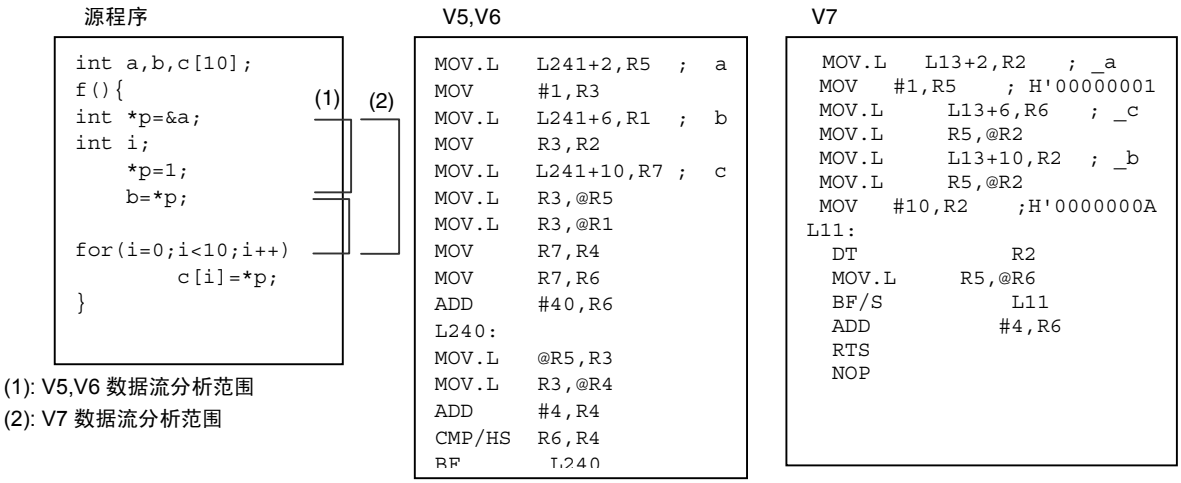
注意： 1. 此功能无法为 SH-1 使用。
2. 此功能无法应用到数据段。
3. 在作为位置无关代码执行时，函数地址无法被指定为初始值。
实例：

```
extern int f();
int (*fp)() = f;
```

函数 f 的地址不确定，直到它被加载到 RAM 内，因此运算在此情况下不被保证。
4. 当使用此功能时，请使用与位置无关代码兼容的标准程序库。 要获取有关创建程序库的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序，及优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, and Optimizing Linkage Editor User's Manual)。

3.10 映像优化

有了针对超型计算机应用的最新优化处理，以及指针与外部变量的别名分析，和包含控制语句的数据流分析，使进一步增进的优化得以实现。



(1): V5,V6 数据流分析范围
(2): V7 数据流分析范围

3.10.1 使用程序

使用由编译及连接分配的符号分配地址进行重新编译。
通过这种方法，依靠分配地址的优化将可由编译程序实现。

使用程序：
第一次编译和连接：
使用普通选项进行编译
使用 -map=<文件>.bls option -> outputs <文件>.bls 进行连接

第二次编译和连接：
使用 -map=<文件>.bls option 进行编译
使用普通选项进行连接

3.10.2 外部变量存取代码获增进的实例 (1)

考虑变量分配的顺序，在相同的寄存器中相对的连续存取分配的变量。

源程序

```
int a,b;
f(){
  a=0;
  b=0;
}
```

V5,V6

```
MOV.L   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   L237+4,R2;_b
 MOV.L   R4,@R3
 RTS
 MOV.L   R4,@R2
L237:
 .DATA.L a
 .DATA.L b
```

V7（MAP 被指定）

```
 MOV.L   L237,R3  ;_a
 MOV     #0,R4
 删除
 MOV.L   R4,@R3
 RTS
 MOV.L   R4,@(1,R3)
L237:
 .DATA.L a
 删除
```

a 被相对的存取

3.10.3 外部变量存取代码获增进的实例 (2)

变量被分配到 0 至 7FFF，同时 FFFF8000 至 FFFFFFFF 由 16 位字面存取。

源程序

```
int a;
f(){
  a=0;
}
```

V5,V6

```
 MOV.L   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   R4,@R3
 RTS
L237:
 .DATA.L a  //ffff0000
```

V7（MAP 被指定）

```
MOV.W   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   R4,@R3
 RTS
L237:
 .DATA.W a  //f000
```

由 16 位引用

3.10.4 增进的外部变量存取代码 (3)

考虑转移目标地址，使用 BSR/BRA 进行转移。

源程序

```
az g();
f(){
   g();
}
```

V5,V6

```
 MOV.L   L237,R2  ;_g
 JMP      @R2
 NOP
L237:
 .RES.W 1
 .DATA.L _g  ;within ±4096
```

V7（MAP 被指定）

```
BRA  _g
NOP
```

使用 BRA 进行转移

3.10.5 增进的外部变量存取代码 (4)

当选项是 gbr=auto (default) 时，GBR 被用作外部变量存取的基址。

源程序

```
int a[101];
int b;

void f()
{
  int i;

  for(i=0;
  i<100;
  i++)
    a[i]=0;

  a[50]=b;
}
```

V5,V6

```
_ f:
 MOV.L    L239+4,R7  ; _a
 MOV      #0,R5
 MOV.W    L239,R6    ; H'0190
 MOV      R7,R4
 ADD      R7,R6
L238:
 MOV.L    R5,@R4
 ADD      #4,R4
 CMP/HS   R6,R4
 BF       L238
 MOV.L    L239+8,R2  ; H'C8+_a
 MOV.L    L239+12,R1 ; _b
 MOV.L    @R2,R3
 RTS
 MOV.L    R3,@R1
L239:
 .DATA.W  H'0190
 .DATA.W  0
 .DATA.L  _a
 .DATA.L  H'000000C8+_a
 .DATA.L  _b
```

V7（MAP 被指定）

```
_ f:
STC      GBR,@-R15
 MOV.W    L13,R0  ; _a
 LDC      R0,GBR
 MOV      #100,R6 ; H'000064
 STC      GBR,R2
 MOV      #0,R5   ; H'000000
L11:
 DT       R6
 MOV.L    R5,@R2
 BF/S     L11
 ADD      #4,R2
 MOV.L    @(200,GBR),R0
 MOV.L    R0,@(400,GBR)
 RTS
 LDC      @R15+,GBR
L13:
 .DATA.W  _a
```

GBR 被相对的引用

3.11 选项

3.11.1 用于代码生成的选项

为使用户能够选择代码生成的程序，SuperH RISC engine C/C++ 编译程序提供了下列选项。

表 3.24 用于代码生成的选项

选项	描述
-SPeed	生成注重执行速度的代码
-SIze	生成注重缩减程序大小的代码
-Goptimize	输出模块间优化的加载信息。
-MAP	根据由优化连接编辑程序生成的外部符号分配信息来设定基址，并生成与基址相对的外部存取执行代码。 若指定了 gbr=auto，根据条件将 GBR 寄存器设定为基址，并生成与 GBR 相对的外部存取执行代码。
-GBr	若指定了 gbr=auto，编译程序根据条件自动生成 GBR 相对的逻辑运算代码。 若指定了 gbr=auto 和 MAP=<文件名>，根据条件将 GBR 设定为基址，并生成与 GBR 相对的外部变量存取执行代码。
-CAse	指定用于 switch 语句的代码扩展方法。若指定了 case=ifthen，switch 语句将使用 if_then 方法被扩展。 因此，在此方法中，目标代码的大小根据 case 标签被包含在 switch 语句中的次数而增加。 若指定了 case=table，switch 语句将使用 table 方法被扩展。 在此情形下，在常数区内分配的跳转表大小，将根据 switch 语句内所包含的 case 标签数酌量增加，然而，执行速度始终为常数。当省略此选项时，编译程序将自动判断要使用哪一种扩展方法。
-SHIft	若指定了 shift=inline，所有移位运算将被指令扩展。 若指定了 shift=runtime，将在存在许多扩展指令的情形中做出运行时例程调用。
-BLOckcopy	若指定了 blockcopy=inline，所有存储器之间的转换代码被指令扩展。 若指定了 blockcopy=runtime，将在转换大小庞大的情形中做出运行时例程调用。
-INLine	指定是否执行函数的自动内联扩展。 若指定了内联选项，自动内联扩展将被执行。 将有可能指定大小。
-DIvision	选择整数类型除法，及程序中余数计算的方法。 若指定了 division=cpu=inline，常数除法将被转换为乘法且会执行内联扩展，同时变量除法使用 DIV1 指令选择一个运行时例程。
-Macsave	指定是否在函数调用前后保证 MACH 和 MACL 寄存器。 当指定了 0 时，MACH 和 MACL 寄存器将不会在函数调用前后被保证。

3.11.2 优化连接的选项

为使用户能够选择优化连接的程序，SuperH RISC engine C/C++ 优化连接编辑程序提供了下列选项。

表 3.25 连接选项

选项	子选项	描述
-OPtimize	-	指定是否执行模块间优化。
	-SPeed	执行除了可能会使对象速度缩减以外的其他优化。 与 optimize=string_unify、symbol_delete、variable_access、register、branch 相同
	-SAFe	执行除了可能会受变量或函数属性限制以外的其他优化。 与 optimize=string_unify、register、branch 相同
	-Branch	基于程序分配信息，转移指令的大小获得优化。若执行了其他优化项目，无论是否指定，它都将被执行。
	-Register	函数调用间的关系将被分析，同时寄存器的重新分配和冗余寄存器的保存/恢复代码将由这项指定删除。
	-STring_unify	统一具有 const 属性的相同值常数。 具有 const 属性的常数包括下列项目。 在 C/C++ 程序中声明为 const 的变量 字符串数据的初始值 字面常数
	-SYmbol_delete	未被参考的变量/函数将由这项指定删除。
	-Variable_access	将被频繁存取的变量分配到可以 8/16 位绝对寻址模式存取的区域。
	-SAMe_code	将多个相似的指令字符串变成子例程。
	-Function_call	在从 0 到 0xFF 的存储器范围具有空间时分配被频繁存取的函数地址。
-SMAESize	-	使用公用代码统一优化指定被优化代码的最小代码大小。
-PROfile	-	指定配置文件的信息文件。使用模块间优化，将可根据动态信息来执行优化。
-CAchesize	-	指定高速缓存大小及高速缓存线大小。若指定了配置文件选项，它将和转移指令优化一同使用。
-SYmbol_forbid	-	禁用通过删除未被参考的符号来实现的优化。
-SAMECode_forbid	-	禁用通过统一公用代码来实现的优化。
-Variable_forbid	-	禁用通过使用短绝对寻址模式来实现的优化。
-FUnction_forbid	-	禁用通过使用间接寻址模式来实现的优化。
-Absolute_forbid	-	禁用地址 + 大小范围的优化。

3.11.3 创建标准程序库的选项

为使用户能够选择创建标准程序库时的优化程序，SuperH RISC engine C/C++ 标准程序库创建工具提供了下列选项。

表 3.26 创建标准程序库的选项

选项	描述
-SPeed	生成注重执行速度的代码
-SIze	生成注重缩减程序大小的代码
-Goptimize	输出模块间优化的加载信息。
-MAP	根据由优化连接编辑程序生成的外部符号分配信息来设定基址，并生成与基址相对的外部存取执行代码。 若设定了 gbr=auto，根据条件将 GBR 寄存器设定为基址，并生成与 GBR 相对的外部存取执行代码。
-GBr	若指定了 gbr=auto，编译程序根据条件自动生成 GBR 相对的逻辑运算代码。 若指定了 gbr=auto 和 MAP=<文件名>，根据条件将 GBR 设定为基址，并生成与 GBR 相对的外部变量存取执行代码。
-CAse	指定用于 switch 语句的代码扩展方法。若指定了 case=ifthen，switch 语句将使用 if_then 方法被扩展。 因此，在此方法中，目标代码的大小根据 case 标签被包含在 switch 语句中的次数而增加。 若指定了 case=table，switch 语句将使用 table 方法被扩展。 在此情形下，在常数区内分配的跳转表大小，将根据 switch 语句内所包含的 case 标签数酌量增加，然而，执行速度始终为常数。 当省略此选项时，编译程序将自动判断要使用哪一种扩展方法。
-SHIft	若指定了 shift=inline，所有移位运算将被指令扩展。 若指定了 shift=runtime，将在存在许多扩展指令的情形中做出运行时例程调用。
-BLOckcopy	若指定了 blockcopy=inline，所有存储器之间的转换代码被指令扩展。 若指定了 blockcopy=runtime，将在转换大小庞大的情形中做出运行时例程调用。
-INLine	指定是否执行函数的自动内联扩展。 若指定了内联选项，自动内联扩展将被执行。 将有可能指定大小。

3.12 SH-DSP 功能

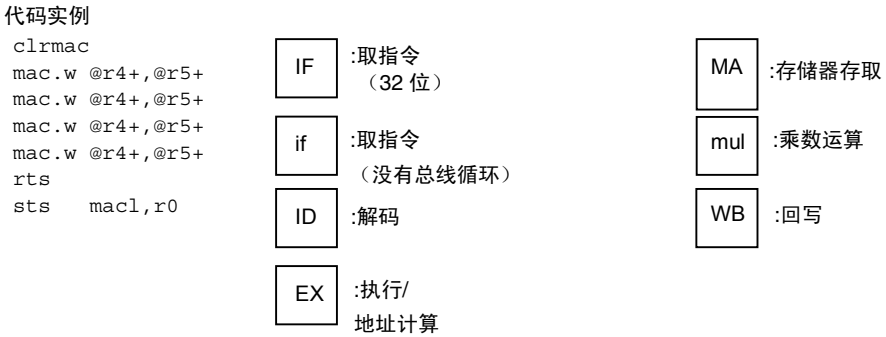
SH-DSP 核心随执行 16 位定点运算的 DSP 装置提供，且非常适用于：

- 乘法累加运算
- 重复处理

它能够高速执行多媒体操作所需的 JPEG 处理、音频处理及过滤器处理。

在之前的 SH 核心（图 3.6 中的 SH-1 核心实例）中，乘法累加运算的性能是由构成流水线操作中乘数运算时间的三个循环所决定。即使乘数运算时间增进到一个循环，然而，流水线仍然会因为指令数据转换而发生停转，因此长期的平均时间将是 2.5 个循环。

在 SH-DSP 核心中，DSP 装置的运算时间是一个循环，且 X 总线/Y 总线作为数据总线提供，因此乘法累加运算仅使用一个循环。（图 3.7）在这里长期平均时间也是一个循环。



流水线操作的实例

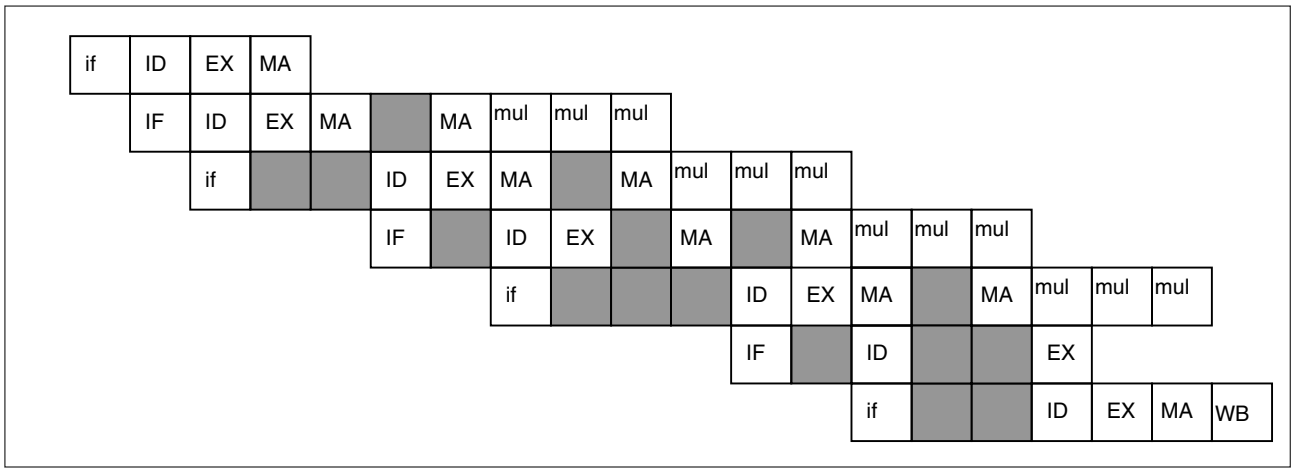
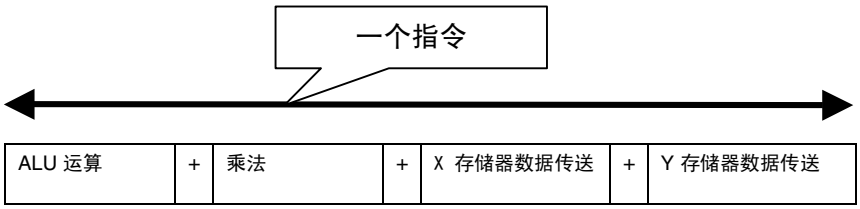


表 3.6 在 SH 核心中执行的乘法累加指令



代码实例

```
指令 1                                          MOVX.W@R4+   X0 MOVY.W@R6+,Y0
指令 2                        PMULSX0,Y0,M0     MOVX.W@R4+   X1 MOVY.W@R6+,Y1
指令 3   PADD A0,M0,A0 PMULSX1,Y1,M1    MOVX.W@R4+   X0 MOVYW@R6+,Y0
指令 4   PADD A0,M1,A0 PMULS X0,Y0,M0   MOVX.W@R4+   X1 MOVYW@R6+,Y1
```

流水线操作的实例

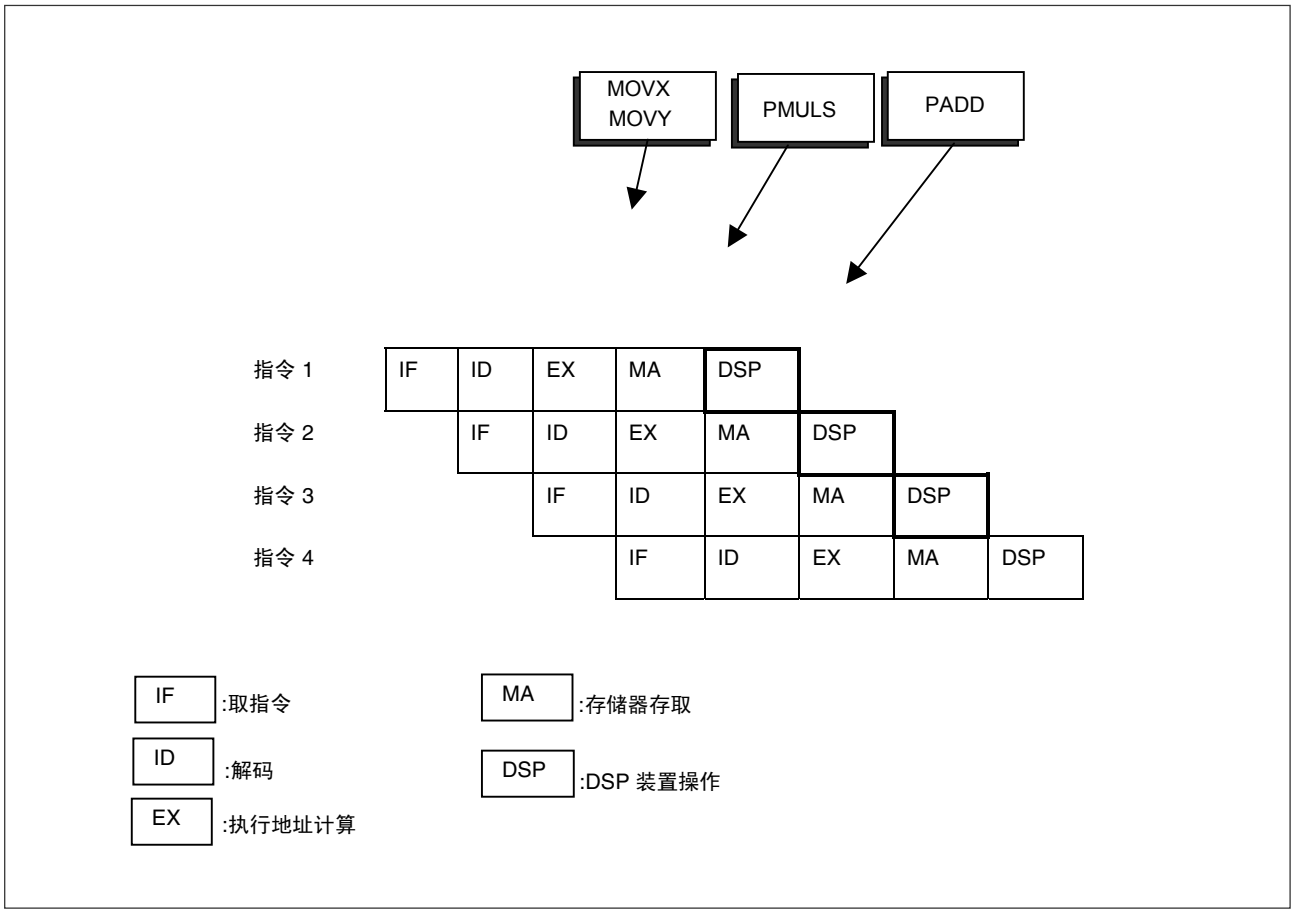


表 3.7 在 SH-DSP 核心中执行的乘法累加指令

此外，SH-DSP 核心配备的硬件机制，可减少因为重复处理所造成流水线中断。

在之前的 SH 核心中，条件转移被用于循环处理。 条件转移会中断流水线，从而添加处理的内务操作。

在 SH-DSP 核心中具有零内务操作机制，会将由这项循环处理引起的流水线中断减少至零。只要设定循环的起始和结束地址以及循环数量，循环处理即可在无需执行条件转移的情况下完成。许多重要的软件操作以循环处理为依据，这是可有效增进软件执行速度的硬件机制。

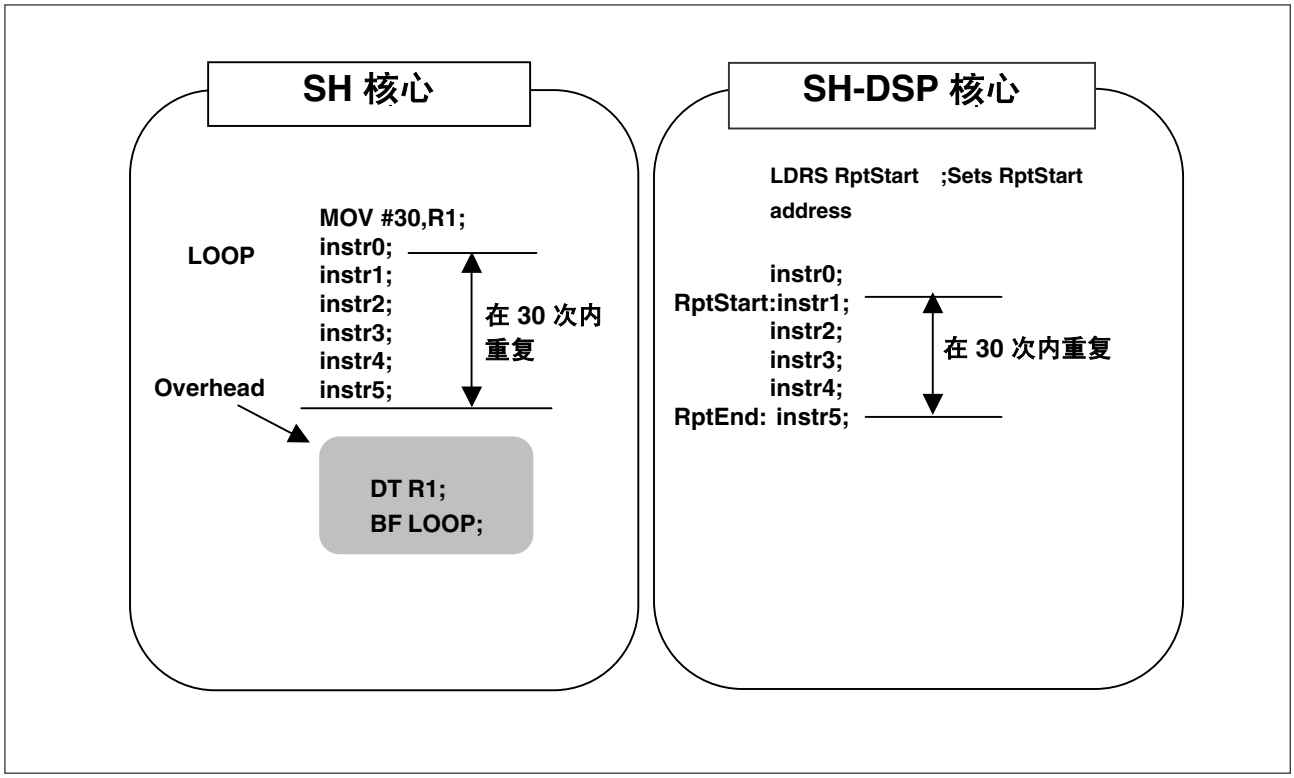


图 3.8 重复处理

SH-DSP 核心能够并行执行五个指令，如图 3.9 中所示： 条件评估、ALU 运算、带符号的乘法、X 存储器存取及 Y 存储器存取。 通过结合这些指令，将可在高速下执行多种乘法累加运算。

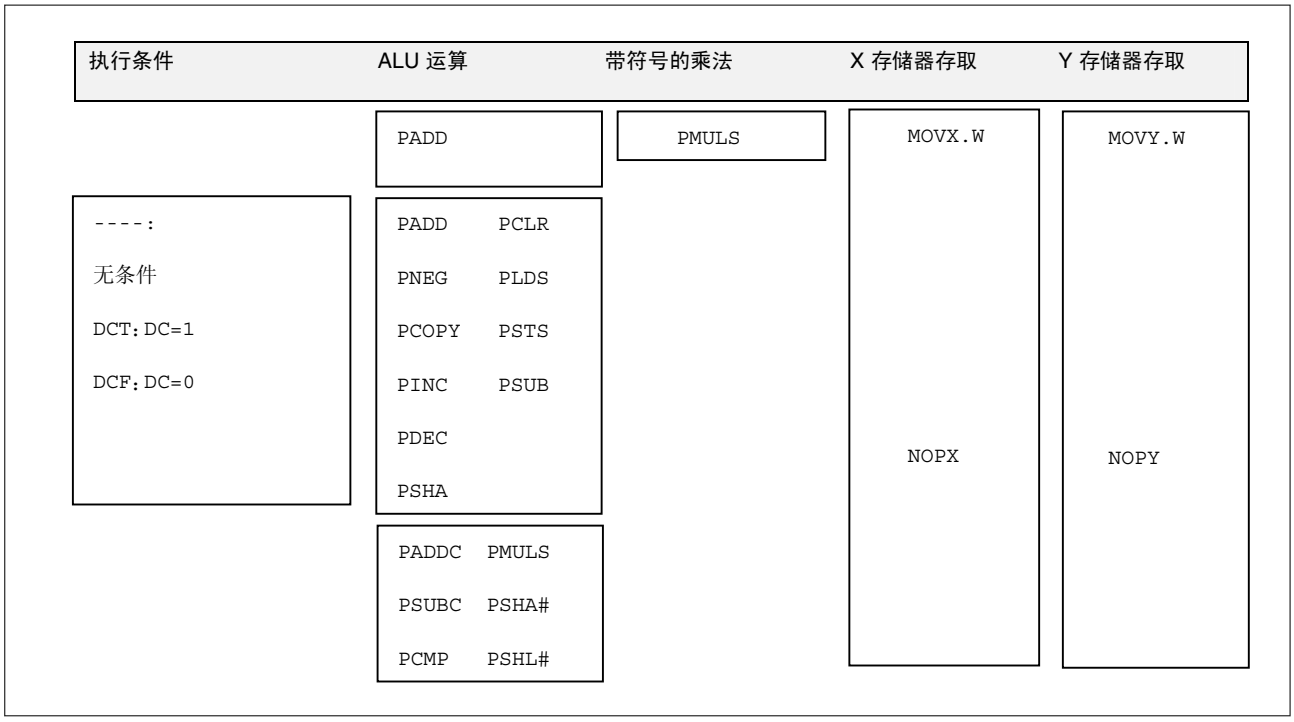


图 3.9 DSP 指令（并行指令）

3.13 DSP 程序库

3.13.1 摘要

本节将解释能和 SH2-DSP 及 SH3-DSP（在此之后联合引述为 SH-DSP）一同使用的数位信号处理 (DSP) 程序库。此程序库包含标准的 DSP 函数，通过单独或连续使用它们，将可执行 DSP 运算。

此程序库包含下列函数。

- 快速傅立叶转换
- 窗口函数
- 过滤器
- 卷积和相关
- 其他

此程序库中的函数是可重入的，除了快速傅立叶转换和过滤器。

当使用此程序库时，包含在表 3.27 中显示的文件。 此外，如表 3.28 所示，连接到与 CPU 和编译选项相应的程序库。

当此程序库被调用时，若函数正常结束，将以 EDSP_OK 作为返回值，若发生错误，则将以 EDSP_BAD_ARG 或 EDSP_NO_HEAP 作为返回值。 有关返回值的详情，请参考各个函数的解释。

表 3.27 和 DSP 程序库一同使用的包含文件

程序库类型	描述	包含文件
DSP 程序库	程序库执行 DSP 运算	<ensigdsp.h>
		<filt_ws.h>*1

注意： *1 当使用过滤器函数时，仅将它们包含在用户程序中一次。

表 3.28 DSP 程序库列表

CPU	选项		程序库名称
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP SH4AL-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

3.13.2 数据格式

此程序库将数据处理为带符号的 16 位定点数。 如图 3.10(a) 中所示，带符号的 16 位定点数所具有的数据格式是点被固定到最高有效位 (MSB) 的右侧，同时可表达 –1 到 $1-2^{-15}$ 的值。

在此程序库中，数据转换使用 short 类型的数据格式。 因此，当从 C/C++ 程序使用此程序库时，以带符号的 16 位定点数表达数据是必要的。

实例：+0.5 表达为带符号的 16 位定点数是 H'4000。因此，传递给程序库函数的 short 类型实际参数是 H'4000。

此程序库中的内部运算使用带符号的 32 位定点数及带符号的 40 位定点数。 带符号的 32 位定点数所具有的数据格式如图 3.10(b) 所示，同时可表达从 –1 到 $1-2^{-31}$ 的值。 带符号的 40 位定点数所具有的是如图 3.10(c) 所示具有附加 8 位警戒位的数据格式，同时可表达从 -2^{8} 到 $2^{8}-2^{-31}$ 的值。

带符号的 16 位定点数的乘法结果被保存为带符号的 32 位定点数。 通过使用 DSP 指令的定点乘法，只有在 H'8000 x H'8000 的情形下需要谨防溢出的发生。 此外，乘法结果的最低有效位 (LSB) 通常是 0。当乘法结果在接下来的操作中被使用时，高 16 位被移除，同时结果被转换为带符号的 16 位定点数。 在此情况下，将有可能会发生下溢或准确度缩减。

在此程序库的乘法累加运算中，加法结果被保存为带符号的 40 位定点数。 请留意在执行加法时不会发生溢出。

若在执行运算时发生溢出，将无法获取正确的结果。 为预防溢出，将有必要执行系数或输入数据的缩放。 缩放函数被内建到此程序库中。 有关缩放的详情，请参考各个函数的解释。

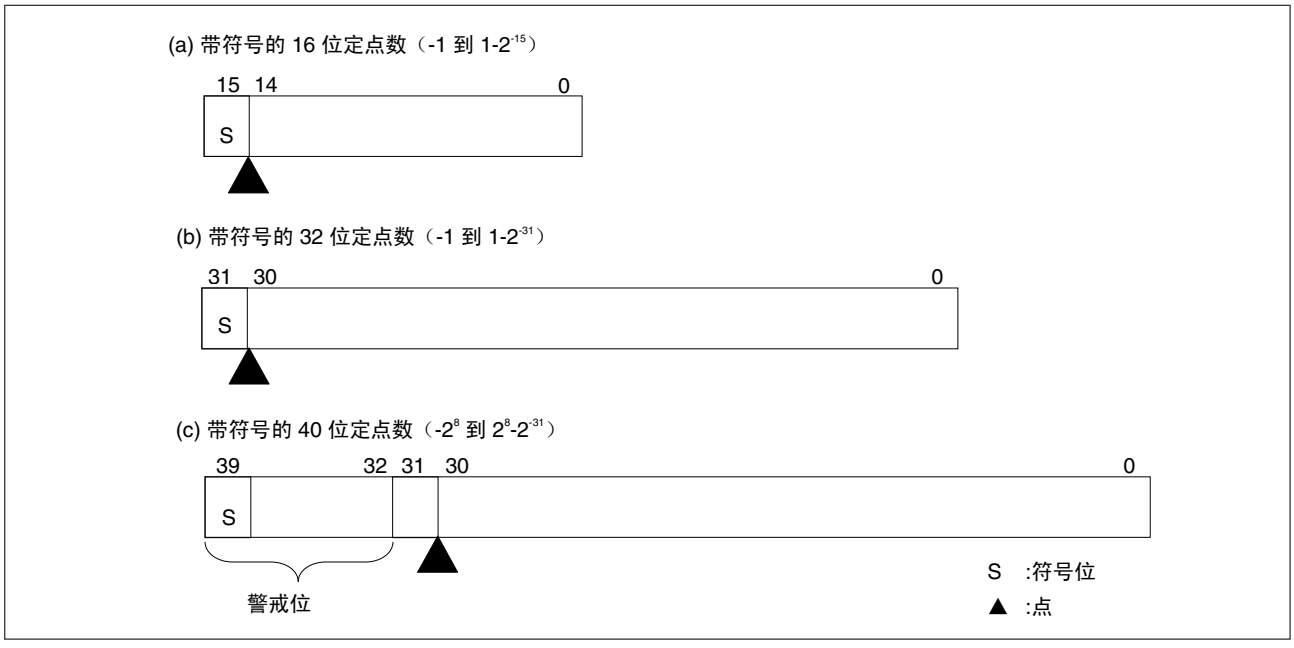


图 3.10 数据格式

3.13.3 效率

此程序库中的函数为在 SH-DSP 上高速执行而被优化。
为有效的使用程序库，当在开发中决定系统的存储器映像时，应尽量观察以下两项建议。

- 为程序代码段分配支持 1 个循环 32 位读取的存储器。
- 为数据段分配支持 1 个循环 16 位（或 32 位）读取及写入的存储器。

若要使用的微型计算机具有足够分配程序库代码及数据的 32 位存储器内建容量，最好将它分配到 32 位存储器。若需要使用其他存储器，请尽量跟随上面的建议。

3.13.4 快速傅立叶转换

(1) 函数列表

表 3.29 DSP 程序库的函数（快速傅立叶转换）列表

编号	类型	函数名称	描述
1	不在位 (not-in-place) 的复数 FFT	FftComplex	执行不在位的复数 FFT
2	不在位 (not-in-place) 的复数 FFT	FftReal	执行不在位的实数 FFT
3	不在位 (not-in-place) 的复数 FFT	IfftComplex	执行不在位的复数反 FFT
4	不在位 (not-in-place) 的复数 FFT	IfftReal	执行不在位的实数反 FFT
5	在位 (in-place) 的复数 FFT	FftInComplex	执行在位的复数 FFT
6	在位 (in-place) 的复数 FFT	FftInReal	执行在位的实数 FFT
7	在位 (in-place) 的复数 FFT	IfftInComplex	执行在位的复数反 FFT
8	在位 (in-place) 的复数 FFT	IfftInReal	执行在位的实数反 FFT
9	对数绝对值	LogMagnitude	将复数数据转换为对数绝对值
10	FFT 旋转因数生成	InitFft	生成 FFT 旋转因数
11	FFT 旋转因数释放	FreeFft	释放用来存储 FFT 旋转因数的存储器

注意： 有关不在位和在位的详情，请参考“(e) FFT 结构”。

因数使用用户定义的缩放，以执行前进方向的高速傅立叶转换以及反方向的高速傅立叶转换。
前进方向的傅立叶转换使用以下方程式来定义。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{-2j\pi n/N}\cdot x_n$$

在这里，s 代表执行缩放的阶段数，N 代表数据元素的数量。
反方向的傅立叶转换使用以下方程式来定义。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{2j\pi n/N}\cdot x_n$$

有关缩放的详情，请参考“(4) 缩放”。

(2) 复数数据组格式

FFT 和 IFFT 复数数据组被分配到 X 存储器（实数）及 Y 存储器（虚数）。然而，实数 FFT 输出数据与实数 IFFT 输入数据的分配不同。若存储实数和虚数的数组分别被定义为 x 和 y，DC 组件的实数组件将进入 x[0]，同时进入 y[0] 的是 Fs/2 组件的实数组件，而非 DC 组件的虚数组件（DC 组件及 Fs/2 组件都是实数组件，虚数组件是 0）。

(3) 实数数据组格式

FFT 及 IFFT 实数数据组格式共有以下 3 种。

- 存储在单一数组中，且被分配到任意存储器块。
- 存储在单一数组中，且被分配到 X 存储器。
- 划分为 2 个数组以进行存储。 各个数组的大小是 N/2，数组的前半部被分配到 X 存储器，后半部则被分配到 Y 存储器。

只有第一种指定方法可用于 FftReal。 用户可为 IfftReal、FftInReal 和 IfftInReal 选择第二或第三种方法。

(4) 缩放

基址 2 FFT 的信号长度将在每个阶段加倍，同时信号幅度峰值也将加倍。 为此原因，当转换到高密度信号时，可能会发生溢出。 然而，通过在每个阶段将信号减半（这称为“缩放”），即可防止溢出。 不过，若过量实行缩放，将可能会发生不必要的量化噪声。

溢出与量化噪声之间的最佳缩放平衡大量取决于输入信号的特性。 为防止信号中大峰值范围的溢出，将需要实行最大缩放，但对于脉冲信号来说，几乎完全不需要实行缩放。

在每个阶段执行缩放是最安全的方法。 若输入数据的密度小于 230，可使用此方法来防止溢出。 通过此程序库，可在每个阶段指定缩放。 因此，通过精确指定缩放，溢出和量化噪声的影响可被抑制到最小。

为指定缩放的方法，每个 FFT 函数参数将包含‘scale’。‘scale’对应于每个阶段的最低有效位到每个个别位。 若每个阶段的对应缩放位被设定为 1，将执行被 2 除的除法。

为增加执行速度，基址 4 FFT 将在此程序库中使用。‘scale’对应于每个阶段的最低有效位到每两位。 若其中一位被设定为 1，将执行被 2 除的除法。 若两位都被设定为 1，将执行被 4 除的除法。 也就是说，这和两个基址 2 FFT 阶段被一个基址 4 FFT 阶段取代是相同的情况。 然而，量化噪声在基址 4 FFT 发生的可能性比在基址 2 FFT 发生的可能性更大。

下面显示一个‘scale’的实例。

- 当 scale = H'FFFFFFFF（或 size-1）时，缩放将会对所有基址 2 FFT 阶段执行。 若所有输入数据的密度少于 230，将不会发生溢出。
- 当 scale = H’55555555 时，缩放将会对每隔一个基址 2 FFT 阶段执行。
- 当 scale = 0 时，将不执行缩放。

这些缩放值被定义为 ensigdsp.h、EFFTALLSCALE(H’FFFFFFFF)、EFFTMIDSCALE(H’55555555)，及 EFFTNOSCALE(0)

(5) FFT 结构

此程序库的 FFT 结构有 2 种，不在位 FFT 及在位 FFT

通过不在位 FFT，输入数据从 RAM 被移除，FFT 被执行，同时输出结果被存储在由用户指定的另一个 RAM 地点。

另一方面，通过在位 FFT，输入数据从 RAM 被移除，FFT 被执行，同时输出结果被存储在相同的 RAM 地点。 若此方法被使用，FFT 的执行时间将增加，但可减少所使用的存储器空间。

当和输入数据一起使用其他 FFT 函数时，请使用不在位 FFT。 此外，当希望保存存储器空间时，请使用在位 FFT。

(6) 各个函数的解释

(a) 不在位的复数 FFT

描述：

格式：

```
int FftComplex (short op_x[], short op_y[],
                                const short  ip_x[], const short ip_y[], long size, long scale)
```

参数：

op_x[]	输出数据的实数组件
op_y[]	输出数据的虚数组件
ip_x[]	输入数据的实数组件
ip_y[]	输入数据的虚数组件
size	FFT 大小
scale	缩放指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 4 • size 不是 2 的幂 • size > max_fft_size

描述：

执行复数快速傅立叶转换。

说明：

由于此函数执行不在位，输入数组及输出数组需个别提供。有关复数数据组的分配详情，请参考“(2) 复数数据组格式”。在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。有关缩放的详情，请参考“(4) 缩放”。‘scale’使用低 log2 (size) 位。 此函数不是可重入的。

使用的实例：

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

#define MAX_FFT_SAMP  64
#define MIN_CFFT_SIZE   4
long ip_scale=0xffffffff;
long size = MIN_CFFT_SIZE;

#pragma section X
short ip_x[MAX_FFT_SAMP];
short op_x[MAX_FFT_SAMP];
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

```
#pragma section Y
short ip_y[MAX_FFT_SAMP];
short op_y[MAX_FFT_SAMP];
#pragma section

/* 用于循环计数的数据 */
#define TWOPI 6.283185307 /* 数据 */
void main()
{
    int i,j;
    long n_samp;

    n_samp=MAX_FFT_SAMP; /* 数据 */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * 8188;
        ip_y[j] = sin(j * TWOPI/n_samp) * 8188;
    }
    if(InitFft(n_samp) != EDSP_OK){
        printf("Initfft != err end");
    }
    if(FftComplex(op_x,op_y,ip_x,ip_y,n_samp,EFFTALLSCALE) != EDSP_OK){
        printf("FftComplex error\n");
    }
    FreeFft();
    for(i=0;i<n_samp;i++){
        printf("[%d] op_x=%d  op_y=%d  \n",i,op_x[i],op_y[i]);
    }
}
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建

FFT 初始化函数；
初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(b) 不在位的实数 FFT

描述：

格式：

```
int FftReal (short op_x[], short op_y[], const short ip[],
                              long size, long scale)
```

参数：

op_x[]	正值输出数据的实数组件
op_y[]	正值输出数据的虚数组件
ip[]	实数输入数据
size	FFT 大小
scale	缩放指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 8 • size 不是 2 的幂 • size > max_fft_size

描述：

执行实数快速傅立叶转换。

说明：

size/2 正值输出数据被存储在 op_x 和 op_y 中。负值输出数据是正值输出数据的共轭复数。 此外，由于 0 和 FS/2 的输出数据值是实数，FS/2 输出的实数被存储在 op_y[0] 中。
由于此函数执行不在位，输入数组及输出数组需个别提供。
有关复数和实数数据组的分配详情，请参考“(2) 复数数据组格式”及“(3) 实数数据组格式”。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define    VLEN  64
#define    TWOPI 6.28318530717959
/* 全局数据声明 */
#pragma section X
short output_x[VLEN];
#pragma section Y
short output_y[VLEN];
#pragma section
void main()
{
    short i;
    int k;
    short input[VLEN];
    short output[VLEN];
/* 生成两个正弦曲线 */
    k = VLEN / 8;
    for (i = 0; i < VLEN; i++)
       input[i] = floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);
    k = VLEN * 3 / 8;
    for (i = 0; i < VLEN; i++)
        input[i] += floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);

/* do FFT */
    if (InitFft(VLEN) != EDSP_OK)
        printf("InitFft problem\n");
    if (FftReal(output_x, output_y, input, VLEN, EFFTALLSCALE) != EDSP_OK)
        printf("FftReal problem\n");
    FreeFft();
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建

FFT 初始化函数：

初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(c) 不在位的复数反 FFT

描述：

格式：

```
int IfftComplex (short op_x[], short op_y[],
                                    const short ip_x[], const short ip_y[],
                                    long size, long scale)
```

参数：

op_x[]	输出数据的实数组件
op_y[]	输出数据的虚数组件
ip_x[]	输入数据的实数组件
ip_y[]	输入数据的虚数组件
size	反 FFT 大小
scale	缩放指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 4 • size 不是 2 的幂 • size > max_fft_size

描述：

执行复数快速傅立叶反转。

说明：

由于此函数执行不在位，输入数组及输出数组需个别提供。
有关复数数据组的分配详情，请参考“(2) 复数数据组格式”。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* 数据 */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* 输入数组 */
short opi_x[MAX_IFFT_SIZE];          /* 普通输出数组 */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* 数据 */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  \n");
        }
        for (j = 0; j < max_size; j++){
            opi_x[j]=0;
            opi_y[j]=0;
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建（用于执行 FftComplex 的数据）

FFT 初始化函数：

初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

这项处理执行 FFT 计算，并将结果用作 FFT 反函数的输入值；这在一般上是不需要的。

```
        }
        if(IfftComplex(opi_x, opi_y, ipi_x, ipi_y, max_size,
                              EFFTALLSCALE)!= EDSP_OK){
            printf("IfftComplex err  end  \n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d \n",j, opi_x[j],opi_y[j]);
        }
        FreeFft();
    }
}
```

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。 若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(d) 不在位的实数反 FFT

描述：

格式：

```
int IfftReal (short op_x[], short scratch_y[],
                              const short ip_x[], const short ip_y[], long size, long scale, int
                              op_all_x)
```

参数：

op_x[]	实数输出数据
scratch_y[]	暂时存储器或实数输出数据
ip_x[]	正值输入数据的实数组件
ip_y[]	正值输入数据的虚数组件
size	反 FFT 大小
scale	缩放指定
op_all_x	输出数据的分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 8 • size 不是 2 的幂 • size > max_fft_size • op_all_x ≠ 0 or 1

描述：

执行实数快速傅立叶反转。

说明：

将 size/2 正值输入数据存储在 ip_x 和 ip_y 中。负值输入数据是正值输入数据的共轭复数。 此外，由于 0 和 FS/2 的输入数据值是实数，FS/2 输入的实数将被存储在 ip_y[0] 中。
输出数据的格式以 op_all_x 来指定。若 op_all_x=1，所有输出数据将被存储在 op_x 中。若 op_all_x=0，第一个 size/2 输出数据将被存储在 op_x 中，其余 size/2 输出数据将被存储在 scratch_y 中。
由于此函数执行不在位，输入数组及输出数组需个别提供。
有关复数和实数数据组的分配详情，请参考“(2) 复数数据组格式”及“(3) 实数数据组格式”。
将 size/2 数据分别存储在 ip_x 和 ip_y 中。视 op_all_x 的值而定，size 或 size/2 数据将被存储在 op_x 中。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* 数据 */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* 输入数组 */
short opi_x[MAX_IFFT_SIZE];          /* 普通输出数组 */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* 数据 */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if (InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
if(IfftReal(opi_x, opi_y, ipi_x, ipi_y, max_size, EFFTALLSCALE,1)!=
   EDSP_OK){
            printf("IfftReal err end  \n");
        }
        for (j = 0; j < max_size; j++){
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建（用于执行 FftReal 的数据）

FFT 初始化函数：

初始化为数据元素的数量执行。 这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。也需要用于 FFT 反函数。

这项处理执行 FFT 计算，并将结果用作 FFT 反函数的输入值；这在一般上是不需要的。

```
            printf("[%d]   opi_x=%d   op_y=%d \n",j, opi_x[j],opi_y[j]);
        }
        FreeFft();
    }
}
```

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(e) 在位的复数 FFT

描述：

格式：

```
int FftInComplex (short data_x[], short data_y[],
                                        long size, long scale)
```

参数：

data_x[]	输入数据的实数组件
data_y[]	输入及输出数据的虚数组件
size	FFT 大小
scale	缩放指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 4 • size 不是 2 的幂 • size > max_fft_size

描述：

执行在位复数快速傅立叶转换。

说明：

有关复数数据组的分配详情，请参考“(2) 复数数据组格式”。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* 数据 */
long ip_scale=0xffffffff;


#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section
void main()
{
    int i,j;
    long max_size;
    long n_samp;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* 数据 */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = sin(j * TWOPI/n_samp) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error\n");
    }
    if(FftInComplex(ip_x, ip_y, n_samp,EFFTALLSCALE ) != EDSP_OK){
        printf("FftInComplex error\n");
    }
    FreeFft();
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  \n",i,ip_x[i],ip_y[i]);
    }
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建

FFT 初始化函数：

初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(f) 在位的实数 FFT

描述：

格式：

```
int FftInReal (short data_x[], short data_y[], long size,
                                long scale, int ip_all_x)
```

参数：

data_x[]	输入时是实数数据，输出时是正值输出数据的实数组件
data_y[]	输入时是实数数据或未使用，输出时是正值输出数据的虚数组件
size	FFT 大小
scale	缩放指定
ip_all_x	输入数据的分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 8 • size 不是 2 的幂 • size > max_fft_size • ip_all_x ≠ 0 or 1

描述：

执行在位实数快速傅立叶转换。

说明：

输入数据的格式以 ip_all_x 指定。若 ip_all_x=1，所有输入数据将从 data_x 移除。若 ip_all_x=0，size/2 输入数据的前半部将从 data_x 移除，size/2 输入数据的后半部将从 data_y 移除。
在执行这项函数后，size/2 正值输出数据被存储在 data_x 和 data_y 中。负值输出数据是正值输出数据的共轭复数。此外，由于 0 和 FS/2 的输出数据值是实数，FS/2 输出的实数被存储在 data_y[0] 中。
有关复数和实数数据组的分配详情，请参考“(2) 复数数据组格式”及“(3) 实数数据组格式”。
将 size/2 数据存储在 data_y 中。视 ip_all_x 的值而定，size 或 size/2 数据将被存储在 data_x 中。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* 数据 */
long ip_scale=8188;
/*long ip_scale=0xffffffff;*/

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section


void main()
{
    int i,j;
    long max_size;
    long n_samp;
    int ip_all_x;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* 数据 */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = 0;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error\n");
    }

    ip_all_x = 1;
    if(FftInReal(ip_x, ip_y, n_samp,EFFTALLSCALE ,ip_all_x) != EDSP_OK){
        printf("FftInReal error\n");
```

包括标头

放置在 X 或 Y 存储器中的变量
通过段内的 pragma 段被定义。

FFT 的数据创建

FFT 初始化函数：

初始化为数据元素的数量执行。 这是必须的。 数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

```
    }


    FreeFft();
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  \n",i,ip_x[i],ip_y[i]);
    }
}
```

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(g) 在位的复数反 FFT

描述：

格式：

```
int IfftInComplex (short data_x[], short data_y[],
                                        long size, long scale)
```

参数：

data_x[]	输入数据的实数组件
data_y[]	输入及输出数据的虚数组件
size	反 FFT 大小
scale	缩放指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 4 • size 不是 2 的幂 • size > max_fft_size

描述：

执行在位复数快速傅立叶反转。

说明：

有关复数数据组的分配详情，请参考“(2) 复数数据组格式”。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* 数据 */

long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* 输入数组 */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* 数据 */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  \n");
        }
        if(IfftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("IfftInComplex err  end  \n");
        }
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建（数据用作 FftInComplex 的输入）

FFT 初始化函数：

初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。也需要用于 FFT 反函数。

这项处理执行 FFT 计算，并将结果用作 FFT 反函数的输入值 这在一般上是不需要的。

```
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d \n",j, ipi_x[j],ipi_y[j]);
        }
        FreeFft();
    }
}
```

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

(h) 在位的实数反 FFT

描述：

格式：

```
int IfftInReal (short data_x[], short data_y[], long size,
                                    long scale, int op_all_x)
```

参数：

data_x[]	输入时是正值输入数据的实数组件，输出时是实数数据
data_y[]	输入时是正值输入数据的虚数组件，输出时是实数数据或未使用
size	反 FFT 大小
scale	缩放指定
op_all_x	输出数据的分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 • size < 8 • size 不是 2 的幂 • size > max_fft_size • op_all_x ≠ 0 or 1

描述：

执行在位实数快速傅立叶反转。

说明：

将 size/2 正值输入数据存储在 data_x 和 data_y 中。负值输入数据是正值输入数据的共轭复数。 此外，由于 0 和 FS/2 的输入数据值是实数，将 FS/2 输入的实数存储在 data_y[0] 中。
输出数据的格式以 op_all_x 指定。若 op_all_x=1，所有输出数据被存储在 data_x 中。若 op_all_x=0，size/2 输出数据的前半部被存储在 data_x 中，size/2 输出数据的后半部则被存储在 data_y 中。
有关复数与实数数据组的分配详情，请参考“(2) 复数数据组格式”及“(3) 实数数据组格式”。
将 size/2 数据存储在 data_y 中。视 op_all_x 的值而定，size 或 size/2 数据将被存储在 data_x 中。
在调用此函数前，先调用 InitFft，然后初始化旋转因数及 max_fft_size。
有关缩放的详情，请参考“(4) 缩放”。
‘scale’使用低 log2 (size) 位。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* 数据 */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* 输入数组 */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* 数据 */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
           printf("InitFft error end  \n");
    }
    else {
        if(FftInRealipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
        if(IfftInReal(ipi_x, ipi_y, max_size, EFFTALLSCALE,1) != EDSP_OK){
            printf("IfftInReal err  end  \n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ipi_y=%d \n",j, ipi_x[j],ipi_y[j]);
        }
        FreeFft();
    }
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建（数据用作 FftInReal 的输入）

FFT 初始化函数：

初始化为数据元素的数量执行。这是必须的。数据元素的数量相等于 FFT 数据大小，同时必须是 2 的幂。

这释放了在 FFT 计算中使用的表。若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。

```
}
```

(i) 对数绝对值

描述：

格式：

```
int LogMagnitude (short output[], const short ip_x[],
                                    const short ip_y[], long no_elements,
                                    float fscale)
```

参数：

output[]	实数输出 z
ip_x[]	输入实数组件 x
ip_y[]	输入虚数组件 y
no_elements	输出数据元素的数量 N
fscale	输出缩放系数

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •no_elements > 32767 •|fscale| > $2^{15}/(10\log_{10}2^{31})$

描述：

以分贝单位计算复数输入数据的对数绝对值，并在输出数组中写入缩放结果。

说明：

z(n)=10fscale · log10(x(n)2+y(n)2)　0 ≤ n < N
有关复数数据组的分配详情，请参考“(2) 复数数据组格式”。

使用的实例：

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* 数据 */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* 输入数组 */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    short output[MAX_IFFT_SIZE];
    max_size=MAX_IFFT_SIZE;/* 数据 */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
        if(LogMagnitude(output, ipi_x,ipi_y, max_size/2, 2) != EDSP_OK){
            printf("LogMagnitude err  end  \n");
        }
        for (j = 0; j < max_size/2; j++){
           printf("[%d]  output=%d  \n",j, output[j]);
        }
        FreeFft();
    }
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

FFT 的数据创建

FFT 函数：
创建由 LogMagnitude 函数使用的数据。

这释放了在 FFT 计算中使用的表。

若这未被完成，存储器资源将被浪费。若 FFT 要使用相同的数据元素数量再执行一次，FFT 函数将在不执行 FreeFft 的情况下被再次使用。这和 LogMagnitude 没有直接关系。

```
}
```

(j) 旋转因数生成

描述：

格式：

```
int InitFft (long max_size)
```

参数：

max_size	所需 FFT 的最大大小

返回的值：

EDSP_OK	成功
EDSP_NO_HEAP	可以 malloc 获取的存储器空间并不足够
EDSP_BAD_ARG	在下列任何情况下 •max_size < 2 •max_size 不是 2 的幂 •max_size > 32,768

描述：

生成 FFT 函数所使用的旋转因数（1/4 大小）。

说明：

旋转因数被存储在由 malloc 获取的存储器中。
当旋转因数被生成时，max_fft_size 全局变量被更新。max_fft_size 显示 FFT 的最大容量大小。
确保在调用第一个 FFT 函数前，先调用此函数一次。
将 max_size 定为 8 或以上。
旋转因数由 max_size 指定的转换大小生成。 在执行大小小于 max_size 的 FFT 函数时使用相同的旋转因数。
旋转因数的地址被存储在内部变量内。 请勿使用用户程序对其进行存取。
此函数不是可重入的。

(k) 旋转因数释放

描述：

格式：

```
void FreeFft (void)
```

参数：

无

返回的值：

无

描述：

释放用来存储旋转因数的存储器。

说明：

将 max_fft_size 全局变量定为 0。当在执行 FreeFft 后再次执行 FFT 函数时，确保首先执行 InitFft。
此函数不是可重入的。

3.13.5 窗口函数

(1) 函数列表

表 3.30 DSP 程序库函数列表 （窗口函数）

编号	类型	函数名称	描述
1	布赖克曼 (Blackman) 窗口	GenBlackman	生成布赖克曼窗口。
2	汉明 (Hamming) 窗口	GenHamming	生成汉明窗口。
3	汉宁 (Hanning) 窗口	GenHanning	生成汉宁窗口。
4	三角窗口	GenTriangle	生成三角窗口。

(2) 各个函数的解释

(a) 布赖克曼窗口

描述：

格式：

```
int GenBlackman (short output[], long win_size)
```

参数：

output[]	输出数据 W(n)
win_size	窗口大小 N

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

描述：

生成布赖克曼窗口，并输出到输出结果。

说明：

在将此窗口应用到实际数据时使用 VectorMult。
下面显示所要使用的函数。

$$W(n) = \left(2^{15} - 1\right)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \quad 0 \le n < N$$

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define MAXN 10
void main()
{
```

} 包括标头

```
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenBlackman(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

(b) 汉明窗口

描述：

格式：

```
int GenHamming (short output[], long win_size)
```

参数：

output[]	输出数据 W(n)
win_size	窗口大小 N

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

描述：

生成汉明窗口，并输出到输出。

说明：

在将此窗口应用到实际数据时使用 VectorMult。
下面显示所要使用的函数。

$$W(n) = \left(2^{15} - 1\right)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define MAXN  10
void main()
```

包括标头

```
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenHamming(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

(c) 汉宁窗口

描述：

格式：

```
int GenHanning (short output[], long win_size)
```

参数：

output[]	输出数据 W(n)
win_size	窗口大小 N

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

描述：

生成汉宁窗口，并输出到输出。

说明：

在将此窗口应用到实际数据时使用 VectorMult。

下面显示所要使用的函数。

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

使用的实例：

```
#include <stdio.h>
```
} 包括标头

```
#include <ensigdsp.h>
#define MAXN  10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenHanning(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

(d) 三角窗口

描述：

格式：

```
int GenTriangle (short output[], long win_size)
```

参数：

output[]	输出数据 W(n)
win_size	窗口大小 N

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

描述：

生成三角窗口，并输出到输出结果。

说明：

在将此窗口应用到实际数据时使用 VectorMult。
下面显示所要使用的函数。

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \quad 0 \le n < N$$

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
```
包括标头

```
#define MAXN 10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenTriangle(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

3.13.6 过滤器

(1) 函数列表

表 3.31 DSP 程序库函数列表 （过滤器）

编号	类型	函数名称	描述
1	FIR	Fir	执行有限推进响应过滤器处理
2	用于单一数据元素的 FIR	Fir1	执行用于单一数据元素的有限推进响应过滤器处理
3	IIR	Iir	执行无限推进响应过滤器处理
4	用于单一数据元素的 IIR	Iir1	执行用于单一数据元素的无限推进响应过滤器处理
5	双精度 IIR	Diir	执行双精度无限推进响应过滤器处理
6	用于单一数据元素的双精度 IIR	Diir1	执行用于单一数据元素的双精度无限推进响应过滤器处理
7	自适应 FIR	Lms	执行自适应 FIR 过滤器处理
8	用于单一数据元素的自适应 FIR	Lms1	执行用于单一数据元素的自适应 FIR 过滤器处理
9	FIR 工作空间分配	InitFir	分配供 FIR 过滤器使用的工作空间
10	IIR 工作空间分配	InitIir	分配供 IIR 过滤器使用的工作空间
11	双精度 IIR 工作空间分配	InitDIir	分配供 DIIR 过滤器使用的工作空间
12	自适应 FIR 工作空间分配	InitLms	分配供 LMS 过滤器使用的工作空间
13	FIR 工作空间释放	FreeFir	释放由 InitFir 分配的工作空间
14	IIR 工作空间释放	FreeIir	释放由 InitIir 分配的工作空间
15	双精度 IIR 工作空间释放	FreeDIir	释放由 InitDIir 分配的工作空间
16	自适应 FIR 工作空间释放	FreeLms	释放由 InitLms 分配的工作空间

注意：当使用这些函数时，仅将 filt_ws.h 包含在用户程序中一次。

(2) 系数缩放

当执行过滤器处理时，有可能发生饱和或量化噪声。 可通过执行这些过滤器系数的缩放来将这些情形制止到最小。 然而，执行缩放时必须谨慎考虑对饱和及量化噪声的影响。 若系数太大，则有可能会发生饱和的情形。 若太小，则可能发生量化噪声。

使用 FIR（有限推进响应）过滤器时，若在过滤器系数的设定上应用了下列方程式，饱和将不会发生。

coeff[i] ≠ H’8000（适用于所有 i）

Σ|coeff| < 224

res_shift = 24

coeff 是过滤器系数，res_shift 是在输出时向右移的位数。

然而，当有许多输入信号时，即使使用了较小的 res_shift 值（或较大的 coeff 值），但是发生饱和的可能性仍然轻微，同时量化噪声可被大幅度减低。此外，若输入值有可能包含 H'8000，将所有 coeff 值设定在 H'8001 到 H'7FFF 的范围内。

IIR（无限推进响应）过滤器具有递归结构。 因此，上面解释的缩放方法不适用。

LMS（最小均方）自适应过滤器与 FIR 过滤器相同。 然而，在适应系数时，可能在某些情况下会发生饱和。 在这种情况下，调整设定使系数中不包含 H'8000。

(3) 工作空间

使用数字过滤器时，有些信息必须在这个处理和下一个处理之间保存。这些信息被保存在可使用最少内务操作来存取的存储器中。 使用此程序库，Y-RAM 区域被用作工作空间。 在执行过滤器处理前，调用 Init 函数并初始化工作空间。

工作空间存储器由程序库函数存取。 请勿直接从用户程序存取工作空间。

(4) 使用存储器

为有效使用 SH-DSP，将过滤器系数分配到 X 存储器。 输入和输出数据可被分配到任意存储器段。

使用 #pragma 段指令将过滤器系数分配到 X 存储器。

每个过滤器通过使用 Init 函数从全局缓冲被分配到工作空间。 全局缓冲被分配到 Y 存储器。

(5) 各个函数的解释

(a) FIR

描述：

格式：

```
int Fir (short output[], const short input[], long no_samples,
                    const short coeff[], long no_coeffs, int res_shift,
                    short *workspace)
```

参数：

output[]	输出数据 y
input[]	输入数据 x
no_samples	输入数据元素的数量 N
coeff[]	过滤器系数 h
no_coeffs	系数数量（过滤器长度）K
res_shift	应用到各个输出的右移
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_samples < 1 •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

描述：

执行有限推进响应 (FIR) 过滤器处理。

说明：

最后的输入数据被保存在工作空间内。 输入的过滤器处理结果被写入到输出。
乘法累加运算的结果被保存为 39 位。 输出 y(n) 是取自 res_shift 右移位结果的低 16 位。 当发生溢出时，这是最大正或负值。
有关系数缩放的详情，请参考“(2) 系数缩放”。
在调用此函数前，先调用 InitFir，然后初始化过滤器的工作空间。
若为输出和输入指定了相同的数组，输入将被覆盖。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define NFN 8 /* 函数数量 */
#define FIL_COUNT   32  /* 数据对象数量 */
#define N   32

#pragma section X
static short coeff_x[FIL_COUNT];
#pragma section
short data[FIL_COUNT] =  {
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,};
short coeff[8] =  {
          0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000,};

void main()
{
    short *work, i;
    short output[N];
    int nsamp, ncoeff, rshift;
    /* 将 coeff 复制到 X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* 设定系数 */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = NFN;/* 设定系数数量  */
    nsamp = FIL_COUNT;/* 设定样品数量 */
    rshift = 12;
    if (InitFir(&work, ncoeff) != EDSP_OK){
        printf("Init Problem\n");
    }
```

包括标头

设定 X 存储器中的过滤器系数。由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 系数数量
这是在 Fir 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（系数数量）*2+8 字节。

```
    if(Fir(output, data, nsamp, coeff_x, ncoeff, rshift, work) != EDSP_OK){
        printf("Fir Problem\n");
    }
    if (FreeFir(&work, ncoeff) != EDSP_OK){
        printf("Free Problem\n");
    }
    for(i=0;i<nsamp;i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

FreeFir 函数将释放用于 Fir 计算的工作区域。在 Fir 执行后，必须始终执行 FreeFir 函数。若这项函数未被执行，存储器资源将被浪费。

(b) 用于单一数据元素的 FIR

描述：

格式：

```
int Fir1 (short *output, short input, const short coeff[],
                    long no_coeffs, int res_shift, short *workspace)
```

参数：

output	输出数据 y(n) 的指针
input	输入数据 x(n)
coeff[]	过滤器系数 h
no_coeffs	系数数量（过滤器长度）K
res_shift	应用到各个输出的右移
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

描述：

执行用于单一数据元素的有限推进响应 (FIR) 过滤器处理。

说明：

最后的输入数据被保存在工作空间内。 输入的过滤器处理结果被写入到*输出。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\, x(n-k)\right] \cdot 2^{-res\_shift}$$

乘法累加运算的结果被保存为 39 位。 输出 y(n) 是取自 res_shift 右移位结果的低 16 位。 当发生溢出时，这是最大正或负值。

有关系数缩放的详情，请参考“(2) 系数缩放”。
在调用此函数前，先调用 InitFir，然后初始化过滤器的工作空间。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define NFN 8  /* 函数数量 */
#define MAXSH 25
#define N   32
#pragma section X
static short coeff_x[NFN];
#pragma section
short data[32] =  {
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[8] =  {
          0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
void main()
{
    short *work, i;
    short output[N];
    int ncoeff, rshift;

    /* 将 coeff 复制到 X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* 设定系数 */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    rshift = 12;
    ncoeff = NFN;/* 设定系数数量 */
    if (InitFir(&work, NFN) != EDSP_OK){
        printf("Init Problem\n");
    }
    for(i=0;i<N;i++) {
        if(Fir1(&output[i], data[i], coeff_x, ncoeff, rshift, work) !=
```

包括标头

设定 X 存储器中的过滤器系数。由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 系数数量
这是在 Fir1 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（系数数量）*2+8 字节。

Fir1 表示设定到 Fir 函数的数据元素数量是 1。当多次执行 Fir1 时，必须分别在 for 语句前后执行 Init Fir 和 FreeFir 函数。

```
    EDSP_OK){
            printf("Fir1 Problem\n");
        }
        printf(" output[%d]=%d \n",i,output[i]);
    }
    if (FreeFir(&work, NFN) != EDSP_OK){
        printf("Free Problem\n");
    }
}
```

Fir1 表示设定到 Fir 函数的数据元素数量是 1。当多次执行 Fir1 时，必须分别在 for 语句前后执行 Init Fir 和 FreeFir 函数。

(c) IIR

描述：

格式：

```
int Iir (short output[], const short input[], long no_samples,
                const short coeff[], long no_sections, short *workspace)
```

参数：

output[]	输出数据 y_{K-1}
input[]	输入数据 x_0
no_samples	输入数据元素的数量 N
coeff[]	过滤器系数
no_sections	二阶滤波器的段数 K
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_samples < 1 •no_sections < 1 •a_{0k} < 0 •a_{0k} > 16

描述：

执行无限推进响应 (IIR) 过滤器处理。

说明：

此过滤器是在 K 数目串联连接了二阶滤波器 (secondary filter) 或双二阶 (biquad) 的情况下配置。 附加缩放以各个双二阶的输出来执行。 过滤器系数以带符号的 16 位定点数指定
各个双二阶的输出根据下列方程式计算。

$d_k(n)=[a1_k d_k(n-1)+a_{2k} d_k(n-2)+2_{15} x(n)] \cdot 2^{-15}$

$y_k(n)=[b_{0k} d_k(n)+b_{1k} d_k(n-1)+b_{2k} d_k(n-2)] \cdot 2^{-a0k}$

k 的输入 x_k (n) 是上一个段的输出 y_{k-1} (n)。 第一段 (k=0) 的输入从输入读取。 最后一段 ($k=_{K-1}$) 的输出写入到输出。
按照下列系数顺序设定 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} 项是在 k 的双二阶输出上执行的右移位数。
各个双二阶执行 32 位乘法累加运算。 各个双二阶的输出是取自 15 位或 a_{0k} 右移结果的低 16 位。 当发生溢出时，这是最大正或负值。
在调用此函数前，先调用 InitIir，然后初始化过滤器的工作空间。
若为输出和输入指定了相同的数组，输入将被覆盖。
此函数不是可重入的。

使用的实例：

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


#define K  4
#define NUMCOEF  (6*K)
#define N 50
#pragma section X
static short coeff_x[NUMCOEF];
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
void main()
{
    short *work, i;
```

包括标头

设定 X 存储器中的过滤器系数。由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

应在一个段内设定六个过滤器系数。一个段中的前导元素是右移数字，且不是过滤器系数。

```
    short output[N];

    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
    }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
    if (Iir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem\n");
    }
    for(i=0;i<N;i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 过滤器段的数量
这是在 Iir 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（过滤器段的数量）*2*2 字节。

Freelir 函数将释放用于 Iir 计算的工作区域。在 Iir 执行后，必须始终执行 Freelir 函数。若这项函数未被执行，存储器资源将被浪费。

(d) 用于单一数据元素的 IIR

描述：

格式：

```
int Iir1 (short *output, short input, const short coeff[],
                                    long no_sections, short *workspace)
```

参数：

output	输出数据 $y_{K-1}(n)$ 的指针
input	输入数据 $x_0(n)$
coeff[]	过滤器系数
no_sections	二阶滤波器的段数 K
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_sections < 1 •a_{ok} < 0 •a_{ok} > 16

描述：

执行用于单一数据元素的无限推进响应 (IIR) 过滤器处理。

说明：

此过滤器是在 K 数目串联连接了二阶滤波器或双二阶的情况下配置。 附加缩放以各个双二阶的输出来执行。 过滤器系数以带符号的 16 位定点数指定。
各个双二阶的输出根据下列方程式计算。
$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)] \cdot 2^{-15}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k}$
k 的输入 x_k (n) 是上一个段的输出 y_{k-1} (n)。 第一段 (k=0) 的输入从输入读取。 最后一段 (k=K-1) 的输出写入到输出。
按照下列系数顺序设定 coeff。
$a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} \ldots b_{2K-1}$
a_{0k} 项是在 k 的双二阶输出上执行的右移位数。
各个双二阶执行 32 位饱和运算。 各个双二阶的输出是取自 15 位或 a_{0k} 右移结果的低 16 位。 当发生溢出时，这是最大正或负值。
在调用此函数前，先调用 InitIir，然后初始化过滤器的工作空间。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K   4
#define NUMCOEF  (6*K)
#define N   50
#pragma section X
static short coeff_x[NUMCOEF];
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
                  32000, 32000, 32000, 32000, 32000,
```

包括标头

设定 X 存储器中的过滤器系数。由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

应在一个段内设定六个过滤器系数。一个段中的前导元素是右移数字，且不是过滤器系数。

```
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
short keisu[5]={ 1,2,20,4,5 };
```

```
void main()
{
    short *work, i;
    short output[N];
    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
  }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
    for(i=0;i<N;i++){
        if (Iir1(&output[i], input[i], coeff_x, K, work) != EDSP_OK){
            printf("EDSP_OK not returned\n");
        }
        printf("output[%d]:%d \n" ,i,output[i]);
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem\n");
    }
}
```

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 过滤器段的数量
这是在 Iir1 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（过滤器段的数量）*2*2 字节。

Iir1 表示设定到 Iir 函数的数据元素数量是 1。当多次执行 Iir1 时，必须分别在 for 语句前后执行 Init Iir 和 Freelir 函数。

Iir1 表示设定到 Iir 函数的数据元素数量是 1。当多次执行 Iir1 时，必须分别在 for 语句前后执行 Init Iir 和 Freelir 函数。

(e) 双精度 IIR

描述：

格式：

```
int DIir (short output[], const short input[], long no_samples,
                        const long coeff[], long no_sections, long *workspace)
```

参数：

output[]	输出数据 y_{K-1}
input[]	输入数据 x
no_samples	输入数据元素的数量 N
coeff[]	过滤器系数
no_sections	二阶滤波器的段数 K
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_samples < 1 •no_sections < 1 •a_{0k} < 3 •k < K-1 及 a_{0k} > 32 •k = K-1 及 a_{0k} > 48

描述：

执行双精度无限推进响应过滤器处理。

说明：

此过滤器是在 K 数目串联连接了二阶滤波器或双二阶的情况下配置。 附加缩放以各个双二阶的输出来执行。 过滤器系数以带符号的 32 位定点数指定。
各个双二阶的输出根据下列方程式计算。
$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)] \cdot 2^{-31}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^2$
k 的输入 x_k (n) 是上一个段的输出 y_{k-1} (n)。 第一段 (k=0) 的输入从输入的 16 位左移值读取。 最后一段 (k=K-1) 的输出写入到输出。
按照下列系数顺序设定 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} 项是在 k 的双二阶输出上执行的右移位数。
DIir 与 Iir 的不同之处在于过滤器系数是以 32 位值指定，而非 16 位值。 乘法累加运算的结果被保存为 64 位。 中期阶段的输出是取自 a_{0k} 位右移结果的低 32 位。 当发生溢出时，这是最大正或负值。 在最后的阶段，低 16 位是取自 a_{0k-1} 的右移位结果。 当发生溢出时，这是最大正或负值。
在调用此函数前，先调用 InitDIir，然后初始化过滤器的工作空间。
延迟的节点 d_k (n) 被舍入到 30 位，当发生溢出时，这是最大的正或负值。
当使用 DIir 时，以带符号的 32 位定点数指定系数。 在这种情况下，当 a_{0k} 是 k < K-1 时，将它设定为 31，当 k=K-1 时，将它设定为 47。
由于 Iir 的执行速度比 DIir 快，若需要进行双精度计算，使用 Iir 来进行。
若为输出和输入指定了相同的数组，输入将被覆盖。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <filt_ws.h>
#include <ensigdsp.h>
#define K  5
#define NUMCOEF  (6*K)
#define N  50
#pragma section X
static long coeff_x[NUMCOEF];
#pragma section
static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[100] = {
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000,
                          32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;
    long nsamp;
```

包括标头

设定 X 存储器中的过滤器系数。

由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

应在一个段内设定六个过滤器系数。一个段中的前导元素是右移数字，且不是过滤器系数。

除了最后一段，右移的数量是 31，最后一段是 47。

```
    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work,K) != EDSP_OK){
        printf("InitDIir Problem\n");
    }
    if(DIir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("DIir Problem\n");
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("FreeDIir Problem\n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 过滤器段的数量
这是在 Dlir 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（过滤器段的数量）*4*2 字节。

FreeDlir 函数将释放用于 Dlir 计算的工作区域。在 Dlir 执行后，必须始终执行 FreeDlir 函数。若这项函数未被执行，存储器资源将被浪费。

(f) 用于单一数据元素的双精度 IIR

描述：

格式：

```
int DIir1  (short output[], const short input[], long no_samples,
                        const long coeff[], long no_sections,
                        long *workspace)
```

参数：

output	输出数据 $y_{K-1}(n)$ 的指针
input	输入数据 $x_0(n)$
coeff[]	过滤器系数
no_sections	二阶滤波器的段数 K
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_sections < 1 •a0k < 3 •k < K-1 及 a_{0k} > 32 •k = K-1 及 a_{0k} > 48

描述：

执行用于单一数据元素的双精度无限推进响应过滤器处理。

说明：

此过滤器是在 K 数目串联连接了二阶滤波器或双二阶的情况下配置。 附加缩放以各个双二阶的输出来执行。 过滤器系数以带符号的 32 位定点数指定。
各个双二阶的输出根据下列方程式计算。

$$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}dk(n-2)+2^{29}x(n)] \cdot 2^{-31}$$

$$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^2$$

k 的输入 x_k (n) 是上一个段的输出 y_{k-1} (n)。 第一段 (k=0) 的输入从输入的 16 位左移值读取。 最后一段 (k=K-1) 的输出写入到输出。
按照下列系数顺序设定 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} 项是在 k 的双二阶输出上执行的右移位数。
DIir1 与 Iir1 的不同之处在于过滤器系数是以 32 位值指定，而非 16 位值。 乘法累加运算的结果被保存为 64 位。 中期阶段的输出是取自 a_{0k} 位右移结果的低 32 位。 当发生溢出时，这是最大正或负值。 在最后的阶段，低 16 位是取自 a_{0k-1} 的右移位结果。 当发生溢出时，这是最大正或负值。
在调用此函数前，先调用 InitDIir，然后初始化过滤器的工作空间。
延迟的节点 d_k (n) 被舍入到 30 位，当发生溢出时，这是最大的正或负值。
当使用 DIir1 时，以带符号的 32 位定点数指定系数。 在这种情况下，当 a_{0k} 是 k < K-1 时，将它设定为 31，当 k=K-1 时，将它设定为 47。
由于 Iir1 的执行速度比 DIir1 快，若需要进行双精度计算，使用 Iir1 来进行。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K  5
#define NUMCOEF  (6*K)
#define N   50
#pragma section X
static long coeff_x[NUMCOEF];
#pragma section
static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};
static short input[N]    = {32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000,
                            32000, 32000, 32000, 32000, 32000 };
void main()
{
    short i;
    short output[N];
    long *work;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work, K) != EDSP_OK){
```

包括标头

应在一个段内设定六个过滤器系数。一个段中的前导元素是右移数字，且不是过滤器系数。

设定 X 存储器中的过滤器系数。
由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

除了最后一段，右移的数量是 31，最后一段是 47。

过滤器初始化：
(1) 工作区域地址
(2) 过滤器段的数量
这是在 Dlir1 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（过滤器段的数量）*4*2 字节。

将 X 存储器中的过滤器系数设定为变量。

Dlir1 表示设定到 Dlir 函数的数据元素数量是 1。当多次执行 Dlir1 时，必须分别在 for 语句前后执行 InitDlir 和 FreeDlir 函数。

```
        printf("Init Problem\n");
    }
    for(i=0;i<N;i++){
        if(DIir1(&output[i], input[i], coeff_x, K, work) !=EDSP_OK){
            printf("DIir1 error\n");
        }
        printf("output[%d]:%d \n" ,i,output[i]);
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("Free DIir error\n");
    }
}
```

DIir1 表示设定到 DIir 函数的数据元素数量是 1。当多次执行 DIir1 时，必须分别在 for 语句前后执行 InitDIir 和 FreeDIir 函数。

(g) 自适应 FIR

描述：

格式：

```
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)
```

参数：

output[]	输出数据 y
input[]	输入数据 x
ref_output[]	所需的输出值 d
no_samples	输入数据元素的数量 N
coeff[]	自适应过滤器系数 h
no_coeffs	系数数量 K
res_shift	应用到各个输出的右移
conv_fact	收敛系数 2m
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_samples < 1 •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

描述：

使用最小均方 (LMS) 算法，执行实数自适应 FIR 过滤器处理。

说明：

FIR 过滤器使用下列方程式定义。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

乘法累加运算的结果被保存为 39 位。 输出 y(n) 是取自 res_shift 右移位结果的低 16 位。 当发生溢出时，这是最大正或负值。
过滤器系数的更新使用 Widrow-Hoff 算法来执行。
hn+1(k)=hn(k)+2μe(n)x(n-k)
在这里，e(n) 是所需符号与实际输出之间的错误差数。
e(n)=d(n)-y(n)
使用 2μe(n)x(n-k) 计算，16 位 x 16 位的乘法被执行 2 次。 乘法结果的高 16 位都被保存，当发生溢出时，这是最大正或负值。若已更新的系数值是 H'8000，溢出可能会随着乘法累加运算发生。将系数值设定在 H'8001 到 H'7FFF 的范围内。
有关系数缩放的详情，请参考“(2) 系数缩放”。由于系数是使用 LMS 过滤器来适应，最安全的缩放方法是设定少于 256 个系数，并将 res_shift 设定为 24。
conv_fact 一般应被设为正值。 请勿设定为 H'8000。
在调用此函数前，先调用 InitLms，然后初始化过滤器。

若为输出和输入或 ref_output 指定了相同的数组，输入或 ref_output 将被覆盖。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25
#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};


static short ref[N] = { -107, -143, 998, 1112, -5956,
                            -10781, 239, 13655, 11202, 2180,
                            -687, -2883, -7315, -6527, 196,
                            4278, 3712, 3367, 4101, 2703,
                            591, 695, -1061, -5626, -4200,
                            3585, 9285, 11796, 13416, 12994,
                            10231, 5803, -449, -6782, -11131,
                            -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
```

包括标头

设定 X 存储器中的过滤器系数。由于 Y 存储器被程序库用作计算过滤器系数的工作区域，因此不应使用 Y 存储器。

```
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* 将 coeff 复制到 X RAM */
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = K;/* 设定系数数量 */
    nsamp = N;/* 设定样品数量 */

for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
    if(Lms(output, data, ref, nsamp, coeff_x, ncoeff, RSHIFT,TWOMU, work) !=
   EDSP_OK){
        printf("Lms Problem\n");
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem\n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 系数数量
这是在 LMS 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（系数数量）*2+8 字节。

FreeLms 函数将释放用于 Lms 计算的工作区域。在 Lms 执行后，必须始终执行 FreeLms 函数。若这项函数未被执行，存储器资源将被浪费。

(h) 用于单一数据元素的自适应 FIR

描述：

格式：

```
int Lms1  (short *output, short input, short ref_output,
                        short coeff[], long no_coeffs, int res_shift,
                        short conv_fact, short  *workspace)
```

参数：

output	输出数据 y(n) 的指针
input	输入数据 x(n)
ref_output	所需的输出值 d(n)
coeff[]	自适应过滤器系数 h
no_coeffs	系数数量 K
res_shift	应用到各个输出的右移
conv_fact	收敛系数 2μ
workspace	工作空间的指针

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

描述：

使用最小均方 (LMS) 算法，执行用于单一数据元素的实数自适应 FIR 过滤器处理。

说明：

FIR 过滤器使用下列方程式定义。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res\_shift}$$

乘法累加运算的结果被保存为 39 位。 输出 y(n) 是取自 res_shift 右移位结果的低 16 位。 当发生溢出时，这是最大正或负值。
过滤器系数的更新使用 Widrow-Hoff 算法来执行。
hn+1(k)=hn(k)+2μe(n)x(n-k)
在这里，e(n) 是所需信号与实际输出之间的错误差数。
e(n)=d(n)-y(n)
使用 2μe(n)x(n-k) 计算，16 位 x 16 位的乘法被执行 2 次。 乘法结果的高 16 位都被保存，当发生溢出时，这是最大正或负值。若已更新的系数值是 H'8000，溢出可能会随着乘法累加运算发生。将系数值设定在 H'8001 到 H'7FFF 的范围内。有关系数缩放的详情，请参考“(2) 系数缩放”。由于系数是使用 LMS 过滤器来适应，最安全的缩放方法是设定少于 256 个系数，并将 res_shift 设定为 24。
conv_fact 一般应被设为正值。 请勿设定为 H'8000。
在调用此函数前，先调用 InitLms，然后初始化过滤器。
此函数不是可重入的。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
          0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
          0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
static short ref[N] = { -107, -143, 998, 1112, -5956,
                              -10781, 239, 13655, 11202, 2180,
                              -687, -2883, -7315, -6527, 196,
                              4278, 3712, 3367, 4101, 2703,
                              591, 695, -1061, -5626, -4200,
                              3585, 9285, 11796, 13416, 12994,
                              10231, 5803, -449, -6782, -11131,
                              -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

```
    /* 将 coeff 复制到 X RAM */


    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;



    ncoeff = K;/* 设定系数数量 */
    nsamp = N;/* 设定样品数量 */


    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
    for(i=0;i<nsamp;i++){
        if(Lms1(&output[i], data[i], ref[i], coeff_x, ncoeff, RSHIFT, TWOMU,
                                                                work) != EDSP_OK){
            printf("Lms1 Problem\n");
        }
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem\n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

将 X 存储器中的过滤器系数设定为变量。

过滤器初始化：
(1) 工作区域地址
(2) 系数数量
这是在 LMS1 函数执行前的必要步骤。在 Y 存储器中的工作区域使用（系数数量）*2+8 字节。

FreeLms 函数将释放用于 Lms 计算的工作区域。在 Lms 执行后，必须始终执行 FreeLms 函数。若这项函数未被执行，存储器资源将被浪费。

(i) FIR 工作空间分配

描述：

格式：

```
int InitFir  (short **workspace, long no_coeffs)
```

参数：

workspace	工作空间的指针
no_coeffs	系数数量 K

返回的值：

EDSP_OK	成功
EDSP_NO_HEAP	可被工作空间使用的存储器空间不足
EDSP_BAD_ARG	no_coeffs ≤ 2

描述：

分配由 Fir 及 Fir1 使用的工作空间。

说明：

将之前的所有输入数据初始化为 0。
只有 Fir、Fir1、Lms 或 Lms 1 能够操作以 InitFir 分配的工作空间。 请勿直接从用户程序存取工作空间。
此函数不是可重入的。

(j) IIR 工作空间分配

描述：

描述：

```
int InitIir  (short **workspace, long no_sections)
```

参数：

workspace	工作空间的指针
no_sections	二阶滤波器的段数 K

返回的值：

EDSP_OK	成功
EDSP_NO_HEAP	可被工作空间使用的存储器空间不足
EDSP_BAD_ARG	no_sections < 1

描述：

分配由 Iir 及 Iir1 使用的工作空间。

说明：

将之前的所有输入数据初始化为 0。
只有 Iir 和 Iir1 能够操作以 InitIir 分配的工作空间。 请勿直接从用户程序存取工作空间。
此函数不是可重入的。

(k) 双精度 IIR 工作空间分配

描述：

格式：

```
int InitDIir  (long **workspace, long no_sections)
```

参数：

workspace	工作空间的指针
no_sections	二阶滤波器的段数 K

返回的值：

EDSP_OK	成功
EDSP_NO_HEAP	可被工作空间使用的存储器空间不足
EDSP_BAD_ARG	no_sections < 1

描述：

分配由 DIir 及 DIir1 使用的工作空间。

说明：

将之前的所有输入数据初始化为 0。
只有 DIir 和 DIir1 能够操作以 InitDIir 分配的工作空间。
此函数不是可重入的。

(l) 自适应 FIR 工作空间分配

描述：

格式：

```
int InitLms  (short **workspace, long no_coeffs)
```

参数：

workspace	工作空间的指针
no_coeffs	系数数量 K

返回的值：

EDSP_OK	成功
EDSP_NO_HEAP	可被工作空间使用的存储器空间不足
EDSP_BAD_ARG	no_coeffs ≤ 2

描述：

分配由 Lms 及 Lms1 使用的工作空间。

说明：

将之前的所有输入数据初始化为 0。
只有 Fir、Fir1、Lms 或 Lms 1 能够操作以 InitLms 分配的工作空间。 请勿直接从用户程序存取工作空间。
此函数不是可重入的。

(m) FIR 工作空间释放

描述：

格式：

```
int FreeFir  (short **workspace, long no_coeffs)
```

参数：

workspace	工作空间的指针
no_coeffs	系数数量 K

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	no_coeffs ≤ 2

描述：

释放由 InitFir 分配的工作空间。

说明：

此函数不是可重入的。

(n) IIR 工作空间释放

描述：

格式：

```
int FreeIir  (short **workspace, long no_sections)
```

参数：

workspace	工作空间的指针
no_sections	二阶滤波器的段数 K

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	no_sections < 1

描述：

释放由 InitIir 分配的工作空间。

说明：

此函数不是可重入的。

(o) 双精度 IIR 工作空间释放

描述：

格式：

```
int FreeDIir  (long **workspace, long no_sections)
```

参数：

workspace	工作空间的指针
no_sections	二阶滤波器的段数 K

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	no_section ≤ 2

描述：

释放由 InitDIir 分配的工作空间存储器。

说明：

此函数不是可重入的。

(p) 自适应 FIR 工作空间释放

描述：

格式：

```
int FreeLms  (short **workspace, long no_coeffs)
```

参数：

workspace	工作空间的指针
no_coeffs	系数数量 K

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	no_coeffs < 1

描述：

释放由 InitLms 分配的工作空间存储器。

说明：

此函数不是可重入的。

3.13.7 卷积和相关

(1) 函数列表

表 3.32 DSP 程序库的函数列表（卷积）

编号	类型	函数名称	描述
1	全面卷积	ConvComplete	计算两个数组的全面卷积
2	周期卷积	ConvCyclic	计算两个数组的周期卷积
3	部分卷积	ConvPartial	计算两个数组的部分卷积
4	相关	Correlate	计算两个数组的相关
5	周期相关	CorrCyclic	计算两个数组的周期相关

当使用这些函数时，将两个输入数组的其中一个分配到 X 存储器，另一个则分配到 Y 存储器。 输出数组可被分配到任何一个存储器。

(2) 各个函数的解释

(a) 全面卷积

描述：

格式：

```
int ConvComplete  (short output[], const short ip_x[], const short ip_y[], long x_size,
                                        long y_size, int res_shift)
```

参数：

output[]	输出 z
ip_x[]	输入 x
ip_y[]	输入 y
x_size	ip_x 的 X 大小
y_size	ip_y 的 Y 大小
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •x_size < 1 •y_size < 1 •res_shift < 0 •res_shift > 25

描述：

全面卷积两个输入数组 x 和 y，然后将结果写入到输出数组 z。

说明：

$$z(m) = \left[\sum_{i=0}^{X-1} x(i)\, y(m-i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < X+Y-1$$

输入数组的外部数据被读取为 0。
ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。
此外，有必要确保数组输出大小大于 (xsize+ysize-1)。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX   8
#define NY   8
#define NOUT  NX+NY-1
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
short w1[5] = {-1, -32768, 32767, 2, -3, };
short x1[5] = {1, 32767, -32767, -32767, -2, };
void main()
{
    short i;
    short output[NOUT];
    int xsize, ysize, rshift;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w1[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x1[i%5];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvComplete(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NX;i++){
        printf("#%3d  dat_x:%6d  dat_y:%6d \n",i,datx[i],daty[i]);
    }
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定用于卷积计算的数据。

```
    for(i=0;i<NOUT;i++){
        printf("#%3d  output:%d \n",i,output[i]);
    }
}
```

(b) 周期卷积

描述：

格式：

```
int ConvCyclic  (short output[], const short ip_x[],
                                 const short ip_y[], long size,
                                 int res_shift)
```

参数：

output[]	输出 z
ip_x[]	输入 x
ip_y[]	输入 y
size	数组的 N 大小
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •size < 1 •res_shift < 0 •res_shift > 25

描述：

周期卷积两个输入数组 x 和 y，然后将结果写入到输出数组 z。

说明：

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|m - i + N|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

在这里，|i|N 表示余数 (i % N)。
ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。
此外，有必要确保数组输出大小大于‘size’。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x2[5] = {1, 32767, -32767, -32767, -2, };
short w2[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w2[i];
        daty[i] = x2[i];
    }
    size = N ;
    rshift = 15;
    if(ConvCyclic(output, datx, daty, size, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  ip_x:%6d  ip_y:%6d  output:%6d \n",i,datx[i],daty[i],
                                                                output[i]);
    }
}
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定用于卷积计算的数据。

(c) 部分卷积

描述：

格式：

```
int ConvPartial  (short output[], const short ip_x[],
                                  const short ip_y[], long x_size,
                                  long y_size,  int res_shift)
```

参数：

output[]	输出 z
ip_x[]	输入 x
ip_y[]	输入 y
x_size	ip_x 的 x 大小
y_size	ip_y 的 y 大小
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •x_size < 1 •y_size < 1 •res_shift < 0 •res_shift > 25

描述：

此函数卷积两个输入数组 x 和 y，然后将结果写入到输出数组 z。

说明：

不包含取自输入数组外部数据的输出。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res\_shift} \quad 0 \le m \le |A\text{-}B|$$

然而，数组的数量是 a < b，而 A 是 a 大小，B 是 b 大小。
输入数组的外部数据被读取为 0。
ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。
此外，有必要确保数组输出大小大于 (|xsize-ysize|+1)。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX    5
#define NY    5
short x3[5] = {1, 32767, -32767, -32767, -2, };
```

} 包括标头

```
short w3[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
void main()
{
    short i;
    short output[NY+NX];
    int ysize, xsize, rshift;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w3[i];
    }
    for(i=0;i<NY;i++){
        daty[i] = x3[i];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvPartial(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NX;i++){
        printf("ip_x=%d  \n",datx[i]);
    }
    for(i=0;i<NY;i++){
        printf("ip_y=%d  \n",daty[i]);
    }
    for(i=0;i<(NY+NX);i++){
        printf("output=%d \n",output[i]);
    }
}
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定用于卷积计算的数据。

(d) 相关

描述：

格式：

```
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)
```

参数：

output[]	输出 z
ip_x[]	输入 x
ip_y[]	输入 y
x_size	ip_x 的 x 大小
y_size	ip_y 的 y 大小
no_corr	用于计算的相关数量 M
x_is_larger	x=y 时的数组指定
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •x_size < 1 •y_size < 1 •no_corr < 1 •res_shift < 0 •res_shift > 25 •x_is_larger ≠ 0 or 1

描述：

找出两个输入数组 x 和 y 的相关，然后将结果写入到输出数组 z。

说明：

在下面的方程式中，数组的数量是 a < b，而 A 是 a 大小。 若 x_is_larger=0 使 x 成为 a，若 x_is_larger=1 则使 x 成为 b。
当 b 数组小于 a 数组时，运算不被保证。
设定输入数组 x 和 y 的大小，以及 x_is_larger，使不存在冲突。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(i+m)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < M$$

A < X + M 没有障碍。在这种情况下，对数组的外部数据使用 0。
res_shift=0 响应正常的整数计算，而 res_shift=15 响应小数计算。
ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。
此外，有必要确保数组输出大小大于 no_corr。

使用的实例：

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

} 包括标头

```
#define NY   5
#define NX   5
#define M    4
#define MAXM  NX+NY
short x4[5] = {1, 32767, -32767, -32767, -2, };
short w4[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
void main()
{
    short i;
    int ysize, xsize, ncorr, rshift;
    short output[MAXM];
    int x_is_larger;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<NX;i++){
         datx[i] = w4[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x4[i%5];
    }
    /* 测试堆栈工作 */
    ysize = NY;
    xsize = NX;
    ncorr = M;
    rshift = 15;
    x_is_larger=0;
    for (i = 0; i < MAXM; output[i++] = 0);
    if (Correlate(output, datx, daty, xsize, ysize, ncorr,x_is_larger,rshift)
                                                                    != EDSP_OK){
       printf("EDSP_OK not returned\n");
    }
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定在计算中使用的数据。

```
    for(i=0;i<MAXM;i++){
        printf("[%d]:output=%d\n",i,output[i]);
    }
}
```

(e) 周期相关

描述：

格式：

```
int CorrCyclic  (short output[], const short ip_x[],
                                  const short ip_y[], long size, int reverse,
                                  int res_shift)
```

参数：

output[]	输出 z
ip_x[]	输入 x
ip_y[]	输入 y
size	数组的 N 大小
reverse	反标志
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •size < 1 •res_shift < 0 •res_shift > 25 •reverse ≠ 0 or 1

描述：

周期性的找出两个输入数组 x 和 y 的相关，然后将结果写入到输出数组 z。

说明：

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\ y\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

在这里，|i|N 表示余数 (i % N)。 若 reverse=1，输出数据被倒转，实际计算如下。

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\ x\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。
此外，有必要确保数组输出大小大于‘size’。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x5[5] = {1, 32767, -32767, -32767, -2, };
short w5[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    int reverse;
    int result;
    /* 测试 X 与 Y 的周期相关 */
    reverse=0;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w5[i];
        daty[i] = x5[i];
    }
    /* 测试堆栈工作 */
    size = N;
    rshift = 15;
    if (CorrCyclic(output, datx, daty, size, reverse, rshift) != EDSP_OK){
        printf("EDSP_OK not returned - this one\n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定在计算中使用的数据。

3.13.8 其他

(1) 函数列表

表 3.33 DSP 程序库函数列表 （其他）

编号	类型	函数名称	描述
1	H'8000 → H'8001 替换	Limit	将 H'8000 数据替换成 H'8001。
2	X 存储器 → Y 存储器复制	CopyXtoY	将数组从 X 存储器复制到 Y 存储器。
3	Y 存储器 → X 存储器复制	CopyYtoX	将数组从 Y 存储器复制到 X 存储器。
4	复制到 X 存储器	CopyToX	将数组从特定地点复制到 X 存储器。
5	复制到 Y 存储器	CopyToY	将数组从特定地点复制到 Y 存储器。
6	从 X 存储器复制	CopyFromX	将数组从 X 存储器复制到特定地点。
7	从 Y 存储器复制	CopyFromY	将数组从 Y 存储器复制到特定地点。
8	高斯白噪声	GenGWnoise	发出高斯白噪声。
9	矩阵乘法	MatrixMult	两个矩阵相乘。
10	乘法	VectorMult	两个数据元素相乘。
11	RMS 值	MsPower	决定 RMS 功率。
12	平均数	Mean	决定平均数。
13	平均数与变异数	Variance	决定平均数与变异数。
14	最大值	MaxI	决定整数数组的最大值。
15	最小值	MinI	决定整数数组的最小值。
16	最大绝对值	PeakI	决定整数数组的最大绝对值。

(2) 各个函数的解释

(a) H'8000 → H'8001 替换

描述：

格式：

```
int Limit (short data[], long no_elements, int data_is_x)
```

参数：

data[]	数据组
no_elements	数据元素数量
data_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •data_is_x ≠ 0 or 1

描述：

将具有 H'8000 值的输入数据替换成 H'8001。这样一来，当使用 DSP 指令执行定点乘法时，不会发生溢出。

说明：

即使在执行过程时，都有可能因为乘法累加运算中的加法而发生溢出。
当 data_is_x=1 时，将数据分配到 X 存储器，当 data_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    int size;
    int src_x;
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

```
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = dat[i%4];
        daty[i] = dat[i%4];
        printf("BEFORE NO %d datx daty :%d:%d \n",i,datx[i], daty[i]);
    }
    size = N;
    src_x = 1;

    if (Limit(datx, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned\n");
    }
    src_x = 0;
    if (Limit(daty, size, src_x) != EDSP_OK){
        printf( "EDSP_OK not returned\n");
    }
    for(i=0;i<N;i++) {
        printf("After NO %d datx daty :%d:%d\n",i,datx[i], daty[i]);
    }
}
```

设定数据。

若使用 X 存储器

若使用 Y 存储器

(b) X 存储器 → Y 存储器复制

描述：

格式：

```
int CopyXtoY  (short op_y[], const short ip_x[], long n)
```

参数：

op_y[]	输出数组
ip_x[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

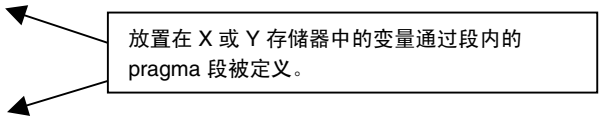
数组从 ip_x 被复制到 op_y。

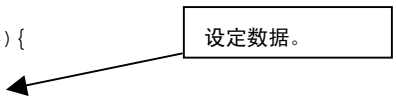
说明：

将 ip_x 分配到 X 存储器，将 op_y 分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=0;
        datx[i]=dat[i%4];
    }
    if(CopyXtoY(daty, datx, N) != EDSP_OK){
        printf("CopyXtoY Problem\n");
    }
    printf("no_elements:%d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  ip_y:%6d \n",i,datx[i],daty[i]);
    }
}
```





(c) Y 存储器 → X 存储器复制

描述：

格式：

```
int CopyYtoX  (short op_x[], const short ip_y[], long n)
```

参数：

op_x[]	输出数组
ip_y[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

数组从 ip_y 被复制到 op_x。

说明：

将 ip_y 分配到 Y 存储器，将 op_x 分配到 X 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[N] = { -32768, 32767, -32768, 0,3};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyYtoX(datx, daty, N)!= EDSP_OK){
     printf("CopyYtoX error!\n");
    }
    printf("no_elements %d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  po_x:%6d  ip_y:%6d \n",i,datx[i],daty[i]);
    }
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

(d) 复制到 X 存储器

描述：

格式：

```
int CopyToX  (short op_x[], const short input[], long n)
```

参数：

op_x[]	输出数组
input[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

数组输入被复制到 op_x。

说明：

将 op_x 分配到 X 存储器，将输入分配到任意存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section
void main()
{
    int   i;
    short data[N];

    for(i=0;i<N;i++){
       data[i]=dat[i];
    }
    if(CopyToX(datx, data, N) !=EDSP_OK){
        printf("CopyToX Problem\n");
    }
```

包括标头

放置在 X 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

```
    printf("no_elements %d\n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  input:%6d \n",i,datx[i],data[i]);
    }
}
```

(e) 复制到 Y 存储器

描述：

格式：

```
int CopyToY  (short op_y[], const short input[], long n)
```

参数：

op_y[]	输出数组
input[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

数组输入被复制到 op_y。

说明：

将 op_y 分配到 Y 存储器，将输入分配到任意存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;
    short data[N] ;
```

包括标头

放置在 Y 存储器中的变量通过段内的 pragma 段被定义。

```
    for(i = 0; i < N; i++){
        data[i] = dat[i%4] ;
    }
    if(CopyToY(daty, data, N) != EDSP_OK){
        printf("CopyToY Problem\n");
    }
    printf("no_elements %ld \n",N);
    for(i = 0; i < N; i++){
        printf("#%2d  op_y:%6d  input:%6d \n",i,daty[i],data[i]);
    }
}
```

设定数据。

(f) 从 X 存储器复制

描述：

格式：

```
int CopyFromX  (short output[], const short ip_x[], long n)
```

参数：

output[]	输出数组
ip_x[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

数组 ip_x 被复制到输出。

说明：

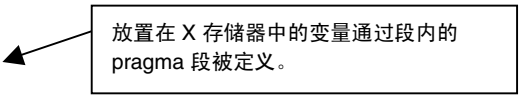
将 ip_x 分配到 X 存储器，将输出分配到任意存储器。

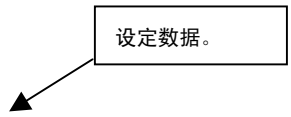
使用的实例：

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

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;
```

包括标头

```
#pragma section X
static short datx[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        datx[i]=dat[i];
    }
    if(CopyFromX(out_dat,datx, N) != EDSP_OK){
        printf("CopyFromX Problem\n");
    }
    for(i=0;i<N;i++){
        printf("#%3d output:%6d  ip_x:%6d \n",i,out_dat[i],datx[i]);
    }
    printf("no_elements:%ld\n",N);
}
```





(g) 从 Y 存储器复制

描述：

格式：

```
int CopyFromY  (short output[], const short ip_y[], long n)
```

参数：

output[]	输出数组
ip_y[]	输入数组
n	数据元素数量

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

描述：

数组 ip_y 被复制到输出。

说明：

将 ip_y 分配到 Y 存储器，将输出分配到任意存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;


#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;


    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyFromY(out_dat,daty, N)!= EDSP_OK){
         printf("CopyFormY Problem\n");
    }
    printf("no_elements:%d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_y:%6d \n",i,out_dat[i],daty[i]);
    }
}
```

包括标头

放置在 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

(h) 高斯白噪声

描述：

格式：

```
int GenGWnoise  (short output[], long no_samples,
                                  float variance)
```

参数：

output[]	输出白噪声数据
no_samples	输出数据的元素数
Variance	噪声发送的变异数 σ2

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_samples < 1 •variance ≤ 0.0

描述：

平均数为 0，高斯白噪声以用户指定的变异数发出。

说明：

生成两组输出数据元素的其中一组。 为生成 1 组输出数据，应使用 rand 函数，直至 x 平方的总和找出少于 1 的一组，1 组随机数，介于 –1 和 1 之间的 γ1 及 γ2 将被生成。 然后使用下列方程式算出 1 组输出数据 o1 和 o2。

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

若数据元素数被设定为奇数，上一组的第二个数据元素将被取消。
由于被此函数调用的标准程序库 rand 函数是不可再入的，因此所生成的随机数 γ1 和 γ2 的顺序并非每次相同。然而，这对所生成的白噪声 o1 和 o2 的特征没有影响。
此函数使用浮点运算。 由于浮点运算的处理速度缓慢，建议将此函数用于评估。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>          } 包括标头
#define MAXG  4.5   /* N(0,1) 随机变量的大约饱和级 */
#define N_SAMP  10  /* 一个帧内生成的样品数量            */
void main()
{
    short out[N_SAMP];
    float var;
    int   i;
    var = 32768 / MAXG * 32768 / MAXG;
    if(GenGWnoise(out, N_SAMP, var) !=EDSP_OK){
        printf("GenGWnoise Problem\n");
    }
    for(i=0;i<N_SAMP;i++){
        printf("#%2d  out:%6d  \n",i,out[i]);
    }
}
```

(i) 矩阵乘法

描述：

格式：

```
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)
```

参数：

op_matrix	第一个输出数据元素的指针
ip_x	输入 x 第一个数据元素的指针
ip_y	输入 y 第一个数据元素的指针
m	矩阵 1 中的行数
n	矩阵 1 中的列数，矩阵 2 中的行数
p	矩阵 2 中的行数
x_first	矩阵乘法的顺序指定
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •m, n, or p < 1 •res_shift < 0 •res_shift > 25 •x_first ≠ 0 or 1

描述：

执行两个矩阵 x 和 y 的乘法，然后将结果分配到 op_matrix。

说明：

当 x_first=1 时，计算 x · y。在这里，ip_x 是 m x n，ip_y 是 n x p，而 op_matrix 是 m x p。
当 x_first=0 时，计算 y · x。在这里，ip_y 是 m x n，ip_x 是 n x p，而 op_maxtrix 是 m x p。
乘法累加运算的结果被保存为 39 位。 输出 y(n) 是取自 res_shift 右移位结果的低 16 位。 当发生溢出时，这是最大正或负值。
每个矩阵被分配到普通 C 格式（以行为主的顺序分配）。

a0 a1 a2 a3
a4 a5 a6 a7
a8 a9 a10 a11

为能指定任意数组大小，为数组参数指定 void*。 将这些参数指向 short 变量。
分别提供输入数组 ip_x 和 ip_y，及输出数组 op_matrix。
将 ip_x 分配到 X 存储器，ip_y 分配到 Y 存储器，及将 op_matrix 分配到任意存储器。

使用的实例：

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

} 包括标头

```
#define NN  N*N
short m1[16] = {  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767, };
short m2[16] = {  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767, };
#pragma section X
static short datx[NN];
#pragma section Y
static short daty[NN];
#pragma section
void main()
{
    short i, j;
    short output[NN];
    int m, n, p, rshift, x_first;
    long sum;
    for (i = 0; i < NN; output[i++] = 0) ;
    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<NN;i++) {
        datx[i] = m1[i%16];
        daty[i] = m2[i%16];
    }
    m = n = p = N;
    rshift = 15;
    x_first = 1;
    if (MatrixMult(output, datx, daty, m, n, p, x_first, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NN;i++) {
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

(j) 乘法

描述：

格式：

```
int VectorMult  (short output[], const short ip_x[],
                                 const short ip_y[],long no_elements,
                                 int res_shift)
```

参数：

output[]	输出
ip_x[]	输入 1
ip_y[]	输入 2
no_elements	数据元素数
res_shift	应用到各个输出的右移

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •res_shift < 0 •res_shift > 16

描述：

每次从 ip_x 和 ip_y 取一个数据元素并执行乘法，然后将结果分配到输出。

说明：

输出是取自 res_shift 右移位结果的低 16 位。
当发生溢出时，这是最大正或负值。
此函数执行数据的乘法。 要计算内部积，可使用 MatrixMult 函数，将 m（矩阵 1 的行数）及 p（矩阵 2 的列数）设定为 1。
ip_x 被分配到 X 存储器，ip_y 被分配到 Y 存储器，而输出被分配到任意存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
#define RSHIFT  15
short y[4] = {1, 32767, -32767, 32767, };
short x[4] = {-1, 32767, -32767, -32767, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i, n ;
    short output[N];
    int size, rshift;

    /* 将数据复制到 X 和 Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = x[i];
        daty[i] = y[i];
    }
    size = N;
    rshift = RSHIFT;
    for (i = 0; i < N; output[i++] = 0) ;
    if (VectorMult(output, datx, daty, size, rshift) != EDSP_OK) {
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_x:%6d  ip_y:%6d \n",i,output[i],datx[i],
                                                                    daty[i]);
    }
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

(k) RMS 值

描述：

格式：

```
int MsPower  (long *output, const short input[],
                            long no_elements, int src_is_x)
```

参数：

output	输出的指针
input[]	输入 x
no_elements	数据元素数 N
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

决定输入数据的 RMS 值。

说明：

$$\text{平均2乘值} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

将除法结果舍入到最近的整数。
运算的结果被保存为 63 位。
若 no_elements 是 2^{32}，将可能发生溢出。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。
将输出分配到任意存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    int    i;
    long   output[1];
    int    src_x;

    /* 将数据复制到 X 和 Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    src_x = 1;
    if (MsPower(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("MsPower:x=%d\n",output[0]);
    src_x = 0;
    if (MsPower(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("MsPower:y=%d\n",output[0]);
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

(l) 平均数

描述：

格式：

```
int Mean  (short *mean, const short input[], long no_elements,
                        int src_is_x)
```

参数：

mean	输入平均数的指针
input[]	输入 x
no_elements	数据元素数 N
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

决定输入数据的平均数。

说明：

$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

将除法结果舍入到最近的整数。
运算结果被保存为 32 位。 若 no_elements 大于 2^{16}-1，将可能发生溢出。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     i,output[1];
    int       size;
    int       src_x;
    int       flag = 1;
    /* 将数据复制到 X 和 Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* 测试堆栈工作 */
    src_x = 1;
    if (Mean(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Mean:x=%d\n",output[0]);
    src_x = 0;
    if (Mean(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Mean:y=%d\n",output[0]);
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

(m) 平均数和变异数

描述：

格式：

```
int Variance  (long *variance, short *mean, const short input[],
                              long no_elements, int src_is_x)
```

参数：

Variance 变异数	输入变异数σ^2的指针
mean	数据平均数 $\bar{x}$ 的指针
input[]	输入 x
no_elements	数据元素数 N
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

决定输入的平均数与变异数。

说明：

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \bar{x}^2$$

将除法结果舍入到最近的整数。
x 被保存为 32 位。 没有溢出检查。
若 no_elements 大于 2^{16}-1，将可能发生溢出。
σ^2 被保存为 63 位。 没有溢出检查。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    long        size,var[1];
    short       mean[1];
    int         i ;
    int         src_x;

    /* 将数据复制到 X 和 Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* 测试堆栈工作 */
    size = N;
    src_x = 1;
    if (Variance(var, mean, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Variance:%d  mean:%d \n ",var[0],mean[0]);
    src_x = 0;
    if (Variance(var, mean, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Variance:%d  mean:%d \n ",var[0],mean[0]);
}
```

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

(n) 最大值

描述：

格式：

```
int MaxI  (short **max_ptr, short input[], long no_elements,
                        int src_is_x)
```

参数：

max_ptr	最大数据的指针
input[]	输入
no_elements	数据元素数
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

在数组输入中搜索最大值，然后将其地址返回到 max_ptr。

说明：

若数个数据元素具有相同的最大值，将返回起始地址与输入最接近的数据。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[131] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short    *outp,**outpp;
    int       size,i;
    int       src_x;
 /* 将数据复制到 X 和 Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MAXI */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MaxI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Max:x = %d\n",**outpp);
    src_x = 0;
    if (MaxI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Max:y = %d\n",**outpp);
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

(o) 最小值

描述：

格式：

```
int MinI  (short **min_ptr, short input[], long no_elements,
                        int src_is_x)
```

参数：

min_ptr	最小数据的指针
input[]	输入
no_elements	数据元素数
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

在数组输入中搜索最小值，然后将其地址返回到 min_ptr。

说明：

若数个数据元素具有相同的最小值，将返回起始地址与输入最接近的数据。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 10
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MINI */
    /* test working of stack */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MinI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Min:x=%d\n",**outpp);
    src_x = 0;
    if (MinI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Min:y=%d\n",**outpp);
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

(p) 最大绝对值

描述：

格式：

```
int PeakI  (short **peak_ptr, short input[], long no_elements,
                          int src_is_x)
```

参数：

peak_ptr	最大绝对值数据的指针
input[]	输入
no_elements	数据元素数
src_is_x	数据分配指定

返回的值：

EDSP_OK	成功
EDSP_BAD_ARG	在下列任何情况下 •no_elements < 1 •src_is_x ≠ 0 or 1

描述：

在数组输入中搜索最大绝对值，然后将其地址返回到 peak_ptr。

说明：

若数个数据元素具有相同的最大绝对值，将返回起始地址与输入最接近的数据。
当 src_is_x=1 时，将输入分配到 X 存储器，当 src_is_x=0 时，将数据分配到 Y 存储器。

使用的实例：

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;
    /* 将数据复制到 X 和 Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    size = N;
    outpp = &outp;
    src_x = 1;
    if (PeakI(outpp, datx, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned\n");
    }
    printf("Peak:x=%d\n",**outpp);
    src_x = 0;
    if (PeakI(outpp, daty, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned\n");
    }
    printf("Peak:y=%d\n",**outpp);
}
```

包括标头

放置在 X 或 Y 存储器中的变量通过段内的 pragma 段被定义。

设定数据。

当使用了 X 存储器时，src_x=1。

当使用了 Y 存储器时，src_x=0。

3.14 DSP 程序库的性能

(1) DSP 程序库的执行周期数

DSP 程序库中函数所需的执行周期数如下所示。
通过将程序段分配到 X-ROM 或 Y-ROM，可使用仿真程序 (SH-DSP, 60 MHz) 来执行测量。

表 3.34 DSP 程序库函数的执行周期列表 (1)

类别	DSP 程序库函数名称	执行周期数（周期）	注意
快速傅立叶转换	FftComplex	29,330	大小：256
	FftReal	25,490	缩放：0xFFFFFFFF
	IfftComplex	30,380	
	IfftReal	29,240	
	FftInComplex	26,540	
	FftInreal	25,260	
	IfftInComplex	27,590	
	IfftInReal	27,470	
	LogMagnitude	1,778,290	
	InitFft	3,116,640	
	FreeFft	780	
过滤器函数	Fir	23,010	系数数量：64
	Fir1	280	数据项目数量：200
	Lms	97,710	收敛系数 2μ = 32767
	Lms1	790	
	InitFir	1,400	
	InitLms	1,400	
	FreeFir	90	
	FreeLms	90	
	Iir	23,530	数据项目数量：200
	Iir1	360	过滤器段的数量：5
	DIir	309,010	
	DIir1	1,860	
	InitIir	280	
	InitDIir	280	
	FreeIir	90	
	FreeDIir	270	

表 3.34 DSP 程序库函数的执行周期列表 (2)

类别	DSP 程序库函数名称	执行周期数（周期）	注意
窗口	GenBlackman	789,950	数据项目数量：100
函数	GenHamming	418,330	
	GenHanning	447,250	
	GenTriangle	744,220	
卷积	ConvComplete	21,890	数据项目数量：100
函数	ConvCyclic	14,790	
	ConvPartial	370	
	Correlate	11,930	
	CorrCylic	15,790	
其他	Limit	480	数据项目数量：100
函数	CopyXtoY	130	
	CopyYtoX	130	
	CopyToX	1,270	
	CopyToY	1,270	
	CopyFromX	1,320	
	CopyFromY	1,320	
	GenGWnoise	2,878,410	
	MatrixMult	2,337,460	
	VectorMult	1,500	
	MsPower	370	
	Mean	270	
	Variance	820	
	MaxI	540	
	MinI	520	
	PeakI	740	

(2) C 语言与 DSP 程序库源代码的比较

在这里，一些 FFT 相关函数（执行 butterfly 计算的函数）的源代码以 C 语言呈现及取自 DSP 程序库。

在 DSP 程序库源代码中，指定 DSP 的指令，如 movx、movy 及 padd 被用于增进 DSP 程序库的性能。

C 源代码

```
void R4add(short *arp, short *brp, short *aip, short *bip, int grpinc, int numgrp)
{
short   tr,ti;
int     grpind;
        for(grpind=0;grpind<numgrp;grpind++) {
                tr  =  *brp;
                ti  =  *bip;
                *brp = sub(*arp,ti);
                *bip = add(*aip,tr);
                *arp = add(*arp,ti);
                *aip = sub(*aip,tr);
                arp += grpinc;
                aip += grpinc;
                brp += grpinc;
                bip += grpinc;
        }
}
```

DSP 程序库源代码

```
_R4add:

MOV.L   Ix,@-R15
        MOV.L   Iy,@-R15

MOV.L   @(2*4,R15),Ix
        SHLL    Ix
        MOV     Ix,Iy
        MOV.L   @(3*4,R15),R1

        REPEAT r4alps,r4alpe
        ADD     #-1,R1
        SETRC   R1
                                movx.w  @ar,X0      movy.w @bi,Y0
        padd    X0,Y0,A0
        psub    X0,Y0,A1        movx.w  @br,X0      movy.w @ai,Y0
        padd    X0,Y0,A0        movx.w  A0,@ar+Ix
        pneg    X0,X0           movx.w  A1,@br+Ix
        padd    X0,Y0,A1        movy.w A0,@bi+Iy
```

```
        movx.w @ar,X0           movy.w @bi,Y0
        .ALIGN 4
r4alps  padd   X0,Y0,A0         movy.w A1,@ai+Iy
        psub   X0,Y0,A1         movx.w  @br,X0     movy.w @ai,Y0
        padd   X0,Y0,A0         movx.w  A0,@ar+Ix
        pneg   X0,X0            movx.w  A1,@br+Ix
        padd   X0,Y0,A1         movy.w A0,@bi+Iy
r4alpe                          movx.w  @ar,X0     movy.w @bi,Y0
                                movy.w A1,@ai+Iy
        MOV.L  @R15+,Iy
        RTS
        MOV.L  @R15+,Ix
```

(3) 个别 FFT 函数的性能

傅立叶转换函数被分类如下。

表 3.35　快速傅立叶转换函数

	不在位函数	在位函数
复数傅立叶转换	FftComplex	FftInComplex
实数傅立叶转换	FftReal	FftReal

表 3.36　快速傅立叶反转函数

	不在位函数	在位函数
复数傅立叶转换	IfftComplex	IfftInComplex
实数傅立叶转换	IfftReal	IfftInReal

在位与不在位函数之间的区别

在位函数将输入数据的数组用作输出数据的数组。 因此输入数据被输出数据覆盖，而未被保存。

当使用不在位函数时，输入和输出数据必须在调用函数前分开准备。输入数据和输出数据是分开的，因此即使在调用函数后，输入数据仍被保存。

在位与不在位函数在性能上几乎没有区别，因此应根据可用的存储器空间来决定要使用的函数类型。

与不在位函数相较，在位函数只需要一半的存储器。

- 关于缩放

  在 FFT 计算的各个阶段，计算将以乘法累加的形式执行，所以可能会发生溢出。 若发生了溢出，所有值将变成最大值或最小值，因此无法正确评估计算结果。

  为防止溢出，可在 FFT 计算的各个阶段执行缩放，缩放是 2，即值是被 2 除（右移）。

表 3.37 缩放值和特性

缩放值	特性
FFTNOSCALE	没有任何移位，可能会发生溢出
EFFTMIDSCALE	每隔一个阶段移位
EFFTALLSCALE	在所有阶段移位，不容易发生溢出

缩放对性能没有很大的影响。 因此当决定进行缩放时，应考虑数据的特性，更甚于性能。

(4) 过滤器函数

使用 Fir 和 Lms

图 3.11 显示 Fir 和 Lms 过滤器在系数与周期数量方面的区别。

由于 Lms 过滤器使用自适应算法，因此计算速度较 Fir 过滤器为慢。 在具有稳定数据波形的系统中，应使用 Lms 来决定过滤器系数，然后以 Fir 过滤器将它替换。

可为数据缩放指定右移数量。由于乘法累加运算在 SH-DSP 程序库函数内部使用，视输入数据而定，可能发生溢出。在这种情况下，应对右移的数量做适当修改，且应参考输出值来做出选择。

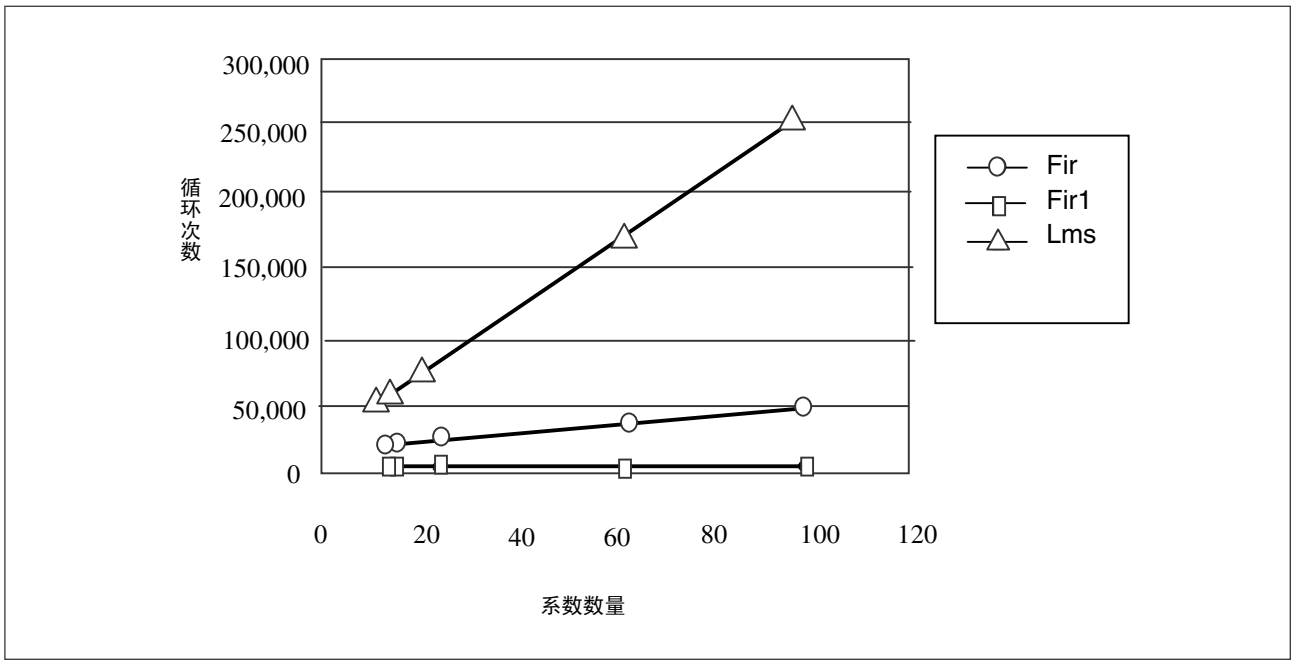


图 3.11 系数数量与周期数量之间的关系

- Iir 和 DIir

  当以性能为优先时，应使用 Iir，而非 DIir。 由于乘法累加运算在 SH-DSP 程序库函数内部使用，视输入数据而定，可能发生溢出。 在这种情况下，应对右移的数量做适当修改，且应参考输出值来做出选择。

  可为数据缩放指定右移数量。然而，右移数量被指定为过滤器系数的部分数组。有关详细信息，请参考第 3.13.16 节，(5)(c) IIR 和 (e) 双精度 IIR。

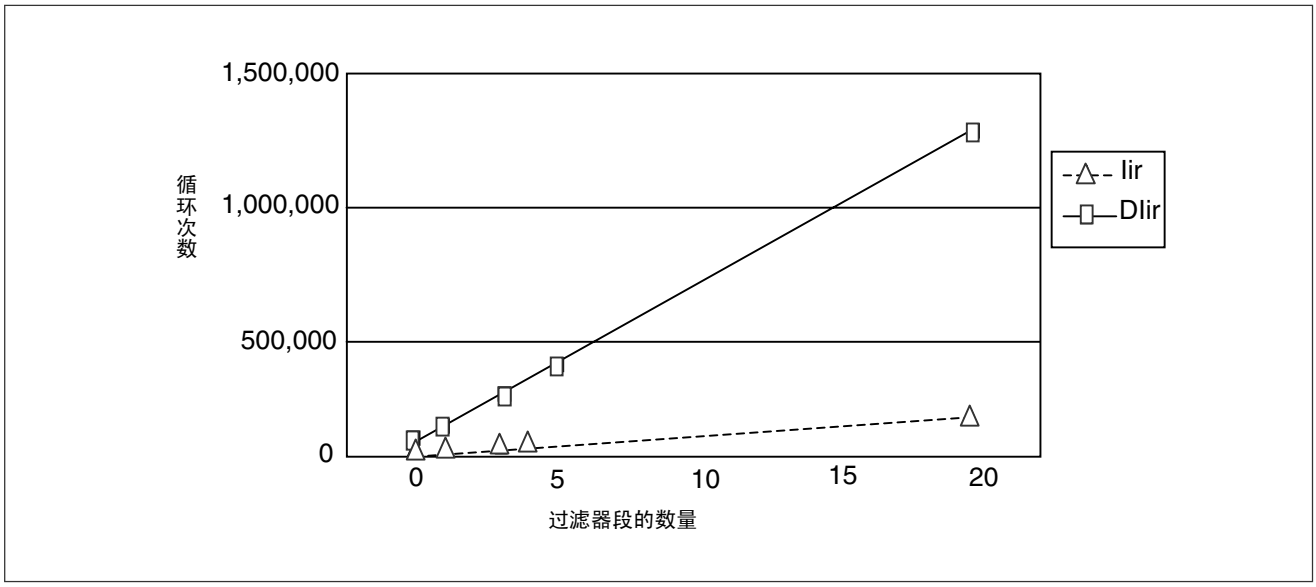


图 3.12 过滤器段数与周期数量之间的关系

- 选择性使用过滤器函数

  Fir 过滤器具有线性相位反映，且始终稳定，使其适合在音频、视频及其他无法承受相位失真的应用中使用。另一方面，Iir 过滤器包括反馈，而且可使用比 Fir 少的系数获得结果，因此执行速度较快，适用于具有时间限制的应用。然而，Iir 过滤器可能在某些状况中不稳定，因此在使用时应格外谨慎。

3.15 交叉软件的相关事项

3.15.1 汇编语言程序的相关事项

由于 SuperH RISC engine C/C++ 编译程序支持 Renesas Technology SuperH RISC engine 家族的特别指令以及标准指令，几乎任何一种程序都可使用 C 语言来编写。 然而，当需要扩展性能时，重要的代码段则经常使用汇编语言来编写，然后与 C 语言程序合并。
本节介绍在结合 C 语言程序与汇编语言代码时所应注意的一些事项。

- 外部名称的相互参考
- 函数调用界面

要获取详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)。

(1) 外部名称的相互参考

(a) 从 C 语言程序参考汇编语言程序中的外部定义名称
以下程序被用于从 C 语言程序参考汇编语言程序中的外部定义名称。

- 在汇编语言程序中，以下划线 ("_") 起始的符号名称（32 个字符内）使用“.EXPORT”或“GLOBAL”汇编程序指令声明为外部定义。
- 在 C 语言程序中，“extern”存储类说明符被用于为外部参考声明符号名称，而没有前导的“_”。

汇编语言程序
（定义变量）

```
      .EXPORT        _a , _b
      .SECTION   D, DATA, ALIGN=4
_a :  .DATA.L      1
_b :  .DATA.L      1
      .END
```

C 语言程序
（引用变量）

```
 extern  int  a   , b;
f ()
{
      a+=b;
}
```

图 3.13 使用 C 语言程序参考在汇编语言程序内的外部定义变量的实例

(b) 从汇编语言程序参考 C 语言程序中的外部定义名称

在 C 语言程序中，外部定义名称包括下列各项。

- 不属于静态存储类变量的全局变量
- 使用 extern 存储类声明的变量名称
- 未指定静态存储类的函数名称

C 语言程序的外部定义名称从汇编语言程序参考如下。

- 符号名称（没有前导的“_”）在 C 语言程序中被外部定义（为全局变量）。
- 在汇编语言程序中，“.IMPORT”或“.GLOBAL”汇编程序指令被用来声明具有前导的“_”的符号名称外部参考。

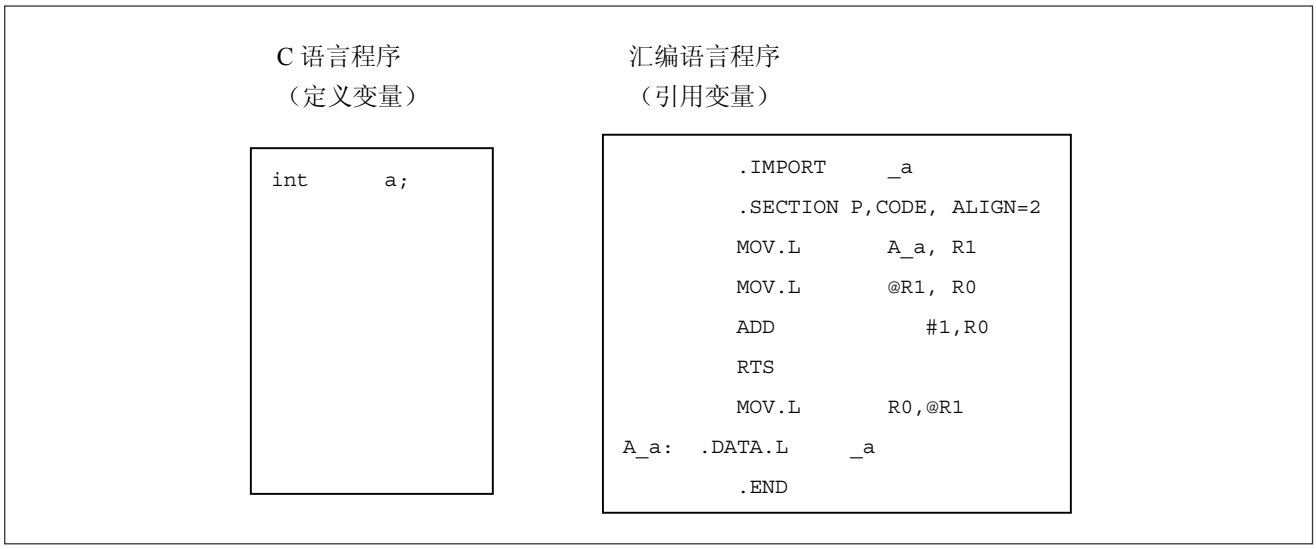


图 3.14 使用汇编语言程序参考在 C 语言程序内的外部定义变量的实例

注意： 从静态数据成员创建的函数名称及外部名称由 C++ 编译程序使用固定规则转换。 当有需要了解由编译程序生成的外部名称时，应使用编译程序选项 code=asm 或 listfile 来查看由编译程序生成的外部名称。 若 C++ 函数由添加 extern C 定义，外部名称使用相似于 C 函数的规则生成。 然而，这类函数将不可能超载。

(2) 函数调用界面

当 C 语言程序或汇编语言程序调用另一语言的函数时，汇编语言程序必须遵守以下四项规则。

(i) 有关堆栈指针的规则

(ii) 分配及释放堆栈帧的规则

(iii) 有关寄存器的规则

(iv) 有关设定及参考参数和返回值的规则

这里将解释规则 (i) 至 (iii)。 有关 (iv) 的信息，请参考第 3.15.1 (3) 节“设定及参考参数和返回值”。

(a) 有关堆栈指针的规则

有效数据不应保存在低于（地址 0 的方向）堆栈指针地址的堆栈区域中。任何保存在低于堆栈指针的地址的数据可能因中断处理而损坏。

(b) 分配及释放堆栈帧的规则

当函数被调用时（紧接着 JSR 或 BSR 指令执行后），堆栈指针将指向堆栈中由调用源函数使用的最低地址。 将数据分配及设定在高于此（地址 H'FFFFFFFF 的方向）的地址是调用源函数的任务。

一般上，RTS 指令被用来在调用目标函数所使用的区域被释放后，将控制返回给调用源函数。高于此（返回值地址与参数区域）的地址区域被调用源函数释放。

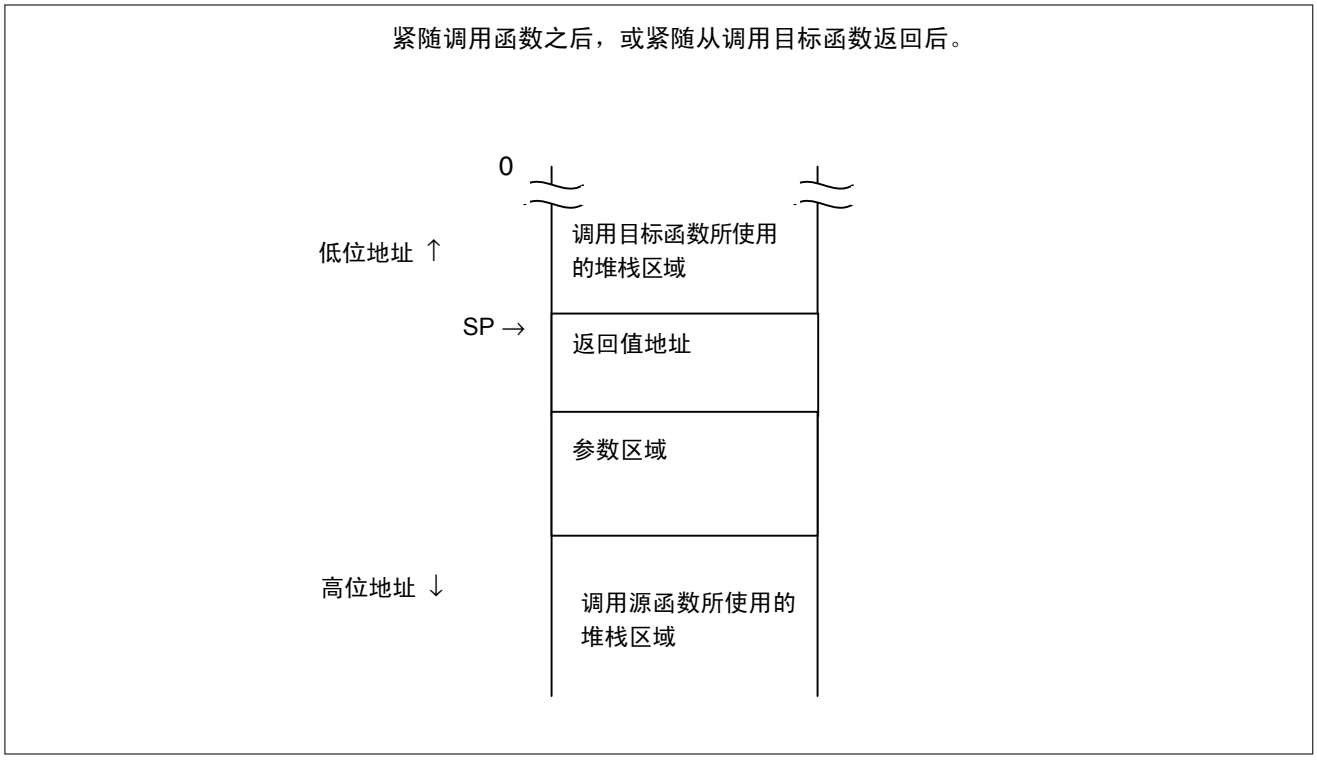


图 3.15 分配及释放堆栈帧

(c) 有关寄存器的规则

对于一些寄存器，C/C++ 编译程序无法保证值是否能在函数调用后保存。 表 3.38 显示保存寄存器内容的规则。

表 3.38 在 C 语言程序的函数调用后保存寄存器内容的规则

编号	类型	寄存器	有关汇编语言代码的注释
1	内容不受保证的寄存器	R0 至 R7、FR0 至 FR11[*1]、DR0 至 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] 及 RE[*3]	若在函数中使用的寄存器在程序调用函数时包含有效的数据，调用者必须在调用函数前将数据保存到堆栈上或寄存器。被调用函数可在不保存包含数据的情况下使用寄存器。 然而，当指定了 **fpscr=safe** 时，**FPSCR** 的内容将被保证。
2	内容受保证的寄存器	R8 至 R15、MACH、MACL、PR、FR12 至 FR15[*1] 和 DR12 至 DR14[*2]	函数中使用的寄存器数据在函数入口被保存到堆栈，然后在函数出口从堆栈恢复。 请注意若指定了 **macsave=0**，MACH 和 MACL 寄存器中的数据将不受保证。 当指定了 **gbr=auto** 时，**GBR** 的内容将被保证。

注意： 1. SH-2E、SH2A-FPU、SH-4 和 SH-4A 的单精度浮点寄存器。
2. SH2A-FPU、SH-4 和 SH-4A 的双精度浮点寄存器。
3. SH2-DSP、SH3-DSP 和 SH4AL-DSP 的 DSP 寄存器。

C 语言程序与汇编语言程序之间的调用应如下。

(i) 从 C 程序调用汇编语言函数

- 当从不同的模块调用汇编语言函数时，PR 寄存器的内容应在汇编语言函数的入口点被保存在堆栈上，并在出口点从堆栈恢复。
- 当使用寄存器 R8 至 R15 时，汇编语言函数内的 MACH 和 MACL 时，寄存器的内容应在使用前被保存在堆栈上，然后在使用后从堆栈恢复。
- 有关传递到汇编语言函数的参数的详细资料，请参考第 3.15.1 (3) 节“设定及参考参数和返回值”。

(ii) 从汇编程序调用 C 语言函数

- 若寄存器 R0 至 R7 中具有有效值，它们应在调用 C 函数前被保存到空的寄存器或堆栈上。
- 有关传递到汇编语言函数的返回值的详细资料，请参考第 3.15.1 (3) 节“设定及参考参数和返回值”。

图 3.16 的例子显示汇编语言函数 g 从 C 语言函数 f 被调用，而汇编语言函数 g 反过来调用了一个 C 语言函数 h。

C 语言函数 f

```
extern   void  g( );


f( )
{
   g( );
}
```

汇编语言函数 g

```
          .EXPORT      _g
          .IMPORT          _h
          .SECTION     P, CODE, ALIGN=2
_g :  STS.L            PR ,@-R15
MOV.L           R14,@-R15
MOV.L           R13,@-R15
:
          MOV.L            R2,@R15
          MOV.L            R1,@R15
MOV.L           L_h,R0
JSR                @R0
NOP
:
MOV.L           @R15+,R13
MOV.L           @R15+,R14
RTS
LDS.L           @R15+,PR

L_h : .DATA.L          _h
          .END
```

在外部定义的函数 g 的声明
在外部引用的函数 h 的声明

保存 PR 寄存器值
保存由函数 g 使用的寄存器

保存由函数 h 使用的寄存器

调用函数 h

恢复由函数 g 使用的寄存器

恢复 PR 寄存器值

C 语言函数 h

```
h( )
{
       :
       :
}
```

图 3.16 C 语言程序与汇编语言程序之间相互调用函数的实例

(3) 设定及参考参数和返回值

由 C/C++ 编译程序施加于设定及参考参数和返回值的规则，视参数或返回值的类型是否在函数声明中被明确声明而异。 在 C 语言中，将参数及返回值类型明确的函数声明称为原型函数声明。

下面，在首先讨论 C 语言中参数及返回值的一般规则后，将解释参数分配的区域与分配方法，及返回值位置的设定。

(a) C 语言程序中参数及返回值的一般规则

(i) 传递参数

函数应仅在参数值被复制到寄存器中或堆栈上为参数分配的区域后被调用。在控制被返回到调用源函数后，调用源函数从不参考分配到参数的区域，因此调用目标函数可在不直接影响调用源函数处理的情况下修改参数值。

(ii) 类型转换的规则

当传递参数，或返回值时，将执行自动类型转换。
表 3.39 显示类型转换的规则。

表 3.39　　类型转换的规则

类型转换	转换方法
已声明类型的参数类型转换	若参数已通过原型声明声明其类型，参数将转换到所声明的类型。
未声明类型的参数类型转换	若参数未通过原型声明声明其类型，参数将根据下列规则转换。 • char、unsigned char、short 和 unsigned short 类型的参数被转换为 int 类型。 • float 类型的参数被转换为 double 类型。 • 以上皆非的类型不会被转换。
返回值的类型转换	返回值将被转换到函数所返回的类型。

实例 1：通过原型声明声明其类型

```
long f();
long f()
{
        float x;
              :
              :
        return x;
}
```

返回值 x 根据原型声明被转换为 long 类型。

实例 2： 类似于实例 1，但未通过原型声明声明其类型

```
void p(int,...);

long f()
{
     char c;
           :
   p(1.0, c);
           :
}
```

由于相应参数的类型是 int，第一个参数被转换为 int 类型。
第二个参数没有相应的参数类型，因此被转换为 int 类型。

实例 3： 类似于实例 2，未通过原型声明声明其类型

当参数不是由原型声明声明其类型时，应调用目标及调用源函数指定相同的类型，以确保参数正确传递。 若类型不匹配，将无法保证正确的操作。

```
void f(x)
float x;
{
        :
        :
}

void main()
{
     float x;
     f(x);
}
```

在此例中，函数 f 的参数没有原型声明，因此当函数 f 由函数 main 调用时，参数 x 被转换为 double 类型。另一方面，参数由函数 f 声明为 float 类型，因此参数无法正确传递。 必须通过原型声明对参数声明其类型，或将函数 f 的参数声明更改为 double 类型。
参数可使用原型声明来正确声明其类型如下。

```
void f(float x)
{
        :
        :
}

void main()
{
```

```
        float x;
        f(x);
    }
```

(b) 在 C 语言程序中为参数分配区域

寄存器可被分配到参数，若无法分配，可将堆栈区域供参数使用。 用于分配到参数的区域在图 3.17 中显示，为参数分配存储器的一般规则在表 3.40 中显示。

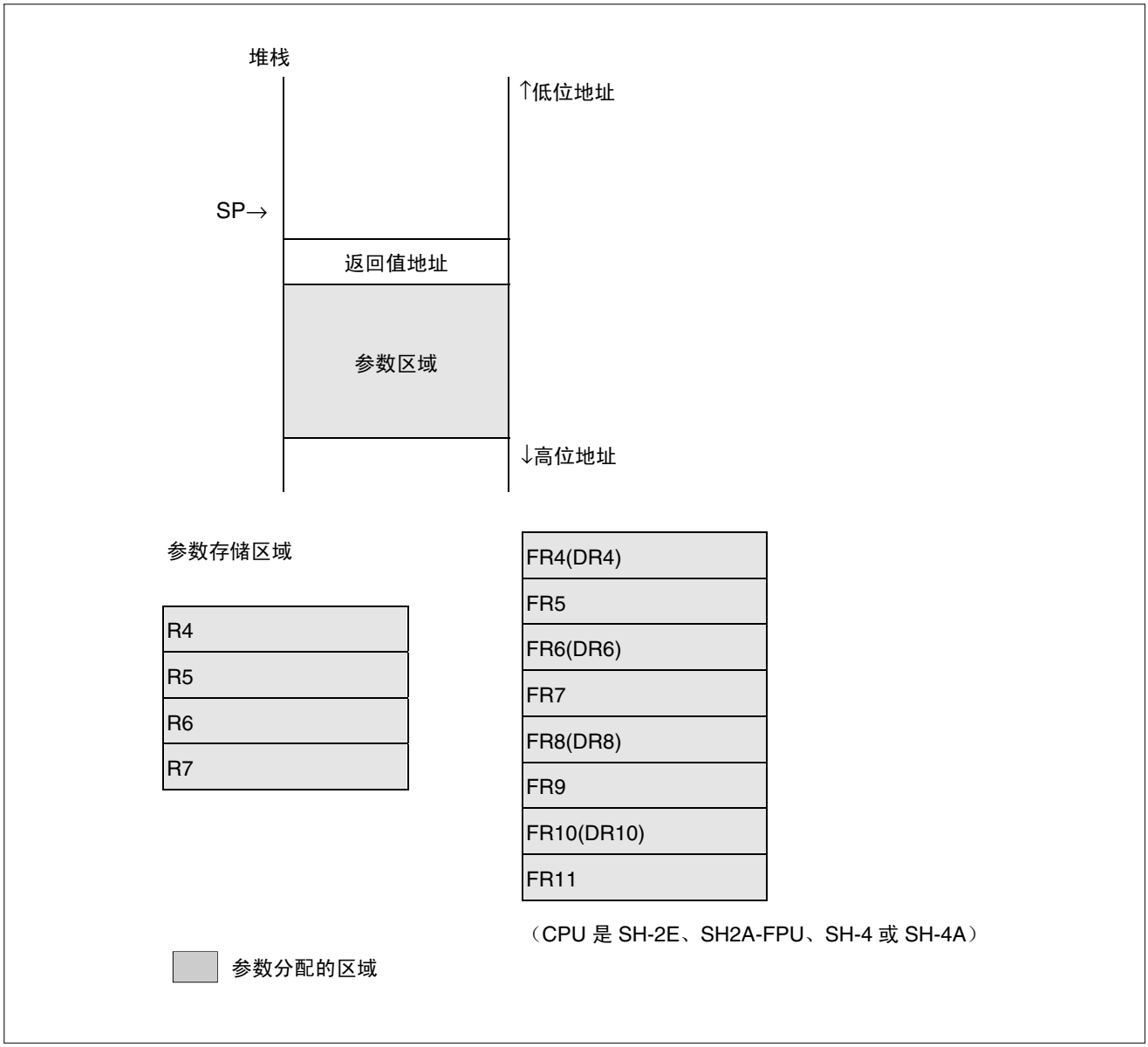


图 3.17 在 C 语言程序中分配到参数的区域

表 3.40　在 C 语言程序中将存储器分配到参数的一般规则

分配规则		
通过寄存器传递的参数		
用于参数存储的寄存器	可用的参数类型	通过堆栈传递的参数
R4 至 R7	char、unsigned char、bool、short、unsigned short、int、unsigned int、long、unsigned long、float（当 CPU 不是 SH-2E、SH2A-FPU、SH-4 或 SH-4A 时），pointer、数据成员的 pointer 和 reference	(1) 其类型不是寄存器传递目标类型的参数 (2) 通过原型声明被声明为具有变量参数的函数参数*3 (3) 当其他参数已分配到 R4 至 R7 时。 (4) 当其他参数已分配到 FR4 (DR4) 至 FR11 (DR10) 时。 (5) long long 类型及 unsigned long long 类型参数 (6) _ _fixed 类型、long _ _fixed 类型、_ _accum 类型及 long _ _accum 类型参数
FR4 至 FR11*1	对于 SH-2E • 参数是 float 类型。 • 参数是 double 类型，同时指定了 double=float。 对于 SH2A-FPU、SH-4 或 SH-4A • 参数类型是 float 类型，同时未指定 fpu=double。 • 参数类型是 double 类型，同时指定了 fpu=single。	
DR4 至 DR10*2	对于 SH2A-FPU、SH-4 或 SH-4A • 参数类型是 double 类型，同时未指定 fpu=single。 • 参数类型是 float 类型，同时指定了 fpu=double。	

注意：
1. SH-2E、SH2A-FPU、SH-4 和 SH-4A 的单精度浮点寄存器。
2. SH2A-FPU、SH-4 和 SH-4A 的双精度浮点寄存器。
3. 若函数由原型声明声明为具有变量参数的函数，在声明中没有相应类型的参数以及紧接在前的参数将被分配到堆栈。

实例：

```
int f2(int, int, int, int,...);
f2(a, b, c, x, y, z)
{
            :
}
```

直到第四个参数，通常会分配寄存器空间，不过这里为 x、y 及 z 分配了堆栈区域。

(i) 为参数存储分配寄存器

寄存器按照源程序中的声明顺序来分配给参数存储，从具有最小编号的寄存器开始。实例 1 中显示参数存储的寄存器分配实例。

(ii) 参数的堆栈区域分配

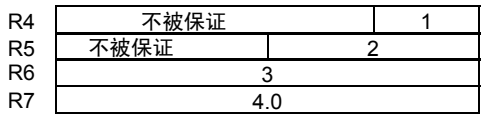
堆栈区域按照源程序声明的顺序分配给参数存储，从具有最低地址的堆栈区域开始。实例 2 到 8 显示参数存储的堆栈区域分配实例。

[有关 structure 和 shared 类型参数的重要信息]

当准备 structure 和 shared 类型的参数时，这些类型始终以四字节边界对齐，同时始终为它们使用四字节倍数存储器区域。这是因为 SuperH 微型计算机的堆栈指针以四个字节为更改单位。

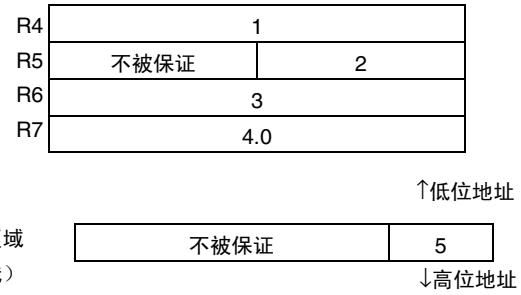
实例 1： R4 到 R7 的寄存器按照声明的顺序被分配到具有寄存器类型的参数。

```
int f(char,short,int,float);
    :
f(1,2,3,4.0);
    :
```



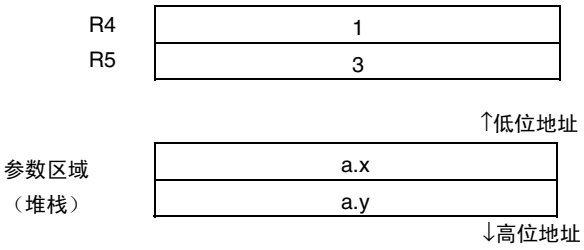
实例 2： 无法分配寄存器的参数会被分配堆栈区域。 当为 (unsigned) char 或 (unsigned) short 类型的参数分配堆栈区域时，它们被扩展为四字节以便分配。

```
int f(int,short,long,float,char);
    :
f(1,2,3,4.0,5);
    :
```



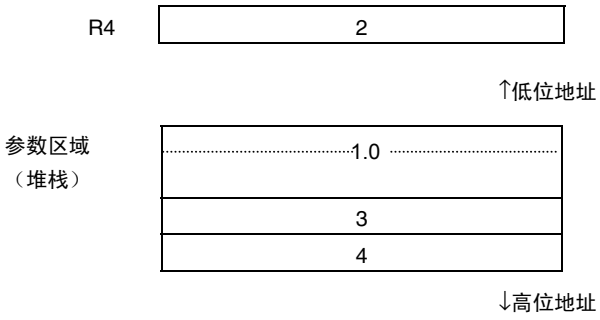
实例 3： 无法分配到寄存器的参数类型会被分配到堆栈区域。

```
struct s{int x,y;}a;
int f(int,struct s,int);
    :
f(1,a,3);
    :
```



实例 4： 当使用原型声明来声明具有变量参数的函数时，没有声明类型的参数，及紧接在前的参数，按照声明的顺序被分配到堆栈上。

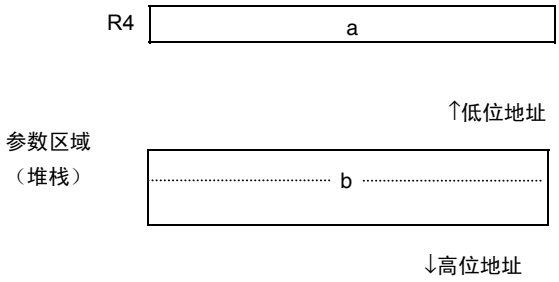
```
int f(double, int, int...)
    :
f(1.0, 2, 3, 4)
    :
```



实例 5： 没有原型声明的情形

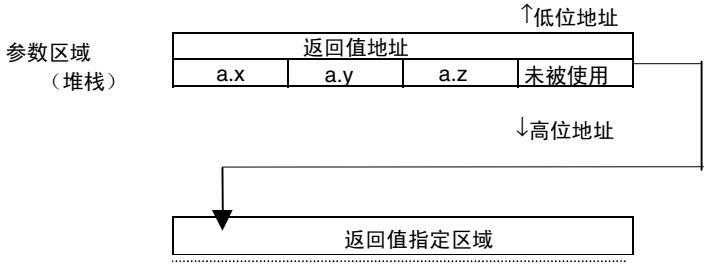
→ char 类型被扩展为 int 类型，float 类型被扩展为 double 类型，以便分配。

```
int f ( )
char  a ;
float  b;

     f (a ,b)
```



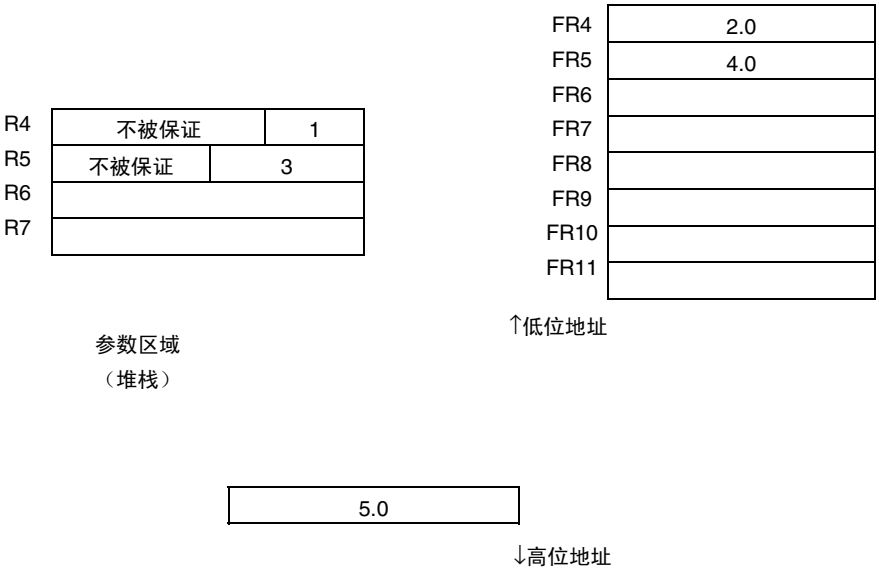
实例 6： 当由函数返回的类型超过四字节或是一个类时，会在紧挨的参数区域前设定返回值地址。同时，当类大小不是四字节倍数时，会发生空区域。

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



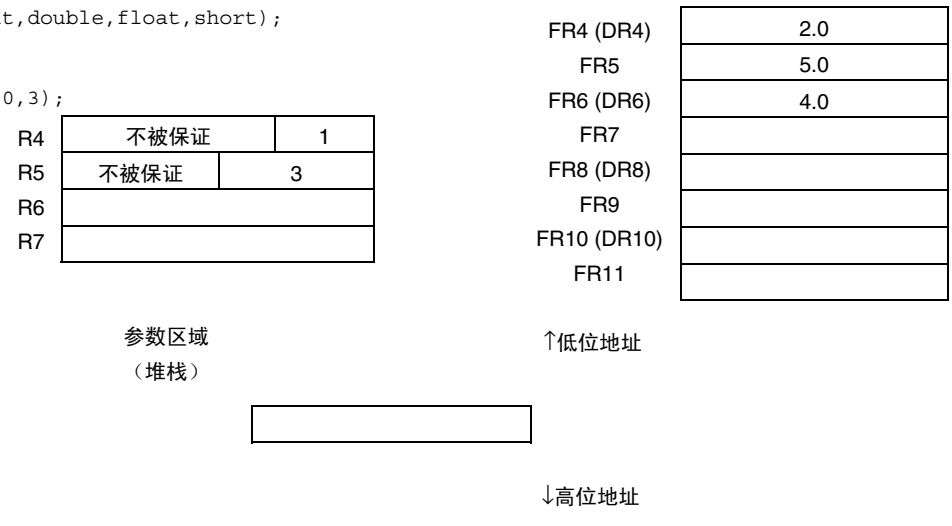
实例 7： 当 CPU 是 SH-2E 时，float 类型参数被分配到 FPU 寄存器。

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



实例 8： 当 CPU 是 SH2A-FPU、SH-4、SH-4A，且未指定任何 -fpu 选项时，float/double 类型参数被分配到 FPU 寄存器。

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



(c) C 语言程序中供设定返回值的位置

视函数的返回值类型而定，返回值会被放置在寄存器中或堆栈上。 表 3.41 中描述了返回值类型和返回位置的关系。

当函数的返回值被放置在堆栈上时，返回值将设定在由返回值地址所指向的区域。对于调用源函数来说，不但保护了参数区域，另外也保护了返回值的区域，同时在将地址设定为返回值地址后，函数会被调用（cf. 图 3.18）。 当函数返回值是 void 类型时，没有设定任何返回值。

表 3.41　　C 语言程序中的返回值类型和位置

编号	返回值类型	返回值存储区域
1	(signed) char、unsigned char、(signed) short、unsigned short、(signed) int、unsigned int、long、unsigned long、float、pointer、bool reference、数据成员的 pointer	R0: 32 位 (signed) char 或 unsigned char 高三字节的内容，及 (signed) short 或 unsigned short 高二字节的内容不被保证。 然而，当指定了 **rtnext** 选项时，(signed) char 或 (signed) short 类型执行了 sign 扩展，unsigned char 或 unsigned short 类型执行了 zero 扩展。 FR0: 32 位 (1) 对于 SH-2E • 返回值是 float 类型。 • 返回值是 double 类型，同时指定了 **double=float**。 (2) 对于 SH2A-FPU、SH-4 或 SH-4A • 返回值是 float 类型，同时未指定 **fpu=double**。 • 返回值是 floating-point 类型，同时指定了 **fpu=single**。
2	double、long double structure、union、class、函数成员的 pointer	返回值设定区域（存储器） DR0: 64 位 对于 SH2A-FPU、SH-4 或 SH-4A • 返回值是 double 类型，同时未指定 **fpu=single**。 • 返回值是 floating-point 类型，同时指定了 **fpu=double**。
3	(signed) long long 和 unsigned long long	返回值设定区域（存储器）
4	_ _fixed、long _ _fixed、_ _accum 和 long _ _accum	返回值设定区域（存储器）

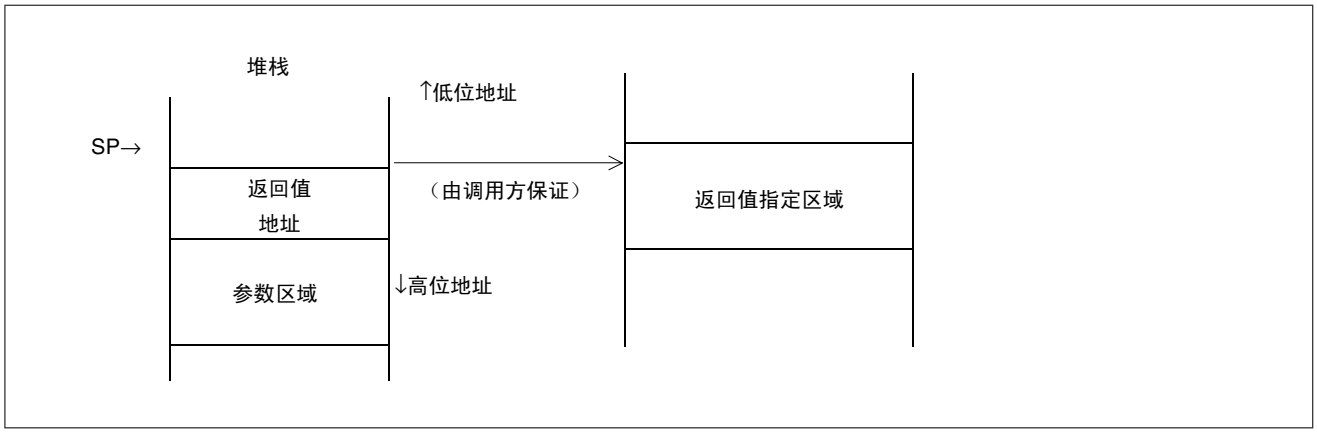


图 3.18 在 C 程序中使用堆栈时的返回值区域

3.15.2 和连接编辑程序一起使用

(1) ROM 支持函数

当把一个装入模块写入到 ROM 时，初始化数据区域也被写入到 ROM。 然而，实际的数据运算必须在 RAM 中执行，因此初始化数据区域必须在启动时从 ROM 复制到 RAM。 通过使用连接编辑程序的 ROM 支持函数，这个过程获得简化。 为使用 ROM 支持函数，必须在连接时指定“ROM D=R”选项（D 是 ROM 中初始化数据区域的段名，R 是 RAM 中初始化数据区域的段名）。
ROM 支持函数执行下列运算。

(a) 必须在 RAM 中确保和 ROM 中初始化数据区域相同大小的区域。 图 3.19 阐述了从 ROM 将初始化数据区域复制到 RAM 的步骤。

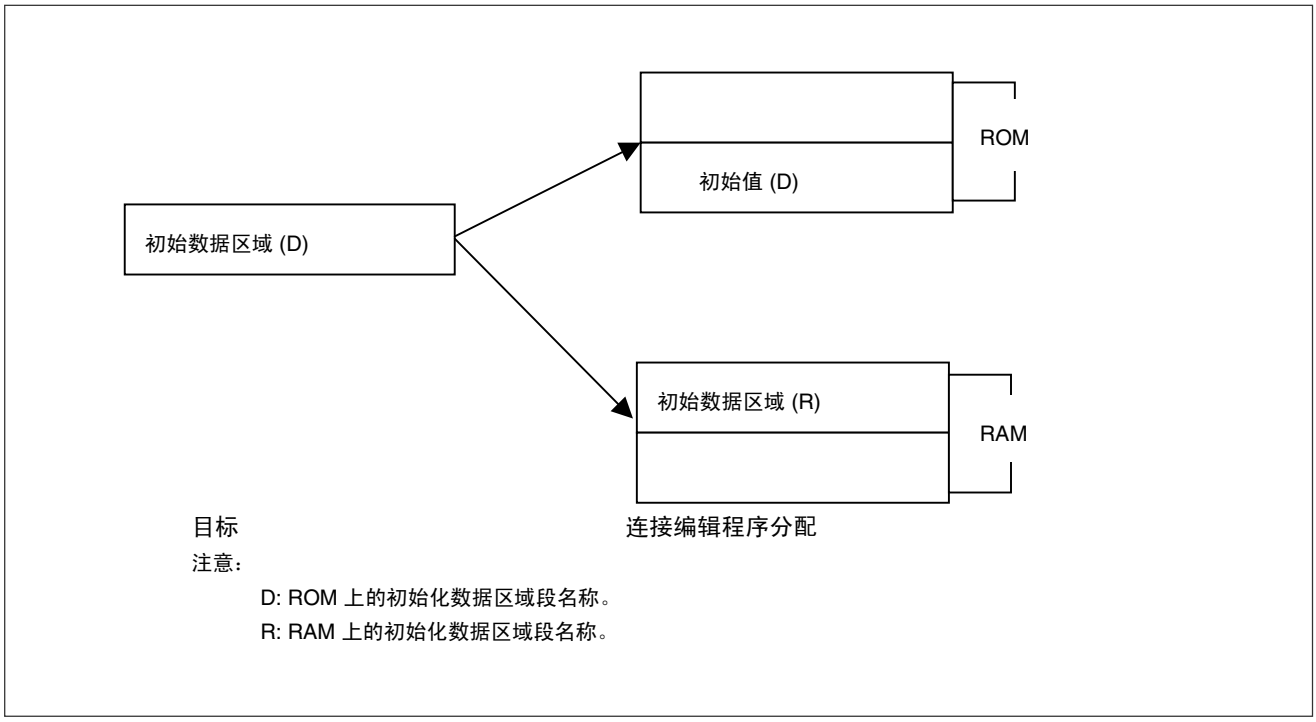


图 3.19 使用 ROM 支持函数的存储器分配

(b) 地址精度会自动执行，以便使用 RAM 中的地址参考在初始化数据区域中声明的符号。

用户必须在启动例程中包含将 ROM 中数据复制到 RAM 的处理。 有关这项操作的实例，请参考第 2.2.4 节“初始化单位的创建”。
要获取 ROM 支持函数的更多信息，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)。
此函数在版本 4 及以上的 H 系列连接编辑程序中支持。

(2) 有关连接的重要信息

表 3.42 描述在连接由 C/C++ 编译程序生成的可再定位目标文件时，对错误信息输出的处理步骤。

表 3.42　在连接时对错误信息的回应

编号	错误信息	要检查的区域	对策
1	在连接时输出了错误编号 L1100(314)*，cannot find section（找不到段）。	是否有在连接编辑程序的起始选项中以大写指定编译程序输出段的名称。	确保指定正确的段名称。
2	在连接时输出了错误编号 L1160(105)*，undefined external symbol（未定义的外部符号）。	当存在 C/C++ 程序与汇编语言程序之间相互参考变量的情形时，检查汇编语言程序中的名称是否在前面标有下划线。	必须在参考中使用正确的变量名称。
		检查是否未在 C/C++ 程序中使用 C 程序库函数。	在连接时，标准程序库应被指定为输入程序库。
		检查未定义的符号名称是否未以下划线开始。 （由运行时例程在标准程序库中使用）	
		检查是否为 C 程序库函数使用了标准 I/O 程序库。	为连接创建低层界面例程。
3	无法进行 C/C++ 源代码级调试。	检查是否已在编译及连接时指定了 debug 选项。	确保在编译及连接时指定 debug 选项。 当在连接期间指定 sdebug 选项时，确保将调试信息文件加载到调试程序内。
		检查所使用的是否是版本 5.3 或以上的连接编辑程序。	应使用版本 5.3 或以上的连接编辑程序。
4	在连接时输出了错误编号 L2330(108)*，relocation size overflow（再定位大小溢出）。	检查在 GBR 基址变量指定中指定变量的偏移是否在限制内。	对于超出限制的变量，删除任何 #pragma gbr_base/gbr_base1 声明。
5	在连接时输出了错误编号 L2300(104)*，duplicate symbol（重复的符号）。	检查是否在多个文件中具有名称相同的变量或函数的外部定义。	更改名称，或使用静态指定。
		检查多个文件中所包含标头文件中的变量或函数的外部定义。 （对使用 #pragma inline/inline_asm 指定的函数执行类似操作）	使用静态指定

注意：*括号前的是版本 7 或以上的连接程序的错误编号，括号中的则是属于版本 6 或以下的错误编号。

3.15.3 和模拟程序调试程序一起使用

当使用模拟程序调试程序来执行装入模块时，可能会生成“MEMORY ACCESS ERROR”（存储器存取错误）。 为了安全起见，应使用下列方法之一以避免这项错误。

(a) 即使在使用模拟程序调试程序时，也使用和实际 CPU 中相同的存储器映像（一段中字节总数应始终为四的倍数）。
(b) 在连接时，在所有段后（除了 P 段）连接使用下列汇编语言程序所创建的虚设的段。

汇编语言程序：

```
.SECTION DM,DUMMY,ALIGN=1
.RES.B      3
.END
```

连接实例

- 当使用命令选项时：

  ```
  -START=P,C,DM/0400,B,DM,D,DM/01000000
  ```

- 当使用子命令文件时：

  ```
  START P,C,DM/0400,B,DM,D,DM/01000000
  ```

下面是有关使用模拟程序调试程序进行源代码级调试的重要信息。

(a) 应使用版本 6.0 或以上的连接编辑程序。
(b) 在编译时，应使用 –debug 选项，而在连接时，应指定 sdebug 选项。
(c) 在某些情况下，函数的局部信号无法在该函数内被参考。
(d) 当在一行源代码中包含多个语句时，只能显示一个语句。
(e) 通过优化消除的源代码行将无法被调试。
(f) 当调试的结果造成行交换及其他更改时，程序执行及反汇编显示的顺序可能和源列表中的顺序有所不同。

实例：
C 语言程序

```
12 for (i=0; i<6; i++)
13 {
14      j = i+1;
15     j++;
16 }
17 j++;
```

模拟程序/调试程序的反汇编显示

```
14 j = i+1;
12 for (i=0; i<6; i++)
17       j++;
```

(g) 在 for 或 while 语句中，反汇编显示可能被显示两次，在循环语句的入口和出口点。

(1) Profile 函数

(a) 使用 Profile 函数

(i) 堆栈信息文件

profile 函数使 HEW 可以读取由优化连接程序（版本 7.0 或以上）输出的 堆栈信息文件（扩展名：“.SNI”） 。 所有这些文件包含和在相应的源文件中调用静态函数有关的信息。 读取堆栈信息文件使 HEW 可在不需执行用户应用程序的情况下（即在测量配置文件数据前），显示与调用函数相关的信息。（然而，此功能在选中了 [仅显示已执行的函数] (Show Only Executed Function) 时不可用。）

当 HEW 不读取任何堆栈信息文件时，有关在测量期间已执行函数的数据将由 profile 函数显示。

无论是否读取，堆栈信息文件都可通过在装入程序对话框中开启或关闭 [加载堆栈信息文件（SNI 文件）] (Load Stack Information file (SNI file)) 复选框来指定。

要使连接程序创建堆栈信息文件，需从“类别” （Category） 列表框选择“其他” (Other)， 然后在 HEW 连接程序选项指定的标准工具链对话框之“连接/程序库”(Link/Library) 面板中选中“堆栈信息输出”(Stack information output) 框。然后创建信息文件。

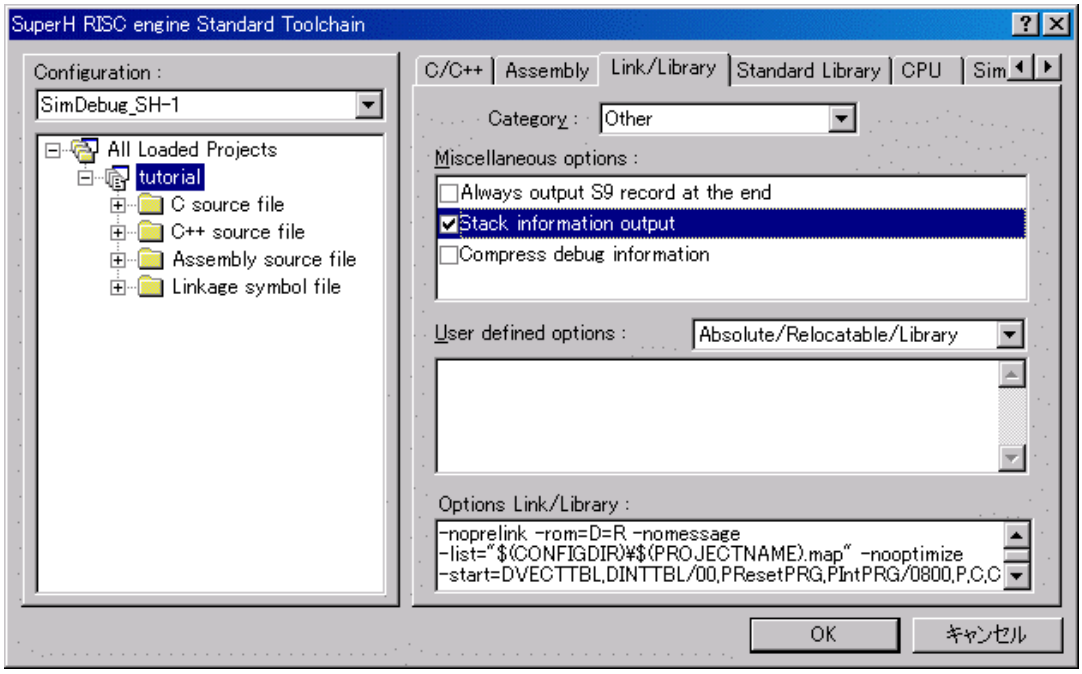


图 3.20 类别 (Category):[其他 (Other)] 对话框

(ii) 配置文件信息文件

若要创建配置文件信息文件，需在测量应用程序的配置文件数据后，从配置文件窗口的弹出式菜单选择“输出配置文件信息文件…” (Output Profile Information Files…) 菜单选项，并指定文件名称。

这个文件包含函数被调用及全局变量被存取的次数等信息。 优化连接程序（版本 7.0 或以上）能够读取配置文件信息文件，及对应程序的实际运作状态优化函数与变量的分配。

若要将剖析器信息文件输入到连接程序，需从“类别” (Category) 列表框选择“优化” (Optimize)， 并在标准工具链对话框之“连接/程序库” (Link/Library) 面板中选中“包含配置文件：” (Include Profile:) 框，然后指定配置文件信息文件的名称。

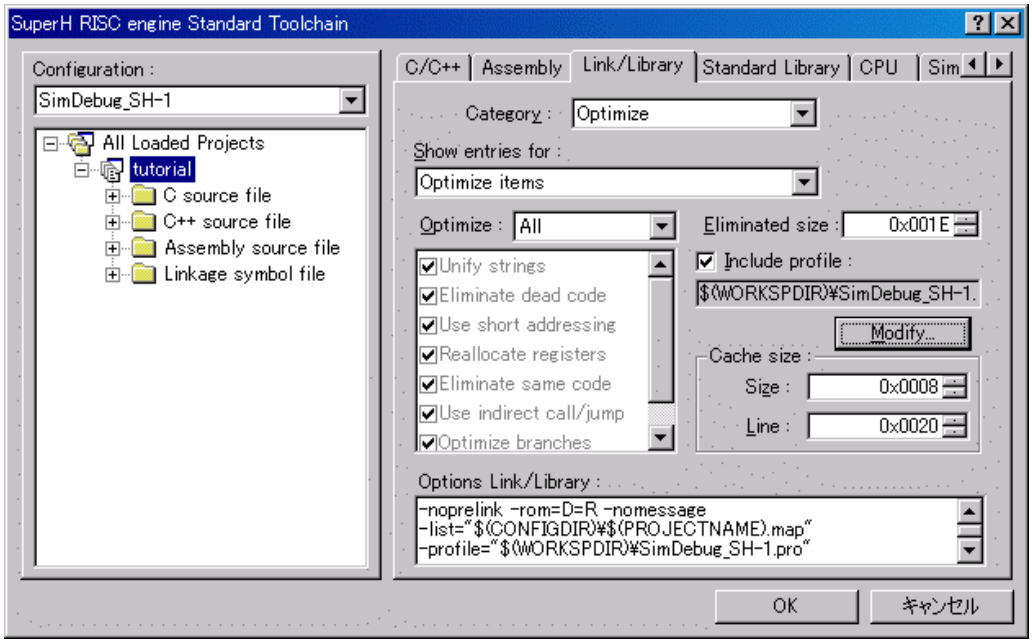


图 3.21 类别 (Category):[优化 (Optimize)] 对话框

(iii) 配置文件窗口

选择 [视图 (View) -> 性能 (Performance) -> 配置文件 (Profile)] 以打开配置文件 (Profile) 窗口。 这个菜单项目会在加载了装入模块时显示。

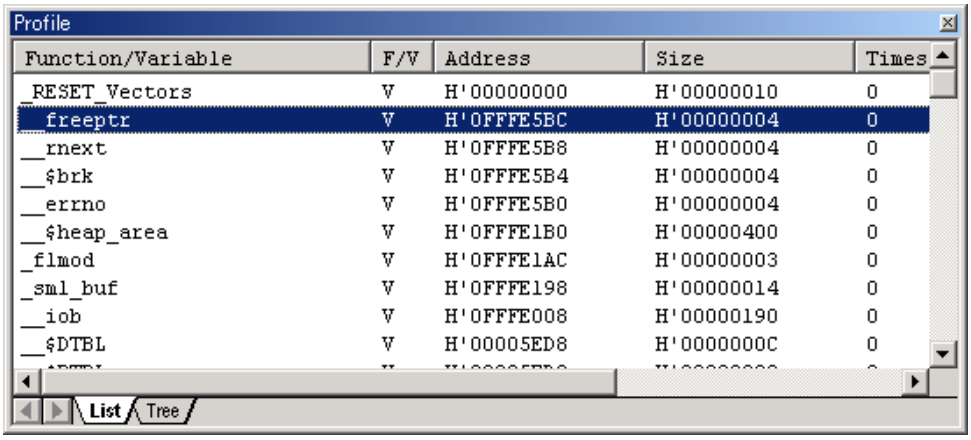


图 3.22 配置文件 (Profile) 窗口

(iv) 配置文件窗口菜单

从配置文件窗口的弹出式菜单选择“启用” (Enable)。（将会选中菜单上的项目。）

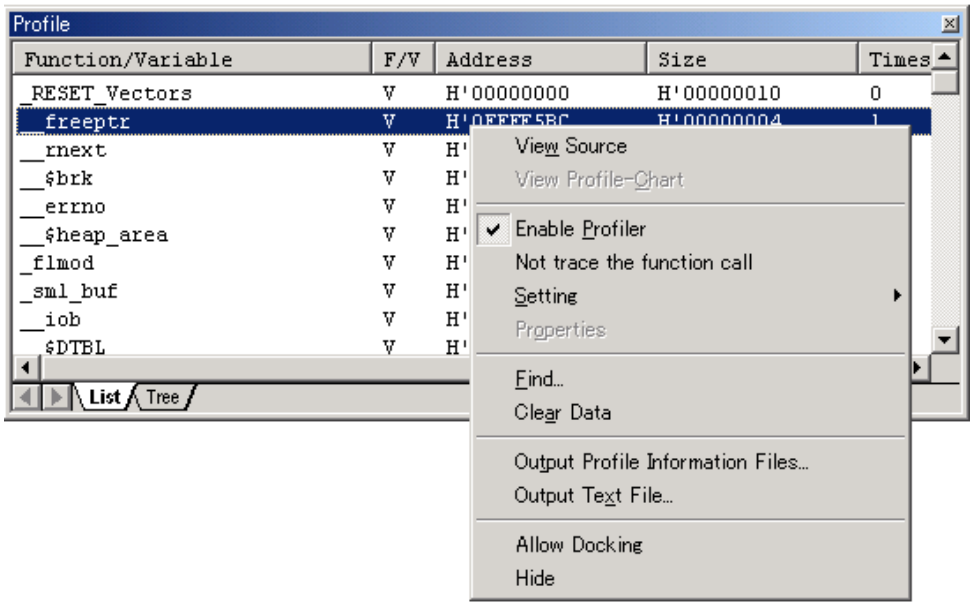


图 3.23 配置文件窗口菜单（启用剖析器）

(v) 设定带条件的断点使配置文件的测量停止。（配置文件测量可在不需设定条件的情况下手动停止。）

(vi) 若满足了在上述 (e) 设定的停止条件，或执行被手动停止或因其他原因停止，测量结果将在配置文件窗口中显示。

(vii) 若要创建配置文件信息文件，需从弹出式菜单选择“保存配置文件信息文件…” (Save Profile Information Files…)。

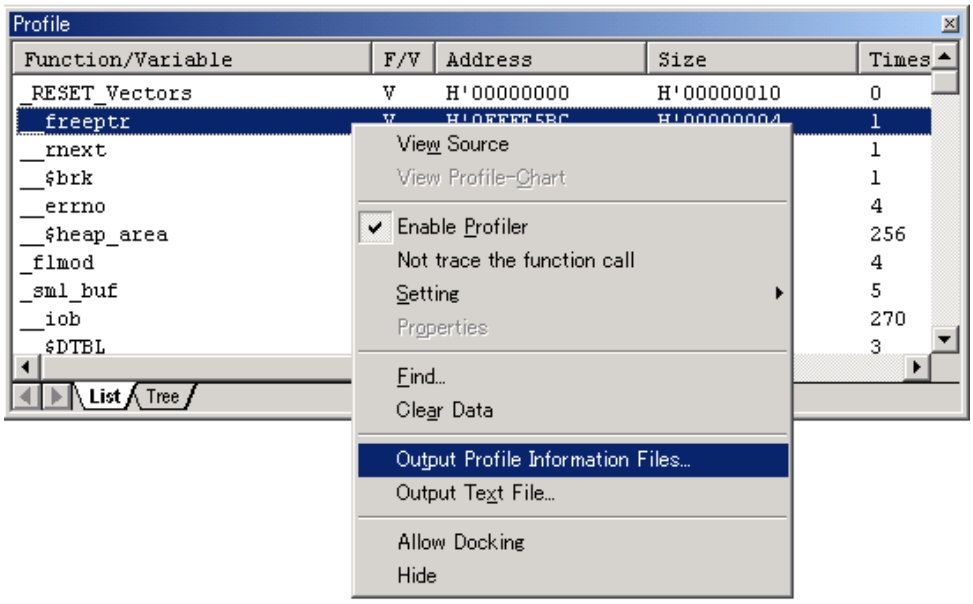


图 3.24 配置文件窗口菜单（保存配置文件信息文件…）

(b) 注意：

(i) 应用程序由 profile 函数测量的执行周期数包含错误差数。 profile 函数仅允许测量函数在应用程序的整体执行中所占用的执行时间比例。 使用性能分析 (Performance Analysis) 功能来准确测量执行周期数。

(ii) 当对装入模块上没有调试信息的配置文件信息进行测量时，相应函数的名称可能无法被显示。

(iii) 堆栈信息文件（扩展名：“.SNI”）必须和装入模块文件处于相同的目录（扩展名：“.ABS”）。

(iv) 测量结果无法被存储。

(v) 测量结果无法被编辑。

(c) Profile 函数的概述

Profile 函数以函数的执行计数来测量应用程序的执行性能。Profile 函数使您能够识别造成性能降级的程序部分以及降级的原因。

(i) 配置文件窗口

配置文件窗口具有两个标签，一个“列表” (List) 标签及“树” (Tree) 标签。

- 列表标签

  这个标签列出函数和全局变量，并显示各个函数及变量的配置文件数据。

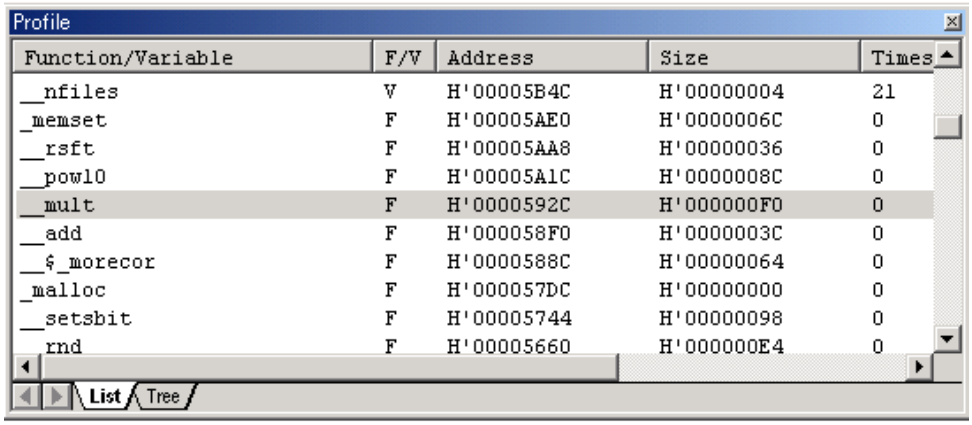


图 3.25 列表标签

- 树标签

  这个标签以树形图以及包含函数被调用时的值的配置文件数据来显示函数调用的关系。

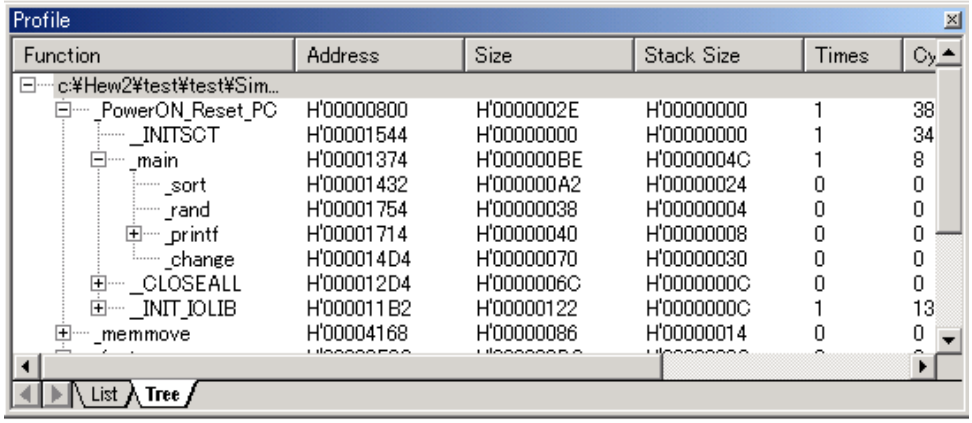


图 3.26 配置文件树窗口

- 配置文件图表窗口

配置文件图表 (Profile-Chart) 窗口显示特定函数的调用关系。这个窗口的中间部分显示指定的函数，而该函数的调用者在左边，函数的被调用者则在右边。函数对调用目标函数的调用或被调用源函数调用的次数也在这个窗口中显示。

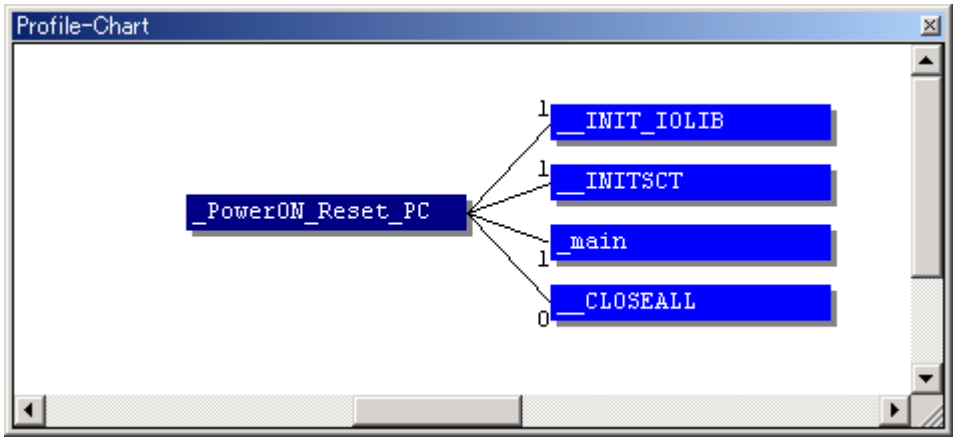


图 3.27 配置文件图表窗口

(ii) 所显示数据的类型和目的

- 地址

  函数： 显示函数的地址（全局变量）。
  用途： 您可以看到函数在存储器中被分配的位置。 根据地址来分类函数及全局变量列表，使用户能够查看项目在存储器空间内的分配方式。
  注意： 分类显示仅在“列表” (List) 标签上提供。

- 大小

  函数： 显示函数的大小（全局变量）。
  用途： 根据大小顺序分类使更容易查找被频繁调用的小型函数。 将这类函数设定为内联将可缩减函数调用的内务操作。
  若您使用结合了高速缓存存储器的微型计算机，更多的高速缓存存储器需要在您执行较大的函数时被更新。 这项信息使您能够检查那些可能造成高速缓存遗漏的函数是否被频繁调用。
  注意： 分类显示仅在“列表” (List) 标签上提供。

- 堆栈大小

  函数： 显示函数所使用的堆栈大小。
  用途： 当函数调用具有深层嵌套时，追踪函数调用的路线，并获取该路线上所有函数的总堆栈大小，以预计所使用的堆栈数量。
  注意： 这将在“树” (Tree) 标签中显示。

函数使用的堆栈大小在堆栈信息文件中设定。 若堆栈信息文件没有被读取，所有堆栈大小将显示为 0。若在此情况下，您在堆栈分析工具（H 系列 Call Walker）中包含了输出配置文件信息文件（扩展名：“.PRO”），将无法显示正确的值。

- 次数

  函数： 显示各个函数被调用或变量被存取的次数。
  用途： 按调用或存取的次数来分类，将能更容易识别被频繁调用的函数及被频繁存取的全局变量。
  注意： 分类显示仅在“列表” (List) 标签上提供。

- 其他

  也提供多种特定目标的数据测量。 有关详细信息，请参考您所使用的目标平台的模拟程序或仿真程序手册。

(iii) 显示设定

若您从弹出式菜单选择了“设定…”(Setting…)，则会显示设定列表。 通过自定义这个列表的显示，将可轻松检测任何有问题的部分。

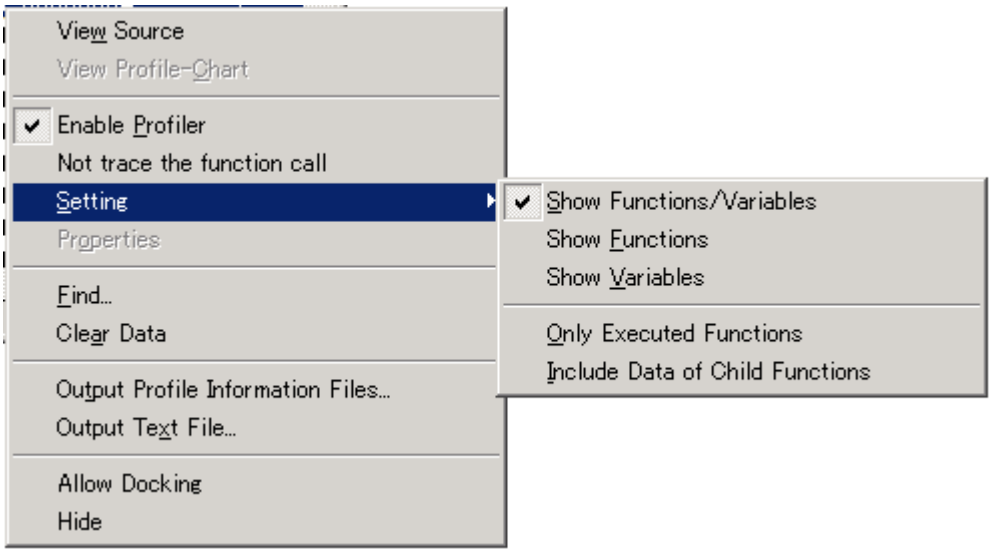


图 3.28 配置文件窗口弹出式菜单

- 显示函数/变量

  功能： 在窗口中显示函数及变量的信息。

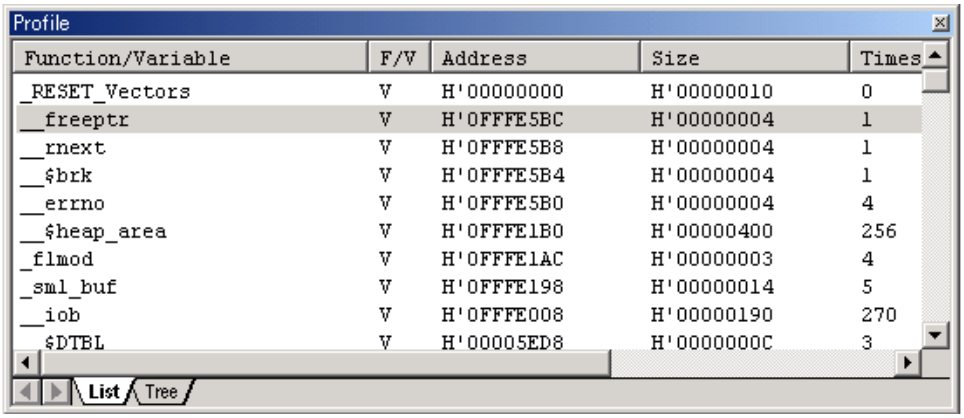


图 3.29 配置文件窗口（显示函数/变量）

- 显示函数

  功能： 在窗口中显示函数的信息。

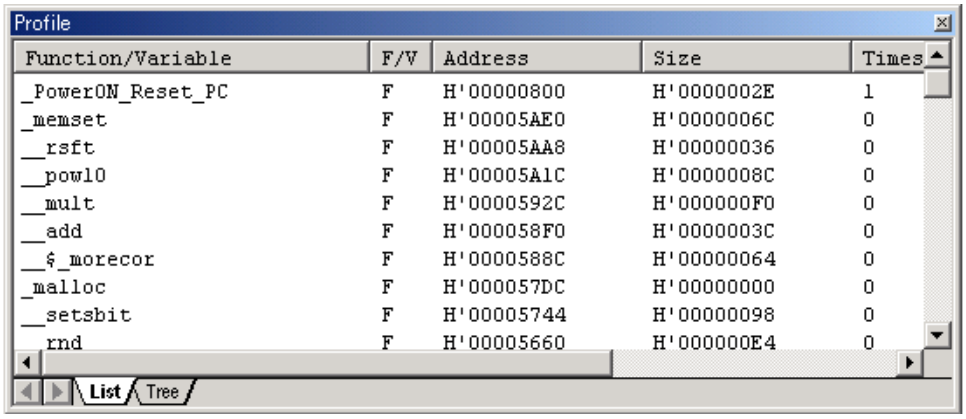


图 3.30 配置文件窗口（显示函数）

- 显示变量

  功能： 在窗口中显示变量的信息。

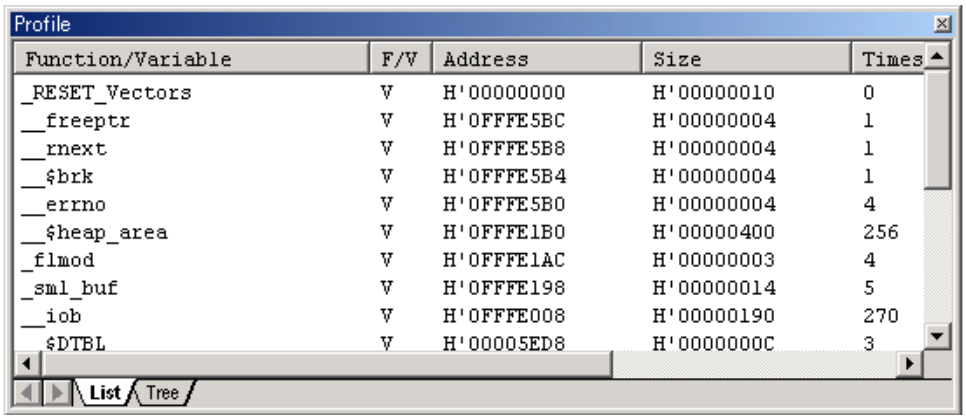


图 3.31 配置文件窗口（显示变量）

- 仅限已执行的函数 (Only executed function(s)) 复选框

  功能： 指定是否在配置文件的数据测量期间仅显示已执行的函数（或已存取的变量），或者同时也显示未被执行的函数（或未被存取的变量）。

  用途： 通过仅显示在数据测量期间已执行的函数，而不显示其他函数，使显示更简单。

  若显示了所有函数，将可确定函数在所有函数之数据测量期间内的执行率，同时也可以确定应用程序在数据测量期间的执行有多满意。

  注意： 若堆栈信息文件未被读取，即使有这项指定，仍然只会显示已被执行的函数（或已被存取的变量）。

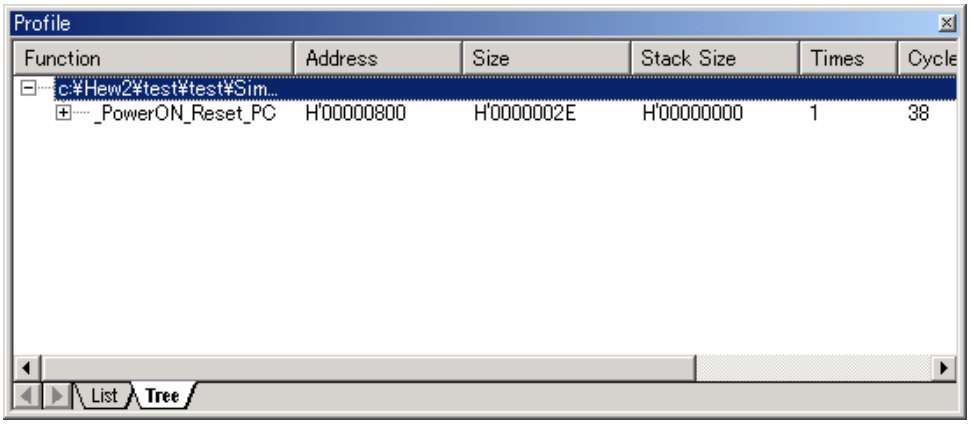


图 3.32 显示当开启了仅限已执行的函数 (Only Executed Function(s)) 复选框时的实例

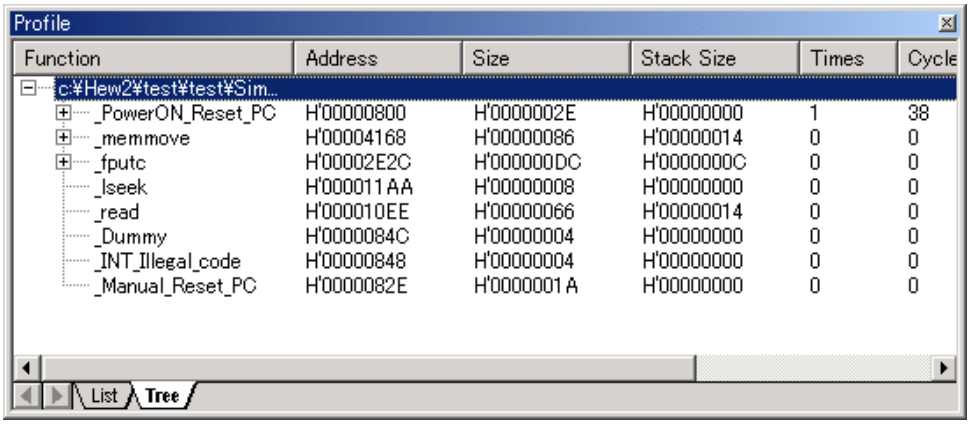


图 3.33 显示当关闭了仅限已执行的函数 (Only Executed Function(s)) 复选框时的实例

- 包含子函数的数据 (Include data of child function(s)) 复选框

  功能： 指定是否在将显示的测量数据中包含子函数的数据。

  用途： 例如，当您使用 SH1 模拟程序测量周期时，选中这个复选框将显示从该函数被调用到其返回的周期数（同时会添加由该函数调用子函数的周期数）。 这在您需要确定各个模块的执行时间时很有用。

  注意： 在地址 (Address)、大小 (Size)、堆栈大小 (Stack Size) 和次数 (Times) 等列中显示的值不更改。

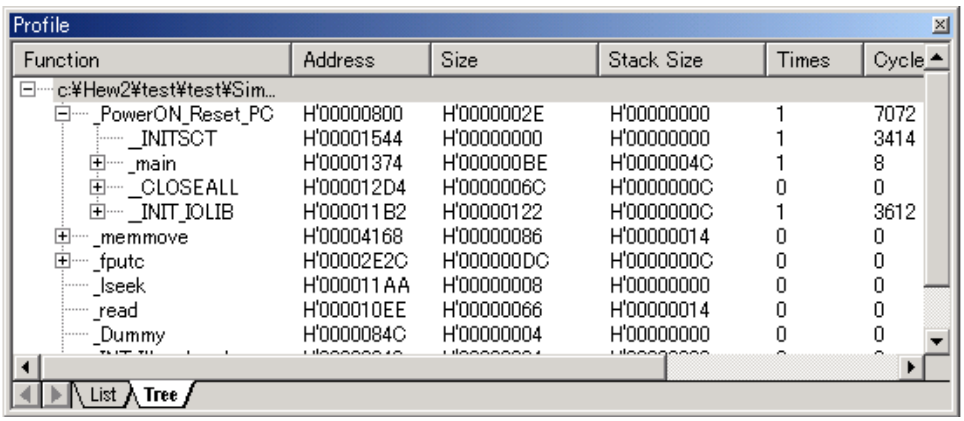


图 3.34 显示当开启了包含子函数的数据 (Include Data Of Child Function(s)) 复选框时的实例

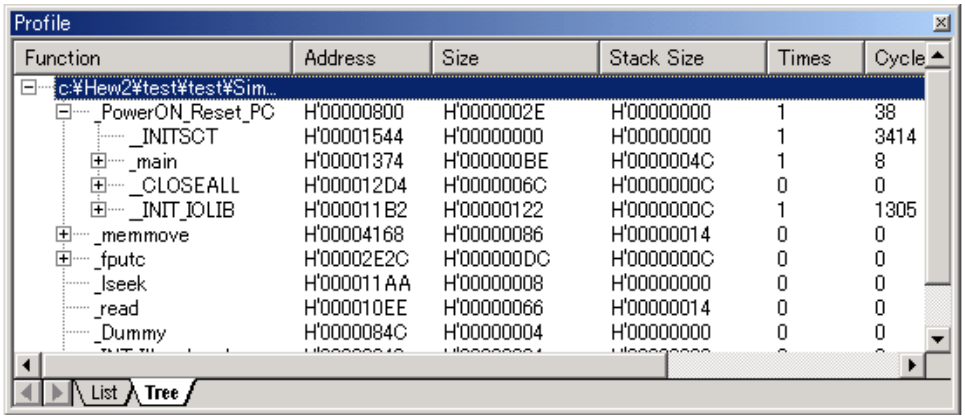


图 3.35 显示当关闭了包含子函数的数据 (Include Data Of Child Function(s)) 复选框时的实例

(iv) 列设定

在配置文件窗口中所显示的列上单击鼠标右键以显示弹出式菜单。

功能： 选择要在窗口中显示的信息。

用途： 仅显示所需的信息，以便使窗口的显示更简单。

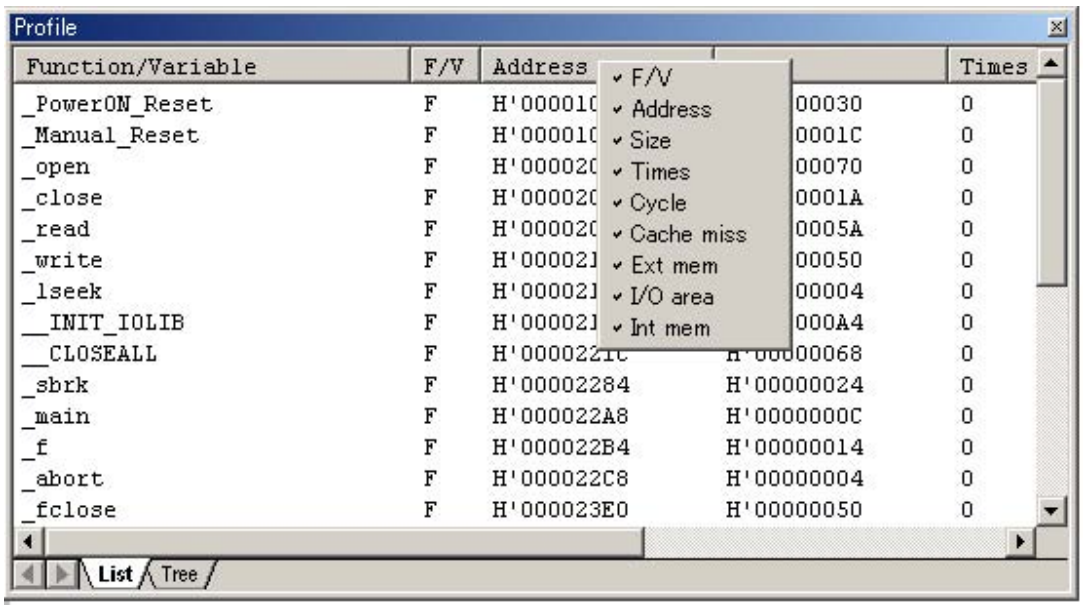


图 3.36 配置文件窗口弹出式窗口

3.16 更改结构的对齐数

描述：

使用 pack 选项 (-pack={1 | 4}) 或 #pragma pack 扩展 (pack 1 | pack 4 | unpack) 来更改结构的对齐数。
若您同时指定了选项和扩展，扩展的指定将优先。
structure、unit 和 class 的对齐数将和成员的最大对齐数相同。
默认值是 pack=4。
下面显示指定和对齐数。

表 3.43 当指定了 pack 选项时，structure、union 及 class 的对齐数

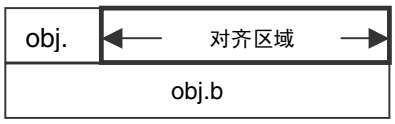
指定	pack=1	pack=4	没有指定
[unsigned]char	1	1	1
[unsigned]short, __fixed	1	2	2
[unsigned]int、[unsigned]long、[unsigned]long long、long、__fixed、__accum、long、__accum、floating-point、pointer	1	4	4
对齐数是 1 的 structure、union 和 class	1	1	1
对齐数是 2 的 structure、union 和 class	1	2	2
对齐数是 4 的 structure、union 和 class	1	4	4

分配 structure 数据

(1) 当您分配 structure 成员时，可能会在成员之间插入空白区域，因为各个成员是由该成员之数据类型的对齐数对齐。

实例：

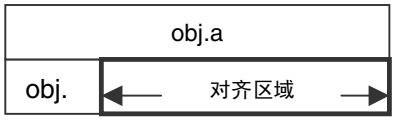
```
struct {
char   a;
int    b;
} obj;
```



(2) 若 structure 的对齐数是 4 字节，而最后一个成员在第一、第二或第三个字节结束，下一个字节也被处理为 structure 类型的区域。

实例：

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

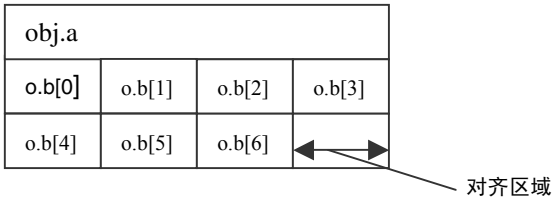


分配 unit 数据

(1) 若 unit 的对齐数是 4 字节，且成员的最大大小不是 4（字节）的倍数，包含 4 之倍数的剩余字节的区域将被处理为 union 类型数据，直到大小达到 4 的倍数。

实例：

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



更改对齐数

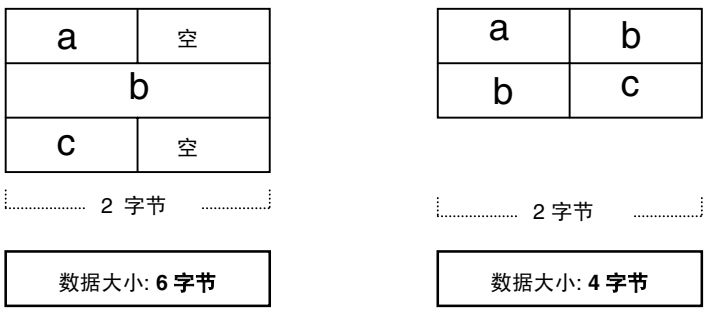
当指定了 #pragma pack 1 时，可能不会插入用于对齐的空白区域，因为一字节以外的其他数据也可以被分配到奇数地址。这将会缩减数据大小。

C/C++ 程序，

```
struct S1{
    char a;
    short b;
    char c;
}
```

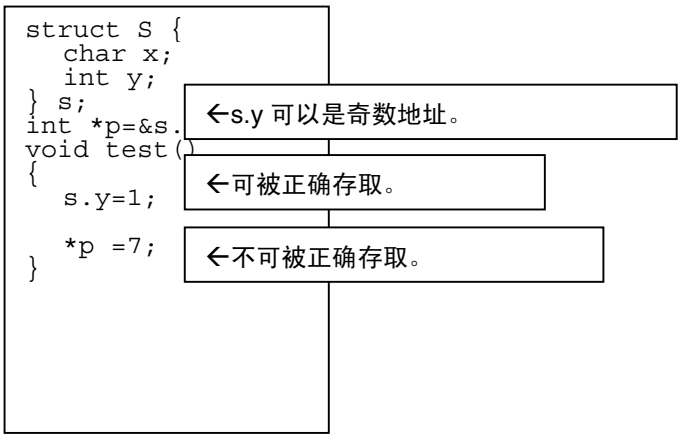
```
#pragma pack 1
struct S1{
    char  a;
    short b;
    char  c;
}
```

数据图像



注意： 若对齐数是 1，各个成员将被字节单位存取。 无法使用指针存取成员。
指向规定会缩减数据大小，使数据块转移更具效率。 然而，若您将对齐数更改为 1，word 或 long word 结构的成员将被逐字节存取。 这会增加代码大小。

C/C++ 程序



3.17 加长类型

描述：

可支持 long long 和 unsigned long long 数据类型。
带符号的整数被描述为 long long，而无符号的整数被描述为 unsigned long long。
要创建 long long 类型的整数常数，需为整数添加 LL 的后缀。要创建 unsigned long long 类型的整数常数，需为整数添加 ULL 的后缀。

表 3.44 整数类型及值的范围

类型	值范围	数据大小
char	-128 至 127	1 字节
signed char	-128 至 127	1 字节
unsigned char	0 至 255	1 字节
short	-32768 至 32767	2 字节
unsigned short	0 至 65535	2 字节
int	-2147483648 至 2147483647	4 字节
unsigned int	0 至 4294967295	4 字节
long	-2147483648 至 2147483647	4 字节
unsigned long	0 至 4294967295	4 字节
long long	-9223372036854775808 至 9223372036754775807	8 字节
unsigned long long	0 至 18446744073709551615	8 字节

3.18 DSP-C 指定

描述：

可支持 DSP-C 语言。
当为 SuperH RISC engine C/C++ 编译程序指定了“dspc”的编译程序选项时，这项指定有效。

3.18.1 定点数据类型

以前，整数类型被用来代表分数值。 现在您可以在不需修改的情况下使用定点数据类型来编写分数值。
SuperH RISC engine C/C++ 编译程序生成适合所使用的定点数据类型的 DSP 指令。 表 3.45 显示定点数据类型的内部表达。

表 3.45 定点数据类型的内部表达

类型	大小（存储器大小）	对齐数（字节）	数据范围		常数索引
			最小值	最大值	
_ _fixed	16 位（16 位）	2	-1.0	$1.0-2^{-15}$ (0.999969482421875)	r
long _ _fixed	32 位（32 位）	4	-1.0	$1.0-2^{-31}$ (0.99999999953433871269226074218 75)	R
_ _accum	24 位（32 位）	4	-256.0	$256.0-2^{-15}$ (255.999969482421875)	a
long _ _accum	40 位（64 位）	4	-256.0	$256.0-2^{-31}$ (255.99999999953433871269226074218 75)	A

重要信息：

(1) 存储在存储器中的 _ _*accum* 和 long _ _accum 数据靠右对齐，起始部分添加了符号扩展。

```
实例：  (_ _accum)128.5a 存储为 "00 40 40 00"。
实例：  (long _ _accum)(-256.0A) 存储为 "FF FF FF 80 00 00 00 00"。
```

(2) 比较 DSP-C 和之前的方法

C 函数 [之前的方法]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short input[NUM] = {0x1000, 0x2000, 0x4000,
                0x6000,

                   0xf000, 0xe000, 0xc000,
                0xa000};

short result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

   result[i] = input[i] + 0x1000;

    }

}

void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

   printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#define NUM 8

__fixed input[8] = { 0.125r, 0.25r, 0.5r, 0.75r,

             -0.125r,  -0.25r, -0.5r, -0.75r};

__fixed result[NUM];

void func()

{

    int i;

    for (i = 0; i < NUM; i++) {

   result[i] = input[i] + 0.125r;

    }

}

void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

   printf("%r\n", result[i]);

    }

}
```

(3) 乘法累加运算的实例

若使用整数类型来代替分数值，积必须与固定位数对齐。 定点数据类型则不需要这种对齐。

C 函数 [之前的方法]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short x input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000, 0xf000, 0xe000, 0xc000, 0xa000};

short y input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000, 0xf000, 0xe000, 0xc000, 0xa000};

int result;

int func(short *x_input, short *y_input)

{

    int i;

    int temp = 0;

    for (i = 0; i< NUM ;i++) {

      temp += (x_input[i] * y_input[i]) >> 15;

    }

    return (temp);

}

void main()

{

    result = func(x_input, y_input);

    printf("%f\n", result/32768.0);

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc -fixed_noround

#include <stdio.h>

#define NUM 8

  X   fixed x input[NUM] = { 0.125r,   0.25r,
 0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};

  Y   fixed y input[NUM] = { 0.125r,   0.25r,
0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};

__accum result;

void func(__accum *result_p,

          __X __fixed *x_input,

         __Y __fixed *y_input)

{

    int i;

    __accum temp = 0.0a;

    for (i = 0; i< NUM ;i++) {

      temp += x_input[i] * y_input[i];

    }

    *result_p = temp;

}

void main()

{

    func(&result, x_input, y_input);

    printf("%a\n", result);

}
```

3.18.2 存储器限定符

将 X/Y 存储器限定符添加到变量将可促使生成比普通存储器存取指令跟更有效的 X/Y 存储器专用存取指令。
使用下列限定符来明确指定用于存储数据的 X 或 Y 存储器。

__X: 将数据存储在 X 存储器中。

__Y: 将数据存储在 Y 存储器中。

SuperH RISC engine C/C++ 编译程序可输出在表 3.46 中所显示的段具有 __X 或 __Y 存储器限定符的目标。 您必须在连接时将这些段分配到 X 或 Y 存储器。

表 3.46 存储器限定符指定

名称	段	描述
常数区域	$XC	const 数据（存储在 X 存储器中）
	$YC	const 数据（存储在 Y 存储器中）
初始化的数据区域	$XD	具有初始值的数据（存储在 X 存储器中）
	$YD	具有初始值的数据（存储在 Y 存储器中）
未初始化的数据区域	$XB	没有初始值的数据（存储在 X 存储器中）
	$YB	没有初始值的数据（存储在 Y 存储器中）

然而，X 或 Y 存储器可能仅存在于 RAM 上。 您在从这类存储器创建 ROM 时必须谨慎。

使用的实例：

(1) 通过使用 __X 或 __Y 存储器限定符，将数据存储在存储器中。

```
__X  int          a;      //存储在 X 存储器中。
     int  __X     b;      //存储在 X 存储器中。
__Y  int        * c;      //Y 存储器中 int 数据的指针（未定义存储器。）
     int   __Y  * d;      //Y 存储器中 int 数据的指针（未定义存储器。）
     int   *__Y   e;      //int 数据的指针（存储在 Y 存储器中）
__X  int   *__Y   f;      //X 存储器中 int 数据的指针（存储在 Y 存储器中）
```

(2) 将常数区域和初始化的数据区域从 ROM 复制到 X/Y RAM。
在这个例子中，连接期间存储在 ROM 中的数据，在程序启动时被复制到 X/Y RAM。您需要使用优化连接编辑程序的 rom 选项来在 ROM 和 X/Y RAM 中分配相同的空间两次。

连接期间的子命令实例：

rom=$XC=XC,$XD=XD,$YC=YC,$YD=YD start

P,C,D,$XC,$XD,$YC,$YD/400,$XB,XC,XD/05007000,$YB,YC,YD/05017000

标准程序库函数 INITSCT() 让您可以从 ROM 轻松将数据复制到 X/Y RAM。

使用的实例： _INITSCT()

```
#include <_h_c_lib.h>
void PowerON_Reset(void)
{
      _INITSCT();
    main();
    sleep();
}

#pragma section $DSEC
static const struct {
    void *rom_s;
    void *rom_e;
    void *ram_s;
} DTBL[] = { {__sectop("$XC"), __secend("$XC"), __sectop("XC")},
            {__sectop("$XD"), __secend("$XD"), __sectop("XD")},
            {__sectop("$YC"), __secend("$YC"), __sectop("YC")},
            {__sectop("$YD"), __secend("$YD"), __sectop("YD")}};
#pragma section
```

(3) 不使用常数区域或初始化区域

通过指定不使用 X/Y 存储器限定符将 const 指定或初始化数据添加到目标，您不需要在 ROM 和 X/Y RAM 中分配相同的空间两次。

例如，您可通过指定以下实例中的动态初始化来消除初始化的数据。

使用的实例

```
#define NUM 8

__X __fixed x_input[NUM];

__Y __fixed y_input[NUM];

__fixed x_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,  -0.125r,  -0.25r,  -0.5r,  -0.75r};

__fixed y_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,  -0.125r,  -0.25r,  -0.5r,  -0.75r};

void xy_init()

{

        int i;


        for (i = 0; i< NUM; i++) {

            x_input[i] = x_init[i];

            y_input[i] = y_init[i];

        }

}


void main()

{

         xy_init();

            :

            :

}
```

(4) 比较 DSP-C 和之前的方法

C 函数 [之前的方法]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                0x6000,

    0xf000, 0xe000, 0xc000, 0xa000};

short y_input[NUM] = {0x2000, 0x4000, 0xe000,
                0xf000,

    0x6000, 0x2000, 0xe000, 0xf000};

short result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] - y_input[i];

    }

}


void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

      printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#define NUM 8

  X   fixed x_input[NUM] = { 0.125r,  0.25r,
            0.5r,  0.75r,

           -0.125r, -0.25r, -0.5r, -0.75r};

__Y __fixed y_input[NUM] = {0.25r, 0.5r, -0.25r,
           -0.125r,

     0.75r, 0.25r, -0.25r, -0.125r};

__fixed result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] - y_input[i];

    }

}


void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

      printf("%r\n", result[i]);

    }

}
```

3.18.3 饱和限定符

若操作造成溢出，饱和操作将以最大或最小的可表示值替换结果。对于 DSP-C，只需添加饱和限定符即可启用饱和操作。使用下列限定符来指定饱和操作：

```
_ _sat
```

您只能为 _ _fixed 或 long _ _fixed 数据指定饱和限定符。 为任何其他数据类型指定饱和限定符将会形成错误。
若表达式包含至少指定了一个饱和限定符 (_ _sat) 的数据块，饱和操作将被执行。

使用的实例：

(1) sat 指定的实例

```
_ fixed         a;
_ _sat   _ _fixed b;
_ _fixed        c;

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

(2) 比较 DSP-C 和之前的方法

C 函数 [之前的方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
    0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000,
     0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short result[NUM];
void func(void)
{
  int i;
  int temp;
  for (i = 0; i < NUM; i++) {
        temp = x_input[i] + y_input[i];
        if (temp > 32767) {
          temp = 32767;
        }
        else if (temp < -32768) {
        temp = -32768;
    }
    result[i] = temp;
  }
}
void main(void)
{
  int i;
  func();
   :
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__sat __X __fixed x_input[NUM] = { 0.125r,  0.25r,  0.5r,  0.75r,
 -0.125r, -0.25r, -0.5r, -0.75r};
__sat __Y __fixed y_input[NUM] = { 0.125r,  0.25r,  0.5r,  0.75r,
-0.125r, -0.25r, -0.5r, -0.75r};
__fixed result[NUM];

void func(void)
{
  int i;
  for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] + y_input[i];
  }
}
void main(void)
{
  int i;
  func();
  for (i = 0; i < NUM; i++) {
        printf("%r\n", result[i]);
  }
}
```

3.18.4 循环限定符

使用下列限定符来指定取模寻址：

```
_ _circ
```

您可以为指定了存储器限定符 (_ _X/_ _Y) 的 _ _fixed 类型一维数组和指针指定取模寻址。 为任何其他条件指定取模寻址将会形成错误。

使用的实例：

(1) 比较 DSP-C 和之前的方法

C 函数 [之前的方法]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

#define BUFFER_SIZE 4

short x input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000,, 0xf000, 0xe000, 0xc000, 0xa000};

short y input[BUFFER SIZE] = {0x2000, 0x4000,
0x2000, 0x1000};

short result[NUM];


void func()

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] +
      y_input[i%(BUFFER_SIZE)];

    }

}


void main()

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

     printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#include <machine.h>

#define NUM 8

#define BUFFER_SIZE 4

  X   fixed x input[NUM] = { 0.125r,  0.25r,
    0.5r,  0.75r, -0.125r, -0.25r, -0.5r,
    -0.75r};

__circ __Y __fixed y_input[BUFFER_SIZE] =
    {0.25r, 0.5r, 0.25r, 0.125r};

__fixed result[NUM];


void func()

{

    int i;

    set_circ_y(y_input, sizeof(y_input));

    for (i = 0; i < NUM; i++) {

     result[i] = x_input[i] + y_input[i];

    }

    clr_circ();

}


void main()

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

     printf("%r\n", result[i]);

    }

}
```

重要信息：

(1) 取模寻址适用于内建函数 clr_circ() 和 set_circ_x() 或 set_circ_y() 之间的一维数组和指针。
(2) 若您同时对多个数组指定取模寻址或您使用不是以上述内建函数指定的 _ _circ 参考数组或指针，将无法保证正确的操作。
(3) 若您以负值方向指定取模寻址，将无法保证正确的操作。
(4) 必须对齐服从于取模寻址的数据，以使连接时的高 16 位相同。 您不可直接参考数组的内容。
(5) 若发生下列其中一种状况（会输出警告），将无法保证正确的操作：

- 指定了 optimize=0。
- 为非局部变量指定了 _ _circ 指针。
- 为 _ _circ 指针指定了 volatile。
- 被更新的 _ _circ 指针没有被参考。
- 存在介于内建函数 clr_circ 和 set_circ_x 或 set_circ_y 之间的函数。

3.18.5 类型转换

表 3.47 显示类型转换的规则。

表 3.47 类型转换的规则

转换	指定
_ _fixed -> long _ _fixed	低 16 位被清零。
_ _accum -> long _ _accum	值保持不变。
long _ _fixed -> _ _fixed	低 16 位被截断。
long _ _accum -> _ _accum	分数部分的准确度降级。
_ _fixed -> _ _accum	为高 8 位执行符号扩展。
long _ _fixed -> long _ _accum	值保持不变。
_ _fixed -> long _ _accum	为高 8 位执行符号扩展。 低 16 位被清零。 值保持不变。
long _ _fixed -> _ _accum	为高 8 位执行符号扩展。 低 16 位被截断。 分数部分的准确度降级。
_ _accum -> _ _fixed	高 8 位被截断。 第 9 位必须是信号位。
long _ _accum -> long _ _fixed	若整数部分是零，值将保持不变。
_ _accum -> long _ _fixed	
long _ _accum -> _ _fixed	高 8 位和低 16 位被截断。 第 9 位必须是信号位。 若整数部分是零，值将保持不变。 分数部分的准确度降级。
_ _fixed -> signed integer type	-1.0r 和 –1.0R 的值是 –1，其他情形是 0。
long _ _fixed -> signed integer type	
_ _accum -> signed integer type	分数部分被截断。
long _ _accum -> signed integer type	转换后的值是从 -256 至 255 的整数。
_ _fixed -> unsigned integer type	对于 –1.0r 和 –1.0R，将假设为转换后类型的最大值。 其他情形则假设为 0。
long _ _fixed -> unsigned integer type	
_ _accum -> unsigned integer type	分数部分被截断。 对于正值，转换后的值是从 -0 至 255 的整数。 对于负值，将假设为（转换前的值 + 1 + 转换后类型的最大值）。
long _ _accum -> unsigned integer type	
signed integer type -> _ _fixed	转换前的最高位必须是转换后的最高位。
signed integer type ->long _ _fixed	所有其他位将是零。
signed integer type -> _ _accum	值的低 9 位必须是整数部分。
signed integer type ->long _ _accum	分数部分必须是零。
unsigned integer type -> _ _fixed	转换后的所有位必须是零。
unsigned integer type ->long _ _fixed	

转换	指定
unsigned integer type -> _ _accum	值的低 9 位必须是整数部分。
unsigned integer type ->long _ _accum	分数部分必须是零。
Fixed-point -> floating-point	可以转换后类型表示的值将和原始值相同。 无法表示的值将被舍入到最近的值。
Floating-point -> fixed point	分数部分的处理和从定点到浮点的转换相同。 整数部分的处理和从浮点到整数的转换相同。 若整数部分是定点的可表示范围，值将保持不变。 若整数部分超出范围，溢出的最低位必须是信号位。即使为转换后类型指定了饱和处理，该处理也不会被执行。

重要信息：

(1) 从 (long)_ _fixed 到整数类型的转换，或相反
可以 (long)_ _fixed 类型表示的整数是 0 和 –1。
这表示上述转换会造成信息遗失。

(2) 从 (long)_ _accum 到整数类型的转换，或相反
可以 (long)_ _accum 类型表示范围介于 -256 到 255 的整数。 此范围内的整数会在转换后保存信息。
然而，注意将负值转换到无符号的整数类型将会造成溢出。
对于一系列只需要整数类型的运算，转换到整数类型将可以增进性能。

(3) 位样式复制
若您使用替代运算符来复制位样式，将会发生类型转换，同时无法获取预计的结果。 在这种情况下，使用 long_as_lfixed 和 lfixed_as_long 等的内建函数。

3.18.6 算术转换

若运算包含两个不同类型的操作数，将使用在图 3.37 的上列中显示的类型。
若您指定在图 3.37 中类型不相关的运算，将会发生错误（例如，integer 和 floating-point 或 _ _accum 和 long_ _fixed 之间）。在这种情况下，您必须使用转型来执行明确的类型转换。然而，只要能保证结果的值，可在一些运算中为获取效率而略过上述转换规则。

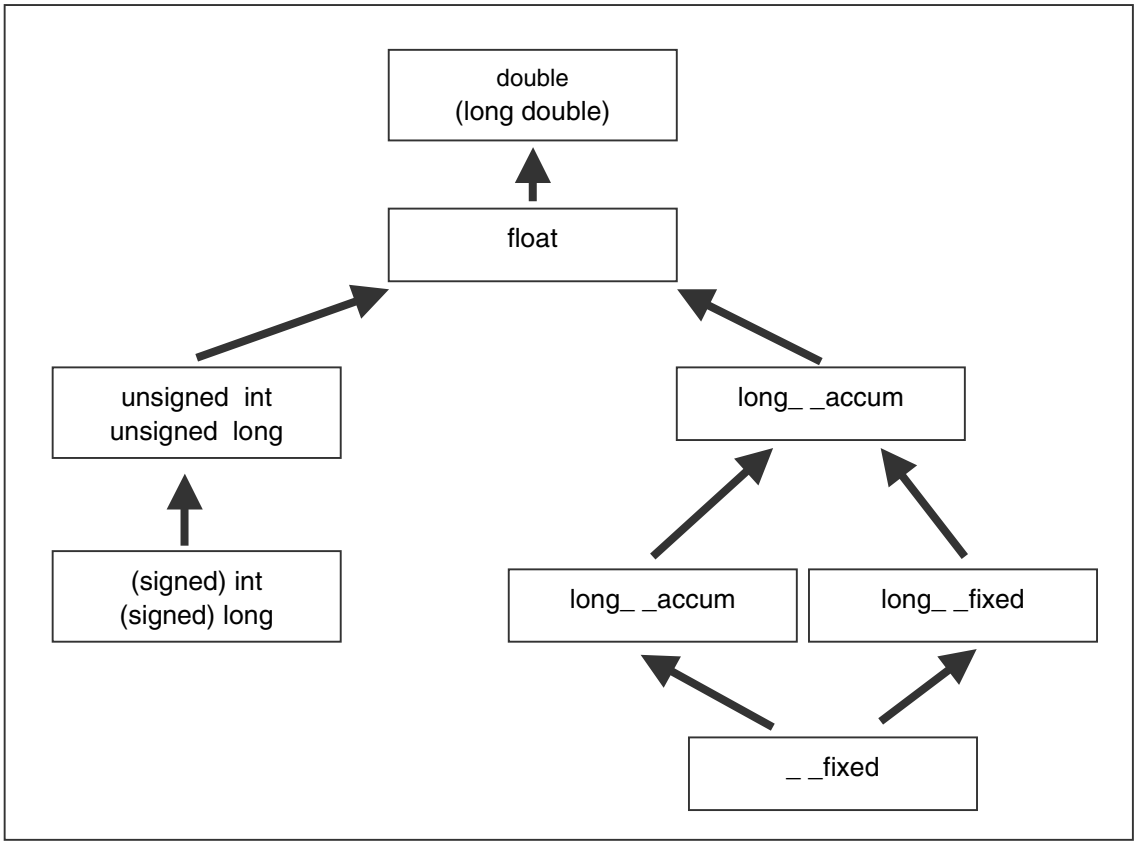


图 3.37 算术转换的规则

3.19 映像优化扩展选项

描述：

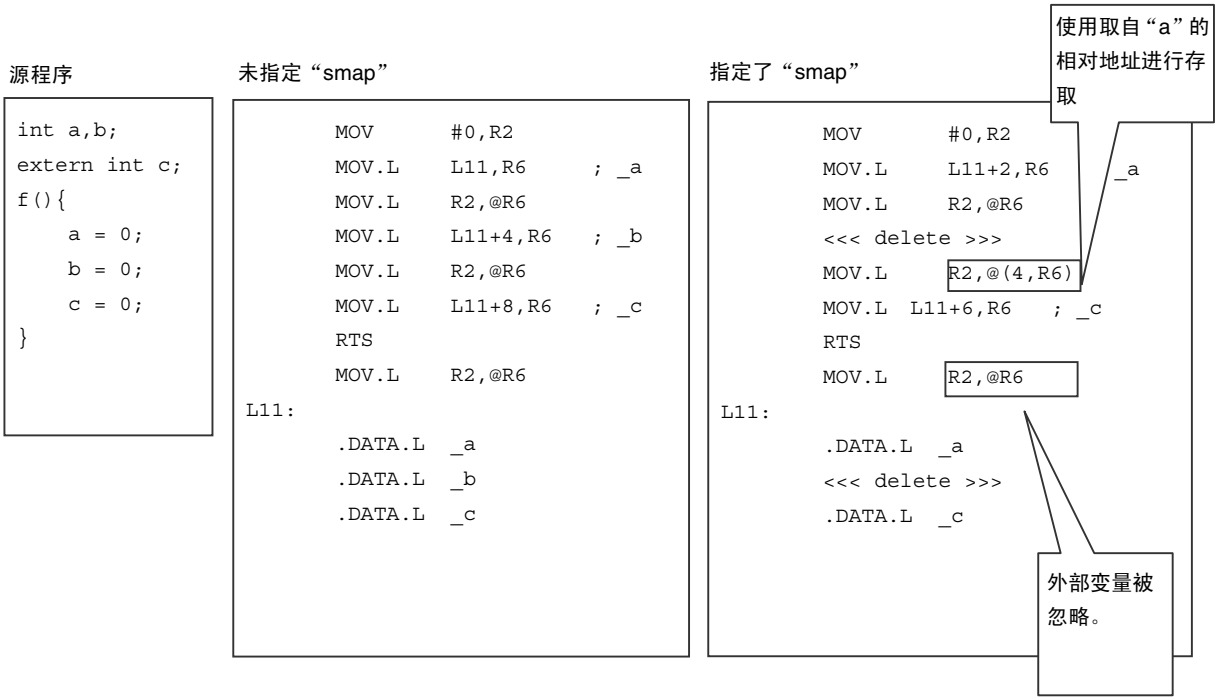
此选项可执行映像优化，而不需使用由连接分配的符号分配地址的相关信息。 因此，不需执行重新编译。
然而，由于优化仅适用于在文件内定义的 static 变量，因此 extern 变量无法被优化。

3.19.1 使用

在编译时指定“-smap”选项。

3.19.2 外部变量存取代码获增进的实例 (1)

考虑到相同段内变量分配的顺序，为相同的寄存器相对的存取连续分配的变量。



3.19.3 外部变量存取代码获增进的实例 (2)

当指定了“gbr=auto”选项（默认）时，GBR 被用作外部变量存取的基址。

源程序

```
int a[100];
f(){
    a[0]=0;
```

未指定“smap”

```
        MOV.L       L11+2,R5    ; _a
        MOV         #-56,R0
        MOV         #0,R4
        EXTU.B      R0,R0
        MOV.L       R4,@R5
        MOV.L       R4,@(R0,R5)
        ADD         #4,R0
        MOV.L       R4,@(R0,R5)
        ADD         #4,R0
        RTS
        MOV.L       R4,@(R0,R5)
L11:
        .RES.W      1
        .DATA.L     _a
```

指定了“smap”

```
        STC         GBR,@-R15
        MOV.L       L11,R0      ; _a
        LDC         R0,GBR
        MOV         #0,R0
        MOV.L       R0,@(0,GBR)
        MOV.L       R0,@(200,GBR)
        MOV.L       R0,@(204,GBR)
        MOV.L       R0,@(208,GBR)
        RTS
        LDC         @R15+,GBR
L11:
        .DATA.L     _a
```

使用 GBR 相对的引用

3.20 TBR 相对函数调用

描述：

对于 SH-2A 和 SH2A-FPU，通过使用表参考子例程调用指令，将跳转表基址寄存器 (TBR) 用来调用函数。
从 TBR 获取的相对值，是在跳转表中其基址由 TBR 包含的偏移。 这个值相等于 $TBR 段的起始处与调用目标函数的地址数据标签之间的距离。
您可以使用“-tbr”选项来指定对所有函数使用 TBR 相对调用。 您也可以使用预处理程序指令“#pragma tbr”来指定个别函数的 TBR 相对调用。
若要执行 TBR 相对函数调用，您必须将 $TBR 段的起始地址设定为 TBR。
若要使用 TBR 相对调用来调用标准程序库函数，请执行下列操作：

(1) 在“tbr.h”系统包含文件中，添加“#pragma tbr”，接着注明要使用 TBR 相对调用的程序库。
(2) 使用标准程序库创建工具“lbgsh”来创建程序库。
(3) 在您要执行 TBR 相对调用的程序库的调用源程序中，使用 #include 指令来包含“tbr.h”。

格式：

```
〈选项〉

-tbr[=<段名称>]


<预处理程序指令>

#pragma tbr (<函数名称>[(sn=<段名称>|ov=<偏移值>)][,...])

<偏移值> 必须是 4 的倍数，从 0 到 1020
```

使用的实例：

实例 1：

若指定了 TBR 相对函数调用，编译程序将对所有函数使用 TBR 相对调用，并为在文件中定义的函数生成跳转表。 此跳转表包含函数地址数据以及它们的标签。
跳转表中函数地址的标签名称是函数名称，以“$_”为前导。对于静态函数，标签名称是函数名称，以“$_$”为前导。

C 语言代码

```
/* -cpu=sh2a -size -tbr */
#include <machine.h>
void f1(){}
void f2(){}
static void f3(){}


main()

{

    set_tbr(__sectop("$TBR")); /* 将 $TBR 段的起始设定到 TBR。 */

    f1();

    f2();

    f3();
```

```
}
```

扩展为汇编语言代码

```
_main:
            STS.L       PR,@-R15
            MOV.L       L14+2,R2    ; STARTOF $TBR
            LDC         R2,TBR
            JSR/N       @@($_f1-(STARTOF $TBR),TBR)      ; TBR 相对函数调用
            JSR/N       @@($_f2-(STARTOF $TBR),TBR)      ; TBR 相对函数调用
            JSR/N       @@($__$f3-(STARTOF $TBR),TBR)    ; TBR 相对函数调用
            LDS.L       @R15+,PR
            RTS/N
L14:
            .RES.W      1
            .DATA.L     STARTOF $TBR
            .SECTION    $TBR,DATA,ALIGN=4                ; TBR 相对跳转表
$_f1:
            .DATA.L     _f1                              ; 函数地址数据
$_f2:
            .DATA.L     _f2                              ; 函数地址数据
$_main:
            .DATA.L     _main                            ; 函数地址数据
$__$f3:
            .DATA.L     __$f3                            ; 静态函数地址数据
```

实例 2：

除了指定对所有函数使用 TBR 相对调用的“-tbr”选项，您可以使用“#pragma tbr”来指定个别函数的 TBR 相对调用。
<函数名称>中指定的函数将使用 TBR 相对调用来进行调用。
若您指定“sn=<段名称>”，函数地址数据将由段名称所表示的段生成，并以“$TBR”为前导。
若您指定“ov=<偏移值>”，TBR 相对值将会是所表示的偏移值。

C 语言代码

```
/* -cpu=sh2a -size */
#pragma tbr (f1(sn=X))
#pragma tbr (f2(ov=0))
f1(){}
f2(){}
```

```
main(){
f1();
f2();
}
```

扩展为汇编语言代码

```
    _main:
          STS.L       PR,@-R15
          JSR/N       @@($_f1-(STARTOF $TBRX),TBR)
          JSR/N       @@(0,TBR)                ; TBR 相对值是 0
          LDS.L       @R15+,PR
          RTS/N
          .SECTION    $TBRX,DATA,ALIGN=4       ; 段名称“$TBRX”
$_f1:
          .DATA.L     _f1
                                               ; 没有为指定了“ov=<偏移值>”（参考实例 3）的
                                               ; 函数（f2）生成函数
                                               ; 地址数据。
```

实例 3：

对于指定了“ov=<偏移值>”的函数，您必须在 TBR 相对跳转表中创建函数地址数据。
若在相同的文件中找不到函数定义，您必须在文件中设定相同的函数定义指定，或在 TBR 相对跳转表中创建函数地址数据。

C 语言代码

```
/* -cpu=sh2a */
#pragma tbr (func1(ov=0))            /* 在跳转表中指定偏移 0 */
#pragma tbr (func2(ov=4))            /* 在跳转表中指定偏移 4 */
#pragma tbr (func3(ov=8))            /* 在跳转表中指定偏移 8 */
extern void func1();
extern void func2();
extern void func3();

#pragma tbr (func4(sn=NEW))          /* 为跳转表中的段指定“$TBRNEW” */
#pragma tbr (func5(sn=NEW))
#pragma tbr (func6(sn=NEW))
extern void func4();
```

```
extern void func5();
extern void func6();


#include<machine.h>
void main()
{
     set_tbr(__sectop("$TBR"));       /* 将 $TBR 段的起始设定到 TBR。 */
     func1();
     func2();
     func3();
     set_tbr(__sectop("$TBRNEW"));    /* 将表切换至“$TBRNEW” */
     func4();
     func5();
     func6();
}
```

扩展为汇编语言代码

```
_main:
            STS.L       PR,@-R15
            MOV.L       L11+2,R1    ; STARTOF $TBR
            LDC         R1,TBR
            JSR/N       @@(0,TBR)
            JSR/N       @@(4,TBR)
            JSR/N       @@(8,TBR)
            MOV.L       L11+6,R4    ; STARTOF $TBRNEW
            LDC         R4,TBR
            JSR/N       @@($_func4-(STARTOF $TBRNEW),TBR)
            JSR/N       @@($_func5-(STARTOF $TBRNEW),TBR)
            JSR/N       @@($_func6-(STARTOF $TBRNEW),TBR)
            LDS.L       @R15+,PR
            RTS/N
L11:
          .RES.W     1
            .DATA.L     STARTOF $TBR
            .DATA.L     STARTOF $TBRNEW
```

要指定 TBR 相对函数调用，您必须设定 TBR。 由于编译程序在调用函数时会参考跳转表上的数据以查找函数地址，这将可缩减数据大小。
下面显示在不使用 TBR 的情况下，调用函数的代码。

```
_main:
            STS.L     PR,@-R15
            MOV.L      L11,R1      ; _func1
            JSR/N      @R1
            MOV.L      L11+4,R4    ; _func2
            JSR/N      @R4
            MOV.L      L11+8,R5    ; _func3
            JSR/N      @R5
            MOV.L      L11+12,R6   ; _func4
            JSR/N      @R6
            MOV.L      L11+16,R7   ; _func5
            JSR/N      @R7
            MOV.L      L11+20,R2   ; _func6
            JMP        @R2
            LDS.L      @R15+,PR
L11:
            .DATA.L    _func1
            .DATA.L    _func2
            .DATA.L    _func3
            .DATA.L    _func4
            .DATA.L    _func5
            .DATA.L    _func6
```

汇编语言代码（跳转表 1）

根据“pragma tbr”中的“ov=<偏移值>”指定，在“$TBR”段中为函数地址数据创建跳转表。

```
            .SECTION   $TBR,DATA,ALIGN=4    ;
            .DATA.L    _func1               ; 跳转表中的偏移应为 0。
            .DATA.L    _func2               ; 偏移应为 4
            .DATA.L    _func3               ;    偏移应为 8
```

汇编语言代码（跳转表 2）

若在相同的文件中找不到函数定义，使用文件中相同的函数定义指定，或创建下列跳转表。

```
            .EXPORT     $_func4
            .EXPORT     $_func5
            .EXPORT     $_func6
            .SECTION    $TBRNEW,DATA,ALIGN=4
$_func4:                                    ; 标签名称应为“$_”+<函数名称>
            .DATA.L     _func4              ; 函数地址数据
$_func5:
            .DATA.L     _func5
$_func6:
            .DATA.L     _func6
```

实例 4：

对于 SH-2A，CPU 将调用与 TBR 相对的 printf 函数。

(1) 在“tbr.h”中指定“#pragma tbr printf”。

```
        :
#if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC)
        :
#pragma tbr printf      // ← 已添加
        :
#endif  /* #if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC) */
        :
```

(2) 创建包含 TBR 相对表的标准程序库。

```
lbgsh -cpu=sh2a
```

(3) 指定在程序中包含使用 printf 的“tbr.h”。

```
#include <tbr.h>        // ← 已添加
#include <stdio.h>

main()
{
    printf("tbr\n");
}
```

重要信息：

(1) 若 BSR 指令可用来调用函数，编译程序将不使用 TBR 相对调用。 然而，若指定了“-size”选项，编译程序仍会使用 TBR 相对调用。
(2) 若您指定“-cpu=sh2a”或“-cpu=sh2afpu”以外的任何其他选项，TBR 相对函数调用将被禁用。
(3) 若您指定“-pic=1”选项，TBR 相对函数调用将被禁用，因为无法确定函数的绝对地址。
(4) 若“$TBR”被用来表示在“-section”选项中为段名称指定的跳转表的段名称，将会在执行目标时发生故障。
(5) 您可以在整个程序中为每个段指定多达 255 个函数。
(6) 您不可为相同函数同时指定“sn=<段名称>”和“ov=<偏移值>”。
(7) 若您为相同函数同时指定下列 #pragma 扩展，将会发生错误。

```
#pragma interrupt

#pargma inline

#pragma inline_asm

#pragma entry
```

3.21 生成 GBR 相对逻辑运算指令

描述：

当与“-logic_gbr”选项指定了“-gbr=user”选项时，相对于 GBR 的逻辑运算指令将用于“#pragma gbr_base”中所指定的 GBR 基址变量以外的外部变量。

格式：

```
-logic_gbr
```

使用的实例：

C 语言代码

```
char a,b,c;
main(){
     a &= 0x0f;
     b |= 0x01;
     c ^= 0x01;
}
```

扩展为汇编语言代码（指定了“-gbr_user”，未指定“-logic_gbr”）

```
          MOV.L       L11+2,R6    ; _a
          MOV.B       @R6,R0
          AND         #15,R0
          MOV.B       R0,@R6
          MOV.L       L11+6,R6    ; _b
          MOV.B       @R6,R0
          OR          #1,R0
          MOV.B       R0,@R6
          MOV.L       L11+10,R6   ; _c
          MOV.B       @R6,R0
          XOR         #1,R0
          RTS
          MOV.B       R0,@R6
L11:
          .RES.W      1
          .DATA.L     _a
          .DATA.L     _b
          .DATA.L     _c
```

扩展为汇编语言代码（指定了“-gbr user”，指定了“-logic gbr”）

```
            MOV.L       L11+2,R0    ; _a-(STARTOF $G0)
            AND.B       #15,@(R0,GBR)                   ; GBR 相对运算指令
            MOV.L       L11+6,R0    ; _b-(STARTOF $G0)
            OR.B        #1,@(R0,GBR)                    ; GBR 相对运算指令
            MOV.L       L11+10,R0   ; _c-(STARTOF $G0)
            RTS
            XOR.B       #1,@(R0,GBR)                    ; GBR 相对运算指令
 L11:
            .RES.W      1
            .DATA.L     _a-(STARTOF $G0)
            .DATA.L     _b-(STARTOF $G0)
            .DATA.L     _c-(STARTOF $G0)
```

要将 GBR 用作基址，您必须事先为 $G0 段的起始地址指定 GBR，好像您为“#pragma gbr_base”指定的情形一样。

重要信息：

(1) 当您指定“-logic_gbr”选项时，您必须映射 $G0 段。
(2) 若您没有指定“-gbr=user”选项，“-logic_gbr”选项将被忽略。

3.22 启用寄存器声明

描述：

编译程序根据编译程序的分析结果，依顺序将寄存器分配到变量，无论寄存器是否有被声明。
当指定了“-enable_register”选项时，寄存器先被分配到具有寄存器声明的变量。

格式：

```
-enable_register
```

使用的实例：

C 语言代码

```
int sum[10],input1[10],input2[10];
int b;

void func()
{
      register int a = 0;
      int i;

      while(b) {
      a++;
      for (i = 0; i < 10; i++) {
      sum[i] = input1[i] + input2[i];
        }
        b--;
      }

      printf("%d\n",a);          // 由于‘a’的值通过 R5 传递到 printf，
                                 //将 R5 分配到‘a’将可改进效率。
}
```

扩展为汇编语言代码（未指定“-enable_register”）

```
_func:
           MOV.L       R12,@-R15
           MOV.L       R13,@-R15
           MOV.L       R14,@-R15
```

```
        MOV.L       L16+2,R12
        MOV         #0,R13          ; 由于 R5 被分配到具有较高优先级的其他变量，
                                    ; R13 被分配到变量 a。
              :           :
        MOV.L       L16+22,R2       ; _printf
        MOV.L       R14,@R12
        MOV         R13,R5          ; 将变量 a 的值从 R13 复制到 R5
        MOV.L       @R15+,R14
        MOV.L       @R15+,R13
        JMP         @R2             ; 调用 printf()
        MOV.L       @R15+,R12
```

扩展为汇编语言代码（指定了“-enable_register”）

```
_func:
        MOV.L       R12,@-R15
        MOV.L       R13,@-R15
        MOV.L       R14,@-R15
        MOV.L       L16,R12         ; _b
        MOV         #0,R5           ; 由于变量 a 具有较高的优先级，R5 被分配。
            :           :
        MOV.L       L16+20,R2       ; _printf
        MOV.L       R13,@R12
        MOV.L       @R15+,R14
        MOV.L       @R15+,R13
        JMP         @R2             ; 调用 printf()
        MOV.L       @R15+,R12
```

重要信息：

若未被分配寄存器，将会显示下列信息。
C0102 (I) Register is not allocated to "variable name" in "function name"（“函数名称”中的“变量名称”未被分配寄存器）
然而，若没有为任何寄存器分配参数，这项信息将不会显示。

3.23 指定变量的绝对地址

描述：

您可以使用预处理程序指令指定被外部参考的变量绝对地址。 编译程序会把在 #pragma address 指令中声明的变量分配到相应的绝对地址。 此项功能使能够通过变量更轻松的存取分配给特定地址的 I/O。

格式：

```
#pragma address （<变量名称> = <地址值>[,<变量名称> = <地址值> ...] ）
```

使用的实例：

变量“io”被分配到绝对地址 0x100。

C 语言代码

```
#pragma address (io=0x100)
int io;
f()
{
     io = 10;
}
```

扩展为汇编语言代码

```
_func:
          MOV         #1,R2
          SHLL8       R2
          MOV         #10,R6
          RTS
          MOV.L       R6,@R2
          .SECTION    $ADDRESS$B100,DATA,LOCATE=H'100
_io:
          .RES.L      1
```

重要信息：

(1) 您必须在变量声明前指定“#pragma address”。
(2) 若您指定复合类型的成员或变量以外的其他类型成员，将会发生错误。
(3) 若您为对齐数是 2 的变量或结构指定奇数地址，将会发生错误。另外，若您为对齐数是 4 的变量或结构指定四的倍数以外的地址，也会发生错误。
(4) 若您为相同变量指定“#pragma address”超过一次，将会发生错误。
(5) 若您为不同变量指定相同地址或您指定相同的变量地址超过一次，将会发生错误。
(6) 若您为相同变量同时指定下列 #pragma 扩展，将会发生错误。

```
#pragma section

#pragma abs16/abs20/abs28/abs32

#pragma gbr_base/gbr_base1

#pragma global_register
```

3.24 加强优化

3.24.1 获增进的字面数据 (1)

常数数据的优化已被加强。

源程序

```
unsigned
short a,b;
f(){
  a|=0x100;
  b=0xffff;
}
```

V5,V6

```
MOV.L    L237+2,R4   ; _a
MOV.W    L237,R3     ; H'0100
MOV.W    @R4,R2
OR        R3,R2
MOV.W    R2,@R4
MOV.L    L237+6,R1   ; H'0000FFFF
MOV.L    L237+10,R0 ; _b
RTS
MOV.W   R1,@R0
L237:
.DATA.W     H'0100
.DATA.L     _a
.DATA.L     H'0000FFFF
.DATA.L     _b
```

V7

使用 “1<<8” 来创建常数 256(0x100)

```
MOV.L       L11,R5      ; _a
MOV         #1,R2       ; H'00000001
SHLL8       R2
MOV.W      @R5,R6
OR          R2,R6
MOV.W      R6,@R5
MOV         #-1,R2      ; H'FFFFFFFF
MOV.L       L11+4,R6    ; _b
RTS
MOV.W      R2,@R6
L11:
.DATA.L     _a
.DATA.L     _b
```

使用 #imm 来设定

3.24.2 获增进的字面数据 (2)

2 字节或以上的常数值被重新使用。

源程序

```
unsigned
short a,b;
f(){
  a=200;
  b=300;
}
```

V5,V6

```
 MOV.W    L237,R3     ; H'00C8
MOV.L     L237+6,R2   ; _a
MOV.W    L237+2,R1   ; H'012C
MOV.L     R3,@R2
MOV.L     L237+10,R0 ; _b
RTS
MOV.L     R1,@R0
L237:
 .DATA.W  H'00C8 ; 200
 .DATA.W  H'012C ; 300
```

V7

```
MOV         #-56,R6   ; H'FFFFFFC8
MOV.L     L11,R5    ; _a
EXTU.B   R6,R6
MOV.L     L11+4,R2 ; _b
MOV.L     R6,@R5
ADD         #100,R6
RTS
MOV.L     R6,@R2
```

使用 200 + 100 来设定

3.24.3 禁用 EXTU (1)

对关于条件表达式中的 AND 结果禁用 EXTU。

源程序

```
unsigned
char a;
f(){
  if(a&120);
   :
}
```

V5,V6

```
MOV.L     L237+2,R4  ; _a
MOV.B     @R4,R0
EXTU.B    R0,R0
TST       #120,R0
BT        L236
```

V7

```
MOV.L     L237+2,R4  ; _a
MOV.B     @R4,R0
TST       #120,R
BT        L236
```

禁用 EXTU

3.24.4 禁用 EXTU (2)

在与常数比较时禁用 EXTU。

源程序

```
unsigned
char a;
f(){
  if(a==10);
   :
}
```

V5,V6

```
MOV.L     L237+2,R4  ; _a
MOV.B     @R4,R0
EXTU.B    R0,R0
CMP/EQ    #10,R0
BT        L236
```

V7

```
MOV.L     L237+2,R4  ; _a
MOV.B     @R4,R0
CMP/EQ    #10,R0
BT        L236
```

禁用 EXTU

3.24.5 获增进的位运算 (1)

增进 1 位数据的比较代码

源程序

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
      :
if(data.p7)
```

V5,V6

```
MOV.L    L239+2,R0   ; _data
MOV.B    @R0,R0
AND      #1,R0
EXTU.B   R0,R0
TST      R0,R0
BT       L238
```

V7

```
MOV.L    L239+2,R6   ; _data
MOV.B    @R6,R0
TST      #1,R0
BT       L238
```

具有 #1 的 TST

3.24.6 获增进的位运算 (2)

增进 1 位数据的替换代码

源程序

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data1,data2;
      :
data1.p7=data2.p6;
```

V5,V6

```
 STS.L      PR,@-R15
 MOV.L      L239+4,R0   ; _data2
 MOV.L      L239+8,R2   ; _data1
 MOV.B      @R0,R0
 MOV.W      L239,R1     ; H'0701
 TST        #2,R0
 MOV.L      L239+12,R3 ; __bfsbu
 MOVT       R0
 ADD        #-1,R0
 JSR        @R3
 NEG        R0,R0
 LDS.L      @R15+,PR
 RTS
 NOP
L239:
 .DATA.W    H'0701
 .DATA.L    __bfsbu0
```

V7

```
MOV.L   L14+2,R6   ; _data2
MOV.B   @R6,R0
MOV.L   L14+6,R6   ; _data1
TST      #2,R0
MOV.B   @R6,R0
BF       L12
BRA      L13
AND      #254,R0
L12:
OR       #1,R0
L13:
RTS
  MOV.B    R0,@R6
```

不使用运行时例程的内联扩展

3.24.7 获增进的位运算 (3)

增进位字段的逻辑运算代码

源程序

```
struct S{
unsigned char p0:4;
unsigned char p1:4;
}data1;
     :
data1.p1|=1;
```

V5,V6

```
STS.L       PR,@-R15
MOV.L    L236+4,R0   ; _data1
MOV.L    L236+4,R2   ; _data1
MOV.B    @R0,R0
MOV.W    L236,R1     ; H'0404
AND      #15,R0
MOV.L    L236+8,R3   ; __bfsbu
JSR       @R3
OR        #1,R0
LDS.L       @R15+,PR
RTS
NOP
L236:
  .DATA.W  H'0404
.DATA.W  0
.DATA.L   _data1
.DATA.L     bfsbu
```

V7

```
MOV.L     L11,R5    ; _data1
MOV.B     @R5,R2
MOV       R2,R0
AND       #15,R0
OR        #1,R0
AND       #15,R0
MOV       R0,R6
MOV       R2,R0
AND       #240,R0
OR        R6,R0
RTS
MOV.B     R0,@R5
L11:
  .DATA.L     _data1
```

不使用运行时例程的内联扩展

3.24.8 获增进的位运算 (4)

增进相同位字段的连续决策处理

源程序

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
     :
if(data.p7==1 &&
      data.p6==1)
```

V5,V6

```
MOV.L       L239+2,R4   ; _data
MOV         R4,R0
MOV.B       @R0,R0
AND         #1,R0
CMP/EQ      #1,R0
BF          L238
MOV         R4,R0
MOV.B       @R0,R0
TST         #2,R0
MOVT        R0
ADD         #-1,R0
NEG         R0,R0
CMP/EQ      #1,R0
BF          L238
```

V7

```
MOV.L       L14+2,R6    ; _data
MOV.B        @R6,R0
AND          #3,R0
CMP/EQ       #3,R0
BF           L12
```

2 位同时求值

3.24.9 获增进的位运算 (5)

增进相同位字段的连续替换

源程序

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
    :
data.p0=0;
data.p1=0;

:
data.p7=0;
```

V5,V6

```
MOV.L       L240,R4     ; _data1
MOV.B       @R4,R0
AND         #127,R0
MOV.B       R0,@R4
MOV.B       @R4,R0
AND         #191,R0
MOV.B       R0,@R4
  :
MOV.B       @R4,R0
AND         #254,R0
RTS
MOV.B       R0,@R4
```

V7

```
MOV.L     L11,R2   ; _data1
MOV       #0,R3    ; H'00000000
RTS
MOV.B     R3,@R2
```

同时设定所有位

3.25 控制未初始化变量的输出顺序

描述

可使用“-bss_order”选项来按声明顺序或定义顺序分配未初始化的变量。

指定 -bss_order=declaration 来按声明顺序分配未初始化的变量，或 -bss_order=definition 来按定义顺序分配未初始化的变量。 若略过此选项，-bss_order=declaration 将被使用。

格式

```
-bss_order={ declaration | definition }
```

使用的实例

C 语言代码

```
extern int a1;
extern int a2;
int a3;
extern int a4;
int a5;
int a2;
int a1;
int a4;
```

扩展为汇编语言代码

当 <指定了 -bss_order=declaration:>

```
        .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
```

当 <指定了 -bss_order=definition:>

```
        .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
```

说明

当指定了“stuff”选项时，bss_order=definition 将始终生效。

3.26 指定变量的放置

描述

通过边界对齐调整数，“-stuff”选项可被用来将变量放置在不同的段中。 这会缩减填充，以保存存储器。

可在“-stuff”选项中指定段类型。 根据数据大小，属于特定段类型的变量将被放置在边界对齐调整数为 4、2 和 1 的段中。 若段类型被略过，选项将以所有变量为目标。

每个段中的数据将按定义顺序输出。若指定了 bss_order=declaration 将被省略。

若指定了“-nostuff”，所有变量将被放置在具有边界对齐调整数 4 的段中。

每个段中的数据将遵循 C 和 D 段的定义顺序，以及 B 段的 bss_order。

若此选项被略过，将会使用“nostuff”。

表 3.48 变量大小与段名称之间的关系

	段类型	变量大小（字节）		
		4n	4n+2	2n+1
Const 类型变量	const	C$4	C$2	C$1
具有初始值的初始化变量	data	D$4	D$2	D$1
没有初始值的未初始化变量	bss	B$4	B$2	B$1

格式

```
-stuff [=section-type[,...]]

-nostuff

section-type:{ Bss | Data | Const }
```

使用的实例：

C 语言代码

```
int a;

char b=0;

const short c=0;

struct {

char x;

char y;

} ST;
```

扩展为汇编语言代码

```
                .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
_sr:
                .RES.B  2
```

说明

指定了 #pragma gbr_base|gbr_base1 或 #pragma global_register 的变量不受此选项影响。

SuperH RISC Engine C/C++编译程序应用笔记

HEW

第 4 节　　HEW

4.1　　在 HEW2.0 或以上版本中指定选项

您可以从［创建 (Build)］菜单指定选项。 此处显示如何从“瑞萨集成开发环境”(Renesas Integrated Development Environment)指定选项。从“创建”(build) 菜单选取“RISC engine 标准工具链”(RISC engine Standard Toolchain) 。

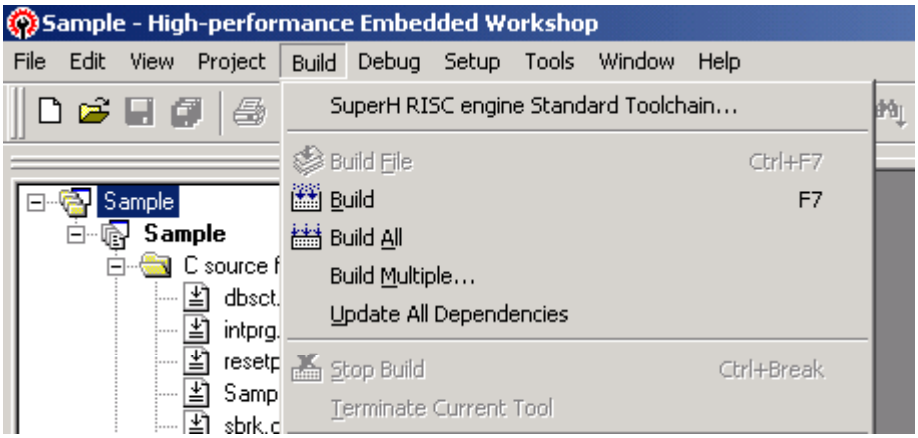


图 4.1 HEW 创建菜单

4.1.1 C/C++ 编译程序选项

从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 对话框中选取“C/C++”标签。

(1) 类别 (Category):[源 (Source)]

表 4.1 类别 (Category):[源 (Source)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
显示有关项目：	
包含文件目录	Include = <路径名称>[,…]
预包含文件	PREInclude = <文件名>[,…]
定义	DEFine = <sub>[,…] <sub> : <宏名称> [= <字符串>]
消息	MEssage
消息级别	CHAnge_message = <sub>[,…] <sub> : <级别>[=<n>[-m],…] <级别> : {Information | Warning | Error}
文件内联路径	FILE_INLINE_PATH = <路径名称>[,…]
显示信息级别消息	NOMEssage [= <错误编号> [- <错误编号>[,…]]

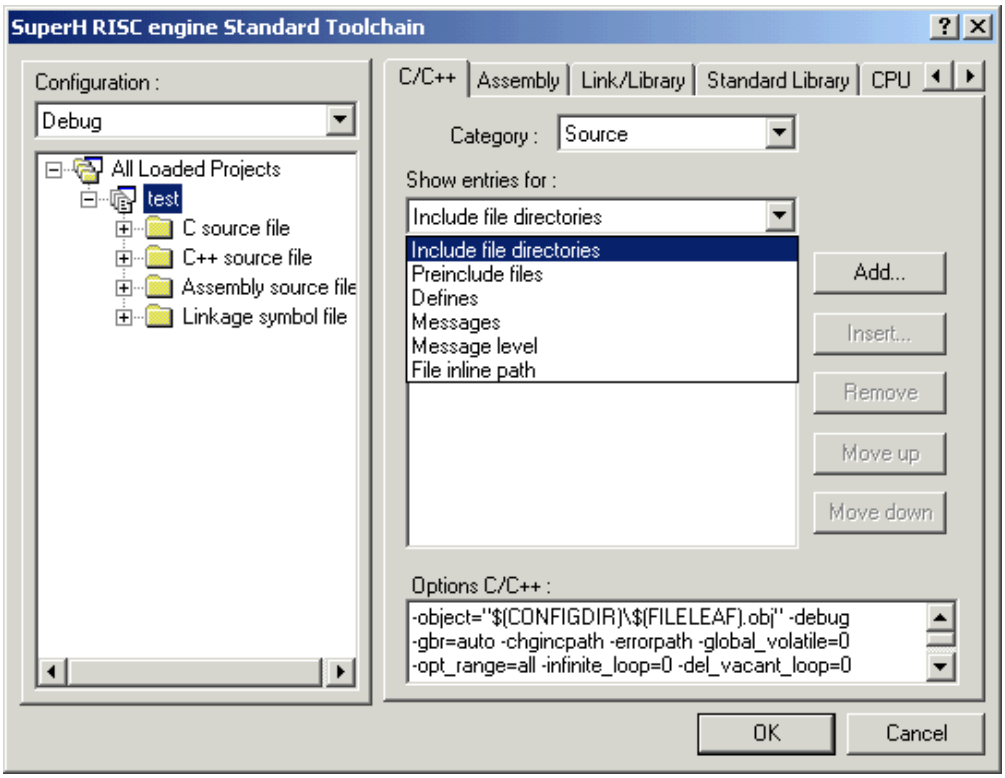


图 4.2 类别 (Category):[源 (Source)] 对话框

(2) 类别 (Category):[目标 (Object)]

表 4.2 类别 (Category):[目标 (Object)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
输出文件类型：	
机器码 (*.obj)	Code = Machinecode
汇编源代码 (*.src)	Code = Asmcode
预处理源文件 (*.p/*.pp)	PREProcessor [= <文件名>]
在预处理的源文件中禁止#line	NOLINe
生成调试信息	DEBug / NODEBug
输出目录：	OBjectfile = <文件名>

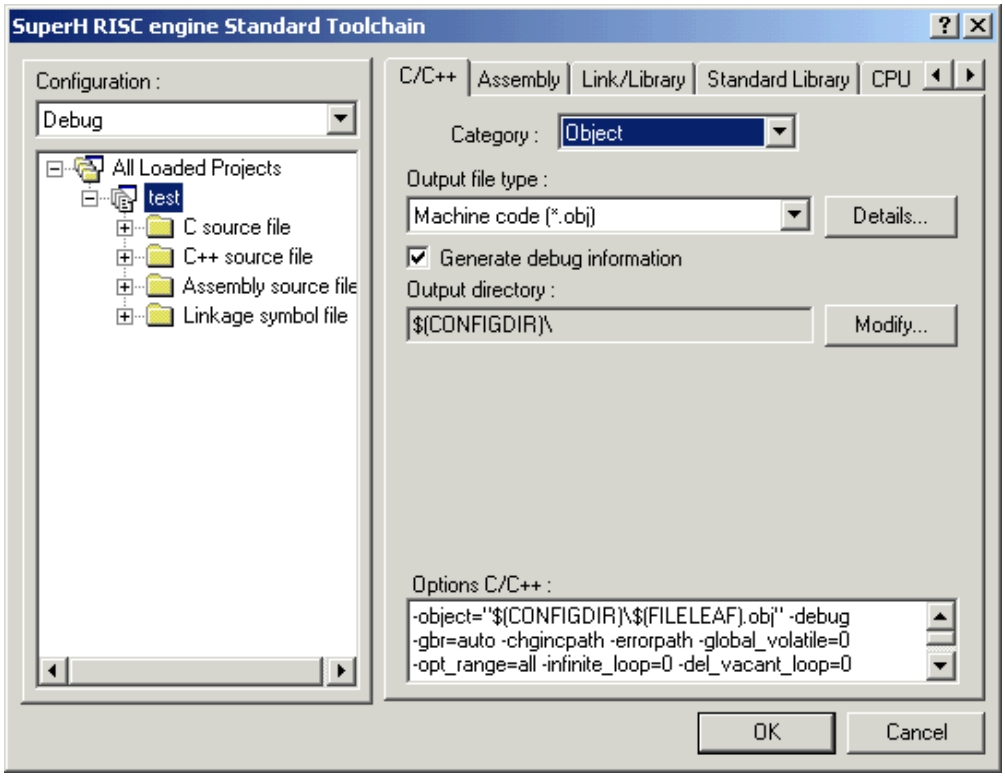


图 4.3 类别 (Category):[目标 (Object)] 对话框

单击 [详细资料 (Details)…] 将会打开“优化详细资料”(Optimize details) 对话框。

(a) “代码生成”(Code generation) 标签

表 4.3 “优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
段：	SEction = <sub>[,…]
程序段 (P)	<sub> : Program = <段名称>
常数段 (C)	<sub> : Const = <段名称>
数据段 (D)	<sub> : Data = <段名称>
未初始化的数据段 (B)	<sub> : Bss = <段名称>
	Default: (p=P, c=C, d=D, b=B)
模板：	Template =
	{ None | Static | Used |
	ALl | AUto }
将字符串数据存储在：	STring = { Const | Data }
除法子选项：	DIvision = Cpu [= { Inline | Runtime }]
不使用 FPU 指令	IFUnc
在无条件转移后对齐标签	ALIGN16
16/32 字节数据边界：	ALIGN32
	NOALign

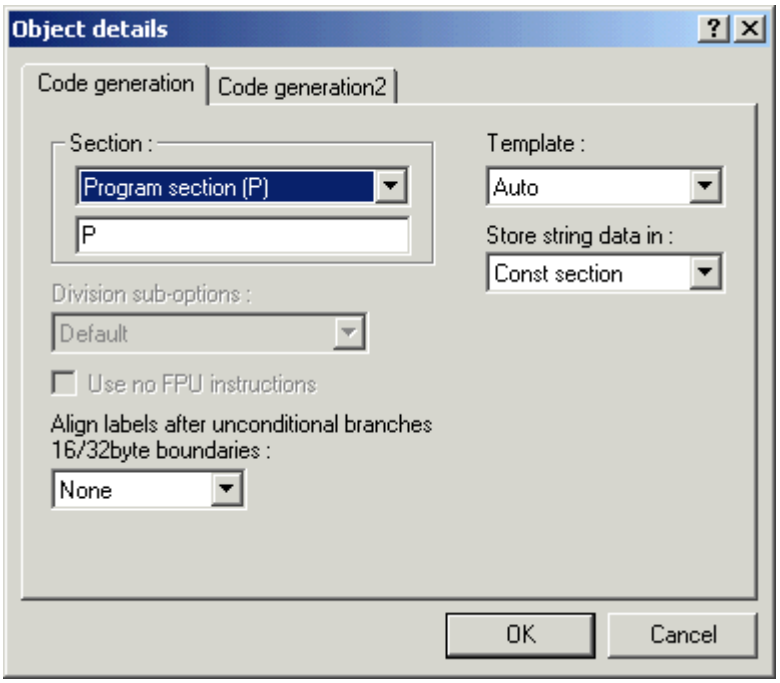


图 4.4 “代码生成”(Code Generation) 标签对话框

(b) “代码生成 2”(Code generation2) 标签

表 4.4 “优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
地址声明：	<ABS> = <sub>[,…] <ABS> : { ABs16 | ABS20 | ABS28 | ABS32 } <sub> : { Program | Const | Data | Bss | Run | All }
TBR 指定：	TBR [= <段名称>]
部署变量：	STUff=<sub>[,...] <sub>: {Bss|Data|Const}
未初始化变量的顺序：	BSs_order=<sub> <sub>: {DEClaration|DEFinition}

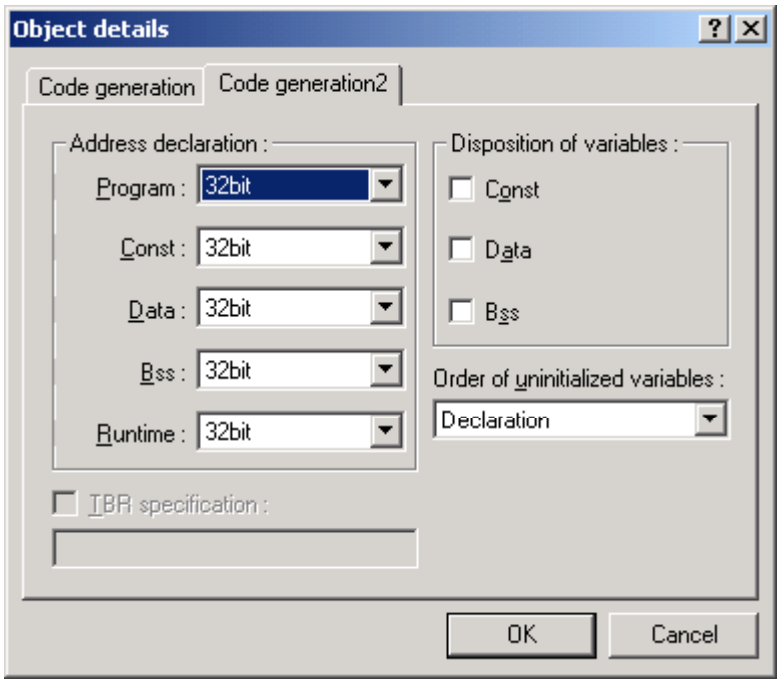


图 4.5 “代码生成 2”(Code Generation2) 标签对话框

(3) 类别 (Category):[列表 (List)]

表 4.5 类别 (Category):[列表 (List)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
生成列表文件	Listfile [= <文件名>] / NOListfile
制表符大小：	SHow = <sub>[,…] <sub> : Tab = { 4 | 8 }
内容：	SHow = <sub>[,…]
目标列表	<sub> : Object / NOObject
统计数据	<sub> : STatistics / NOSTatistics
源代码列表	<sub> : SOurce / NOSOurce
包括扩展后	<sub> : Include / NOInclude
宏扩展后	<sub> : Expansion / NOExpansion

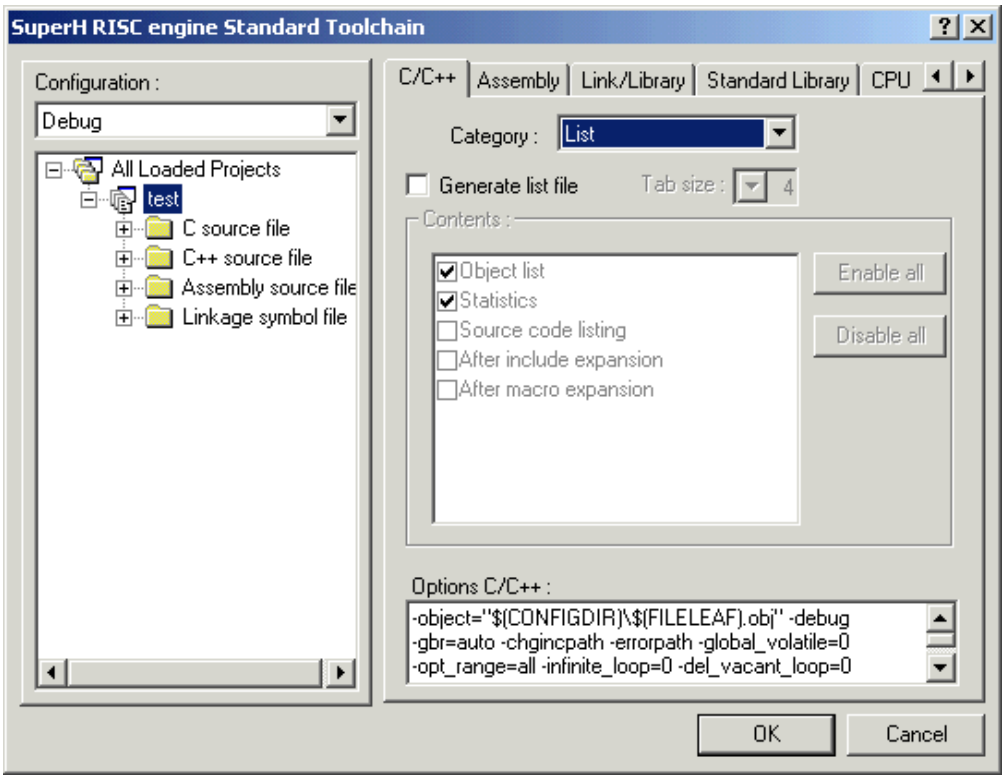


图 4.6 类别 (Category):[列表 (List)] 对话框

同时指定 -nolist 和 -show 选项时，-nolist 选项将拥有优先级。

(4) 类别 (Category):[优化 (Optimize)]

表 4. 6 类别 (Category):[优化 (Optimize)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
优化	OPtimize = 1 / OPtimize = 0
速度或大小：	
优化速度	SPeed
优化大小	SIze
优化速度和大小	NOSPeed
为模块间优化生成文件	Goptimize
优化对外部变量的存取：	MAP = <文件名>
Gbr 相对操作：	GBr = { Auto | User }
未对齐的移动：	Unaligned = { Inline | Runtime }
切换语句：	CAse = { Ifthen | Table }
移位运算：	SHIft = { Inline | Runtime }
转移代码开发：	BLOckcopy = { Inline | Runtime }

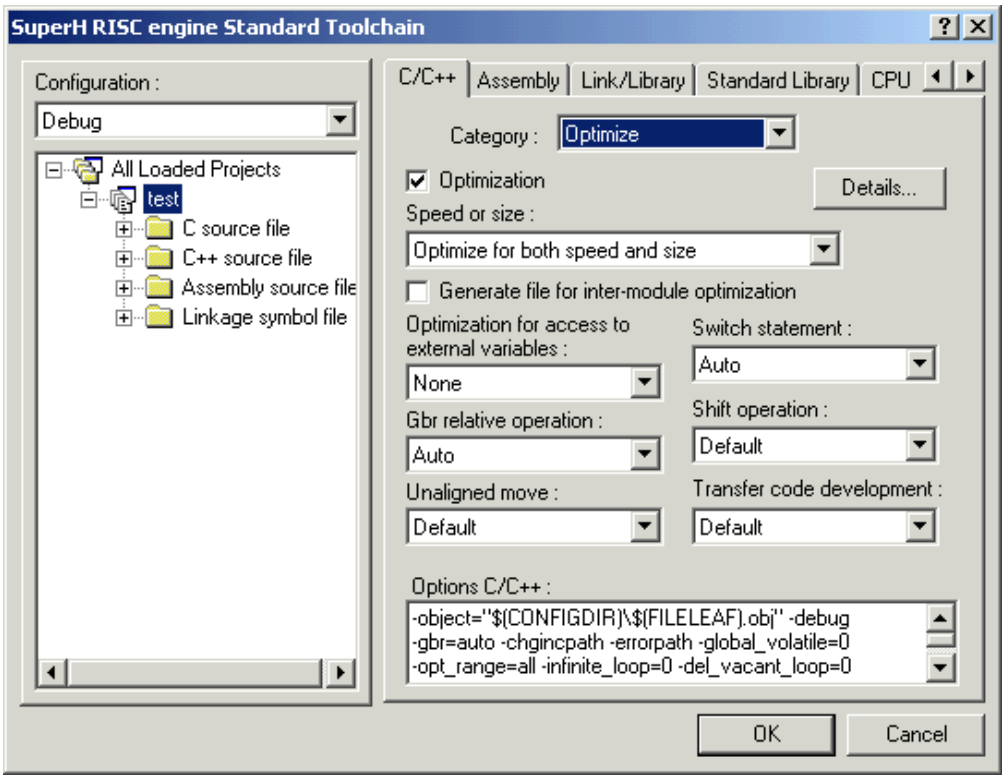


图 4.7 类别 (Category):[优化 (Optimize)] 对话框

- 对于“速度”(Speed) 或“大小”(Size) 选项，选择“优化速度和大小”(Optimize for both speed and size)。

单击 [详细资料 (Details)…] 将会打开“优化详细资料”(Optimize details) 对话框。

在 V.7.0.06 中添加的选项必须在此对话框中指定。

(a) “内联”(Inline) 标签

表 4.7 “优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
内联	-
内联文件路径：	File_inline = <文件名>[,…]
自动内联扩展：	INLine [= <数值>] / NOINLine

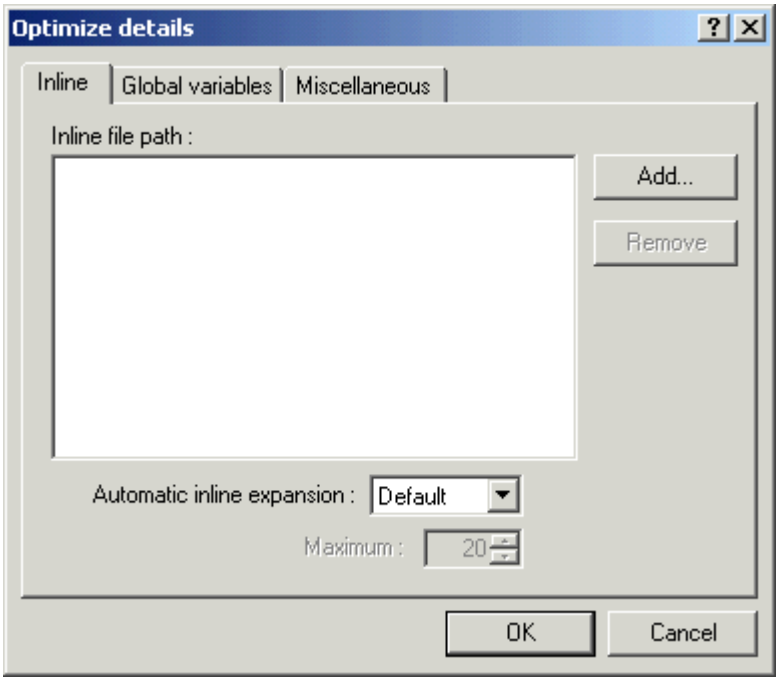


图 4.8 “内联”(Inline) 标签 对话框

(b) “全局变量”(Global Variables) 标签

表 4.8 “优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
级别：	-
内容：	
将全局变量视为符合易失性标准来处理	GLOBAL_Volatile = 1 / GLOBAL_Volatile = 0
在无穷循环之前删除到全局变量的赋值	INFinite_loop = 1 / INFinite_loop = 0
指定优化的范围：	OPT_Range = { All | NOLoop | NOBlock }
将寄存器分配到全局变量：	
禁用	GLOBAL_Alloc = 0
允许	GLOBAL_Alloc = 1
默认	-
传播符合常数标准的变量：	
禁用	CONST_Var_propagate = 0
允许	CONST_Var_propagate = 1
默认	-
预设指令：	
禁用	SChedule = 0
允许	SChedule = 1
默认	-

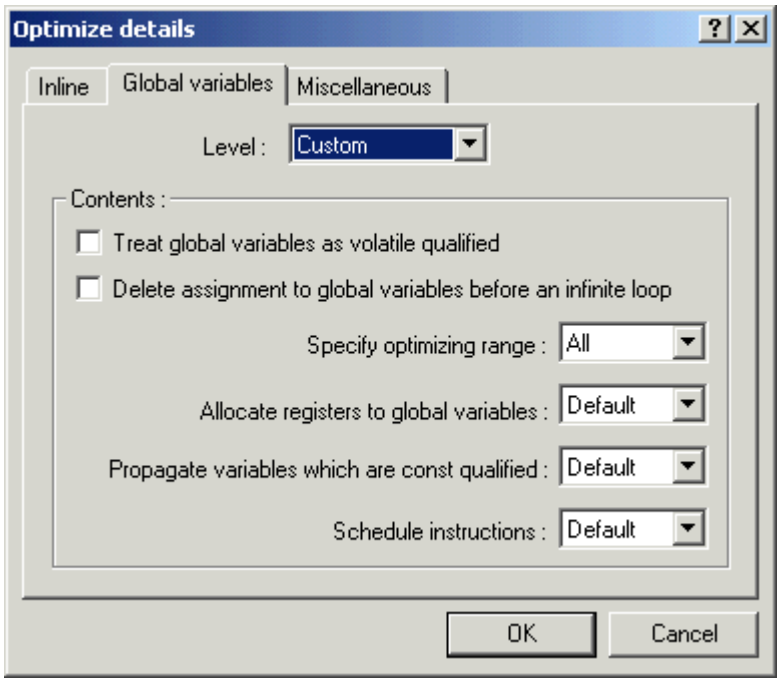


图 4.9 “全局变量”(Global Variables) 标签对话框

通过设定级别，可以全体控制外部变量的优化。

级别 1 (Level1)
禁用外部变量的优化。
gloal_volatile = 1
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 0

级别 2 (Level2)
优化外部变量且在转移范围内（包括循环）不进行易失性指定。
gloal_volatile = 0
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 1

级别 3 (Level3)
优化所有外部变量而不进行易失性指定。
gloal_volatile = 0
opt_range = all
infinite_loop = 0
global_alloc = 1
const_var_propagate = 1
schedule = 1

定制 (Custom)
用户根据程序指定外部变量的优化。

(c) “杂项”(Miscellaneous) 标签

表 4.9 “杂项”(Miscellaneous) 标签对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
删除空的循环	DEL_vacant_loop = 1 / DEL_vacant_loop = 0
指定最大的解开因数：	MAX_unroll = <numeric value> : 1 至 32
将常数值装入为：	
内联	CONST_Load = Inline
文字	CONST_Load = Literal
默认	-
将寄存器分配到结构/联合成员：	STRUCT_Alloc = 1 / STRUCT_Alloc = 0
软件流水线技术：	SOftpipe

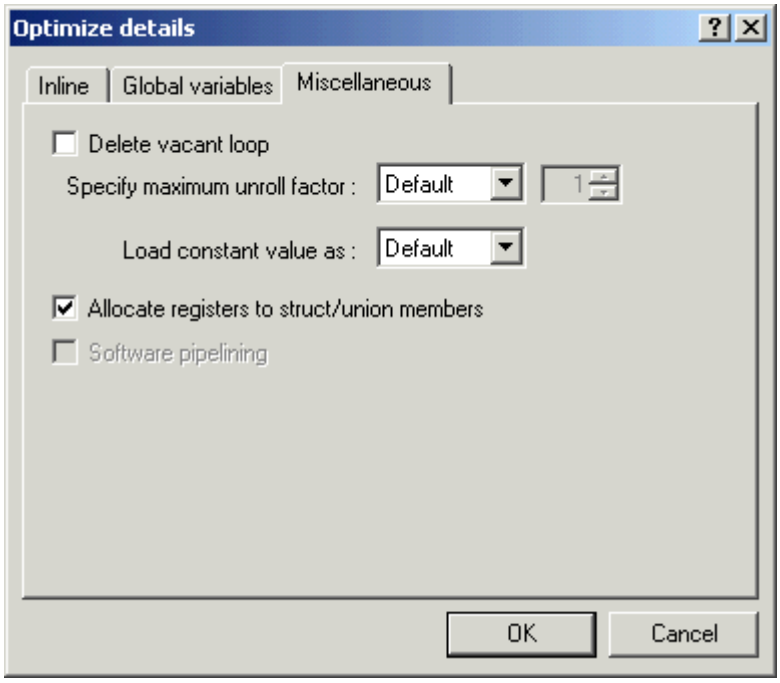


图 4.10 “杂项”(Miscellaneous) 标签对话框

(5) 类别 (Category):[其他 (Other)]

表 4.10 类别 (Category):[其他 (Other)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
杂项：	
对照 EC++ 语言规格	ECpp
对照 DSP-C 语言规格	DSpc
允许注解嵌套	COMment = Nest / COMment = NONest
Callee 保存/恢复 MACH	Macsave = 1 / Macsave = 0
和 MACL 寄存器（若使用）	
保存/恢复 SSR 及 SPC 寄存器	SAve_cont_reg = 0 / SAve_cont_reg = 1
扩展返回值到 4 字节	RTnext / NORTnext
解开循环	LOop / NOLOop
近似浮点常数除法	APproxdiv
避免非法 SH7055 指令	PAtch = 7055
使用双精度数据时更改 FPSCR 寄存器	FPScr = Safe / FPScr = Aggressive
将循环条件视为符合易失性标准来处理	Volatile_loop
使枚举大小为最小	AUto_enum
将浮点常数当作定点常数处理	FIXED_Const
将 1.0 视为定点类型的最大数字	FIXED_Max
定点乘法后删除类型转换	FIXED_Noround
使用 DSP 重复循环	REPeat
允许寄存器声明	ENAble_register declaration
更严格遵循 ansi 规格	STRIct_ansi
将整数除法改成浮点	FDIv

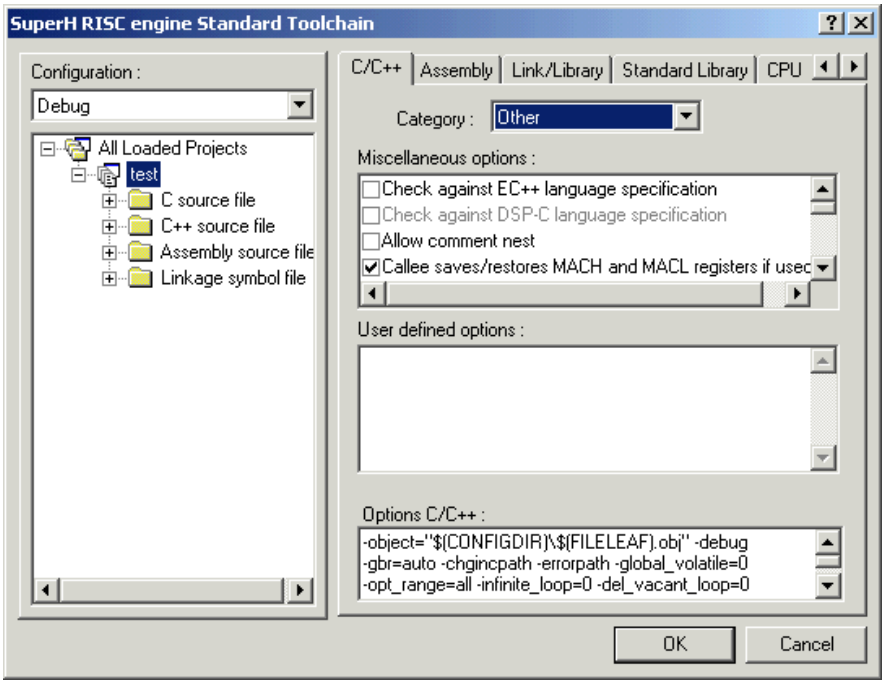


图 4.11 类别 (Category):[其他 (Other)] 对话框

4.1.2 汇编选项

从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 对话框中选取“汇编”(Assemby) 标签。

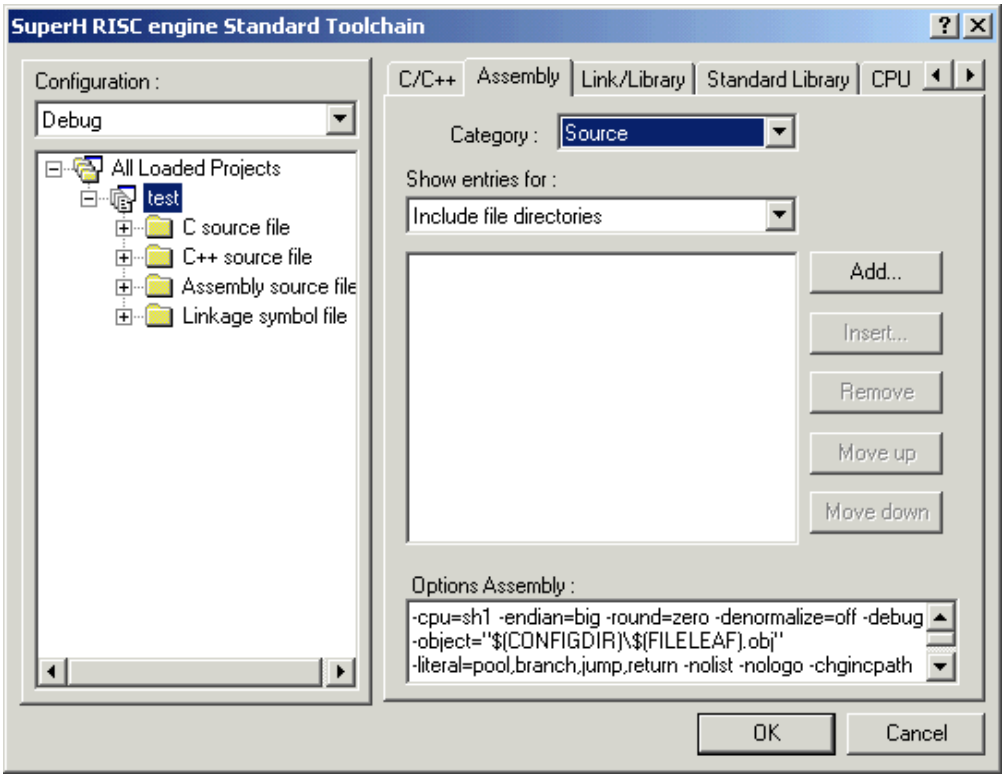


图 4.12“汇编”(Assembly) 标签对话框

(1) 类别 (Category):[源 (Source)]

表 4.11 类别 (Category):[源 (Source)] 中的项目和汇编程序选项 (Assembler Option) 之间的对应

对话框	选项
显示有关项目：	
包含文件目录	Include = <路径名称>[,…]
定义	DEFine = <sub>[, …]
	<sub> :
	<替换符号> = "<字符串文字>"
预处理程序变量	ASsignA = <sub>[, …]
	<sub> :
	<变量名称> = <整数常数>
	ASsignC = <sub>[, …]
	<sub> :
	<变量名称> = "<字符串文字>"

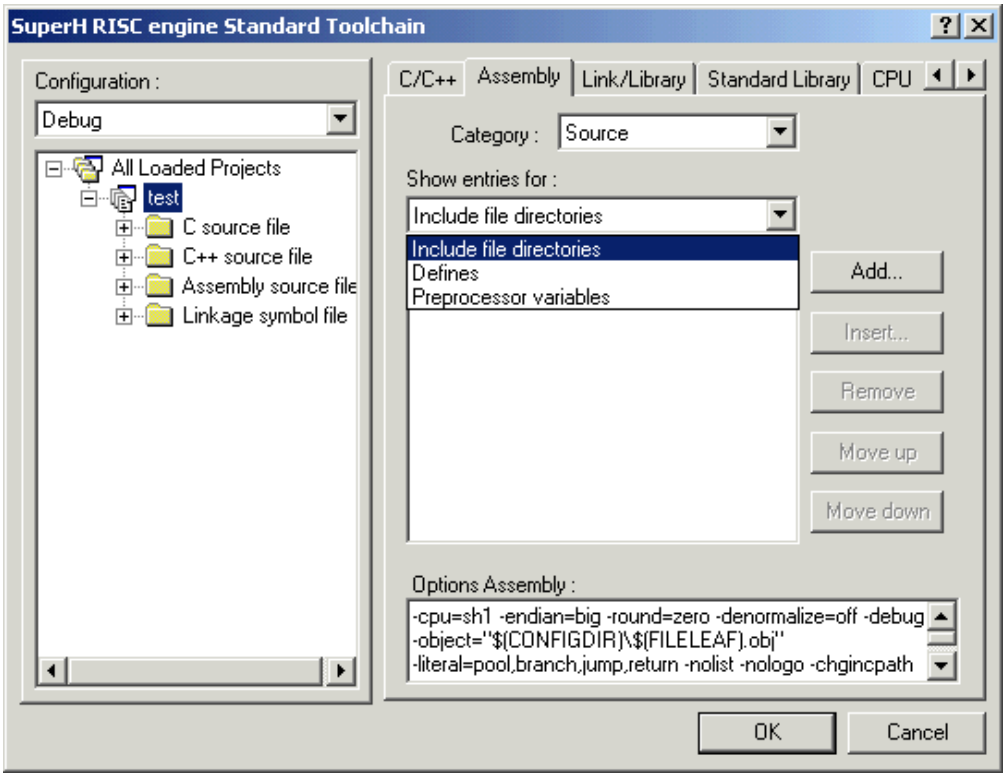


图 4.13 类别 (Category):[源 (Source)] 对话框

(2) 类别 (Category):[目标 (Object)]

表 4.12 类别 (Category):[目标 (Object)] 中的项目和汇编程序选项 (Assembler Option) 之间的对应

对话框	选项
调试信息：	
默认	-
具有调试信息	Debug
没有调试信息	NODebug
在预处理后生成汇编源文件	EXPand [= <输出文件名>]
在以下项目后面生成文字库：	LITERAL = <point>[,…]
.POOL 指令	<点> : Pool
BRA、BRAF	<点> : Branch
JMP	<点> : Jump
RTS、RTE	<点> : Return
选取位移大小：	DIspsize = { 4 | 12 }
输出文件目录：	Object [= <输出文件名>] / NOObject

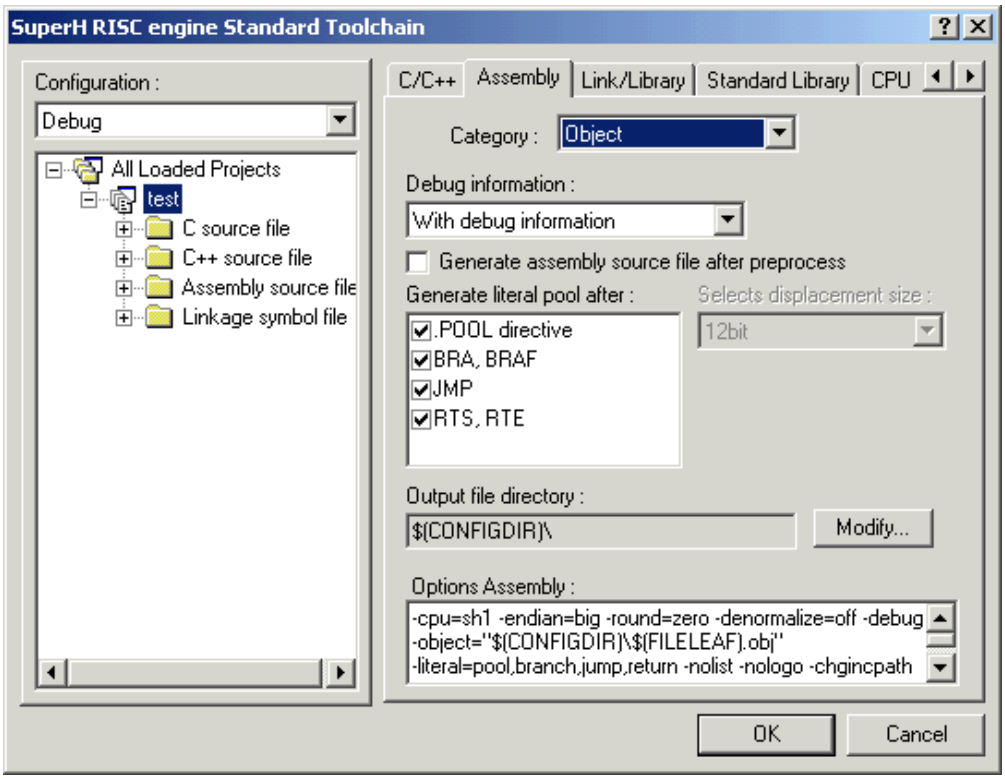


图 4.14 类别 (Category):[目标 (Object)] 对话框

(3) 类别 (Category):[列表 (List)]

表 4.13 类别 (Category):[列表 (List)] 中的项目和汇编程序选项 (Assembler Option) 之间的对应

对话框	选项
生成列表文件	LISt [= <output file name>] / NOLISt
源程序：	
默认	-
显示	SOurce
不显示	NOSOurce
交叉参考：	
默认	-
显示	CRoss_reference
不显示	NOCRoss_reference
段：	
默认	-
显示	SEction
不显示	NOSEction
源程序列表的内容：	
内容	
默认/显示/不显示	- / SHow [= <项目>[,…]] / NOSHow [= <项目>[,…]]
状态	
条件	<项目> : CONditionals
定义	<项目> : Definitions
调用	<项目> : CAlls
扩展	<项目> : Expansions
代码	<项目> : CODe
制表符大小	<项目> : TAB = { 4 / 8 }

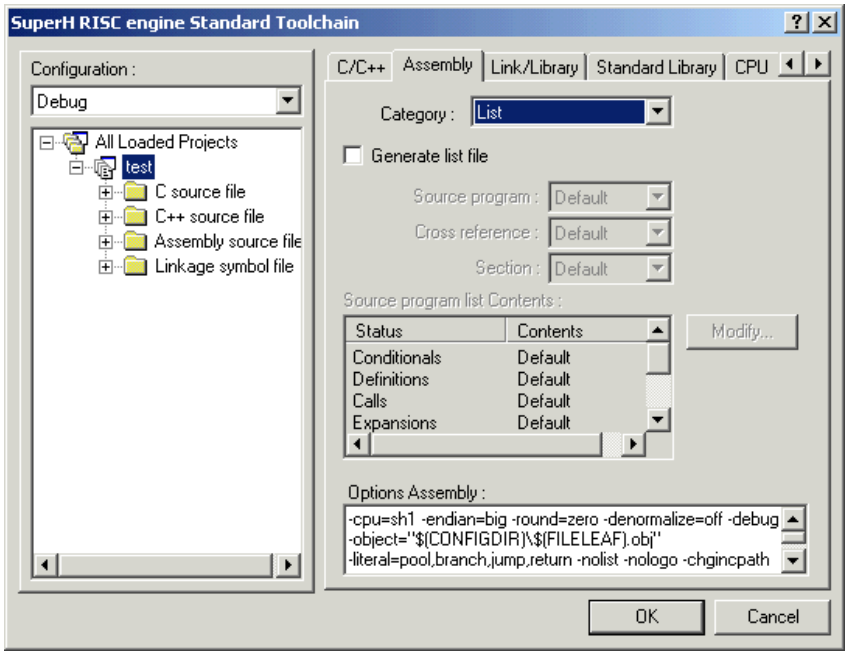


图 4.15 类别 (Category):[列表 (List)] 对话框

(4) 类别 (Category):[其他 (Other)]

表 4.14 类别 (Category):[其他 (Other)] 中的项目和汇编程序选项 (Assembler Option) 之间的对应

对话框	选项
杂项:	
自动生成立即值的文字库	AUTO_literal
移除未被参考的外部符号	Exclude / NOExclude
检查有权限的指令	CHKMd
检查 LDTLB 指令	CHKTlb
检查高速缓冲指令	CHKCache
检查 DSP 指令	CHKDsp
检查 FPU 指令	CHKFpu
检查 8 字节对齐	CHKAlign8

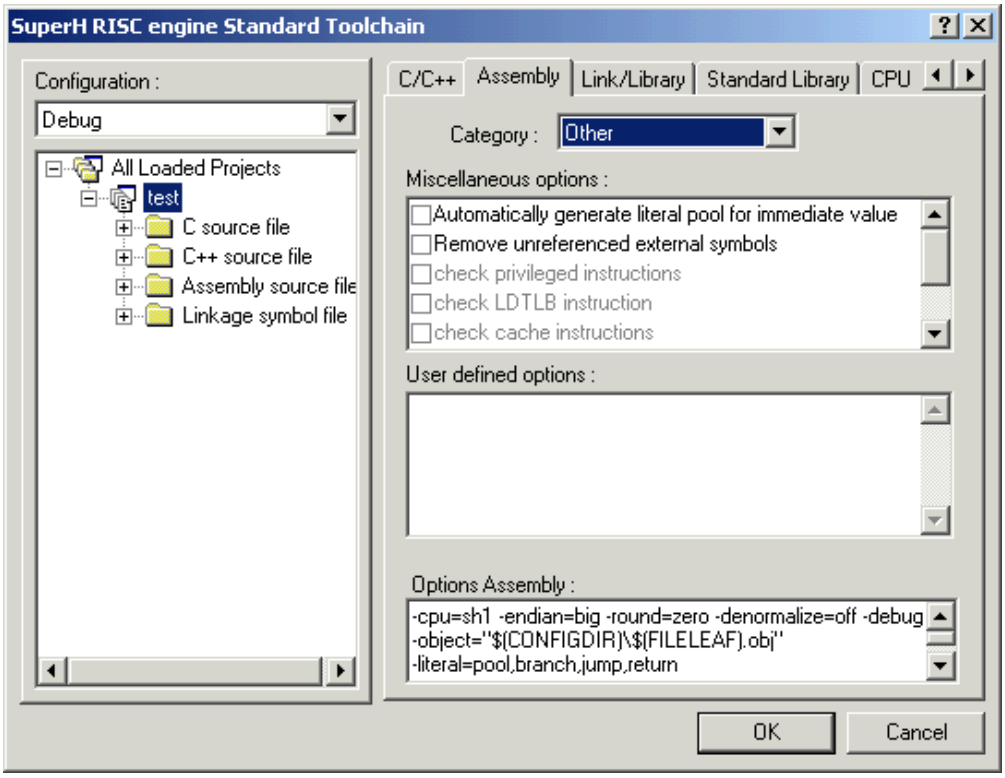


图 4.16 类别 (Category):[其他 (Other)] 对话框

4.1.3 优化连接编辑程序选项

从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 对话框中选取“连接/程序库”(Link/Library) 标签。

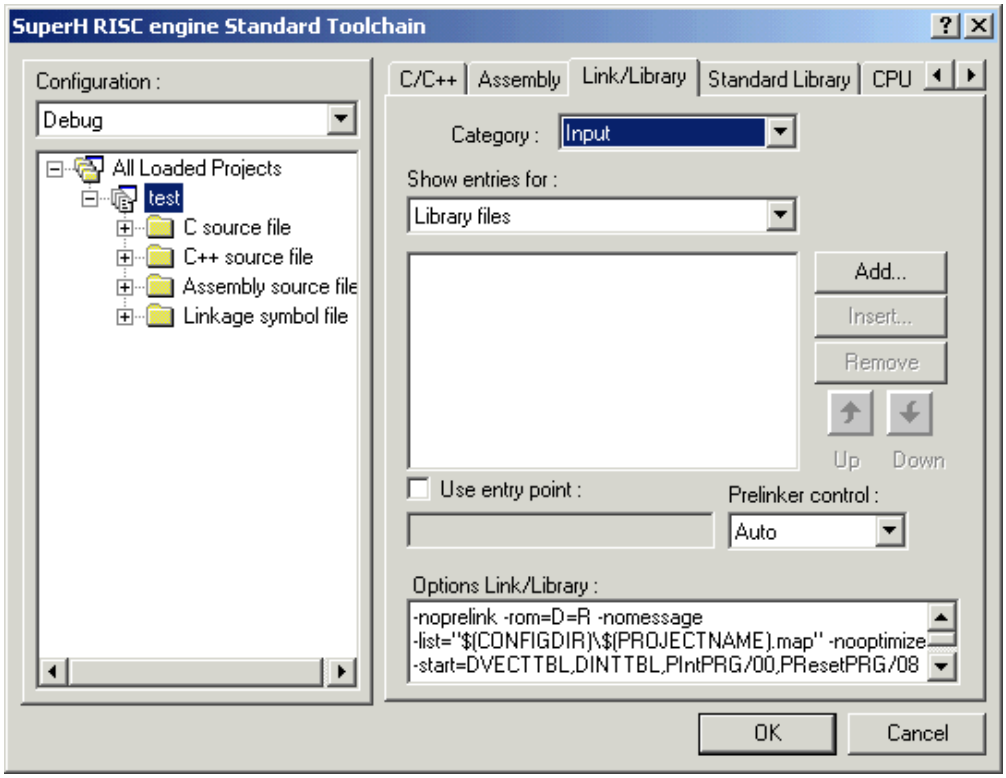


图 4.17“连接/程序库”(Link/Library) 标签对话框

(1) 类别 (Category):[输入 (Input)]

表 4.15 类别 (Category):[输入 (Input)] 中的项目和连接编辑程序选项 (Linkage Editor Option) 之间的对应

对话框	选项
显示有关项目：	
程序库文件	LIBrary = <文件名>[,…]
可再定位的文件和目标文件	Input = <sub> [{,|Δ}...] <sub> : <文件名>[(<模块名称>[,...])]
二进制文件	Binary = <sub>[,...] <sub> : <文件名>(<段名称> [:<边界对齐>] [,<符号名称>])
定义	DEFine = <sub>[,...] <sub> : <符号名称> = { <符号名称> | <数值> }
使用入口点：	ENTry = { <符号名称> | <地址> }
预连接程序控制项：	
自动	NOPRElink
跳过预连接程序	NOPRElink
运行预连接程序	-
*1 包括在工程中的文件不需要明确地添加；这会在连接未编译或汇编的目标时指定。	

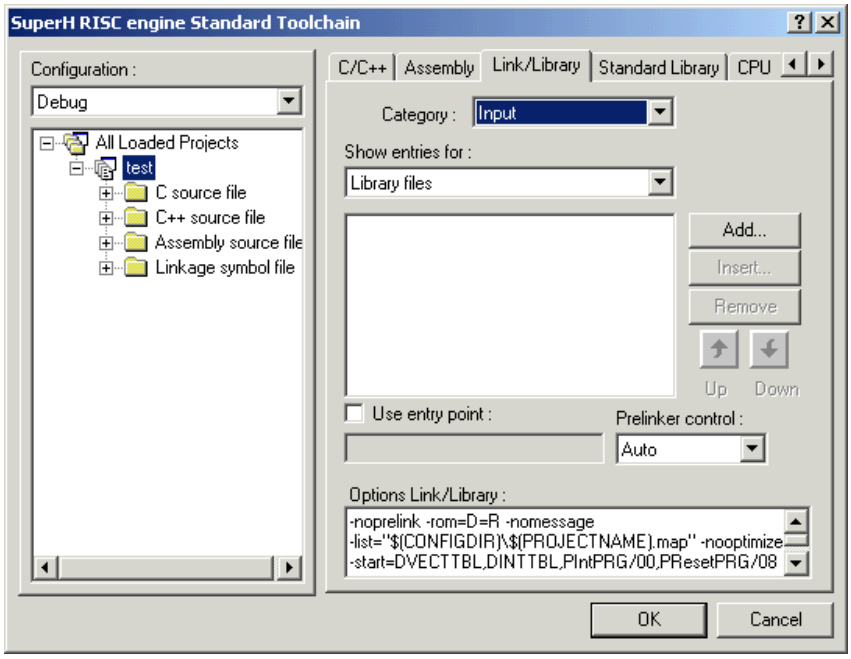


图 4.18 类别 (Category):[输入 (Input)] 对话框

(2) 类别 (Category):[输出 (Output)]

表 4.16 类别 (Category):[输出 (Output)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
输出文件的类型：	
绝对 (ELF/DWARF)	FOrm = Absolute
绝对 (SYSROF)	FOrm = Absolute
可再定位的	FOrm = Relocate
系统程序库	FOrm = Library = S
用户程序库	FOrm = Library = U
通过绝对来输出十六进制	FOrm = Hexadecimal
通过绝对来输出 S 类型	FOrm = Stype
通过绝对来输出二进制	FOrm = Binary
数据记录标头：	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }
调试信息：	
无	NODEBug
以输出到装入模块的形式	DEBug
以个别调试文件 (*.dbg) 的形式	SDebug
显示有关项目：	
输出文件路径/消息	
ROM 到 RAM 的映像段	ROm = <sub>[,…] <sub> : <ROM 段名称>=<RAM 段名称>
分隔的输出文件	OUtput = <sub>[,…] <sub> : <文件名>[=<输出范围>] <输出范围> : { <起始地址> - <终止地址> |
输出填充数据	<段名称>[: ...] }
压制的信息级别消息	SPace = [<数值>] NOMessage [= <sub>[,…]] / Message <sub> : <错误代码> [- <错误代码>]
生成映像文件	MAp [= <文件名>]

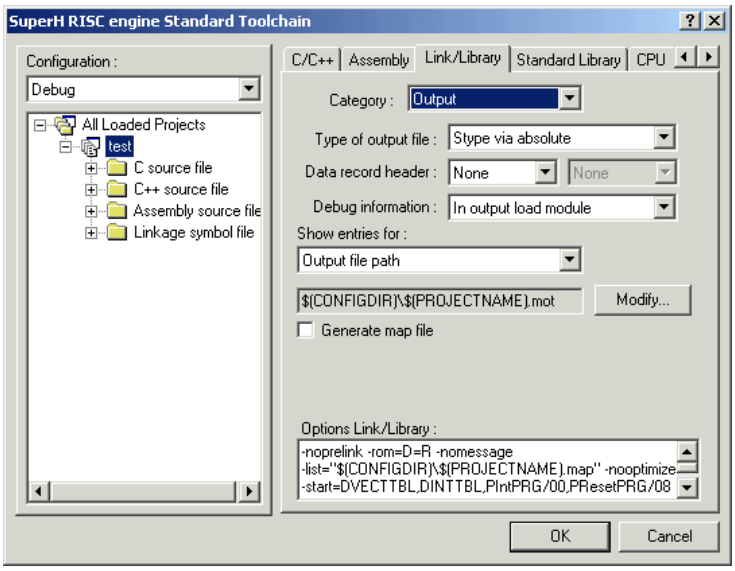


图 4.19 类别 (Category):[输出 (Output)] 对话框

(3) 类别 (Category):[列表 (List)]

表 4.17 类别 (Category):[列表 (List)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
生成列表文件	LISt [= <文件名>] / -
内容：	SHow [= <sub>[,...]]
显示符号	<sub> : SYmbol
显示参考	<sub> : Reference
显示段	<sub> : SEction
显示交叉参考	<sub> : Xreference

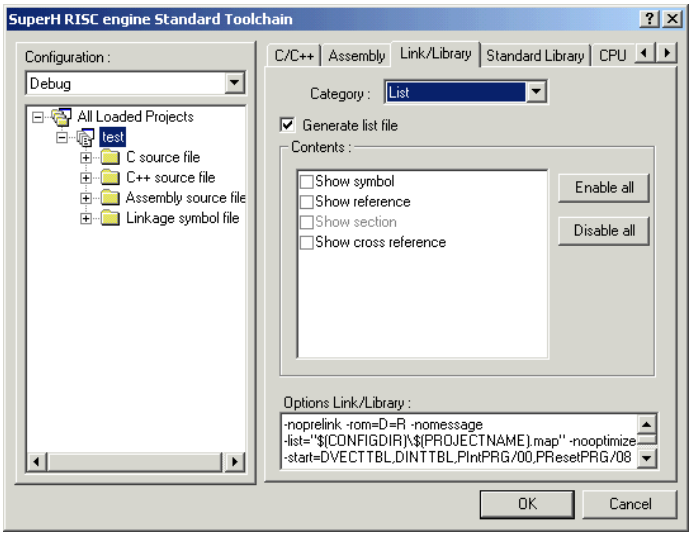


图 4.20 类别 (Category):[列表 (List)] 对话框

(4) 类别 (Category):[优化 (Optimize)]

表 4.18 类别 (Category):[优化 (Optimize)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
显示有关项目：	
优化项目	
优化：	OPtimize [= <sub>[,...]]
全部	<sub> : STring_unify,SYmbol_delete, Variable_access,Register, SAMe_code,SHort_format, Function_call,Branch
速度	<sub> : SPeed
安全	<sub> : SAFe
定制	Optionally specify the folloing:
统一字符串	<sub> : STring_unify
删除死码	<sub> : SYmbol_delete
再分配寄存器	<sub> : Register
删除相同的代码	<sub> : SAMe_code
优化转移	<sub> : Branch
无	NOOPtimize
删除的大小：	SAMESize = <大小> (default:sames=1e)
包含配置文件：	PROfile = <文件名>
高速缓存大小	CAchesize = Size = <大小>, Align = <行大小> (default:ca=s=8,a=20)
显示有关项目：	
禁止项目	
死码的删除	SYmbol_forbid = <符号名称>[,...]
相同代码的删除	SAMECode_forbid = <函数名称>[,...]
存储器分配在	Absolute_forbid = <地址> [+ <大小>] [,...]

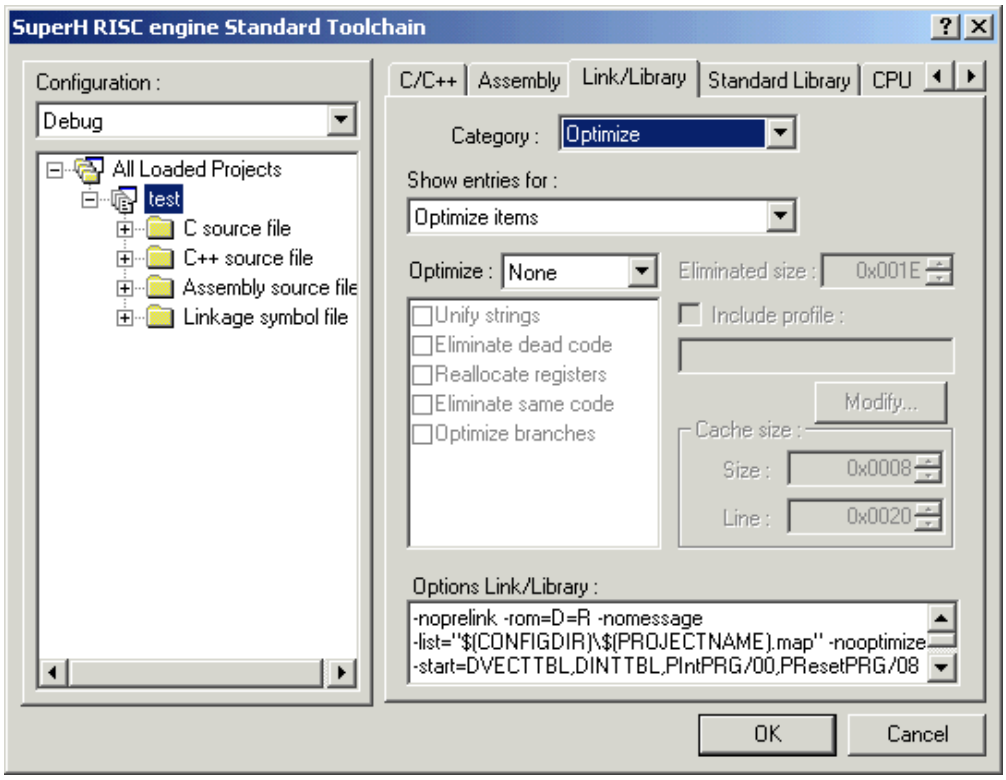


图 4.21 类别 (Category):[优化 (Optimize)] 对话框

(5) 类别 (Category):[段 (Section)]

表 4.19 类别 (Category):[段 (Section)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
显示有关项目：	
段	STAR t= <sub>[,...] <sub> : <段名称> [{: | ,} <段名称>[,...]] [/<地址>]
符号文件	FSymbol = <段名称>[,...]

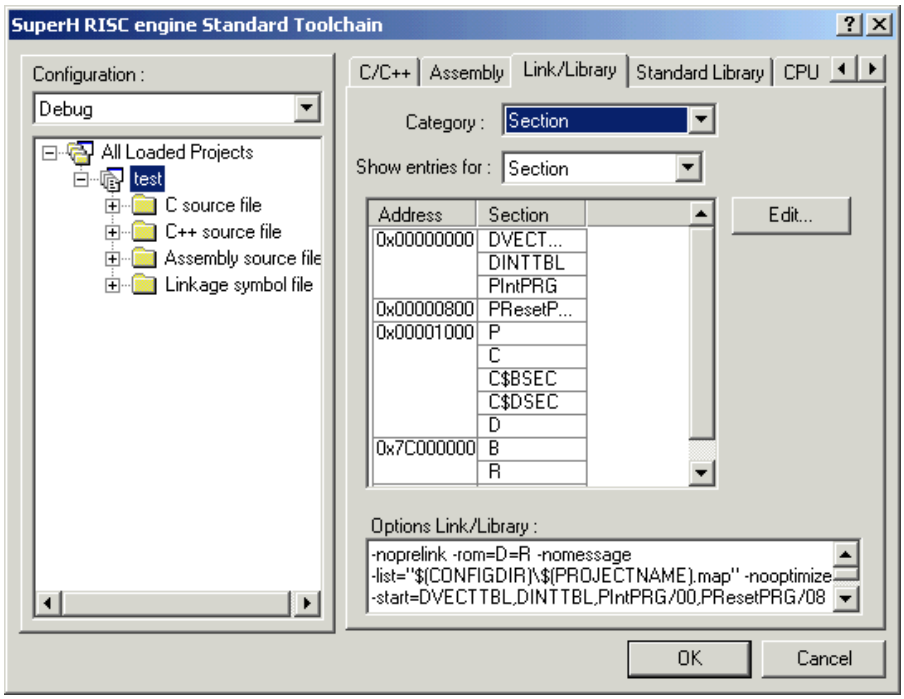


图 4.22 类别 (Category):[段 (Section)] 对话框

附加的段可以使用 [编辑(Edit)] 按钮指定。

指定的段名称和地址可以使用 [添加(Add)] 按钮添加。

已经指定的段名称和地址可以使用 [修改(Modify)] 按钮编辑。

多个段可以使用 [新的覆盖(New Overlay)] 按钮分配到同个地址。

已经指定的段可以使用 [删除(Remove)] 按钮删除。

段的顺序可以使用 [上(UP)] 和/或 [下(DOWN)] 按钮改变。

如果前页的对话框内容写入“连接编辑程序”(Linkage Editor) 的子命令文件：

```
START RSTHandler
START INITHandler,VECTTBL,INITTBL,IntPRG/800
START RestPRG/1000
START P,C,D/2000
START RAM_sct1:RAM_sct2/F00000
  START B,R/7F000000
  START Stack/7FFFFBF0
```

在图 4.19 中有相关显示。

RAM_sct1 和 RAM_sct2 将会分配到同一个段。

注意： 要获取有关创建“连接编辑程序”(Linkage Editor) 的子命令文件的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)。

(6) 类别 (Category)：[验证 (Verify)]

表 4.20 类别 (Category):[验证 (Verify)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
CPU 信息检查：	
不检查	-
检查	CPu = {<cpu 信息文件名> | {<存储器类型>} = <地址范围>[,…]} <存储器类型> : {ROm | Ram | XROm | XRAm | YROm | YRAm } <地址范围> : <起始地址> - <终止地址>|
使用 CPU 信息文件	CPu = <cpu 信息文件名>

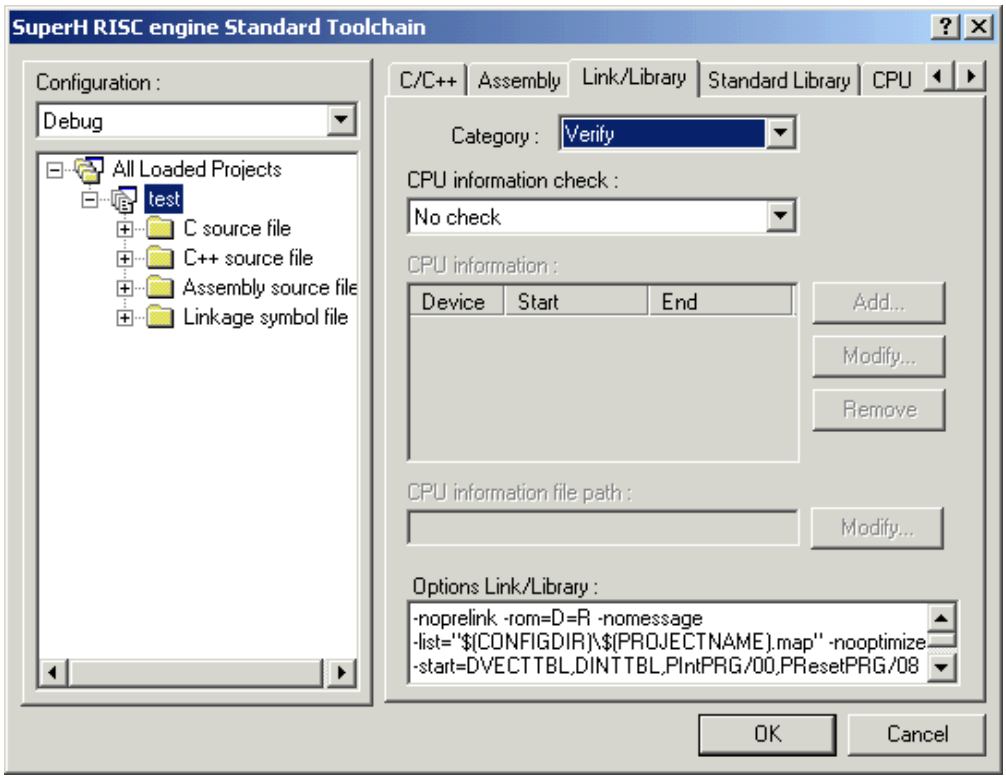


图 4.23 类别 (Category):[验证 (Verify)] 对话框

(7) 类别 (Category)： [其他 (Other)]

表 4.21 类别 (Category):[其他 (Other)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
杂项：	
始终在结束部分输出 S9 记录	S9
堆栈信息输出	STACk
压缩调试信息	COmpress / NOCOmpress
连接期间使用低存储	MEMory = [High | Low]
用户定义的选项：	
绝对/可再定位的/程序库	
十六进制/S 类型/二进制	

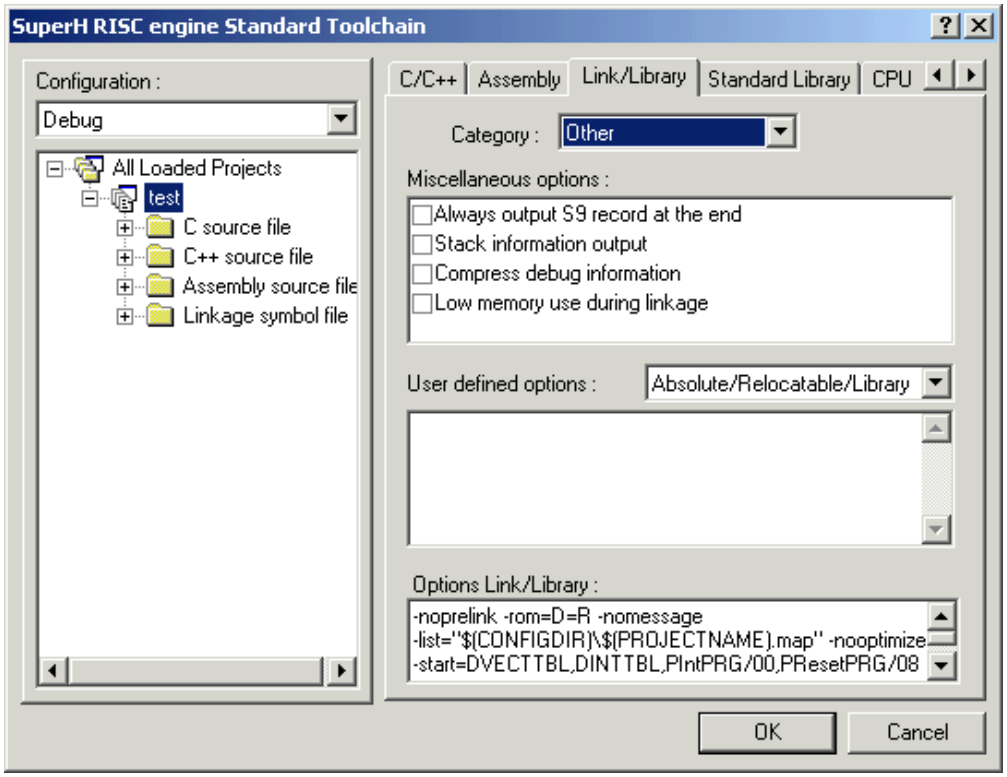


图 4.24 类别 (Category):[其他 (Other)] 对话框

(8) 类别 (Category):[子命令文件 (Subcommand file)]

表 4.22 类别 (Category):[子命令文件 (Subcommand file)] 中的项目和优化连接编辑程序选项 (Optimizing Linkage Editor Options) 之间的对应

对话框	选项
使用外部子命令文件	SUbcommand = <文件名>

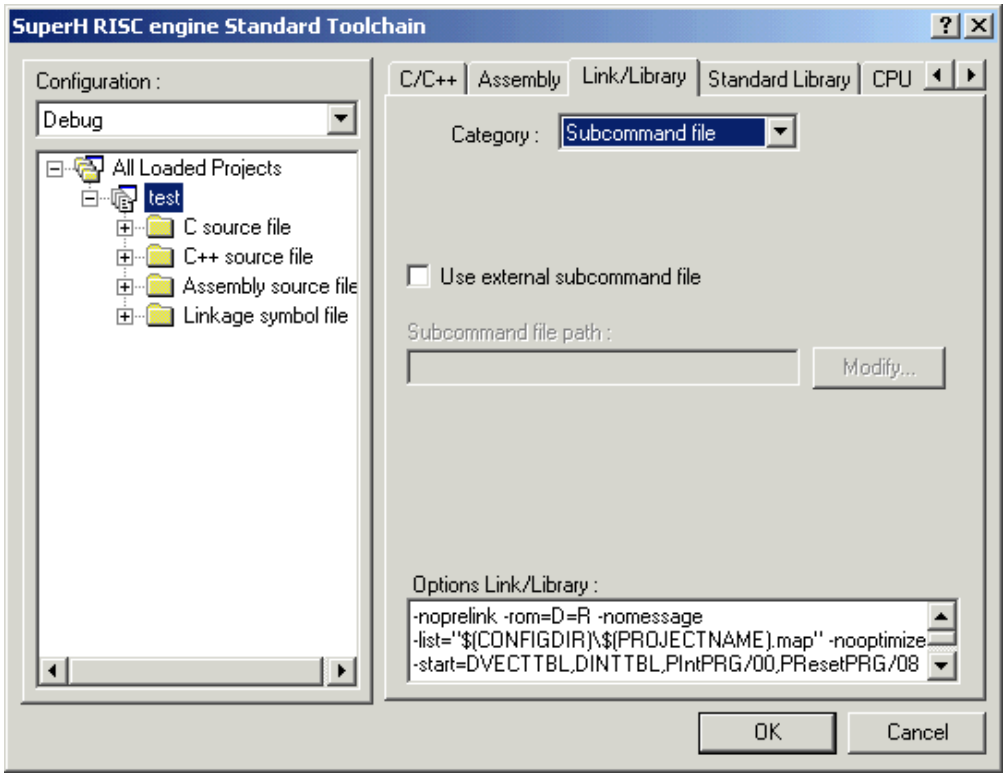


图 4.25 类别 (Category):[子命令文件 (Subcommand file)] 对话框

4.1.4 标准程序库生成程序选项

从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 对话框中选取“标准程序库”(Standard Library) 标签。

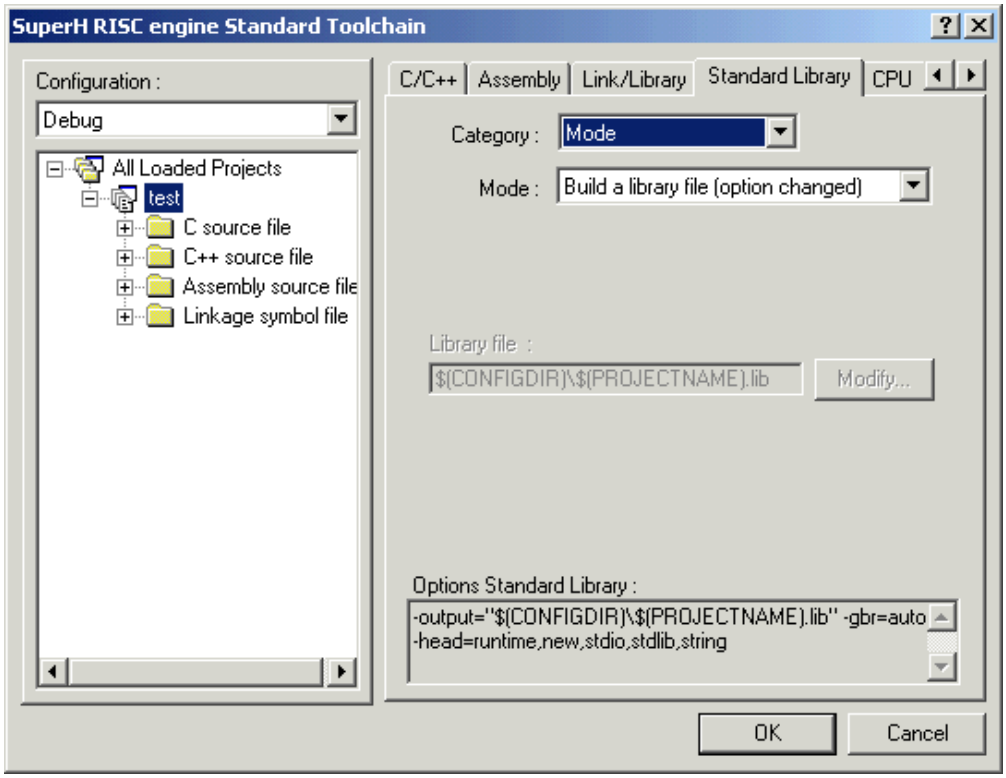


图 4.26“标准程序库”(Standard Library) 标签对话框

(1) 类别 (Category)：[模式 (Mode)]

表 4.23 类别 (Category):[模式 (Mode)] 中的项目和功能之间的对应

对话框	功能
模式：	
创建程序库文件（任何时候）	创建当前的标准程序库。
创建程序库文件（选项更改）	在选项更改时，创建当前的标准程序库。
使用现有的程序库文件	连接现有的标准程序库。
不添加程序库文件	不连接标准程序库。

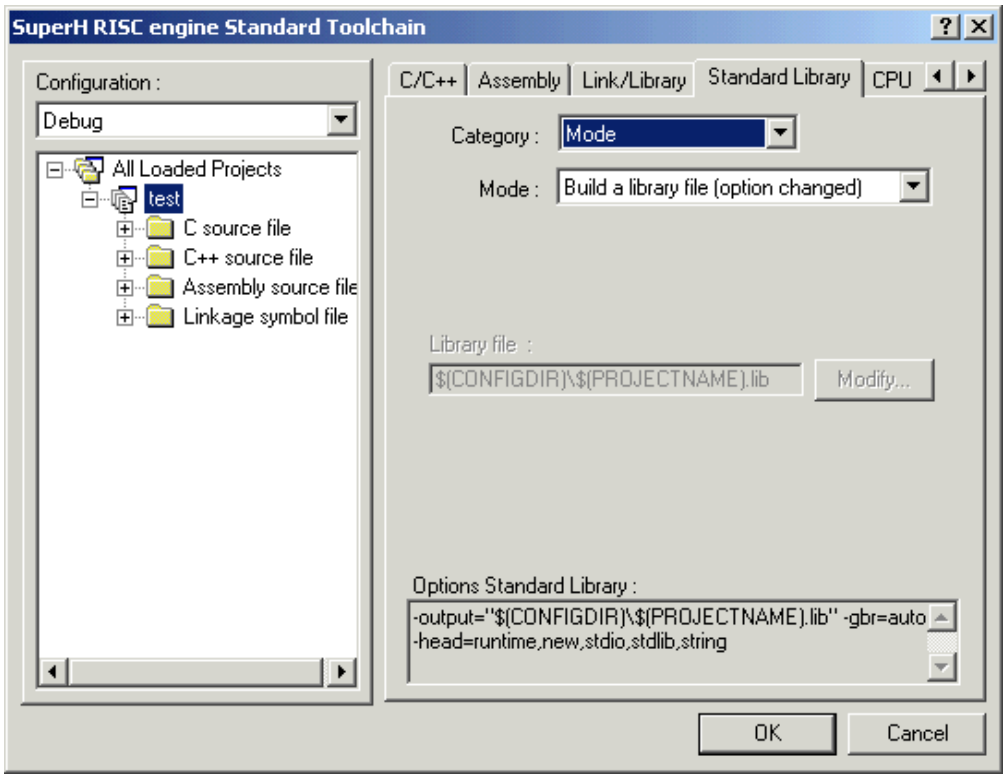


图 4.27 类别 (Category):[模式 (Mode)] 对话框

(2) 类别 (Category)：[标准程序库 (Standard Library)]

表 4.24 类别 (Category):[标准程序库 (Standard Library)] 中的项目和“标准程序库生成程序选项” (Standard Library Generator Options) 之间的对应

对话框	选项
类别：	Head = <sub>[,….]
运行时	<sub> : RUNTIME
新建	<sub> : NEW
ctype.h	<sub> : CTYPE
math.h	<sub> : MATH
mathf.h	<sub> : MATHF
stdarg.h	<sub> : STDARG
stdio.h	<sub> : STDIO
stdlib.h	<sub> : STDLIB
string.h	<sub> : STRING
ios(EC++)	<sub> : IOS
complex(EC++)	<sub> : COMPREX
string(EC++)	<sub> : CPPSTRING

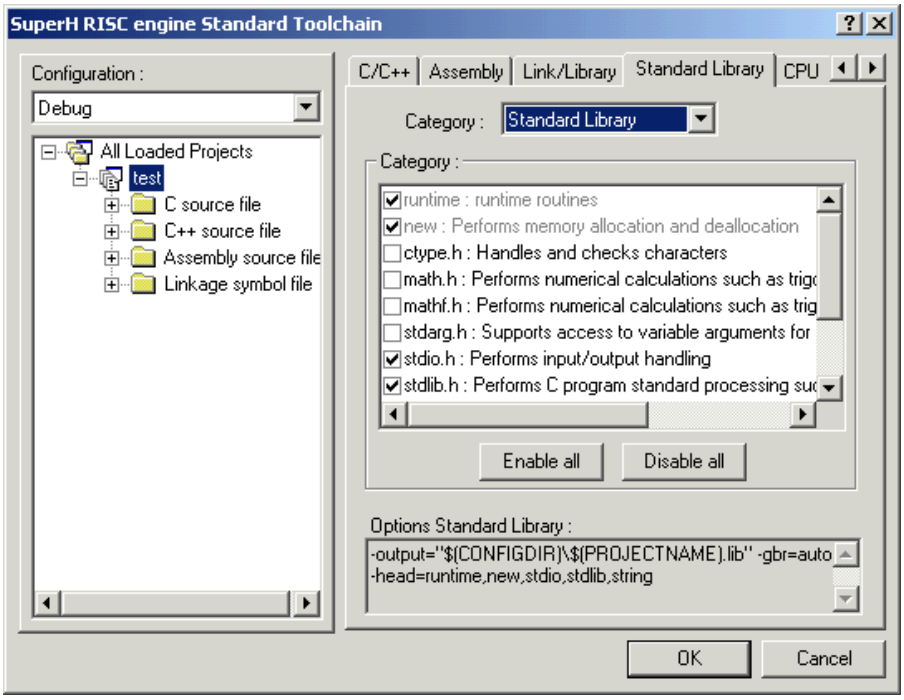


图 4.28 类别 (Category):[标准程序库 (Standard Library)] 对话框

(3) 类别 (Category)：[目标 (Object)]

表 4.25 类别 (Category):[目标 (Object)] 中的项目和选项之间的对应

对话框	选项
简单 I/O 函数	NOFLoat
生成可重入程序库	REent

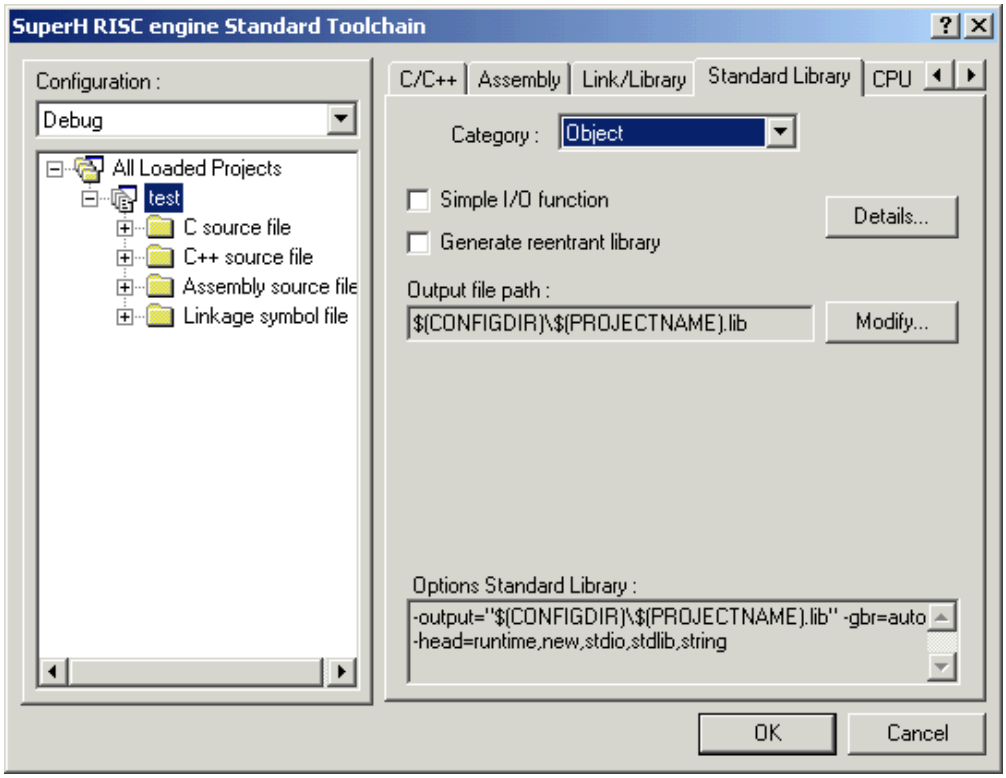


图 4.29 类别 (Category):[目标 (Object)] 对话框

单击 [详细资料 (Details)…] 将会打开“优化详细资料”(Optimize details) 对话框。

(a) “代码生成”(Code generation) 标签

表 4.26“优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
段	SEction = <sub>[,…]
程序段 (P)	<sub> : Program = <段名称>
常数段 (C)	<sub> : Const = <段名称>
数据段 (D)	<sub> : Data = <段名称>
未初始化的数据段 (B)	<sub> : Bss = <段名称>
	Default: (p=P, c=C, d=D, b=B)
将字符串数据存储在：	STring = { Const | Data }
除法子选项：	DIvision = Cpu [= { Inline | Runtime }]
不使用 FPU 指令：	IFUnc
在无条件转移后对齐标签	ALIGN16
16/32 字节数据边界：	ALIGN32
	NOALign
NOFLoat、REent： 标准程序库生成程序选项	
其他： 编译程序选项	

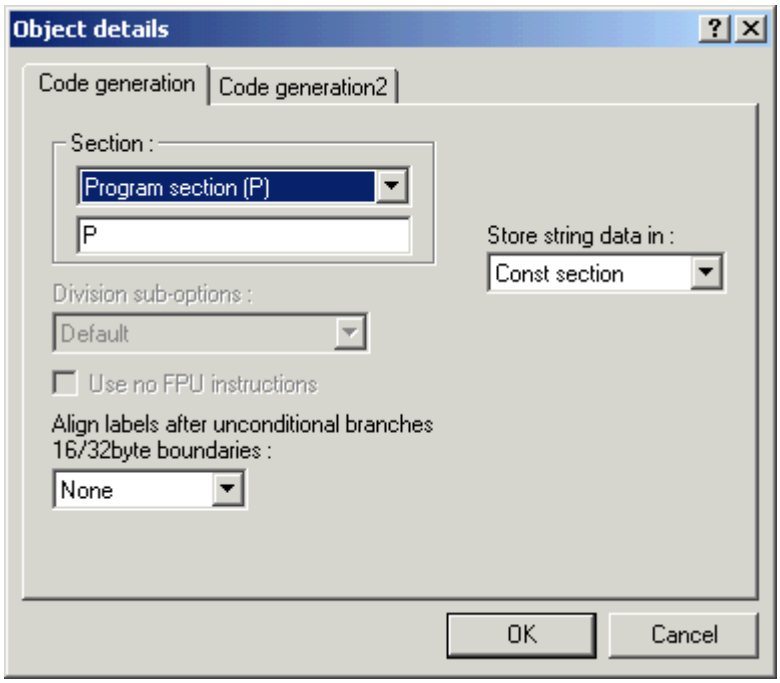


图 4.30 类别 (Category):[目标 (Object)] 对话框

(b) “代码生成 2”(Code generation2) 标签

表 4.27“优化详细资料”(Optimize details) 对话框中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
地址声明：	<ABS> = <sub>[,…] <ABS> : { ABs16 | ABS20 | ABS28 | ABS32 } <sub> : { Program | Const | Data | Bss | Run | All }
TBR 指定：	TBR [= <段名称>]

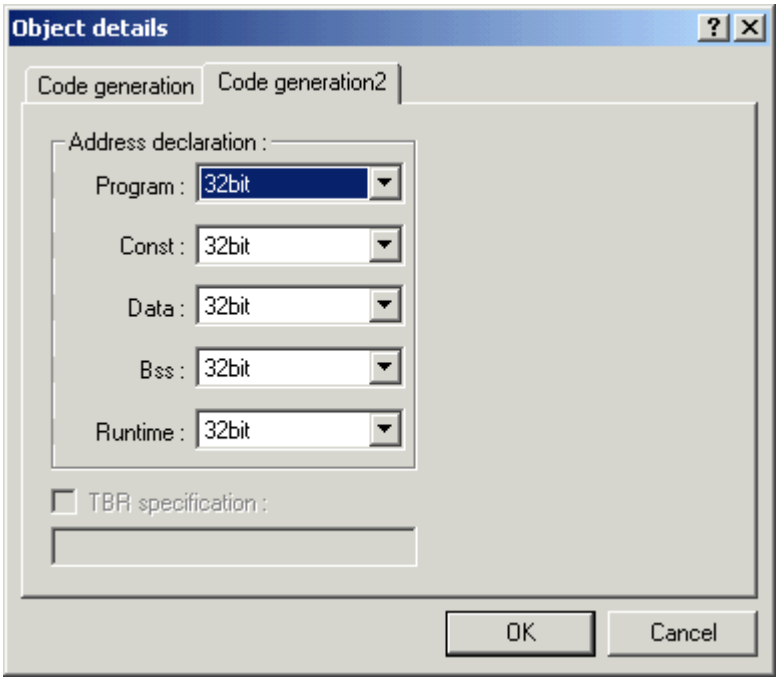


图 4.31 类别 (Category):[目标 (Object)] 对话框

(4) 类别 (Category):[优化 (Optimize)]

表 4.28 类别 (Category):[优化 (Optimize)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
优化	OPtimize = 1 / OPtimize = 0
速度或大小：	
优化速度	SPeed
优化大小	SIze
优化速度和大小	NOSPeed
为模块间优化生成文件	Goptimize
Gbr 相对操作：	GBr = { Auto | User }
未对齐的移动：	Unaligned = { Inline | Runtime }
自动内联扩展	INLine [= <数据>] / NOINLine
切换语句：	CAse = { Ifthen | Table }
移位运算：	SHIft = { Inline | Runtime }
转移代码开发：	BLOckcopy = { Inline | Runtime }

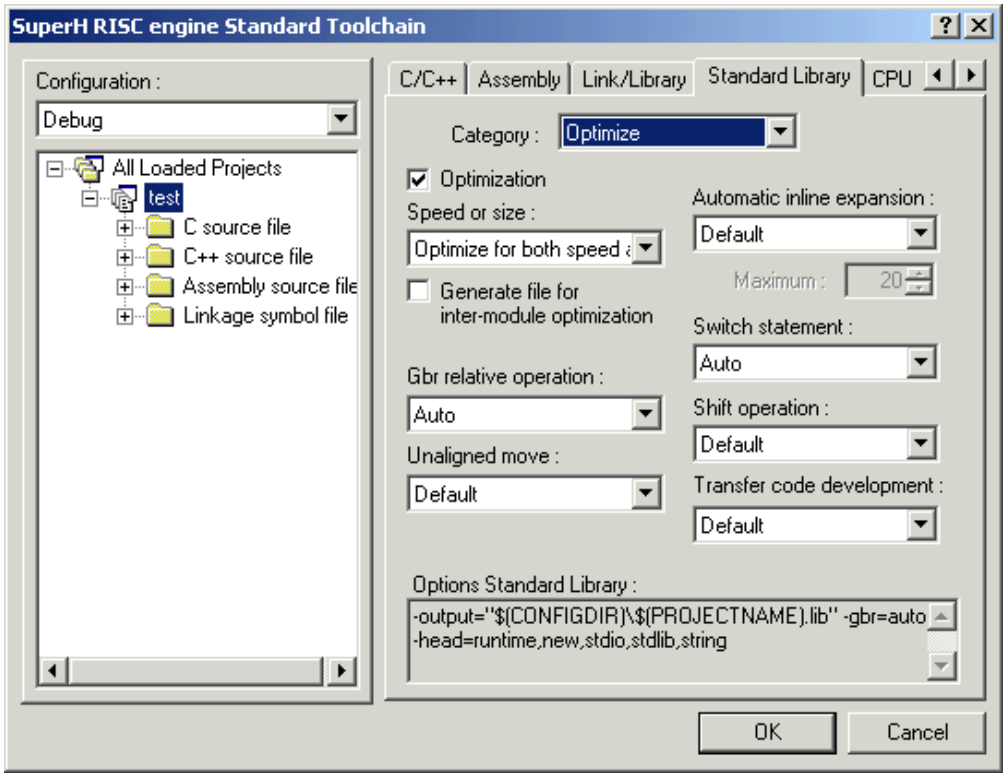


图 4.32 类别 (Category):[优化 (Optimize)] 对话框

- 对于“速度”(Speed) 或“大小”(Size) 选项，选择“优化速度和大小”(Optimize for both speed and size)。

(5) 类别 (Category):[其他 (Other)]

表 4.29 类别 (Category):[其他 (Other)] 中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
杂项：	
对照 EC++ 语言规格	ECpp
对照 DSP-C 语言规格	DSpc
保存/恢复 SSR 及 SPC 寄存器	SAve_cont_reg = { 0 | 1 }
扩展返回值到 4 字节	RTnext / NORTnext
解开循环	LOop / NOLOop
近似浮点常数除法	APproxdiv
避免非法 SH7055 指令	PAtch = 7055
使用双精度数据时更改 FPSCR 寄存器	FPScr = Safe / FPScr = Aggressive
将循环条件视为符合易失性标准来处理	Volatile_loop
使枚举大小为最小	AUto_enum
将浮点常数当作定点常数处理	FIXED_Const
将 1.0 视为定点类型的最大数字	FIXED_Max
定点乘法后删除类型转换	FIXED_Noround
使用 DSP 重复循环	REPeat
允许寄存器声明	ENAble_register
更严格遵循 ansi 规格	STRIct_ansi
将整数除法改成浮点	FDIv

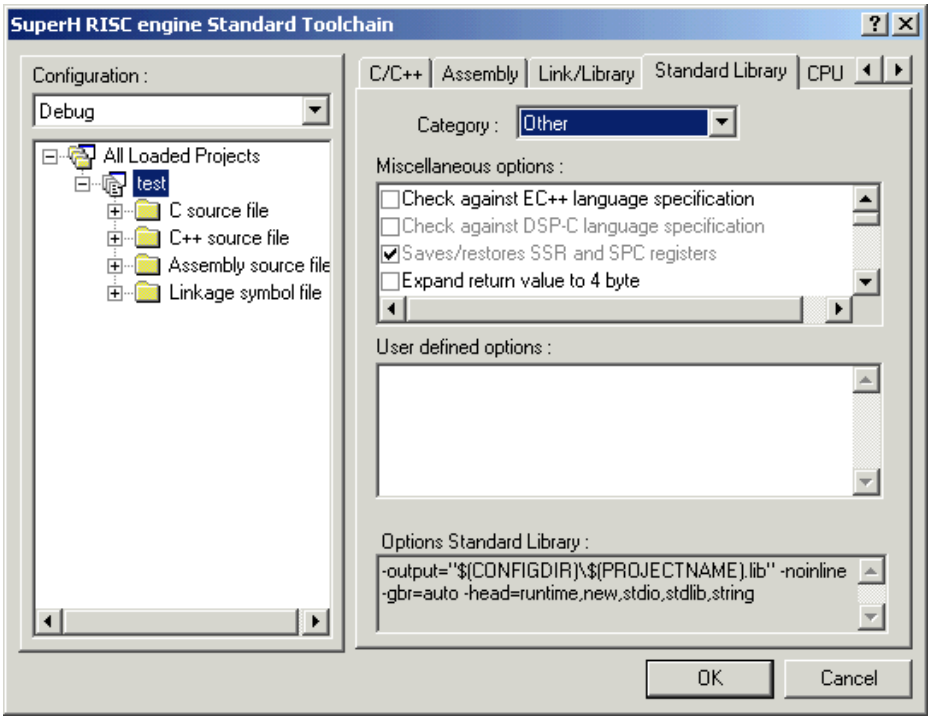


图 4.33 类别 (Category):[其他 (Other)] 对话框

4.1.5 CPU 选项

从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 对话框中选取“CPU”标签。

表 4.30 [CPU] 标签中的项目和编译程序选项 (Compiler Option) 之间的对应

对话框	选项
CPU：	CPu = { SH1 | SH2 | SH2E | SH2A | SH2AFPU | SH2DSP |SH3 | SH3DSP | SH4 | SH4A | SH4ALDSP }
除法：	DIvision = { Cpu = [= { Inline | Runtime }] | Peripheral | Nomask }
Endian：	ENdian = { Big | Little }
FPU：	Fpu = { Single | Double }
舍入：	Round = { Zero | Nearest }
允许非正常化数字为结果	DENormalize = ON / DENormalize = OFF
位置无关代码 (PIC)	Pic = 1 / Pic = 0
将双精度当作浮点处理	DOuble = Float
位字段的成员将从较低的位分配	Bit_order = { Left | Right }
对齐 struct, union 和 class 类型	PACK = 1 / PACK = 4
使用 C++ 的 try、throw 和 catch	EXception / NOEXception
允许/禁用运行时类型信息	RTTI = ON / RTTI = OFF

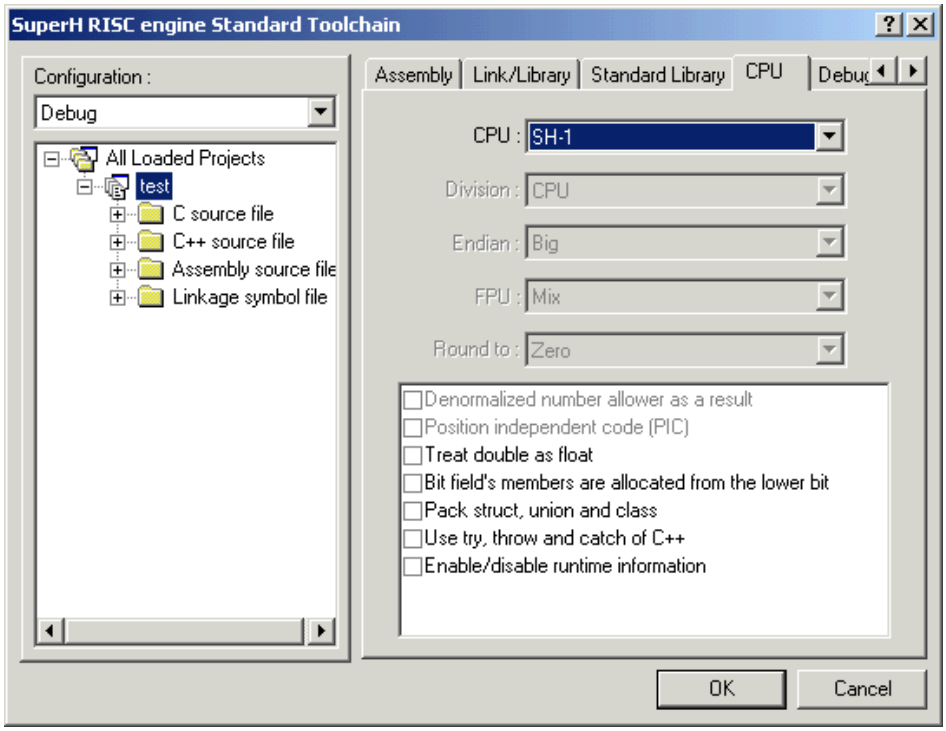


图 4.34 [CPU] 标签对话框

4.2 从“瑞萨集成开发环境”(Renesas Integrated Development Environment) 指定编译程序版本

这里为您说明在“瑞萨集成开发环境”(Renesas Integrated Development Environment) 中指定编译程序版本的方法。 编译程序版本可以通过升级“瑞萨集成开发环境”(Renesas Integrated Development Environment) 来指定。
如果将旧版本（如 HEW1.1 或 SH5.1B）中创建的工作空间在新版本（如 HEW3.01 或 SH9.0）中打开，下列对话框将会出现。

(1) 检查要升级的工程。
检查要升级的工程的名称。

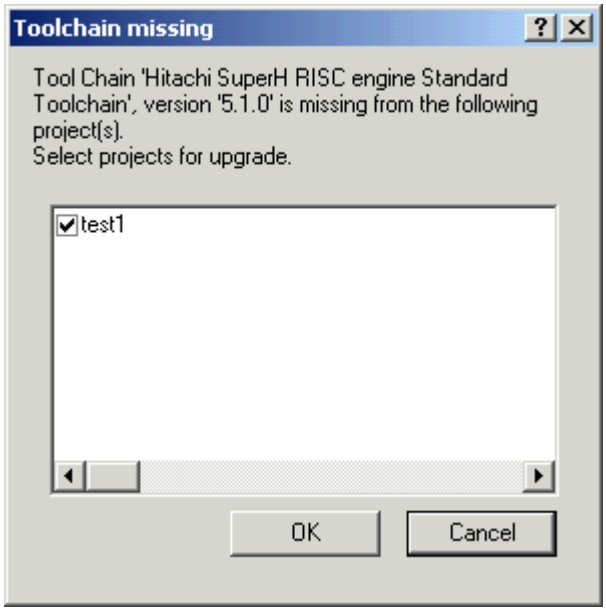


图 4.35 高性能嵌入式工作区 (High-performance Embedded Workshop)

(2) 指定编译程序版本

选取可以升级的编译程序版本。

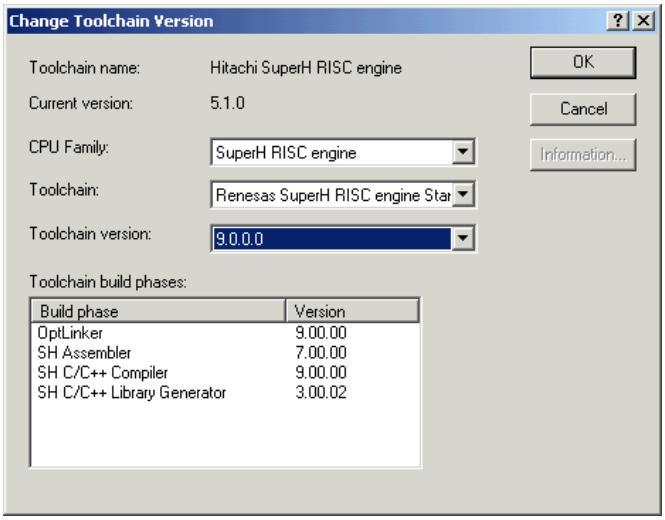


图 4.36 “更改工具链版本”(Change Toolchain Version) 对话框

(3) 确认信息

C/C++ 编译程序 7.1 或以上版本仅支持要输出之目标的 ELF/DWARF 文件格式。

文件格式将会在升级时更改为 ELF/DWARF 格式。如果当前的调试环境不支持 ELF/DWARF 格式，请将 ELF/DWARF 格式转换为升级后调试环境所支持的格式。

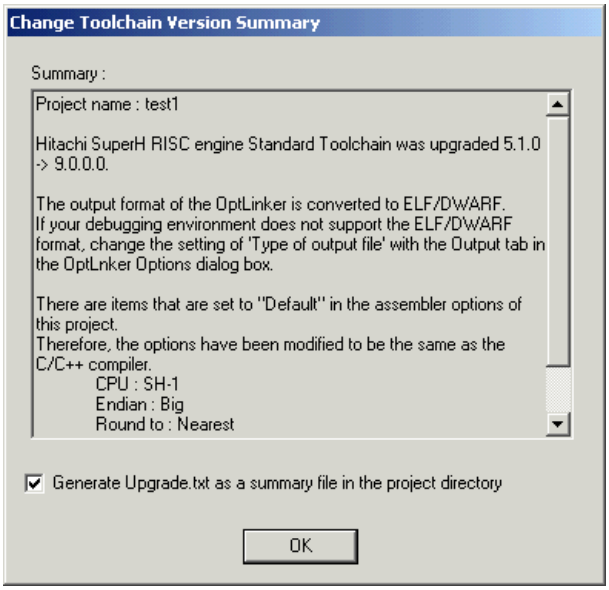


图 4.37“确认信息”对话框

SuperH RISC Engine C/C++编译程序应用笔记

有效的编程技术

第 5 节　　有效的编程技术

虽然 SuperH RISC engine C/C++ 编译程序提供了各种优化，但通过编程上的创新则可获得更佳的性能。

本节描述用户可尝试用于有效编程的建议技术。

评估程序的标准包括程序的执行速度和程序大小。

通过指定“-speed”编译选项，SuperH RISC engine C/C++ 编译程序可接受指令执行强调执行速度的优化。

以下为有效创建程序的规则。

(1) 增进执行速度的规则
执行速度同时由经常执行的语句和复杂的语句决定。 应找出这些语句，并加以特别处理来增进它们。

(2) 缩减程序大小的规则
为缩减程序大小，应使用公用代码来执行类似的处理，同时应修订复杂的函数。

因为编译程序优化的结果，实际执行速度有时会异于理论速度。 为增进性能，应透过使用各种技术，在编译程序中实际运行程序来检查性能。

本节显示的汇编语言扩展代码是使用以下命令行获得

shcΔ (C language file) Δ-code=asmcodeΔ-cpu=sh2

然而，“-cpu”选项可能会使 SH-1、SH-2、SH-2E、SH-3 和 SH-4 之间的汇编语言扩展代码相异。将来在编译程序中的改进以及其他更改可能造成汇编语言扩展代码的更改。

本节中显示的代码大小和执行速度值是从 SH-1、SH-2、SH-2A、SH2A-FPU、SH-2E、SH2-DSP (SH7065)、SH-3、SH3-DSP、SH-4、SH4A 和 SH4AL-DSP 测量所得。 表 5.1 显示编译时的 CPU 选项。

表 5.1 CPU 选项列表

编号	CPU 类型	CPU 选项
1	SH-1	-cpu=sh1
2	SH-2	-cpu=sh2
3	SH-2A	-cpu=sh2a
4	SH2A-FPU	-cpu=sh2afpu
5	SH-2E	-cpu=sh2e
6	SH-DSP(SH7065):	-cpu=sh2
7	SH-3	-cpu=sh3
8	SH3-DSP	-cpu=sh3
9	SH-4	-cpu=sh4Δ-fpu=single
10	SH-4A	-cpu=sh4Δ-fpu=single
11	SH4AL-DSP	-cpu=sh4aldsp

对于 SH-2A、SH2A-FPU、SH3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP，高速缓存遗漏仅在一些测量中考虑。 外部存储器的存取周期数被假设为 1。

表 5.2 列出有效的编程技术。

表 5.2 有效编程技术列表

编号	函数	ROM 效率	RAM 效率	执行速度	参考的节
1	局部变量（数据大小）	O	-	O	5.1.1
2	全局变量（符号）	O	-	O	5.1.2
3	数据大小（乘法）	O	-	O	5.1.3
4	数据结构	O	-	O	5.1.4
5	数据对齐	-	O	-	5.1.5
6	初始值和常数类型	-	O	-	5.1.6
7	局部变量和全局变量	O	-	O	5.1.7
8	指针变量的使用	O	-	O	5.1.8
9	引用常数 (1)	O	-	-	5.1.9
10	引用常数 (2)	O	-	-	5.1.10
11	保持为常数的变量 (1)	-	-	-	5.1.11
12	保持为常数的变量 (2)	-	-	-	5.1.12
13	在模块中结合函数	O	-	O	5.2.1
14	使用指针变量的调用源函数	O	-	O	5.2.2
15	函数界面	-	O	O	5.2.3
16	尾递归	O	-	O	5.2.4
17	使用 FSQRT 和 FABS 指令	O	-	O	5.2.5
18	不变量表达式在循环内的移动	-	-	O	5.3.1
19	减少循环次数	x	-	O	5.3.2
20	乘法和除法的使用	-	-	-	5.3.3
21	标识的应用	-	-	O	5.3.4
22	表的使用	O	-	O	5.3.5
23	条件	O	-	O	5.3.6
24	删除加载/存储指令	O	-	O	5.3.7
25	转移	O	-	O	5.4
26	函数的内联扩展	x	-	O	5.5.1
27	带嵌入式汇编语言代码的内联扩展	-	-	O	5.5.2
28	使用全局基址寄存器 (GBR) 的偏移引用	O	-	O	5.6.1
29	全局基址寄存器 (GBR) 区域的选择性使用	O	-	O	5.6.2
30	寄存器保存/恢复操作的控制	O	-	O	5.7
31	使用二字节地址的规格	O	-	-	5.8
32	预取指令	-	-	O	5.9.1
33	平铺	x	-	O	5.9.2
34	矩阵运算	O	-	O	5.10
35	软件流水线	-	-	O	5.11

注意： 圆圈 (O) 和 X 在表中具有下列意义：

O： 可有效增进性能

X： 可能降低性能

5.1 数据规格

表 5.3 列出应加以考虑的数据相关事项。

表 5.3 数据规格的建议

区域	建议	参考的节
数据类型说明符、类型修饰符	若尝试缩减数据大小，程序大小可能因此增加。 应根据用途来声明数据类型。	5.1.1 至 5.1.3、
	取决于所使用的是带符号或无符号类型，程序大小可能会更改；应谨慎选择数据类型。	5.1.6
	若是其值不会在程序内更改的初始化数据，使用常数运算符将减低存储器要求。	
数据调整	应分配数据以使存储器中不会出现未使用的区域。	5.1.5
结构的定义和引用	在一些情况下，经常引用或修改的数据可被结合到用来缩减程序大小的结构和指针变量。	5.1.4
	位字段可用来缩减数据大小。	
局部变量和全局变量	局部变量更具效率；应将所有可用作局部变量的变量声明为局部变量，而非全局变量。	5.1.7
指针类型的使用	应将使用数组类型的程序修改为使用指针类型。	5.1.8
内部 ROM/RAM 的使用	由于对内部存储器的存取比外部存储器更快速，应将公用变量存储在内部存储器内。	-

5.1.1 局部变量（数据大小）

重点：

当使用了四字节大小的局部变量时，ROM 效率和执行速度可在一些情况下获得增进。

描述：

Renesas Technology SuperH RISC engine 家族中的一般用途寄存器是四字节的，因此处理的基本单位是四字节。

所以当操作使用了一字节或二字节的局部变量时，将添加代码把它们转换为四字节。在一些情况下，为变量采用四字节将可获得较小的程序大小和更快的执行速度，即使其实采用一字节或二字节就已足够。

使用的实例：

计算从 1 到 10 的整数总和：

优化前的代码

```
int f (void)
{
     char a = 10;
   int  c = 0;

     for ( ; a > 0; a-- )
            c += a;

     return(c);
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV           #10,R4
          MOV           #0,R5
L217:
          EXTS.B        R4,R3
          ADD           R3,R5
          ADD           #-1,R4
          EXTS.B        R4,R2
          CMP/PL        R2
          BT            L217
          RTS
          MOV           R5,R0
```

优化后的代码

```
int f( void )
{
     long a = 10;
     int  c = 0;

     for ( ; a > 0; a-- )
            c += a;

     return(c);
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV           #10,R4
          MOV           #0,R5
L217:
          ADD           R4,R5
          ADD           #-1,R4
          CMP/PL        R4
          BT            L217
          RTS
          MOV           R5,R0
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	18	16	73	63
SH-2	18	16	73	63
SH-2A	16	14	62	52
SH2AL-FPU	16	14	62	52
SH-2E	18	16	73	63
SH2-DSP(SH7065)	18	16	73	63
SH-3	18	16	73	63
SH3-DSP	18	16	73	63
SH-4	18	16	64	54
SH-4A	18	16	54	44
SH4AL-DSP	18	16	54	44

5.1.2 全局变量（符号）

重点：

当一个语句包含全局变量的类型转换时，若整数变量无论是带符号或无符号都没有分别，将它声明为带符号将可增进 ROM 效率和执行速度。

描述：

当 Renesas Technology SuperH RISC engine 家族使用 MOV 指令从存储器转移一或二字节的数据时，添加了用于无符号的 EXTU 指令。 这使得声明为无符号类型的变量效率较带符号类型差。

使用的实例：

代入变量 a 和变量 b 以求总和变量 c：

优化前的代码

```
unsigned short    a;
unsigned short    b;
int               c;
void f(void)
{
      c = b + a;
}
```

扩展为汇编语言代码（优化前）

```
_f:
        MOV.L       L11,R1
        MOV.L       L11+4,R2
        MOV.W       @R1,R5
        EXTU.W      R5,R4
        MOV.L       L11+8,R5
        MOV.W       @R5,R7
        EXTU.W      R7,R7
        ADD         R7,R4
        RTS
        MOV.L       R4,@R2
L11:
        .DATA.L     _b
        .DATA.L     _c
        .DATA.L     _a
```

优化后的代码

```
short a;
short b;
int      c;
void f(void)
{
      c = b + a;
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L       L11,R1
          MOV.L       L11+4,R4
          MOV.W       @R1,R5
          MOV.W       @R4,R7
          MOV.L       L11+8,R2
          ADD         R7,R5
          RTS
          MOV.L       R5,@R2
L11:
          .DATA.L     _b
          .DATA.L     _a
          .DATA.L     _c
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	32	28	15	11
SH-2	32	28	15	11
SH-2A	32	28	8	8
SH2A-FPU	32	28	15	11
SH-2E	32	28	15	11
SH2-DSP(SH7065)	32	28	15	11
SH-3	32	28	15	11
SH3-DSP	32	28	15	13
SH-4	32	28	13	9
SH-4A	32	28	10	8
SH4AL-DSP	32	28	10	8

5.1.3 数据大小（乘法）

重点：

在乘法中，若将乘数或被乘数声明为 [unsigned] char 或 [unsigned] short，则将可增进执行速度。

描述：

在 SH-2、SH-2E、SH2-DSP、SH-3、SH-3DSP 和 SD-4 中，若乘法中的乘数或被乘数是一或二字节，操作将被扩展为 MULS.W 或 MULU.W 指令；但若其中一个是四字节，则将使用 MUL.L 指令。

在 SH-1 中，若乘数和被乘数是一或二字节，MULS 或 MULU 指令将被使用；但若它们是四字节，则运行时库将被调用。

使用的实例：

取变量 a 和变量 b 的积，然后返回结果：

注意： 在此实例中，编译选项是 –cpu=sh1。

优化前的代码

```
int f( long a, long b )
{
   return( a * b );
}
```

扩展为汇编语言代码（优化前）

```
_f:
           MOV.L       L11,R2
           MOV         R5,R1
           JMP         @R2
           MOV         R4,R0
L11:
           .DATA.L     _muli
```

优化后的代码

```
int f( short a, short b )
{
   return( a * b );
}
```

扩展为汇编语言代码（优化后）

```
f:
           STS.L       MACL,@-R15
           MULS        R5,R4
           STS         MACL,R0
           RTS
           LDS.L       @R15+,MACL
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	12	10	23	8

注意： a=1, b=2

5.1.4 数据结构

重点：

若将相关的数据声明为结构，可在一些情况下增进执行速度。

描述：

当数据在相同函数中被多次引用时，通过将基址分配到寄存器并创建数据结构将可增进效率。若将数据作为参数传递也可增进效率。应把频繁存取的数据放在结构的起始部分，以获得最佳效果。

当为数据创建结构后，修改数据呈现方式等的调整将变得更轻松。

使用的实例：

为变量 a、b 和 c 代入数值：

优化前的代码

```
int a, b, c;
void f(void)
{
    a = 1;
    b = 2;
    c = 3;
}
```

优化后的代码

```
struct s{
    int a;
    int b;
    int c;
} s1;
void f(void)
{
    register struct s *p=&s1;

    p->a = 1;
    p->b = 2;
    p->c = 3;
}
```

扩展为汇编语言代码（优化前）

```
_f:
            MOV.L       L11,R7
            MOV         #1,R1
            MOV.L       R1,@R7
            MOV.L       L11+4,R1
            MOV.L       L11+8,R2
            MOV         #2,R4
            MOV         #3,R5
            MOV.L       R4,@R1
            RTS
            MOV.L       R5,@R2
L11:
            .DATA.L     _a
            .DATA.L     _b
            .DATA.L     _c
```

扩展为汇编语言代码（优化后）

```
_f:
            MOV.L       L11,R2
            MOV         #1,R1
            MOV         #2,R4
            MOV         #3,R5
            MOV.L       R1,@R2
            MOV.L       R4,@(4,R2)
            RTS
            MOV.L       R5,@(8,R2)
L11:
            .DATA.L     _s1
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	32	20	12	9
SH-2	32	20	12	9
SH-2A	32	20	9	6
SH2A-FPU	32	20	9	6
SH-2E	32	20	12	9
SH2-DSP(SH7065)	32	20	12	9
SH-3	32	20	14	10
SH3-DSP	32	20	15	11
SH-4	32	20	8	7
SH-4A	32	20	10	8
SH4AL-DSP	32	20	10	8

5.1.5 数据对齐

重点：

在一些情况中，可通过更改数据声明顺序来缩减所需的 RAM 数量。

描述：

在以不同的大小类型声明变量时，应连续声明具有相同大小类型的变量。以此对齐数据，将可最小化数据空间中的空区域。

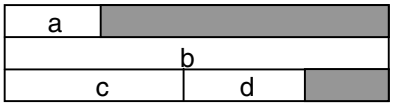
使用的实例：

声明总计八字节的数据：

优化前的代码

```
char    a;
int     b;
short   c;
char    d;
```

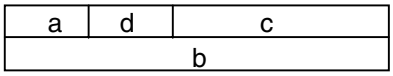
优化前的数据整理



优化后的代码

```
char    a;
char    d;
short   c;
int     b;
```

优化后的数据整理



5.1.6 初始值和常数类型

重点：

在程序执行期间不会更改的初始值，应使用 const 来声明。

描述：

初始化数据通常会在启动时从 ROM 转移到 RAM，同时 RAM 区域将用来进行处理。因此若程序中的初始数据值不更改，所预备的 RAM 区域将被浪费。 通过在声明初始数据时使用 const 运算符，可防止在启动时被转移到 RAM，由此缩减了存储器的用量。

此外，通过创建规定不更改初始值的程序，更方便在 ROM 中储存。

使用的实例：

指定 5 项初始数据：

优化前的代码	优化后的代码

`char a[] =`
`{1, 2, 3, 4, 5};`
 	
初始值在处理前从 ROM 转移到 RAM。
	`const char a[] =`
	`{1, 2, 3, 4, 5};`
	
	存储在 ROM 中的初始值用来进行处理。

5.1.7 局部变量和全局变量

重点：

若将局部使用的变量，如临时变量或循环计数器等，声明为局部变量，将可增进执行速度。

描述：

应始终将可用作局部变量的变量声明为局部变量，而非全局变量。全局变量的值可能因为函数调用或指针操作的结果而改变，因此全局变量不接受全局优化。

使用局部变量具有下列好处：

更低的存取成本

具有寄存器分配的可能性

优化

使用的实例：

执行十次循环重复：

优化前的代码

```
int i;

void f(void)
{
   for ( i = 0; i < 10; i++ );
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.L       L218+2,R4
          MOV         #0,R3
          MOV         #10,R5
          BRA         L216
          MOV.L       R3,@R4
L217:
          MOV.L       @R4,R1
          ADD         #1,R1
          MOV.L       R1,@R4
L216:
          MOV.L       @R4,R3
          CMP/GE      R5,R3
          BF          L217
          RTS
          NOP
L218:
          .DATA.W     0
```

优化后的代码

```
void f(void)
{
   int i;

   for ( i = 0; i < 10; i++ );
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV         #10,R4
L216:
          DT          R4
          BF          L216
          RTS
          NOP
```

```
        .DATA.L     _i
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	20	12	54	52
SH-2	20	10	45	42
SH-2A	20	8	42	42
SH2A-FPU	20	8	42	42
SH-2E	20	10	45	42
SH2-DSP(SH7065)	20	10	54	51
SH-3	20	10	45	42
SH3-DSP	20	10	53	51
SH-4	20	10	34	33
SH-4A	20	10	25	23
SH4AL-DSP	20	10	50	46

5.1.8 指针变量的使用

重点：

在一些情况中，将使用数组的程序重写为使用指针类型将可增进执行速度。（版本 6）

描述：

在引用数组元素 a[i] 时，生成的代码将 i 添加到 a[0] 的地址。通过使用指针变量，有时将可缩减变量和操作的数量。

使用的实例：

计算数组总数：

优化前的代码

```
int f1( int data[], int count )
{
        int ret = 0, i;

        for (i = 0; i < count; i++)
                ret += data[i]*i;
        return ret;
}
```

扩展为汇编语言代码（优化前）

```
_f1:
        STS.L       MACL,@-R15
        MOV         #0,R7
        CMP/PL      R5
        BF/S        L219
        MOV         R7,R6
L220:
        MOV         R6,R0
        SHLL2       R0
        MOV.L       @(R0,R4),R3
        MUL.L       R6,R3
        ADD         #1,R6
        STS         MACL,R3
        CMP/GE      R5,R6
        BF/S        L220
        ADD         R3,R7
L219:
        MOV         R7,R0
        RTS
        LDS.L       @R15+,MACL
```

优化后的代码

```
int f2( int *data, int count )
{
        int ret = 0, i;

        for (i = 0; i < count; i++)
                ret += *data++ *i;
         return ret;
}
```

扩展为汇编语言代码（优化后）

```
_f2:
        STS.L       MACL,@-R15
        MOV         #0,R7
        CMP/PL      R5
        BF/S        L221
        MOV         R7,R6
L222:
        MOV.L       @R4+,R3
        MUL.L       R6,R3
        ADD         #1,R6
        STS         MACL,R3
        CMP/GE      R5,R6
        BF/S        L222
        ADD         R3,R7
L221:
        MOV         R7,R0
        RTS
        LDS.L       @R15+,MACL
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小［字节］		执行速度 ［周期］	
	优化前	优化后	优化前	优化后
SH-1	40	36	318	318
SH-2	34	30	179	159
SH-2E	34	30	178	158
SH2-DSP(SH7065)	34	30	207	187
SH-3	34	30	149	129
SH3-DSP	34	30	168	148
SH-4	34	30	117	97

注意： 周期数用于 count=10。

5.1.9 引用常数 (1)

重点：

尽可能使用单一字节来代表立即值将可缩减代码大小。

描述：

当使用单一字节的立即值时，它将被嵌入到代码中。相对的，二字节和四字节的立即值则被放置到存储器中，然后被存取。

使用的实例：

将立即值代入变量：

源代码（1）

```
int i;
void f(void)
{
     i = 0x10000;
}
```

扩展为汇编语言代码（1）

```
_f:
          MOV       #1,R2
          MOV.L     L12+2,R6
          SHLL16    R2
          RTS
          MOV.L     R2,@R6
L12:
          .RES.W    1
          .DATA.L   _i
```

源代码（2）

```
int i;
void f(void)
{
     i = 0x01;
}
```

扩展为汇编语言代码（2）

```
_f:
          MOV.L     L12+2,R6
          MOV       #1,R2
          RTS
          MOV.L     R2,@R6
L12:
          .RES.W    1
          .DATA.L   _i
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	源代码 (1)	源代码 (2)	源代码 (1)	源代码 (2)
SH-1	16	12	7	5
SH-2	16	12	7	5
SH-2A	16	12	4	4
SH2AL-FPU	16	12	4	4
SH-2E	16	12	7	5
SH2-DSP(SH7065)	16	12	7	6
SH-3	16	12	7	5
SH3-DSP	16	12	6	6
SH-4	16	12	5	4
SH-4A	16	12	5	4
SH4AL-DSP	16	12	5	4

5.1.10 引用常数 (2)

重点：

可将使用常数的表达式结合起来，而不会增加所生成代码的大小。

描述：

可使用常数卷积功能。 即使常数以表达式来呈现，它只在编译时计算而不会在所生成的代码中反映出来。

使用的实例：

为变量 a 代入常数：

优化前的代码

```
#define MASK1  0x1000
#define MASK2  0x10

int a = 0xffffffff;

void f(void)
{
      int x;

    x = MASK1;
    x |= MASK2;
    a &= x;
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.W       L217,R4
          MOV.L       L217+4,R5
          MOV.L       @R5,R3
          AND         R4,R3
          RTS
          MOV.L       R3,@R5
L217:
          .DATA.W     H'1010
          .DATA.W     0
          .DATA.L     _a
```

优化后的代码

```
#define MASK1  0x1000
#define MASK2  0x10

int a = 0xffffffff;

void f(void)
{
      a &= MASK1 | MASK2;
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L       L216+4,R4
          MOV.W       L216,R3
          MOV.L       @R4,R2
          AND         R3,R2
          RTS
          MOV.L       R2,@R4
L216:
          .DATA.W     H'1010
          .DATA.W     0
          .DATA.L     _a
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	20	20	9	9
SH-2	20	20	9	9
SH-2A	20	20	7	7
SH2A-FPU	20	20	7	7
SH-2E	20	20	9	9
SH2-DSP(SH7065)	20	20	9	9
SH-3	20	20	9	9
SH3-DSP	20	20	9	9
SH-4	20	20	8	8
SH-4A	20	20	6	6
SH4AL-DSP	20	20	6	6

5.1.11 保持为常数的变量 (1)

重点：

当一个变量的值保持为常数时，它将被当作常数处理；即使该变量未被预先计算，也不会对存储器效率或执行速度具有任何影响。

描述：

常数卷积 功能也可应用于具有常数特征的变量，追踪该变量的值并执行常数计算。 所以即使为方便读取而如此编写源代码也不会增加所生成的代码大小。

使用的实例：

根据变量 rc 的结果更改返回值：

预先计算变量值

源代码 (1)

```
#define ERR        -1
#define NORMAL     0

int f(void)
{
      int rc, code;

      rc = 0;
      code = NORMAL;
      return( code );
}
```

扩展为汇编语言代码 (1)

```
_f:
           RTS
           MOV          #0,R0
```

让 C 编译程序计算该值

源代码 (2)

```
#define ERR        -1
#define NORMAL     0

int f(void)
{
      int rc, code;

      rc = 0;
      if ( rc ) code = ERR;
          else  code = NORMAL;
      return( code );
}
```

扩展为汇编语言代码 (2)

```
_f:
           RTS
           MOV          #0,R0
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	源代码 (1)	源代码 (2)	源代码 (1)	源代码 (2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2A	4	4	4	4
SH2A-FPU	4	4	4	4
SH-2E	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.1.12 保持为常数的变量 (2)

重点：

当一个变量的值保持为常数时，它将被当作常数处理；即使该变量未被预先计算，也不会对存储器效率或执行速度具有任何影响。

描述：

常数卷积功能也可应用于具有常数特征的变量，追踪该变量的值并执行常数计算。所以即使为方便读取而如此编写源代码也不会增加所生成的代码大小。

使用的实例：

计算变量 a 和 c 的积，然后将结果代入变量 b。

预先计算变量值

源代码 (1)

```
int f(void)
{
     int a, b;

     a = 3;
     b = 15;
     return b;
}
```

为上述扩展为汇编语言代码 (1)

```
_f:
          RTS
          MOV          #15,R0
```

让 C 编译程序计算该值

源代码 (2)

```
int f(void)
{
     int a, b, c;

     a = 3;
     c = 5;
     b = c * a;
     return b;
}
```

为上述扩展为汇编语言代码 (2)

```
_f:
          RTS
          MOV          #15,R0
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	源代码 (1)	源代码 (2)	源代码 (1)	源代码 (2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2A	4	4	4	4
SH2A-FPU	4	4	4	4
SH-2E	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.2 函数调用

表 5.4 中列出在调用函数时所应考虑的事项。

表 5.4 有关函数调用的建议

区域	建议	参考的节
函数位置	密切关联的函数应放在同一文件中	5.2.1
界面	应严格限制参数数量（最多四个），以便它们都能被分配到寄存器。 当有大量的参数时，它们应结合到一个结构中，然后使用指针来传递。	5.2.3
函数划分	在一些情况中，极大量的函数可能无法有效执行多种优化。 使用称为尾递归的功能，函数将被划分到能够有效执行优化的大小。	5.2.4
宏替换	对于频繁调用的函数，可使用宏来替换以增进执行速度。 然而，使用宏将增加程序大小，所以必须视情况来使用宏。	-

5.2.1 在模块中结合函数

重点：

密切关联的函数应放在同一文件中，以增进程序执行速度。

描述：

当调用不同文件中的函数时，将使用 JSR 指令来扩展它们；但若所调用的是相同文件中的函数，同时调用的范围很小，则将使用 BSR 指令，这将使执行速度更快，生成的目标更简练。

通过将函数结合到模块中，可更轻松进行调整性的修改。

使用的实例：

要从函数 f 调用函数 g：

优化前的代码

```
extern g(void);
int f(void)
{
     g();
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.L       L216+2,R3
          JMP         @R3
          NOP
L216:
          .DATA.W     0
          .DATA.L     _g
```

优化后的代码

```
int g(void)
{
}

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

扩展为汇编语言代码（优化后）

```
_g:
          RTS
          NOP
_f:
          BRA         _g
          NOP
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小［字节］		执行速度 ［周期］	
	优化前	优化后	优化前	优化后
SH-1	12	4	8	6
SH-2	12	4	8	6
SH-2A	12	4	8	6
SH2A-FPU	12	4	8	6
SH-2E	12	4	8	6
SH2-DSP(SH7065)	12	4	9	6
SH-3	12	4	8	6
SH3-DSP	12	4	9	6
SH-4	12	4	8	5
SH-4A	12	4	5	4
SH4AL-DSP	12	4	5	4

注解：

BSR 指令可调用范围介于 ±4096 字节（±2048 指令）内的函数。

若文件过大，将无法有效的使用 BSR 指令。

在这种情况下，建议在定位时让经常彼此调用的函数互相靠近，以便可以使用 BSR 指令。

5.2.2 使用指针变量的调用源函数

重点：

与其使用 switch 语句来进行转移，使用表将可增进执行速度。

描述：

若 switch 语句在每个情况下的处理基本上是相同的，应对使用表的可能性进行研究。

使用的实例：

根据变量 a 的值更改调用目标函数：

优化前的代码

```
extern void nop(void);
extern void stop(void);
extern void play(void);

void f(int a)
{
     switch (a)
      {
     case 0:
                 nop(); break;
     case 1:
                  stop(); break;
     case 2:
                  play(); break;
      }
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV          R4,R0
          CMP/EQ       #0,R0
          BT           L220
          CMP/EQ       #1,R0
          BT           L221
          CMP/EQ       #2,R0
          BT           L222
          BRA          L223
          NOP
L220:
          MOV.L        L224,R3
          JMP          @R3
          NOP
L221:
          MOV.L        L224+4,R3
          JMP          @R3
```

优化后的代码

```
extern void nop(void);
extern void stop(void);
extern void play(void);

static int (*key[3])() =
            {nop, stop, play};
void f(int a)
{
       (*key[a])();
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L        R4,@-R15
          MOV          R4,R3
          MOV.L        L241+2,R0
          SHLL2        R3
          MOV.L        @(R0,R3),R3
          JMP          @R3
          ADD          #4,R15
L241:
          .DATA.W      0
          .DATA.L      _$key
          .SECTION     D,DATA,ALIGN=4
_$key:
          .DATA.L      _nop,_stop,_play
```

```
        NOP
L222:
        MOV.L       L224+8,R3
        JMP         @R3
        NOP

L223:
        RTS
        NOP
L224:
        .DATA.L     _nop
        .DATA.L     _stop
        .DATA.L     _play
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	52	16	15	11
SH-2	52	16	15	11
SH-2A	52	16	14	10
SH2A-FPU	52	16	14	10
SH-2E	52	16	15	11
SH2-DSP(SH7065)	52	16	16	12
SH-3	52	16	15	11
SH3-DSP	52	16	16	13
SH-4	52	16	13	10
SH-4A	52	16	10	8
SH4AL-DSP	52	16	10	8

5.2.3 函数界面

重点：

若谨慎声明函数参数，将可削减所需的 RAM 数量，同时执行速度获得增进。

有关详细资料，请参考 3.15.1 (2) 节，函数调用界面。

描述：

应谨慎选择函数参数，以确保所有参数可被分配到寄存器（最多四个参数）。若必须使用多个参数，则必须将它们结合到结构中，并使用指针进行传递。 若寄存器可容纳所有参数，函数调用和函数入口与出口的处理将被简化。 堆栈用量也将被缩减。

寄存器 R0 至 R3 是工作寄存器，R4 至 R7 供参数使用，R8 至 R14 则供局部变量使用。

在 SH-2E 和 SH-4 中，浮点寄存器被用来处理浮点数据。 寄存器 FR0 至 FR3 是工作寄存器，FR4 至 FR11 供参数使用，FR12 至 FR14 供局部变量使用。

使用的实例：

函数 f 的参数数量是五个，超过参数寄存器的数量。

优化前的代码

```
int f(int, int, int, int, int);

void g(void)
{
        f(1, 2, 3, 4, 5);
}
```

优化后的代码

```
struct b{
        int a, b, c, d, e;
} b1 = {1, 2, 3, 4, 5};

int f(struct b *p);

void g(void)
{
        f(&b1);
}
```

扩展为汇编语言代码（优化前）

```
_g:
        STS.L       PR,@-R15
        MOV         #5,R3
        MOV.L       L216+2,R2
        MOV         #4,R7
        MOV.L       R3,@-R15
        MOV         #3,R6
        MOV         #2,R5
        JSR         @R2
        MOV         #1,R4
        ADD         #4,R15
        LDS.L       @R15+,PR
        RTS
        NOP
L216:
```

扩展为汇编语言代码（优化后）

```
_g:
        MOV.L       L217,R4
        MOV.L       L217+4,R3
        JMP         @R3
        NOP
L217:
        .DATA.L     _b1
        .DATA.L     _f
```

```
.DATA.W     0
.DATA.L     _f
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	32	16	17	7
SH-2	32	16	20	10
SH-2A	28	16	17	9
SH2A-FPU	28	16	17	9
SH-2E	32	16	20	10
SH2-DSP(SH7065)	32	16	28	14
SH-3	32	16	22	10
SH3-DSP	32	16	25	15
SH-4	32	16	18	10
SH-4A	32	16	15	6
SH4AL-DSP	32	16	15	6

5.2.4 尾递归

重点：

若大型函数被划分为一系列较小型的函数，下一个函数将在上一个函数结束时被调用，因此不会使执行速度变慢。

描述：

当函数 funk3() 被本身由函数 funk1() 调用的函数 funk2() 调用时，将通过 BRA 或 JMP 指令将控制传递给函数 funk3()。一般上，在函数 funk3() 完成处理后，RTS 指令将把控制返回给函数 funk2()，同时当函数 funk2() 完成处理后，另一个 RTS 指令将把控制返回给函数 funk1()。（参阅图 5.1 的左边）

在这里，当 funk3() 在函数 funk2() 结束后被调用时，控制也通过 BSR 或 JSR 指令转移给 funk3()，而在 funk3() 完成处理后，可通过 RTS 指令直接将控制返回给函数 funk1()。（参阅图 5.1 的右边） 这项功能称为尾递归。

在一些情况中，极大量的模块可能无法有效执行多种优化。通过使用尾递归，较大型的模块将被划分为可有效执行优化的模块大小，以增强性能。

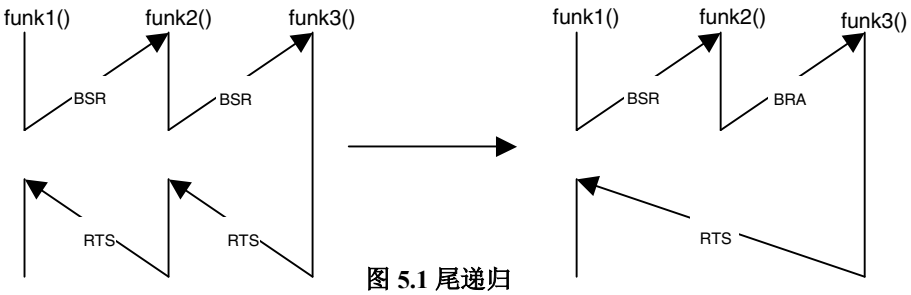


图 5.1 尾递归

使用的实例：

从函数 f 调用函数 g 及 h： 当从 g 和 h 返回时，控制将直接传递给 f 的调用源函数，而略过 f 本身。

应用前的源代码（版本 2.0）

```
void f(int x)
{
      if (x==1)
            g();
      else
            h();
}
```

扩展为汇编语言代码（应用前）

```
_f:
          STS.L       PR,@-R15
          MOV         R4,R0
          CMP/EQ      #1,R0
          BF          L207
          BSR         _g
          NOP
          BRA         L208
```

应用后的源代码（版本 3.0 或以上）

```
void f(int x)
{
      if (x==1)
            g();
      else
            h();
}
```

扩展为汇编语言代码（应用后）

```
_f:
          MOV         R4,R0
          CMP/EQ      #1,R0
          BF          L12
          BRA         _g
        NOP
L12:
          BRA          _h
```

```
NOP                                     NOP
L207:
        BSR         _h
        NOP
L208:
        LDS.L       @R15+,PR
        RTS
        NOP
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	应用前	应用后	应用前	应用后
SH-2	24	14	14	8

注意： x = 2

5.2.5 使用 FSQRT 和 FABS 指令

重点：

与其从程序库调用数学函数 sqrt 和 fabs，若目标为 SH-4，则为那些处理程序使用指令集内的 FSQRT 和 FABS 指令。

描述：

fabs（浮点绝对值）函数是数学函数程序库的一部分，但程序库在没有函数地址的程序中变得不必要，因此直接使用 FABS 指令。

然而，这项指令的使用需要包含 <math.h> 或 <mathf.h>。

若这些未被包含，编译程序将把 fabs 调用为普通函数，结果导致的程序库调用将会降低性能。

用户不需要定义宏。

<宏实例>

#define fabs(a) ((a)>=0?0:(-(a))) /*不扩展为 FABS 指令*/

使用的实例：

下面显示当不包含 <math.h>（程序库调用的结果）及包含 <math.h>（使用 FABS 指令）的区别。

注意： 在此实例中，使用了下列编译选项。

-cpu=sh4 -fpu=single

当使用 fabsf() 时，必须包含 <mathf.h>。

优化前的代码

```
float a,b;
f()
{
     :
     :
b=fabs(a);
   :
}
```

扩展为汇编语言代码（优化前）

```
_f:
        STS.L       PR,@-R15
        MOV.L       L12,R6
        MOV.L       L12+4,R1
        JSR         @R1
        FMOV.S      @R6,FR4
        MOV         R0,R4
        LDS         R4,FPUL
        FLOAT       FPUL,FR8
        MOV.L       L12+8,R5
        LDS.L       @R15+,PR
```

优化后的代码

```
#include <math.h>
float a,b;
f()
{
     :
     :
b=fabs(a);
   :
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L     L12,R1
          MOV.L     L12+8,R4
          FMOV.S    @R1,FR9
          FABS      FR9
          RTS
          FMOV.S    FR9,@R4
_L12:
          .DATA.L   _a
          .DATA.L   _fabs
          .DATA.L   _bW
```

```
        RTS
        FMOV.S      FR8,@R5

L12:
        .DATA.L     _a
        .DATA.L     _fabs
        .DATA.L     _bW
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH2A-FPU	68	40	49	21
SH-2E	36	20	33	9
SH-4	36	20	29	8
SH-4A	36	20	22	6

5.3 运算

表 5.5 列出应给予考虑的运算相关区域。

表 5.5 有关运算的建议

区域	建议	参考的节
不变量和公用表达式的统一、移动	应研究使用临时变量取代函数的公用常量方程式的可能性 for 语句中的任何不变量表达式应被移出 for 语句。	5.3.1
缩减循环迭代的次数	应研究合并具有相同或类似条件的循环语句的可能性。 尝试扩展循环语句。	5.3.2
运算的优化	合并相同的运算，以缩减运算迭代的次数。	5.3.3
标识的使用	应研究使用数学标识以缩减运算次数的可能性。	5.3.4
快速算法的使用	应研究使用占用较少处理时间的有效算法，如数组的快速分类。	-
表的使用	若 switch 语句在每个情况下的处理几乎是相同的，应对使用表的可能性进行研究。 通过预先执行运算，将结果存储在表中，并在需要运算结果时引用表中的值，有时可能会增进执行速度。 然而，这种方法需要更多的 ROM，并应在使用时特别注意以在所需的执行速度和可用的 ROM 之间取得平衡。	5.3.5
条件	当和常数比较时，若常数的值是 0，将会生成更有效率的代码。	5.3.6
删除加载/存储	通过删除存储器存取（加载、存储）指令，将可缩减执行周期的数量。	5.3.7

5.3.1 不变量表达式在循环内的移动

重点：

若在循环中具有其值不变更的表达式，则可通过在循环开始前计算表达式来增进执行速度。（版本 6）

描述：

通过在循环开始前计算表达式在循环中不变更的值，将可省略在每次迭代中的计算，从而缩减执行指令的数量。

使用的实例：

将数组元素 b[5] 代入数组 a[ ]：

优化前的代码

```
extern int a[100], b[100];
void f(void)
{
        int i,j;

        j = 5;
        for ( i=0; i < 100; i++)
         a[i] = b[j];
}
```

扩展为汇编语言代码（优化前）

```
_f:
            MOV.L       L240+4,R5
            MOV         R5,R4
            MOV.W       L240,R6
            ADD         R5,R6
            MOV.L       L240+8,R5
L239:
            MOV.L       @R5,R3
            MOV.L       R3,@R4
            ADD         #4,R4
            CMP/HS      R6,R4
            BF          L239
            RTS
            NOP
L240:
            .DATA.W     H'0190
            .DATA.W     0
            .DATA.L     _a
            .DATA.L     H'00000014+_b
```

优化后的代码

```
extern int a[100], b[100];
void f(void)
{
        int i,j,t;

        j = 5;
        for ( i=0, t=b[j];i < 100; i++)
         a[i] = t;
}
```

扩展为汇编语言代码（优化后）

```
_f:
            MOV.L       L241+4,R5
            MOV.L       @R5,R5
            MOV.L       L241+8,R7
            MOV         R7,R4
            MOV.W       L241,R6
            ADD         R7,R6
L240:
            MOV.L       R5,@R4
            ADD         #4,R4
            CMP/HS      R6,R4
            BF           L240
            RTS
            NOP
L241:
            .DATA.W     H'0190
            .DATA.W     0
            .DATA.L     H'00000014+_b
            .DATA.L     _a
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	36	36	809	611
SH-2	36	36	809	611
SH-2E	36	36	809	611
SH2-DSP(SH7065)	36	36	908	611
SH-3	36	36	909	711
SH3-DSP	36	36	1008	711
SH-4	36	36	608	407

5.3.2 减少循环次数

重点：

当循环被扩展时，将可增进执行速度。

描述：

循环扩展对内部循环特别有效。循环扩展将增加程序大小，所以应仅在有必要增进执行速度时，以增加程序大小的代价使用这项技术。

使用的实例：

初始化数组 a[]：

优化前的代码

```
extern int a[100];
void f(void)
{
        int i;

        for ( i = 0; i < 100; i++)
                a[i] = 0;
}
```

扩展为汇编语言代码（优化前）

```
_f:
            MOV.L       L238+2,R7
            MOV         #0,R5
            MOV.W       L238,R6
            MOV         R7,R4
            ADD         R7,R6
L237:
            MOV.L       R5,@R4
            ADD         #4,R4
            CMP/HS      R6,R4
            BF          L237
            RTS
            NOP
L238:
            .DATA.W     H'0190
            .DATA.L     _a
```

优化后的代码

```
extern int a[100];
void f(void)
{
        int i;

        for ( i = 0; i < 100; i+=2)
        {
                a[i] = 0;
                a[i+1] = 0;
        }
}
```

扩展为汇编语言代码（优化后）

```
_f:
            MOV.L       L239+2,R7
            MOV         #0,R5
            MOV.W       L239,R0
            MOV         R7,R6
            ADD         #4,R6
            MOV         R7,R4
            ADD         R7,R0
L238:
            MOV.L       R5,@R4
            MOV.L       R5,@R6
            ADD         #8,R4
            CMP/HS      R0,R4
            BF/S        L238
            ADD         #8,R6
            RTS
            NOP
L239:
            .DATA.W     H'0190
            .DATA.L     _a
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	24	28	805	455
SH-2	24	24	506	356
SH-2A	20	24	403	253
SH2A-FPU	20	24	403	253
SH-2E	24	24	506	356
SH2-DSP(SH7065)	24	24	605	605
SH-3	24	24	606	407
SH3-DSP	24	24	705	503
SH-4	24	24	305	204
SH-4A	24	24	405	255
SH4AL-DSP	24	24	405	255

5.3.3 乘法和除法的使用

重点：

当不确定应使用乘法/除法或移位运算时，应尝试使用乘法和除法。

描述：

在编写程序时应尽量使其容易读取。 在乘法和除法运算中，当乘数/除数和被乘数/被除数无符号时，这些运算将因为编译程序优化的结果被移位运算的组合替换。

使用的实例：

执行乘法和除法运算：

源代码（乘法）

```
unsigned int a;

int f(void)
{
      return(a*4);
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L       L217,R3
          MOV.L       @R3,R0
          RTS
          SHLL2       R0
L217:
          .DATA.L     _a
```

源代码（除法）

```
unsigned int b;

int f(void)
{
      return(b/2);
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L       L217,R3
          MOV.L       @R3,R0
          RTS
          SHLR        R0
L217:
          .DATA.L     _b
```

5.3.4 标识的应用

重点：

通过应用数学标识，有时可缩减运算次数，使执行速度获得增进。

描述：

必须要注意的是分析上简单的数学标识可能会在实际的数学应用中增加运算次数。

使用的实例：

计算从 1 到 n 的整数总和：

优化前的代码

```
int f( long n )
{
      int i, s;

      for (s = 0, i = 1;
              i <= n; i++)
              s += i;
      return( s );
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV           #1,R5
          CMP/GT        R4,R5
          BT/S          L218
          MOV           #0,R6
L219:
          ADD           R5,R6
          ADD           #1,R5
          CMP/GT        R4,R5
          BF            L219
L218:
          RTS
          MOV           R6,R0
```

优化后的代码

```
int f( long n )
{
          return( n*(n+1) >> 1 );
}
```

扩展为汇编语言代码（优化后）

```
_f:
          STS.L         MACL,@-R15
          MOV           R4,R0
          ADD           #1,R0
          MUL.L         R4,R0
          STS           MACL,R0
          SHAR          R0
          RTS
          LDS.L         @R15+,MACL
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	18	24	609	31
SH-2	18	16	609	21
SH-2A	16	12	608	10
SH2A-FPU	16	12	608	10
SH-2E	18	16	609	14
SH2-DSP(SH7065)	18	16	710	15
SH-3	18	16	609	18
SH3-DSP	18	16	710	18
SH-4	18	16	507	14
SH-4A	18	16	407	8
SH4AL-DSP	18	16	407	8

注意： 周期数 n = 100

5.3.5 表的使用

重点：

与其使用 switch 语句来进行转移，使用表将可增进执行速度。

描述：

若 switch 语句在每个情况下的处理基本上是相同的，应对使用表的可能性进行研究。

使用的实例：

更改将根据变量 I 的值代入到变量 ch 中的字符常数：

优化前的代码

```
char f (int i)
{
      char ch;

      switch (i)
      {
          case 0:
               ch = 'a'; break;
          case 1:
               ch = 'x'; break;
          case 2:
               ch = 'b'; break;
      }
      return (ch);
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV          R4,R0
          CMP/EQ       #0,R0
          BT           L218
          CMP/EQ       #1,R0
          BT           L219
          CMP/EQ       #2,R0
          BT           L220
          BRA          L221
          NOP
L218:
          BRA          L221
          MOV          #97,R4
L219:
          BRA          L221
          MOV          #120,R4
L220:
          MOV          #98,R4
```

优化后的代码

```
char chbuf[] = { 'a', 'x', 'b' };

char f(int i)
{
        return (chbuf[i]);
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L        L218+2,R0
          RTS
          MOV.B        @(R0,R4),R0
L218:
          .DATA.W      0
          .DATA.L      _chbuf
```

```
L221:
          RTS
          MOV           R4,R0
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		[cycle]执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	18	24	609	31
SH-2	18	16	609	21
SH-2A	16	12	608	10
SH2A-FPU	16	12	608	10
SH-2E	18	16	609	14
SH2-DSP(SH7065)	18	16	710	15
SH-3	18	16	609	18
SH3-DSP	18	16	710	18
SH-4	18	16	507	14
SH-4A	18	16	407	8
SH4AL-DSP	18	16	407	8

注意： i = 2

5.3.6 条件

重点：

当和常数比较时，若常数的值是 0，将会生成更有效率的代码。

描述：

当和 0 比较时，不会生成加载常数值的指令，因此相对于和 0 以外的其他常数值比较，其代码的长度更短。循环和 if 语句的条件应设计为和 0 进行比较。

使用的实例：

根据参数值是否是 1 或以上来更改返回值：

优化前的代码

```
int f (int x)
{
        if ( x >= 1 )
             return 1;
        else
             return 0;
}
```

扩展为汇编语言代码（优化前）

```
_f:
           MOV          #1,R3
           CMP/GE       R3,R4
           MOVT         R0
           RTS
           NOP
```

优化后的代码

```
int f (int x)
{
        if ( x > 0 )
             return 1;
        else
                return 0;
}
```

扩展为汇编语言代码（优化后）

```
_f:
           CMP/PL       R4
           MOVT         R0
           RTS
           NOP
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	8	6	5	4
SH-2	8	6	5	4
SH-2A	8	6	6	5
SH2A-FPU	8	6	6	5
SH-2E	8	6	5	4
SH2-DSP(SH7065)	8	6	5	4
SH-3	8	6	5	4
SH3-DSP	8	6	5	4
SH-4	8	6	5	4
SH-4A	8	6	4	3
SH4AL-DSP	8	6	4	3

5.3.7 删除加载/存储指令

重点：

通过删除存储器存取（加载、存储）指令，将可缩减执行周期的数量。

描述：

在座标计算中，每个迭代过程从存储器加载 x、y 和 z 值，及将这些值存储到存储器的过程是降低性能的主要因素。 座标计算应尽可能在 FPU 寄存器中执行，而避免在结构中执行，这样可以缩减存储器加载和存储指令的次数，并增进执行速度。

使用的实例：

计算定点 P 和点 P0、P1 和 P2 所组成的面之间的顶点距离（平方），并根据距离来做决定：

注意： 在此实例中，编译选项是 -cpu=sh4Δfpu=single。

优化前的代码

```
#define SCAL2(v)   ((v)->x*(v)->x \
                +(v)->y*(v)->y\
                +(v)->z*(v)->z)
#define SubVect(a,b)
                ((a)->x-=(b)->x,\
                (a)->y -= (b)->y,\
                (a)->z -= (b)->z)

typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

int f(POINT3 *p, POLI *poli,
                              float rad)
{
    float  dst2;
    POINT3 dv;

    dv=poli->v[0];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    dv=poli->v[1];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;
```

优化后的代码

```
typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

float scal2(POINT3 *p1, POINT3 *q1)
{
    float a,b,c;
    float d,e,f;
    float *p=(float *)p1,*q=(float *)q1;

    a=*p++; d=*q++;
    b=*p++; e=*q++; a-=d;
    c=*p++; f=*q++; b-=e;
                        c-=f;
    return a*a+b*b+c*c;
}


int f(POINT3 *p,POLI *poli, float rad)
{
    float         d;
    POINT3     *q;

    q=poli->v;
    d2=scal2(q++,p);
    if (d2>rad) return 0;

    d2=scal2(q++,p);
```

```
    dv=poli->v[2];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    return 1;
}
```

```
if (d2>rad) return 0;

    d2=scal2(q++,p);
    if (d2>rad) return 0;

    return 1;
}
```

扩展为汇编语言代码（优化前）

```
_f:
        MOV.L       R8,@-R15
        MOV.L       @R5,R3
        ADD         #-12,R15
        MOV         R15,R2
        MOV.L       @R3,R1
        MOV.L       R1,@R2
        MOV.L       @(4,R3),R1
        MOV.L       R1,@(4,R2)
        MOV.L       @(8,R3),R1
        MOV.L       R1,@(8,R2)
        MOV         R15,R0
        NOP
        FMOV.S      @R0,FR2
        MOV         R15,R3
        FMOV.S      @R4,FR3
        MOV         R15,R2
        FSUB        FR3,FR2
        FMOV.S      FR2,@R0
        MOV         #4,R0
        MOV         R0,R1
        ADD         R4,R1
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @R1,FR3
        FSUB        FR3,FR2
        FMOV.S      FR2,@(R0,R3)
        MOV         #8,R0
        MOV         R0,R1
        ADD         R4,R1
        MOV         R15,R3
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @R1,FR3
        MOV         R15,R1
        FSUB        FR3,FR2
        FMOV.S      FR2,@(R0,R3)
        MOV         R15,R3
        MOV         R15,R0
```

扩展为汇编语言代码（优化后）

```
_scal2:
        FMOV.S      @R5+,FR5
        FMOV.S      @R4+,FR6
        FMOV.S      @R5+,FR7
        FMOV.S      @R4+,FR4
        FSUB        FR5,FR6
        FMOV.S      @R5,FR8
        FSUB        FR7,FR4
        FMOV.S      @R4,FR5
        FMOV.S      FR6,FR0
        FSUB        FR8,FR5
        FMOV.S      FR4,FR3
        FMUL        FR4,FR3
        FMAC        FR0,FR6,FR3
        FMOV.S      FR5,FR0
        FMAC        FR0,FR5,FR3
        RTS
        FMOV.S      FR3,FR0
_f:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV         R4,R13
        FMOV.S      FR15,@-R15
        STS.L       PR,@-R15
        MOV.L       @R5,R14
        MOV         R4,R5
        FMOV.S      FR4,FR15
        MOV         R14,R4
        BSR         _scal2
        ADD         #12,R14
        FMOV.S      FR0,FR4
        FCMP/GT     FR15,FR4
        BT          L258
        MOV         R14,R4
        MOV         R13,R5
        BSR         _scal2
        ADD         #12,R14
```

```
        NOP
        MOV         #4,R8
        MOV         R15,R7
        ADD         R7,R8
        MOV         R15,R6
        MOV         #4,R7
        FMOV.S      @R8,FR0
        ADD         R6,R7
        FMOV.S      @R1,FR2
        FMOV.S      @R7,FR3
        FMUL        FR0,FR3
        FMOV.S      @R0,FR0
        MOV         #8,R0
        FMAC        FR0,FR2,FR3
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @(R0,R2),FR0
        FMAC        FR0,FR2,FR3
        FMOV.S      FR3,FR5
        FCMP/GT     FR4,FR5
        BT          L284
        MOV.L       @R5,R2
        MOV         R15,R3
        ADD         #12,R2
        MOV.L       @R2,R1
        MOV.L       R1,@R3
        MOV.L       @(4,R2),R1
        MOV.L       R1,@(4,R3)
        MOV.L       @(8,R2),R1
        MOV.L       R1,@(8,R3)
        MOV         R15,R0
        NOP
        FMOV.S      @R0,FR2
        MOV         R15,R3
        FMOV.S      @R4,FR3
        MOV         R15,R2
        FSUB        FR3,FR2
        FMOV.S      FR2,@R0
        MOV         #4,R0
        MOV         R0,R1
        ADD         R4,R1
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @R1,FR3
        FSUB        FR3,FR2
        FMOV.S      FR2,@(R0,R3)
        MOV         #8,R0
        MOV         R0,R1
        ADD         R4,R1
        MOV         R15,R3
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @R1,FR3
        MOV         R15,R1
        FMOV.S      FR0,FR4
        FCMP/GT     FR15,FR4
        BT          L258
        MOV         R13,R5
        BSR         _scal2
        MOV         R14,R4
        FMOV.S      FR0,FR4
        FCMP/GT     FR15,FR4
        BF          L256
L258:
        BRA         L254
        MOV         #0,R0
L256:
        MOV         #1,R0
L254:
        LDS.L       @R15+,PR
        FMOV.S      @R15+,FR15
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
```

```
FSUB        FR3,FR2
FMOV.S      FR2,@(R0,R3)
MOV         R15,R3
MOV         R15,R0
NOP
MOV         #4,R8
MOV         R15,R7
ADD         R7,R8
MOV         R15,R6
MOV         #4,R7
FMOV.S      @R8,FR0
ADD         R6,R7
FMOV.S      @R1,FR2
FMOV.S      @R7,FR3
FMUL        FR0,FR3
FMOV.S      @R0,FR0
MOV         #8,R0
FMAC        FR0,FR2,FR3
FMOV.S      @(R0,R3),FR2
FMOV.S      @(R0,R2),FR0
FMAC        FR0,FR2,FR3
FMOV.S      FR3,FR5
FCMP/GT     FR4,FR5
BT          L284
MOV.L       @R5,R2
MOV         R15,R3
ADD         #24,R2
MOV.L       @R2,R1
MOV.L       R1,@R3
MOV.L       @(4,R2),R1
MOV.L       R1,@(4,R3)
MOV.L       @(8,R2),R1
MOV.L       R1,@(8,R3)
MOV         R15,R0
NOP
FMOV.S      @R0,FR2
MOV         R15,R3
FMOV.S      @R4,FR3
MOV         R15,R2
FSUB        FR3,FR2
FMOV.S      FR2,@R0
MOV         #4,R0
MOV         R0,R1
ADD         R4,R1
FMOV.S      @(R0,R3),FR2
FMOV.S      @R1,FR3
FSUB        FR3,FR2
FMOV.S      FR2,@(R0,R3)
MOV         #8,R0
MOV         R0,R1
ADD         R4,R1
```

```
        MOV         R15,R3
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @R1,FR3
        MOV         R15,R1
        FSUB        FR3,FR2
        FMOV.S      FR2,@(R0,R3)
        MOV         R15,R3
        MOV         R15,R0
        NOP
        MOV         #4,R8
        MOV         R15,R7
        ADD         R7,R8
        MOV         R15,R6
        MOV         #4,R7
        FMOV.S      @R8,FR0
        ADD         R6,R7
        FMOV.S      @R1,FR2
        FMOV.S      @R7,FR3
        FMUL        FR0,FR3
        FMOV.S      @R0,FR0
        MOV         #8,R0
        FMAC        FR0,FR2,FR3
        FMOV.S      @(R0,R3),FR2
        FMOV.S      @(R0,R2),FR0
        FMAC        FR0,FR2,FR3
        FMOV.S      FR3,FR5
        FCMP/GT     FR4,FR5
        BF          L282
L284:
        ADD         #12,R15
        MOV         #0,R0
        RTS
        MOV.L       @R15+,R8
L282:
        MOV         #1,R0
L280:
        ADD         #12,R15
        RTS
        MOV.L       @R15+,R8
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH2A-FPU	196	90	115	106
SH-2E	196	62	141	128
SH-4	196	62	123	87
SH-4A	196	62	97	93

优化前和优化后的程序分析：

下面比较加载和存储指令在两种情况下的数量。

在 x、y、z 中的单一迭代：

优化前，我们有

```
dv=poli->v[0];              一个加载指令
SubVect(&dv,p);             两个加载指令和一个存储指令
dst2=SCAL2(&dv);            两个加载指令
if (dst2>rad) rerun 0;
```

重复三次，总共 18 次加载/存储指令

优化后，我们有

```
a=*p++; d=*q++;
b=*p++; e=*q++; a-=d;
c=*p++; f=*q++; b-=e;
                c-=f;
return a*a+b*b+c*c;
```

在此，p 和 q 使用两个加载指令，乘上三次迭代则是总共六个加载/存储指令。

通过这个方法，存储器存取指令可减少至 1/3。由于 SuperH 微型计算机的指令集基本上不包含任何可直接计算存储器数据的指令，因此和在 FPU 寄存器中的运算比较起来，其指令数量有所增加。

此外，存储到存储器将中断流水线。 因此存储器存取操作次数的减少也能使流水线操作更顺畅。

补充：

在已优化的程序中，定点 P 被加载三次。

如果改进到只需要一次加载操作，则能获得更佳性能。

一般上对于定点来说，考虑到会进行多次循环处理，所以应在执行操作时将定点加载到 FPU 寄存器变量，而非结构。

5.4 转移

应加以考虑的转移相关事项如下所示。

- 应合并相同的决定。
- 当 switch 语句和“else if”语句很长时，需要快速确定及频繁转移的 case 应被放置到起始部分。
- 当 switch 和“else if”语句很长时，将它们划分为数个阶段将可增进程序执行速度。

重点：

具有五或六个 case 的 switch 语句可被更改为 if 语句，以增进执行速度。

描述：

具有较少 case 的 switch 语句应以 if 语句替换。

当在 switch 语句中将变量值引用到 case 值的表前，将检查变量值范围是否导致额外操作。

另外，if 语句涉及大量比较，这在增加所涉及 case 数量的同时将减低效率。

使用的实例：

根据变量 a 的值更改返回值：

优化前的代码

```
int  x(int a)
{
     switch (a)
     {
        case 1:
               a = 2; break;
        case 10:
               a = 4; break;
        default:
               a = 0; break;
     }
     return (a);
}
```

优化后的代码

```
int x (int a)
{
     if (a==1)
            a = 2;
     else if (a==10)
            a = 4;
     else
            a = 0;
     return (a);
}
```

扩展为汇编语言代码（优化前）

```
_x:
          MOV          R4,R0
          CMP/EQ       #1,R0
          BT           L16
          CMP/EQ       #10,R0
          BT           L17
BRA         L18
          NOP
L16:
```

扩展为汇编语言代码（优化后）

```
_x:
          MOV          R4,R0
          CMP/EQ       #1,R0
          BF           L22
          BRA          L23
          MOV          #2,R4
L22:
          CMP/EQ       #10,R0
          BF/S         L23
```

```
        BRA     L19             MOV     #0,R4
        MOV     #2,R2           MOV     #4,R4
L17:                            L23:
        BRA     L19             RTS
        MOV     #4,R2           MOV     R4,R0
L18:
        MOV     #0,R2
L19:
        RTS
        MOV     R2,R0
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	28	22	11	9
SH-2	28	22	11	9
SH-2A	22	20	8	5
SH2A-FPU	22	20	8	5
SH-2E	28	22	11	9
SH2-DSP(SH7065)	28	22	12	10
SH-3	28	22	11	9
SH3-DSP	28	22	12	10
SH-4	28	22	8	7
SH-4A	28	22	7	7
SH4AL-DSP	28	22	7	7

注意： a=1

5.5 内联扩展

表 5.6 列出应加以考虑的内联扩展相关事项。

表 5.6 有关内联扩展的建议

区域	建议	节
函数的内联扩展	对频繁调用的函数进行内联扩展会带来好处。 然而，函数扩展会增加程序大小。 在选择这项功能时应考虑到执行速度与可用 ROM 之间的平衡。	5.5.1
带嵌入式汇编语言的内联扩展	使用和 C 语言函数相同的界面将可调用以汇编语言编写的代码。	5.5.2

5.5.1 函数的内联扩展

重点：

频繁调用的函数可被内联扩展，以增进执行速度。

描述：

通过对频繁调用的函数进行内联扩展，执行速度可获得增进。尤其对循环内的调用目标函数进行扩展将可获得更出色效果。 应仅在有必要增进执行速度时，以增加程序大小的代价使用此选项。

使用的实例：

交换数组 a 和数组 b 的元素：

优化前的代码

```
 int x[10], y[10];
 static void g(int *a, int *b, int i)
{
           int temp;

           temp = a[i];
           a[i] = b[i];
           b[i] = temp;
}

void f (void)
{
           int i;

           for (i=0;i<10;i++)
                        g(x, y, i);
}
```

优化后的代码

```
int x[10], y[10];
#pragma inline (g)
static void g(int *a, int *b, int i)
{
          int temp;

            temp = a[i];
            a[i] = b[i];
            b[i] = temp;
}

void f (void)
{
            int i;

            for (i=0;i<10;i++)
                          g(x, y, i);
}
```

扩展为汇编语言代码（优化前）

```
_$g:
            SHLL2       R6
            MOV         R6,R0
            MOV.L       @(R0,R4),R1
            MOV.L       @(R0,R5),R2
            MOV.L       R2,@(R0,R4)
            RTS
            MOV.L       R1,@(R0,R5)
_f:
            MOV.L       R11,@-R15
            MOV.L       R12,@-R15
            MOV.L       R13,@-R15
            MOV.L       R14,@-R15
            STS.L       PR,@-R15
            MOV         #0,R14
            MOV.L       L14+2,R12
            MOV.L       L14+6,R13
            MOV         #10,R11
L12:
            MOV         R14,R6
            MOV         R12,R4
            MOV         R13,R5
            BSR         _$g
            ADD         #1,R14
            CMP/GE      R11,R14
            BF          L12
            LDS.L       @R15+,PR
            MOV.L       @R15+,R14
            MOV.L       @R15+,R13
            MOV.L       @R15+,R12
            RTS
            MOV.L       @R15+,R11
L14:
            .RES.W      1
            .DATA.L     _x
            .DATA.L     _y
```

扩展为汇编语言代码（优化后）

```
_f:
            MOV         #10,R1
            MOV.L       L13+2,R4
            MOV.L       L13+6,R5
L11:
            MOV.L       @R5,R6
            MOV.L       @R4,R2
            DT           R1
            MOV.L       R2,@R5
            MOV.L       R6,@R4
            ADD         #4,R5
            BF/S        L11
            ADD         #4,R4
            RTS
            NOP
L13:
            .RES.W      1
            .DATA.L     _y
            .DATA.L     _x
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	54	36	210	137
SH-2	54	36	210	118
SH-2A	38	32	164	74
SH2A-FPU	50	32	187	74
SH-2E	54	36	210	118
SH2-DSP(SH7065)	52	36	305	138
SH-3	54	36	234	147
SH3-DSP	52	36	294	156
SH-4	54	36	203	97
SH-4A	54	36	155	85
SH4AL-DSP	52	36	185	85

5.5.2 使用嵌入式汇编语言的内联扩展

重点：

汇编语言代码可包含在 C 程序内以增进执行速度。

描述：

有时为增强性能，尤其为增进执行速度，将有需要以汇编语言来编写代码。在这种情况下，可以只在编写重要代码时使用汇编语言，并以调用 C 语言函数的相同方式来进行调用。 这项功能必须与 -code=asmcode 选项配合使用。

使用的实例：

交换数组 big 中元素的高位及低位字节，并将结果存储在数组 little 中。

优化前的代码

```
#define A_MAX 10
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
      short ret;

     *((UChar *)(&ret)+1) =
      *((UChar *)(&p1));
     *((UChar *)(&ret)) =
      *((UChar *)(&p1)+1);
    return ret;
}

void f (void)
{
      int  i;
      short *x, *y;

      x = little;
      y = big;
      for(i=0; i<A_MAX; i++,
                          x++, y++){
            *x = swap(*y);
      }
}
```

优化后的代码

```
#define A_MAX 10
#pragma inline_asm (swap)
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
     SWAP.B R4,R0
}

void f (void)
{
     int  i;
     short *x, *y;

     x = little;
     y = big;
     for(i=0; i<A_MAX; i++, x++, y++){
                    *x = swap(*y);
    }
}
```

扩展为汇编语言代码（优化前）

```
_swap:
          ADD         #-8,R15
          MOV         R4,R0
          MOV.L       R4,@R15
          MOV.W       R0,@(2,R15)
          MOV.B       @(2,R15),R0
```

扩展为汇编语言代码（优化后）

```
_swap:
SWAP.B R4,R0
          .ALIGN      4
          RTS
          NOP
_f:
```

```
        MOV.B   R0,@(5,R15)
        MOV.B   @(3,R15),R0
        MOV.B   R0,@(4,R15)
        MOV.W   @(4,R15),R0
        RTS
        ADD     #8,R15
_f:
        MOV.L   R12,@-R15
        MOV.L   R13,@-R15
        MOV.L   R14,@-R15
        STS.L   PR,@-R15
        MOV.L   L14+2,R13
        MOV.L   L14+6,R14
        MOV     #10,R12
L12:
        BSR     _swap
        MOV.W   @R14+,R4
        DT      R12
        MOV.W   R0,@R13
        BF/S    L12
        ADD     #2,R13
        LDS.L   @R15+,PR
        MOV.L   @R15+,R14
        MOV.L   @R15+,R13
        RTS
        MOV.L   @R15+,R12
L14:
        .RES.W  1
        .DATA.L _little
        .DATA.L _big
```

```
MOV.L   R12,@-R15
        MOV.L   R13,@-R15
        MOV.L   R14,@-R15
        MOV.L   L15,R13
        MOV.L   L15+4,R14
        MOV     #10,R12
L12:
        BRA     L14
        MOV.W   @R14+,R4
L15:
        .DATA.L _little
        .DATA.L _big
L14:
         SWAP.B R4,R0
        .ALIGN  4
        DT      R12
        MOV.W   R0,@R13
        BT/S    L17
        ADD     #2,R13
        MOV.L   L16+2,R3
        JMP     @R3
        NOP
L17:
        MOV.L   @R15+,R14
        MOV.L   @R15+,R13
        RTS
        MOV.L   @R15+,R12
L16:
        .RES.W  1
        .DATA.L L12
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	46	68	285	172
SH-2	46	64	286	171
SH-2A	30	60	202	126
SH2A-FPU	42	76	215	148
SH-2E	46	64	286	171
SH2-DSP(SH7065)	46	64	318	193
SH-3	46	64	333	185
SH3-DSP	46	64	332	177
SH-4	46	64	249	140
SH-4A	46	64	182	117
SH4AL-DSP	46	64	182	117

5.6 全局基址寄存器的使用 (GBR)

5.6.1 使用全局基址寄存器 (GBR) 的偏移引用

重点：

通过 GBR 和偏移量来引用外部变量，将可增进性能。

描述：

以 GBR 为基址寄存器，使用偏移来引用被频繁存取的外部变量，将可生成更简练的目标代码。 此外，指令的数量也将被缩减，从而增进了执行速度。

使用的实例：

将结构 y 的内容代入到结构 x：

注意： 在此实例中，编译选项是 –cpu=sh2 –gbr=user。

优化前的代码

```
struct {
      char  c1;
      char  c2;
      short s1;
      short s2;
      long  l1;
      long  l2;
} x, y;

void f (void)
{
    x.c1 = y.c1;
    x.c2 = y.c2;
    x.s1 = y.s1;
    x.s2 = y.s2;
    x.l1 = y.l1;
    x.l2 = y.l2;
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.L       L12,R5
          MOV.L       L12+4,R6
          MOV.B       @(1,R5),R0
          MOV.B       @R5,R1
          MOV.B       R0,@(1,R6)
          MOV.W       @(2,R5),R0
          MOV.L       @(8,R5),R4
          MOV.W       R0,@(2,R6)
```

优化后的代码

```
#pragma gbr_base(x,y)
struct {
      char  c1;
      char  c2;
      short s1;
      short s2;
      long  l1;
      long  l2;
} x, y;

void f (void)
{
      x.c1 = y.c1;
      x.c2 = y.c2;
      x.s1 = y.s1;
      x.s2 = y.s2;
      x.l1 = y.l1;
      x.l2 = y.l2;
}
```

扩展为汇编语言代码（优化后）

```
_f:
   MOV.B  @(_y2-(STARTOF $G0),GBR),R0
   MOV.B  R0,@(_x2-(STARTOF $G0),GBR)
   MOV.B  @(_y2-(STARTOF $G0)+1,GBR),R0
   MOV.B  R0,@(_x2-(STARTOF $G0)+1,GBR)
   MOV.W  @(_y2-(STARTOF $G0)+2,GBR),R0
   MOV.W  R0,@(_x2-(STARTOF $G0)+2,GBR)
   MOV.W  @(_y2-(STARTOF $G0)+4,GBR),R0
   MOV.W  R0,@(_x2-(STARTOF $G0)+4,GBR)
```

```
        MOV.W       @(4,R5),R0          MOV.L   R0,@(_x2-(STARTOF $G0)+8,GBR)
        MOV.L       @(12,R5),R7         MOV.L   @(_y2-(STARTOF $G0)+12,GBR),R0
        MOV.B       R1,@R6              RTS
        MOV.W       R0,@(4,R6)          MOV.L   R0,@(_x2-(STARTOF $G0)+12,GBR)
        MOV.L       R4,@(8,R6)
        RTS
        MOV.L       R7,@(12,R6)
L12:
      .DATA.L     _y
      .DATA.L     _x
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	40	26	22	25
SH-2	40	26	22	25
SH-2A	40	26	17	18
SH2A-FPU	40	26	17	18
SH-2E	40	26	22	25
SH2-DSP(SH7065)	40	26	22	25
SH-3	40	26	26	27
SH3-DSP	40	26	36	31
SH-4	40	26	18	21
SH-4A	40	26	15	13
SH4AL-DSp	40	26	15	13

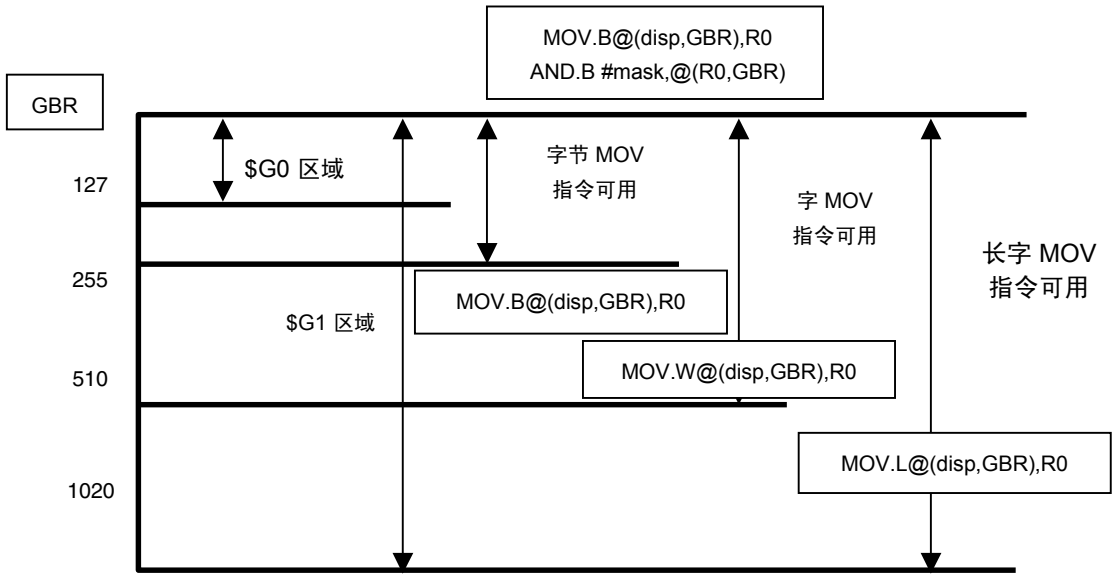
5.6.2 全局基址寄存器 (GBR) 区域的选择性使用

重点：

GBR0 和 GBR1 区域的选择性使用将可增进性能。

描述：

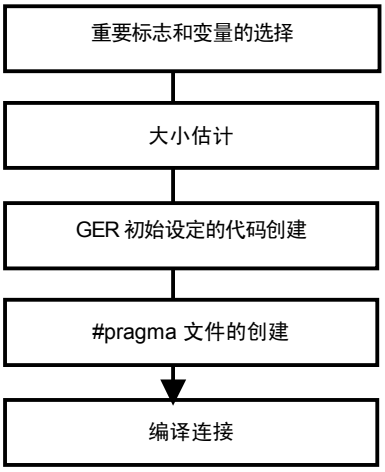
- #pragma gbr_base 区域



- GBR 基本寻址的特性与应用。

段（区域）	特性	应用
$G0 (bgr_base)	有效进行字节数据的位处理、设置和引用。	字节的标志数据
$G1 (bgr_base1)	有效进行数据的设置和引用。	一般变量

- 使用 GBR 基址的步骤：



选择重要的标志和变量，并提名它们为 #pragma gbr_base 的候选者。

检查 $G0 和 $G1 各个段的大小，并确保 $G1 的段大小分别处于字节、字和长字的区域大小范围内。

```
MOV #STARTOF $G0,R0
LDC  R0,GBR
```

在对 gbr_base 进行初始存取前需要进行初始设定。

在连接上，$G1 段应紧随 $G0 段之后分配。

使用的实例：

存取位字段：

注意： 在此实例中，编译选项是 –cpu=sh2 –gbr=user。

C 源

```
#pragma gbr_base (bitf)

struct BitField {
  unsigned char a : 1 ;
  unsigned char b : 1 ;
  unsigned char c : 1 ;
  unsigned char d : 1 ;
  unsigned char e : 1 ;
  unsigned char f : 1 ;
  unsigned char g : 1 ;
  unsigned char h : 1 ;
} bitf ;

main()
{
  bitf.a = 1 ; // bit set
  bitf.b = 0 ; // bit
clear

  if (bitf.c)
    bitf.d = 1 ;
  else
    bitf.e = 1 ;
}
```

未指定 #pragma

```
.EXPORT     _bitf
.EXPORT     _main
.SECTION    P,CODE,ALIGN=4
_main:      ; function: main
            ; frame size=0
MOV.L       L241,R4  ;  _bitf
MOV.B       @R4,R0
OR          #128,R0
MOV.B       R0,@R4
MOV.B       @R4,R0
AND         #191,R0
MOV.B       R0,@R4
MOV         R4,R0
MOV.B       @R0,R0
TST         #32,R0
BT          L238
MOV.B       @R4,R0
BRA         L240
OR          #16,R0
L238:
MOV.B       @R4,R0
OR          #8,R0
L240:
RTS
MOV.B       R0,@R4
L241:
.DATA.L     _bitf
.SECTION    B,DATA,ALIGN=4
_bitf:      ; static: bitf
            .RES.B      1
            .END
```

指定了 #pragma

```
.EXPORT     _bitf
.EXPORT     _main
.SECTION    ,CODE,ALIGN=4
 main:       ;function:main
             ; frame size=0
  MOV #_bitf-(STARTOF$G0),R0
  OR.B    #128,@(R0,GBR)
  AND.B   #191,@(R0,GBR)
  TST.B    #32,@(R0,GBR)
  BT          L238
  BRA         L239
  OR.B    #16,@(R0,GBR)
L238:
  OR.B    #8,@(R0,GBR)
L239:
  RTS
  NOP
.SECTION    G0,DATA,ALIGN=4
_bitf:        ; static: bitf
.DATAB.B    1,0
.END
```

5.7 寄存器保存/恢复操作的控制

重点：

通过寄存器保存/恢复操作的创新，使执行速度获得增进。（版本 8）

描述：

通过在终端函数的入口和出口删除变量寄存器的保存和恢复操作，将可增进执行速度和 ROM 的效率。然而，下列必须执行的其中一种处理可能会降低，而非提高性能。因此必须谨慎研究要应用此方法的代码。

(1) 寄存器变量的寄存器必须在寄存器保存/恢复操作被省略之函数的调用源函数保存/恢复。
(2) 目标不可为生成函数调用的寄存器变量分配寄存器。

使用的实例：

使用函数表合并堆栈保存/恢复：

优化前的代码

```
#define LISTMAX 2
typedef
   int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
```

优化后的代码

```
#pragma regsave (table)
#pragma noregalloc (table)
#pragma noregsave (init, copy, sum)
#define LISTMAX 2
typedef
   int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY  ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
{
    int    i, j, k;
```

```
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                                    q[i][j][j];
}
```

扩展为汇编语言代码（优化前）

```
_table:
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV.L       L270+6,R14
          MOV.L       L270+10,R4
          BSR         _init
          MOV         R14,R5
          MOV.L       L270+14,R5
          BSR         _copy
          MOV         R14,R4
          MOV         R14,R4
          LDS.L       @R15+,PR
          MOV.L       L270+18,R6
          MOV.L       L270+14,R5
          BRA         _sum
          MOV.L       @R15+,R14
_init:
          MOV         #2,R6
          MOV.L       R13,@-R15
          MOV         #0,R13
          MOV.L       R12,@-R15
          MOV         R5,R12
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV.L       R8,@-R15
          MOV         R5,R8
          STS.L       MACL,@-R15
          ADD         #32,R8
          MOV.W       L270,R9
          MOV.W       L270+2,R10
L261:
```

```
    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                                    q[i][j][j];
}
```

扩展为汇编语言代码（优化后）

```
_table:
          MOV.L       R14,@-R15
          MOV.L       R13,@-R15
          MOV.L       R12,@-R15
          MOV.L       R11,@-R15
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV.L       R8,@-R15
          FMOV.S      FR15,@-R15
          FMOV.S      FR14,@-R15
          FMOV.S      FR13,@-R15
          FMOV.S      FR12,@-R15
          STS.L       PR,@-R15
          MOV.L       L270+10,R4
          MOV.L       L270+6,R5
          STS.L       MACH,@-R15
          STS.L       MACL,@-R15
          BSR         _init
          NOP
          MOV.L       L270+6,R4
          MOV.L       L270+14,R5test3
          BSR         _copy
          NOP
          MOV.L       L270+6,R4
          MOV.L       L270+14,R5test3
          MOV.L       L270+18,R6
          BSR         _sum
          NOP
          LDS.L       @R15+,MACL
          LDS.L       @R15+,MACH
          LDS.L       @R15+,PR
```

```
        MOV     R13,R1
        MOV     R12,R0
L262:
        MOV     R13,R7
        MOV     R0,R5
L263:
        MUL.L   R10,R4
        ADD     #1,R7
        STS     MACL,R3
        MOV     R3,R4
        AND     R9,R4
        CMP/GE  R6,R7
        MOV.L   R4,@R5
        BF/S    L263
        ADD     #4,R5
        ADD     #1,R1
        CMP/GE  R6,R1
        BF/S    L262
        ADD     #8,R0
        ADD     #16,R12
        CMP/HS  R8,R12
        BF       L261
        LDS.L   @R15+,MACL
        MOV.L   @R15+,R8
        MOV.L   @R15+,R9
        MOV.L   @R15+,R10
        MOV.L   @R15+,R12
        RTS
        MOV.L   @R15+,R13
_copy:
        MOV.L   R14,@-R15
        MOV.L   R13,@-R15
        MOV     #2,R13
        MOV.L   R11,@-R15
        MOV.L   R10,@-R15
        MOV.L   R9,@-R15
        MOV     #0,R9
        MOV.L   R8,@-R15
        MOV     R9,R14
        ADD     #-4,R15
        MOV     R5,R8
        ADD     #32,R8
L264:
        MOV     R9,R7
        MOV     R14,R10
        SHLL2   R10
        SHLL    R10
        MOV     R14,R3
        SHLL2   R3
        SHLL2   R3
        ADD     R4,R3
```

```
        FMOV.S  @R15+,FR12
        FMOV.S  @R15+,FR13
        FMOV.S  @R15+,FR14
        FMOV.S  @R15+,FR15
        MOV.L   @R15+,R8
        MOV.L   @R15+,R9
        MOV.L   @R15+,R10
        MOV.L   @R15+,R11
        MOV.L   @R15+,R12
        MOV.L   @R15+,R13
        RTS
        MOV.L   @R15+,R14
_init:
        MOV.W   L270+2,R10
        MOV     R5,R8
        MOV.W   L270,R9
        ADD     #32,R8
        MOV     #2,R6
        MOV     R5,R12
        MOV     #0,R13
L261:
        MOV     R12,R0
        MOV     R13,R1
L262:
        MOV     R0,R5
        MOV     R13,R7
L263:
        MUL.L   R10,R4
        ADD     #1,R7
        CMP/GE  R6,R7
        STS     MACL,R3
        MOV     R3,R4
        AND     R9,R4
        MOV.L   R4,@R5
        BF/S    L263
        ADD     #4,R5
        ADD     #1,R1
        CMP/GE  R6,R1
        BF/S    L262
        ADD     #8,R0
        ADD     #16,R12
        CMP/HS  R8,R12
        BF      L261
        RTS
        NOP
_copy:
        ADD     #-4,R15
        MOV     R5,R8
        MOV     #0,R9
        ADD     #32,R8
        MOV     R9,R14
```

```
        MOV.L   R3,@R15
L265:
        MOV.L   @R15,R3
        MOV     R5,R6
        MOV     R7,R11
        SHLL2   R11
        SHLL    R11
        ADD     R3,R11
        MOV     R7,R1
        SHLL2   R1
L266:
        MOV.L   @R11+,R3
        MOV     R6,R0
        ADD     R10,R0
        ADD     #16,R6
        CMP/HS  R8,R6
        BF/S    L266
        MOV.L   R3,@(R0,R1)
        ADD     #1,R7
        CMP/GE  R13,R7
        BF      L265
        ADD     #1,R14
        CMP/GE  R13,R14
        BF      L264
        ADD     #4,R15
        MOV.L   @R15+,R8
        MOV.L   @R15+,R9
        MOV.L   @R15+,R10
        MOV.L   @R15+,R11
        MOV.L   @R15+,R13
        RTS
        MOV.L   @R15+,R14
L270:
        .DATA.W H'3FFF
        .DATA.W H'051D
        .DATA.W 0
        .DATA.L _ary1
        .DATA.L H'00012403
        .DATA.L _ary2
        .DATA.L _ary3
_sum:
        MOV.L   R14,@-R15
        MOV.L   R13,@-R15
        MOV.L   R12,@-R15
        MOV     #0,R12
        MOV.L   R11,@-R15
        MOV     #2,R11
        MOV.L   R10,@-R15
        MOV.L   R9,@-R15
        MOV.L   R8,@-R15
        ADD     #-4,R15
        MOV     #2,R13
L264:
        MOV     R14,R3
        SHLL2   R3
        SHLL2   R3
        MOV     R14,R10
        ADD     R4,R3
        MOV     R9,R7
        SHLL2   R10
        MOV.L   R3,@R15
        SHLL    R10
L265:
        MOV     R7,R11
        MOV.L   @R15,R3
        SHLL2   R11
        MOV     R7,R1
        SHLL    R11
        MOV     R5,R6
        SHLL2   R1
        ADD     R3,R11
L266:
        MOV     R6,R0
        ADD     #16,R6
        MOV.L   @R11+,R3
        CMP/HS  R8,R6
        ADD     R10,R0
        BF/S    L266
        MOV.L   R3,@(R0,R1)
        ADD     #1,R7
        CMP/GE  R13,R7
        BF      L265
        ADD     #1,R14
        CMP/GE  R13,R14
        BF      L264
        RTS
        ADD     #4,R15
_sum:
        ADD     #-4,R15
        MOV     #0,R12
        MOV     R12,R8
        MOV     #2,R11
L267:
        MOV     R8,R13
        SHLL2   R13
        SHLL2   R13
        MOV     R12,R10
L268:
        MOV     R10,R14
        MOV     R10,R3
        SHLL2   R3
        MOV     R12,R9
```

```
        MOV     R12,R8
L267:
        MOV     R12,R10
        MOV     R8,R13
        SHLL2   R13
        SHLL2   R13
L268:
        MOV     R12,R9
        MOV     R12,R7
        MOV     R10,R14
        SHLL2   R14
        SHLL    R14
        MOV     R10,R3
        SHLL2   R3
        MOV.L   R3,@R15
L269:
        MOV     R13,R0
        ADD     R6,R0
        ADD     R14,R0
        ADD     R7,R0
        MOV     R13,R3
        MOV.L   R0,@-R15
        MOV     R13,R2
        MOV.L   @(4,R15),R0
        ADD     #1,R9
        ADD     R5,R3
        ADD     R14,R3
        MOV.L   @(R0,R3),R3
        ADD     R4,R2
        ADD     R14,R2
        ADD     R7,R2
        MOV.L   @R2,R1
        CMP/GE  R11,R9
        MOV.L   @R15+,R2
        ADD     R1,R3
        MOV.L   R3,@R2
        BF/S    L269
        ADD     #4,R7
        ADD     #1,R10
        CMP/GE  R11,R10
        BF      L268
        ADD     #1,R8
        CMP/GE  R11,R8
        BF      L267
        ADD     #4,R15
        MOV.L   @R15+,R8
        MOV.L   @R15+,R9
        MOV.L   @R15+,R10
        MOV.L   @R15+,R11
        MOV.L   @R15+,R12
        MOV.L   @R15+,R13
        SHLL2   R14
        MOV.L   R3,@R15
        SHLL    R14
        MOV     R12,R7
L269:
        MOV     R13,R0
        ADD     R6,R0
        ADD     R14,R0
        MOV     R13,R2
        ADD     R7,R0
        MOV     R13,R3
        ADD     R4,R2
        MOV.L   R0,@-R15
        ADD     R14,R2
        MOV.L   @(4,R15),R0
        ADD     R5,R3
        ADD     R7,R2
        ADD     R14,R3
        MOV.L   @R2,R1
        MOV.L   @(R0,R3),R3
        ADD     #1,R9
        MOV.L   @R15+,R2
        CMP/GE  R11,R9
        ADD     R1,R3
        MOV.L   R3,@R2
        BF/S    L269
        ADD     #4,R7
        ADD     #1,R10
        CMP/GE  R11,R10
        BF      L268
        ADD     #1,R8
        CMP/GE  R11,R8
        BF      L267
        RTS
        ADD     #4,R15
L270:
        .DATA.W H'3FFF
        .DATA.W H'051D
        .DATA.W 0
        .DATA.L _ary1
        .DATA.L H'00012403
        .DATA.L _ary2
        .DATA.L _ary3
```

```
        RTS
        MOV.L       @R15+,R14
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	292	288	684	669
SH-2	238	242	446	426
SH-2E	238	258	446	438
SH2-DSP(SH7065)	236	252	490	470
SH-3	238	242	476	458
SH3-DSP	236	252	489	487
SH-4	238	258	301	313

5.8 使用二字节地址的规格

重点：

通过使用二字节来代表变量和函数地址，将可增进 ROM 的使用效率。

描述：

若变量和函数的所在地址可使用二字节来代表，同时引用侧的代码地址以二字节指定时，代码的大小将会缩小。

使用的实例：

在变量 x 的值是 1 时调用外部函数 g：

优化前的代码

```
extern int  x;
extern void g(void);

void f (void)
{
   if (x == 1)
        g();
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.L       L218+2,R3
          MOV.L       @R3,R0
          CMP/EQ      #1,R0
          BF          L219
          MOV.L       L218+6,R2
          JMP         @R2
          NOP
L219:
          RTS
          NOP
L218:
          .DATA.W     0
          .DATA.L     _x
          .DATA.L     _g
```

优化后的代码

```
#pragma abs16(x,g)
extern int  x;
extern void g(void);

void f (void)
{
   if (x == 1)
        g();
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.W       L238+2,R3
          MOV.L       @R3,R0
          CMP/EQ      #1,R0
          BF          L239
          MOV.W       L238,R2
          JMP         @R2
          NOP
L239:
          RTS
          NOP
L238:
          .DATA.W     _g
          .DATA.W     _x
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-1	28	28	15	11
SH-2	28	28	15	11
SH-2A	24	24	13	11
SH2A-DSP	24	24	13	11
SH-2E	28	28	15	11
SH2-DSP(SH7065)	28	28	16	12
SH-3	28	28	15	11
SH3-DSP	28	28	17	12
SH-4	28	28	13	10
SH-4A	28	28	9	6
SH4AL-DSP	28	28	9	6

注意： x =1，函数 g 是 void g(){}

5.9 使用高速缓存

性能可通过高速缓存的有效使用获得增进。

5.9.1 预取指令

重点：

当存取数组变量时，通过在使用前执行预取指令将可增进执行速度。（仅限 SH-2A、SH2A-FPU、SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP）

描述：

当在循环中连续存取数组时，通过在引用数组成员前执行预取将可增进执行速度。同时，通过扩展循环可能获得更有效的预取操作。

然而，持续执行预取指令无法增进速度。 接下来的预取指令仅可在之前的预取指令完成后执行。

使用的实例：

在元素 d 中存储使用数组“数据”的元素 a、b 和 c 执行操作的结果 (SH-4)：

注意： 在此实例中，编译选项是 –cpu=sh4 –fpu=single。

优化前的代码

```
typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;

    for( p1=data, p2=data+1; p1<end;
 p1+=2, p2+=2 ){
        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
        c1 = 1/t11;
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;
```

优化后的代码

```
#include <machine.h>

typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;
    data_t *next = data+4;

    for( p1=data, p2=data+1; p1<end;
 p1+=2, p2+=2 ){
        prefetch(next);
        next += 2;

        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
```

```
        a2 = p2->a;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

扩展为汇编语言代码（优化前）

```
_f:
          MOV.L      L252+6,R6
          MOV.L      L252+2,R7
          MOV         R7,R5
          MOV         R7,R4
          ADD         R7,R6
          CMP/HS      R6,R4
          ADD         #16,R5
          BT/S        L250
          FLDI1       FR5
L251:
          MOV         #4,R0
          FMOV.S      @R4,FR4
          FMOV.S      @(R0,R4),FR6
          MOV         #8,R0
          FMOV.S      FR4,FR7
          FMUL        FR4,FR7
          FMOV.S      FR6,FR8
          FMUL        FR6,FR8
          FADD        FR6,FR4
          FADD        FR8,FR7
          FMOV.S      FR7,FR3
          FMOV.S      FR5,FR7
          FDIV        FR3,FR7
          FADD        FR7,FR4
          FMOV.S      FR7,@(R0,R4)
          MOV         #12,R0
          FMOV.S      FR4,@(R0,R4)
          MOV         #4,R0
          FMOV.S      @(R0,R5),FR6
```

```
        c1 = 1/t11;
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;

        a2 = p2->a;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

扩展为汇编语言代码（优化后）

```
_f:
          MOV.L      L253+6,R7
          MOV.L      L253+2,R0
          MOV         R0,R4
          MOV         R0,R5
          ADD         R0,R7
          MOV         R0,R6
          CMP/HS      R7,R4
          ADD         #16,R5
          ADD         #64,R6
          BT/S        L251
          FLDI1       FR5
L252:
          PREF        @R6
          MOV         #4,R0
          FMOV.S      @R4,FR4
          FMOV.S      @(R0,R4),FR6
          MOV         #8,R0
          FMOV.S      FR4,FR7
          FMUL        FR4,FR7
          FMOV.S      FR6,FR8
          FMUL        FR6,FR8
          FADD        FR6,FR4
          ADD         #32,R6
          FADD        FR8,FR7
          FMOV.S      FR7,FR3
          FMOV.S      FR5,FR7
          FDIV        FR3,FR7
          FADD        FR7,FR4
          FMOV.S      FR7,@(R0,R4)
```

```
        MOV     #8,R0
        FMOV.S  @R5,FR4
        FMOV.S  FR6,FR8
        FMUL    FR6,FR8
        FMOV.S  FR4,FR7
        FMUL    FR4,FR7
        FADD    FR6,FR4
        FADD    FR8,FR7
        FMOV.S  FR7,FR3
        FMOV.S  FR5,FR7
        FDIV    FR3,FR7
        FADD    FR7,FR4
        FMOV.S  FR7,@(R0,R5)
        ADD     #32,R4
        MOV     #12,R0
        CMP/HS  R6,R4
        FMOV.S  FR4,@(R0,R5)
        BF/S    L251
        ADD     #32,R5
L250:
        RTS
        NOP
L252:
        .DATA.W 0
        .DATA.L _data
        .DATA.L H'00008000
```

```
        MOV     #12,R0
        FMOV.S  FR4,@(R0,R4)
        MOV     #4,R0
        FMOV.S  @(R0,R5),FR6
        MOV     #8,R0
        FMOV.S  @R5,FR4
        FMOV.S  FR6,FR8
        FMUL    FR6,FR8
        FMOV.S  FR4,FR7
        FMUL    FR4,FR7
        FADD    FR6,FR4
        FADD    FR8,FR7
        FMOV.S  FR7,FR3
        FMOV.S  FR5,FR7
        FDIV    FR3,FR7
        FMOV.S  FR7,@(R0,R5)
        FADD    FR7,FR4
        ADD     #32,R4
        MOV     #12,R0
        CMP/HS  R7,R4
        FMOV.S  FR4,@(R0,R5)
        BF/S    L252
        ADD     #32,R5
L251:
        RTS
        NOP
L253:
        .DATA.W 0
        .DATA.L _data
        .DATA.L H'00008000
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-3	18	24	609	31
SH3-DSP	18	16	609	21
SH-4	16	12	608	10

注意： 1. 对于 SH-3、SH3-DSP 和 SH-4，将程序加载入外部存储器。
2. 对于 SH-3 和 SH3-DSP，在将外部存储器的存取周期数设置到 16 后执行测量。
3. 对于 SH-4，在将存储器的存取等待周期数设置到 15 后执行测量。
4. 应考虑高速缓存遗漏。

5.9.2 平铺

重点：

在此方法中，所创建的程序将专注于数据存取，从而减少数据的高速缓存遗漏。

换言之，在此技术中，可在命中高速缓存时执行的计算将首先被执行。

描述：

举一个简单的例子，考虑取两个数组 A 和 B 的差别总和所创建的一个数组。

通过创建具有不同存取顺序的程序，数据的高速缓存遗漏将可被减少。

使用的实例：

对于数组成员 a、b、c 和 d，使用结构来执行计算

di=∑j bj-aj

优化前的代码

```
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q;
    data_t *p_end = &data[n];
    data_t *q_end = p_end;
    float a,d;

    for (p = data; p < p_end; p++){
        a = p->a;
        d = 0.0f;
        for (q = data; q < q_end; q++){
            d += q->b -a;
        }
        p->d=d;
   }
}
```

优化后的代码

```
#define STRIDE 512
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q, *end=&data[n];
    data_t *pp, *qq;
    data_t *pp_end, *qq_end;
    float a,d;


    for (p = data; p < end; p = pp_end){
        pp_end = p + STRIDE;
        for (q = data; q < end; q = qq_end){
            qq_end = q + STRIDE;
            for (pp = p; pp < pp_end && pp

                a = pp->a;
                d = pp->d;
                for (qq = q; qq < qq_end
                                        && qq < end;
qq++){
                    d += qq->b -a;
                }
                p->d = d;
            }
        }
   }
}
```

扩展为汇编语言代码（优化前）

```
_f:
            MOV             R5,R1
            SHLL2           R1
            SHLL2           R1
            FLDI0           FR6
            ADD             R4,R1
            BRA             L244
            MOV             R4,R6
L245:
            MOV             R4,R5
            FMOV.S          @R6,FR5
            CMP/HS          R1,R5
            BT/S            L246
            FMOV.S          FR6,FR4
L247:
            STS             FPSCR,R3
            MOV.L           L248,R2
            MOV             #4,R0
            FMOV.S          @(R0,R5),FR3
            ADD             #16,R5
            AND             R2,R3
            CMP/HS          R1,R5
            LDS             R3,FPSCR
            FSUB            FR5,FR3
            BF/S            L247
            FADD            FR3,FR4
L246:
            MOV             #12,R0
            FMOV.S          FR4,@(R0,R6)
            ADD             #16,R6
L244:
            CMP/HS          R1,R6
            BF              L245
            RTS
            NOP
L248:
              .DATA.L       H'FFE7FFFF
              .END
```

扩展为汇编语言代码（优化后）

```
_f:
            MOV.L           R14,@-R15
            MOV             R5,R7
            MOV.L           R13,@-R15
            SHLL2           R7
            MOV.L           R11,@-R15
            SHLL2           R7
            MOV.L           R10,@-R15
            ADD             R4,R7
            MOV.W           L259,R11
            BRA             L249
            MOV             R4,R13
L250:
            MOV             R13,R10
            ADD             R11,R10
            BRA             L251
            MOV             R4,R14
L252:
            MOV             R14,R1
            ADD             R11,R1
            BRA             L253
            MOV             R13,R6
L254:
            MOV             #12,R0
            FMOV.S          @R6,FR5
            FMOV.S          @(R0,R6),FR4
            BRA             L255
            MOV             R14,R5
L256:
            STS             FPSCR,R3
            MOV.L           L259+2,R2
            MOV             #4,R0
            FMOV.S          @(R0,R5),FR3
            ADD             #16,R5
            AND             R2,R3
            LDS             R3,FPSCR
            FSUB            FR5,FR3
            FADD            FR3,FR4
L255:
            CMP/HS          R1,R5
            BT              L257
            CMP/HS          R7,R5
            BF              L256
L257:
            MOV             #12,R0
            ADD             #16,R6
            FMOV.S          FR4,@(R0,R13)
L253:
            CMP/HS          R10,R6
            BT              L258
```

```
                CMP/HS      R7,R6
                BF          L254
L258:
                MOV         R1,R14
L251:
                CMP/HS      R7,R14
                BF          L252
                MOV         R10,R13
L249:
                CMP/HS      R7,R13
                BF          L250
                MOV.L       @R15+,R10
                MOV.L       @R15+,R11
                MOV.L       @R15+,R13
                RTS
                MOV.L       @R15+,R14
L259:
                .DATA.W     H'2000
                .DATA.L     H'FFE7FFFF
                .END
                .END
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-3	76	132	931×10^3	725×10^3
SH3-DSP	76	132	940×10^3	697×10^3
SH-4	52	104	315×10^3	43×10^3

注意： 1. n = 4096, STRIDE = 512
2. 应考虑高速缓存遗漏。

优化前和优化后的程序分析：

在优化后，循环将具有四重嵌套，同时处理复杂，代码大小将增加。 然而，通过这种处理方法，与高速缓存遗漏关联的内务操作将可被缩减。 这项技术在只有小量数据要处理时不具效率，但对于大型数据大小较具效率。

在优化前，data[0] 到 data[n-1] 的值被连续引用以计算 data[0]->d。

然后为了计算 data[1]->d，data[0] 到 data[n-1] 的值将再次被引用，但数组数据的大小比高速缓存的大小较大，data[0] 的值将不再存在于高速缓存内，造成高速缓存遗漏。

因为大区域中的数据被连续引用，当相同的数据在下一次被引用时，它已不再存在于高速缓存中。

在已被优化的程序中，数据被划分为较小的区域以供存取，因此在数据存取的过程中将较少发生高速缓存遗漏的情形。计算的顺序将被更改，以便使用相同数据的其他计算在高速缓存命中期间也会执行。

5.10 矩阵运算

重点：

若在矩阵运算中使用了固有函数，将可增进执行速度。

在这里作为乘数的数组必须预先存储在浮点扩展寄存器中。

描述：

具有四行和四列的矩阵的积通常透过使用循环的连续运算来计算，造成过程复杂，很少能够快速执行。 然而，SH-4 支持用于矩阵运算的固有函数，通过使用这些函数，可能可以在执行速度上获得明显的增进。

使用的实例：

将矩阵 data 和矩阵 tbl 的积存储在矩阵 ret 内：

注意： 在此实例中，编译选项是 –cpu=sh4 –fpu=single。

优化前的代码

```
void mtrx4mul1 (float data[4][4],
  float tbl[4][4], float ret[4][4])
{
int i,j,k;

for(i=0;i<4;i++){
  for(j=0;j<4;j++){
    for(k=0;k<4;k++){
      ret[i][j]+=
        data[i][k]*tbl[k][j];
      }
    }
  }
}
```

扩展为汇编语言代码（优化前）

```
_mtrx4mull:
          MOV.L       R14,@-R15
          MOV.L       R13,@-R15
          MOV.L       R11,@-R15
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV.L       R8,@-R15
          ADD         #-4,R15
          MOV         #0,R8
          MOV.L       R8,@R15
          MOV         #4,R14
L244:
          MOV.L       @R15,R11
          MOV         R8,R9
          SHLL2       R11
```

优化后的代码

```
#include <machine.h>
void _mtrx4mul (float data[4][4],
  float tbl[4][4],float ret[4][4])
{
   ld_ext(tbl);
   mtrx4mul(data,ret);
}
```

扩展为汇编语言代码（优化后）

```
_mtrx4mul:
    ADD         #-12,R15
    MOV.L       R4,@(8,R15)
    MOV.L       R5,@(4,R15)
    MOV.L       R6,@R15
    MOV.L       @(8,R15),R2
    FRCHG
    FMOV.S      @R2+,FR0
    FMOV.S      @R2+,FR1
    FMOV.S      @R2+,FR2
    FMOV.S      @R2+,FR3
    FMOV.S      @R2+,FR4
    FMOV.S      @R2+,FR5
    FMOV.S      @R2+,FR6
    FMOV.S      @R2+,FR7
```

```
          SHLL2       R11
L245:
          MOV         R9,R1
          MOV         #0,R7
          SHLL2       R1
          MOV         R8,R10
          MOV         #0,R13
          ADD         R5,R7
L246:
          MOV         R11,R0
          ADD         R6,R0
          ADD         R1,R0
          MOV         R11,R3
          MOV.L       R0,@-R15
          ADD         R4,R3
          MOV.L       @R15+,R2
          MOV         R1,R0
          ADD         R13,R3
          FMOV.S      @(R0,R7),FR0
          FMOV.S      @R2,FR2
          ADD         #1,R10
          FMOV.S      @R3,FR3
          CMP/GE      R14,R10
          ADD         #16,R7
          FMAC        FR0,FR3,FR2
          FMOV.S      FR2,@R2
          BF/S        L246
          ADD         #4,R13
          ADD         #1,R9
          CMP/GE      R14,R9
          BF          L245
          MOV.L       @R15,R3
          ADD         #1,R3
          CMP/GE      R14,R3
          BF/S        L244
          MOV.L       R3,@R15
          ADD         #4,R15
          MOV.L       @R15+,R8
          MOV.L       @R15+,R9
          MOV.L       @R15+,R10
          MOV.L       @R15+,R11
          MOV.L       @R15+,R13
          RTS
          MOV.L       @R15+,R14
```

```
    FMOV.S      @R2+,FR8
    FMOV.S      @R2+,FR9
    FMOV.S      @R2+,FR10
    FMOV.S      @R2+,FR11
    FMOV.S      @R2+,FR12
    FMOV.S      @R2+,FR13
    FMOV.S      @R2+,FR14
    FMOV.S      @R2+,FR15
    FRCHG
    MOV.L       @(4,R15),R3
    MOV.L       @R15,R1
    FMOV.S      @R3+,FR0
    FMOV.S      @R3+,FR1
    FMOV.S      @R3+,FR2
    FMOV.S      @R3+,FR3
    FTRV          XMTRX,FV0
    ADD           #16,R1
    FMOV.S      FR3,@-R1
    FMOV.S      FR2,@-R1
    FMOV.S      FR1,@-R1
    FMOV.S      FR0,@-R1
    FMOV.S      @R3+,FR0
    FMOV.S      @R3+,FR1
    FMOV.S      @R3+,FR2
    FMOV.S      @R3+,FR3
    FTRV          XMTRX,FV0
    ADD           #32,R1
    FMOV.S      FR3,@-R1
    FMOV.S      FR2,@-R1
    FMOV.S      FR1,@-R1
    FMOV.S      FR0,@-R1
    FMOV.S      @R3+,FR0
    FMOV.S      @R3+,FR1
    FMOV.S      @R3+,FR2
    FMOV.S      @R3+,FR3
    FTRV          XMTRX,FV0
    ADD           #32,R1
    FMOV.S      FR3,@-R1
    FMOV.S      FR2,@-R1
    FMOV.S      FR1,@-R1
    FMOV.S      FR0,@-R1
    FMOV.S      @R3+,FR0
    FMOV.S      @R3+,FR1
    FMOV.S      @R3+,FR2
    FMOV.S      @R3+,FR3
    FTRV          XMTRX,FV0
    ADD           #32,R1
    FMOV.S      FR3,@-R1
    FMOV.S      FR2,@-R1
    FMOV.S      FR1,@-R1
    FMOV.S      FR0,@-R1
```

```
                                    ADD         #12,R15
                                    RTS
                                    NOP
```

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]		执行速度 [周期]	
	优化前	优化后	优化前	优化后
SH-4	110	118	603	113

5.11 软件流水线

重点：

设计能够消除运算结果等待的程序，即可达成顺畅的流水线流程。

描述：

软件流水线可消除伴随数据流的指令等待（值的定义和使用）。 例如，在取值的和的代码中，当 ADD 指令的加法紧随加载指令的定义时，将产生等待。 然而，若加载指令较早发出，将可消除这项等待。 若处理是在循环内，下一迭代的数据加载将在当前迭代期间执行。

这个方法的典型实例是除法和平方根计算。 SH-4 随附有 FDIV 和 FSQRT 指令，但是由于大量的等待时间（从发出指令到生成结果的周期，对于 SH-4 是 12 个周期），在立即使用结果的程序中，将出现执行下一指令前的等待周期。

使用的实例：

取平方根的和的循环实例。

注意： 在此实例中，编译选项是 –cpu=sh4 –fpu=single。

优化前的代码

```
#include <mathf.h>

float func1(float *p, int cnt){
    float ret=0.0f;
    do {
       ret+=sqrtf(*p++);
       x();

    } while(cnt--);
    return ret;
 }
```

扩展为汇编语言代码（优化前）

```
_func1:
          MOV.L       R14,@-R15
          FMOV.S      FR15,@-R15
          STS.L       PR,@-R15
          ADD         #-8,R15
          MOV.L       L262,R14
          FLDI0       FR15
          MOV.L       R4,@(4,R15)
          MOV.L       R5,@R15
L260:
          MOV.L       @(4,R15),R3
          ADD         #4,R3
          MOV.L       R3,@(4,R15)
          ADD         #-4,R3
          FMOV.S      @R3,FR3
```

优化后的代码

```
#include <mathf.h>

float func21(float *p, int cnt){
    float ret=0.0f;
    float sq=0.0f;
    do {
       ret+=sq;
       sq=sqrtf(*p++);
       x();
    } while (cnt--);
    return ret;
 }
```

扩展为汇编语言代码（优化后）

```
_func21:
          MOV.L       R14,@-R15
          FMOV.S      FR15,@-R15
          FMOV.S      FR14,@-R15
          STS.L       PR,@-R15
          ADD         #-8,R15
          FLDI0       FR4
          MOV.L       L263,R14
          FMOV.S      FR4,FR15
          MOV.L       R4,@(4,R15)
          MOV.L       R5,@R15
          FMOV.S      FR4,FR14
L261:
          MOV.L       @(4,R15),R3
          FADD        FR15,FR14
          ADD         #4,R3
```

```
        FSQRT   FR3
        JSR     @R14
        FADD    FR3,FR15
        MOV.L   @R15,R3
        ADD     #-1,R3
        MOV.L   R3,@R15
        ADD     #1,R3
        TST     R3,R3
        BF      L260
        FMOV.S  FR15,FR0
        ADD     #8,R15
        LDS.L   @R15+,PR
        FMOV.S  @R15+,FR15
        RTS
        MOV.L   @R15+,R14
L262:
        .DATA.L _x
```

```
        MOV.L   R3,@(4,R15)
        ADD     #-4,R3
        FMOV.S  @R3,FR15
        JSR     @R14
        FSQRT   FR15
        MOV.L   @R15,R3
        ADD     #-1,R3
        MOV.L   R3,@R15
        ADD     #1,R3
        TST     R3,R3
        BF      L261
        FMOV.S  FR14,FR0
        ADD     #8,R15
        LDS.L   @R15+,PR
        FMOV.S  @R15+,FR14
        FMOV.S  @R15+,FR15
        RTS
        MOV.L   @R15+,R14
L263:
        .DATA.L _x
```

优化前和优化后的程序分析：

在优化前，FADD 指令在 FSQRT 后立即执行，所以在 FSQRT 完成及 FADD 可被执行前出现等待周期。

在已被优化的程序中，当执行 FSQRT 后，FADD 指令将在下一循环中发出，所以 FADD 之前的等待将被消除。

优化前及优化后的代码大小和执行速度：

CPU 类型	代码大小 [字节]（一个循环）		执行速度 [周期]（FSQRT 中的等待周期数量）	
	优化前	优化后	优化前	优化后
SH-4	28	28	9	0

5.12 关于高速缓存存储器

SuperH 系列包含配备内部高速缓存存储器的产品。

高速缓存是减少存储器中程序和数据的存取频率，并增进程序操作的机制。

通过使用高速缓存存储器，程序的执行速度将可获得增进。然而，高速缓存具有多种类型，对它们的结构和功能性的透彻了解将可使编程更具效率。

在这里将说明 SuperH 系列中所使用的几种高速缓存结构，同时也提供充分利用高速缓存存储器的编程建议。

5.12.1 术语解释

高速缓存命中

当 CPU 尝试存取外部存储器时，它将检查所要检索的数据是否已存在于高速缓存存储器中。若数据存在于高速缓存存储器中，称为高速缓存命中。

简单来说，当发生高速缓存命中时，高速缓存存储器将被存取，而免却存取较慢的外部存储器的需要。

高速缓存遗漏

当 CPU 尝试存取外部存储器时，它将检查所要检索的数据是否已存在于高速缓存存储器中。若数据不存在于高速缓存存储器中，称为高速缓存遗漏。

高速缓存填充

当发生高速缓存遗漏时，CPU 将把所存取的存储器内容存储在高速缓存内。这个过程称为高速缓存填充。

高速缓存线长

当 CPU 执行高速缓存填充时，它不只将所存取的存储器内容存储在高速缓存中，在高速缓存填充过程中，它也存储了存储器连续区域的内容，包括所存取数据的之前和之后的区域。这个范围的大小称为高速缓存线长。线长对于特定的 CPU 来说是固定的长度（大小）。这个线长是在高速缓存中存储数据时的单位。

高速缓存大小、项目数量、线数

数据存储在高速缓存中的容量称为高速缓存大小。

项目数量（或线数）是由高速缓存线长和高速缓存大小所决定，如下所示。

（高速缓存大小）=（项目数量）x（高速缓存线长）

回写和写通

当处于高速缓存命中状态时，将执行尝试覆盖存储器内容的操作，有两种选择可执行该覆盖。

(1) 高速缓存存储器的内容及外部存储器的内容被同时覆盖。
(2) 只有高速缓存存储器被覆盖。
在 (1) 的状况中，高速缓存存储器的内容和外部存储器的内容永远是一致的。

这个方法被称为写通。

在 (2) 的状况中，只会在高速缓存存储器中保留最近的数据，外部存储器不被覆盖，但保留了旧的数据。 当使用这种方法时，在丢弃高速缓存项目的内容之前，项目中的数据将被回写到外部存储器。 这个方法被称为回写。 （一般上有一个标志会指示高速缓存中的项目有被写入一次或多次，只有在标志指示高速缓存有发生写入时，才会执行回写到外部存储器。）

高速缓存一致性

这是指外部存储器的内容与高速缓存存储器的内容的一致性。

换言之，当对高速缓存使用回写方法时，可能存在高速缓存存储器的内容与外部存储器的内容不一致的问题，同时若有 CPU 以外的其他设备存储外部存储器，由于这项数据没有被更新，因此将发生软件操作错误。 当其他设备存取相同的存储器时（若使用共用存储器），应使用写通方法，否则应在其他设备存取前对存储器区域使用高速缓存项目的回写。

直接映射

这是使用高速缓存的另一种方法。

简单来说，高速缓存存储器用于存储数据的地址将特别根据外部存储器中的地址来决定。外部存储器的偏移被用来将数据存储在高速缓存存储器中具有相同偏移的地址内。

当使用高速缓存中的特定地址来检查数据地点时，存储地点将由存储器地址明确决定，因此这个方法可以减轻硬件的负担。然而，当被频繁存取的存储器地点具有相同的偏移地址时，相同的项目将频繁被替换，而另一方面有些项目则甚少被使用。

所有这个方法在一些情况下会造成不具效率的高速缓存存储器使用，因此在选择这个方法时应谨慎考虑程序设计。

完全相关方法

这是使用高速缓存的另一种方法。

和直接映射相反，外部存储器的所有地址和数据被存储在项目中。 从最长时间未被存取的项目开始进行替换（LRU 方法），以便使高速缓存的使用具有最大效率。然而，为了决定外部存储器的地址应存储在哪个高速缓存项目中，必须对所有高速缓存项目进行检查，这使得硬件机制变得复杂。

集相关方法

这个方法介于直接映射和完全相关方法之间，在这里一组直接映射高速缓存将被使用（所使用的数量称为路数）。在单个直接映射高速缓存中，所使用的高速缓存项目将由外部存储器中地址的偏移值来决定，但对于整组高速缓存，将使用最长时间未被存取的那一路。

当搜索存储地址的高速缓存区域时，与其搜索所有高速缓存项目，只需要进行和路数相等的搜索，以便使所涉及的硬件操作不会太繁复。

SH7604、SH7708、SH7707、SH7709 和 SH7718 使用 4 路集相关高速缓存。

5.13 SuperH 系列高速缓存

在 SuperH 系列中使用的各种高速缓存解释如下。

5.13.1 SH7032、SH7034、SH7020 和 SH7021 系列 (SH-1)

此系列未随附高速缓存存储器。 这些产品基于配备内部 ROM/RAM 的 CPU。 若执行仅限于内部 ROM/RAM，将可获得相等于或超越使用高速缓存存储器的性能。

5.13.2 SH704x 系列 (SH-2)

此系列基于配备内部 ROM/RAM 的 CPU。 它们具有性能较 SH7034 为佳的 CPU，并提供允许将部分内部 RAM 用作指令高速缓存的功能。

高速缓存方法： 指令高速缓存（直接映射）

高速缓存大小 1 kB（可用，2 kB 内部 RAM）
高速缓存线长： 4 字节（两个指令）
项目数量： 256

高速缓存仅用于指令。 由于高速缓存填充的低内务操作，此高速缓存对于 1 kB 内的循环处理具高效率。 高速缓存的有效范围是外部存储器，它对于内部 ROM/RAM 不具效率。（内部 ROM/RAM 可被快速存取，因此无需使用高速缓存。）

然而，4 kb 的内部 RAM 中有 2 kb 被用作 1 kb 的高速缓存存储器，因此当使用高速缓存时，内部 RAM 的大小是 2 kb。

由于这不是数据高速缓存，在一些情况中若将所有 4 kb 的 RAM 用于数据，而不使用高速缓存，而对于频繁使用的代码优先使用内部 ROM，将可增进整体性能。

5.13.3 SH7604 系列 (SH-2)

此系列基于配备高速缓存的 ROM-less 处理器型 CPU。

高速缓存方法： 4 路集相关高速缓存（混合指令和数据）

高速缓存大小 4 kb
高速缓存线长： 16 字节
项目数量： 256 (64x4)
其他： 写通方法

5.13.4 SH7707、SH7708 和 SH7709 系列 (SH-3)

此系列基于配备高速缓存的 ROM-less 处理器型 CPU。

高速缓存方法： 4 路集相关高速缓存（混合指令和数据）

高速缓存大小 8 kb
高速缓存线长： 16 字节
项目数量： 512 (128x4)
其他： 可选择写通或回写方法

5.13.5 SH7750 系列 (SH-4)

高速缓存方法： 直接映射（混合指令和数据）

指令高速缓存

高速缓存大小 8 kb
高速缓存线长：32 字节
项目数量：256
其他：可使用 4 kb x 2 索引模式

数据高速缓存（操作数高速缓存）

高速缓存大小 16 kb
高速缓存线长：32 字节
项目数量：512
其他：8 kb 可被用作内部 RAM

存储队列

为对外部存储器的高速转移提供了两个 32 kb 的存储队列。

存储队列是对外部存储器进行高速转移的缓冲。

通过使用它，将可对外部存储器进行高速转移。

数据高速缓存无法执行高速缓存阻塞，因此高速缓存有可能被替换。

存储队列是用于高速转移的可靠机制，不会因为高速缓存替换或其他问题而造成性能下降。

5.14 高速缓存的使用技巧

下面描述有效使用高速缓存的技巧。

(1) 性能不佳的原因
表 5.7 列出最可能阻止达到更佳性能的原因。

一般上相信要获得高速缓存最好的使用效果所需的方法和措施如下：

[1] 使用调试程序或分析工具来检查函数及执行频率的依赖性。
[2] 将高速缓存放置在具有密切依赖性的地点以减少高速缓存遗漏。
[3] 执行面向大小的优化以将频繁执行的部分变成函数。

表 5.7 性能不佳的主要原因

项目	可能的原因	动作
高速缓存上的地址移位	高速缓存上或项目中的地址移位可能更改高速缓存的线路争夺。	更改对齐。
程序大小的增加	高速缓存命中率可能因为用于对齐的虚拟区域的增加及程序大小的增加而被降级。	执行面向大小的优化

下面详细描述您必须采取的动作。 然而，请注意这些动作并非每次都有效。

(2) 高速缓存上地址移位的更正动作
有效的方法是将程序对齐 (Alignment of program) 更改为高速缓存线长。 在默认情况下，由 SuperH RISC engine C/C++ 编译程序输出的程序对齐是四个字节。 使用 SuperH RISC engine C/C++ 编译程序的“align16”编译程序选项来将对齐更改为 16 字节，这相等于高速缓存上的一个线长。 这将确保从高速缓存线的起始部分开始放置地址。

然而，请注意这个方法仅在高速缓存线长为 16 字节时有效。 程序大小将增加。

[1] 指定“align16”选项的最有效方法
- 为简练的函数组（仅在高速缓存上使用少数线）指定“align16”选项。
- 在程序中的一般公用例程组内为小函数组指定“align16”选项，若有。
  （定义邻近相同模块的小函数组将可减少高速缓存项目竞争。）

(3) 程序大小的增加
用于对齐的虚拟区域及程序大小的增加将使高速缓存命中率降级。

[1] 如何应付程序大小的增加

您可以制止程序大小增加的方法如下：

- 通过为 SuperH RISC engine C/C++ 编译程序指定 “Size-Oriented” 的优化选项来执行优化。
- 使用较小的函数来重新创建程序。
- 创建一个特殊的一般目的例程。

(4) 如何使用高速缓存项目

您必须检查高速缓存中的项目是否被平均使用而不是让替换集中发生在少数项目上。这项检查对于直接映射高速缓存尤其重要。

检查这些项目需要一些方法来追踪高速缓存的内容。

然而，在许多情况下并无法进行实际追踪。

因此，增进整体性能的方法包括以下各项：

[1] 选择被频繁执行的函数。

[2] 从连接编辑程序的映射文件决定各个函数的起始及终止地址。

[3] 检查函数所使用的高速缓存项目。

[4] 为各个函数所使用的项目编译统计数据。

[5] 检查是否集中于高速缓存上的特定项目。

在这个阶段，若多个函数使用相同的高速缓存线，更改函数地址以消除集中。

您可通过在编译时更改段名或在连接时更改输入顺序来更改地址。

第 [3] 项中用于检查已用项目的方法视 CPU 及高速缓存方法而异。 一般上，若使用了直接映射高速缓存，项目编号将由绝对地址的偏移值决定。（偏移范围视高速缓存大小而定。）

若您认为使用 [3] 到 [5] 的步骤来检查竞争很困难，您可以使用下列方法：

更改在 [1] 中所选定函数的段名，然后重新编译。

这将把在 [1] 中所选定的函数分配到连续地址。 换言之，在这些函数之间没有发生高速缓存竞争。 这表示要更改段名之函数的总大小必须等于或小于高速缓存大小。

(5) 编程技巧

为您提供一些有效使用高速缓存的有效编程技巧。 参考下列内容以编程：

■ 平铺程序

请参阅第 5.9.2 节“平铺”。

■ 预取

请参阅第 5.9.1 节“预取指令”。

第 6 节　有效的编程技术（补遗）

6.1　如何指定选项

要创建有效的程序，有效的方法是指定优化选项。您可以依靠优化选项的选择或将一个选项与其他选项结合使用来实现提高的效力。

6.1.1　用于启动 HEW 的选项（浮点设定）

可以设定为启动 Hew 的选项将设定与一般使用 CPU 有关的环境，包括 endian 和浮点的处理。

当程序使用浮点时，浮点的设定将会大大影响性能。在大小和速度方面，单精度浮点模式（32 位）会比双精度浮点模式更有效率。请使用单精度浮点模式，如果它可以满足您的应用需要。单精度提供约七个十进制数字的精度，而双精度则提供 17 个十进制数字的精度。

注意： 按照本节说明指定的浮点模式将会影响整个工程。因此，您不能为每一个文件指定不同的模式。

(1) 对于 SH-1、SH-2、SH-2E、SH-2A、SH2-DSP、SH3、SH3-DSP 以及 SH4AL-DSP

单击图 6.1 所示的“将双精度当作浮点处理”(Treat double as float)复选框。此操作将使所有浮点（包括那些声明为双精度的浮点）以单精度模式处理。

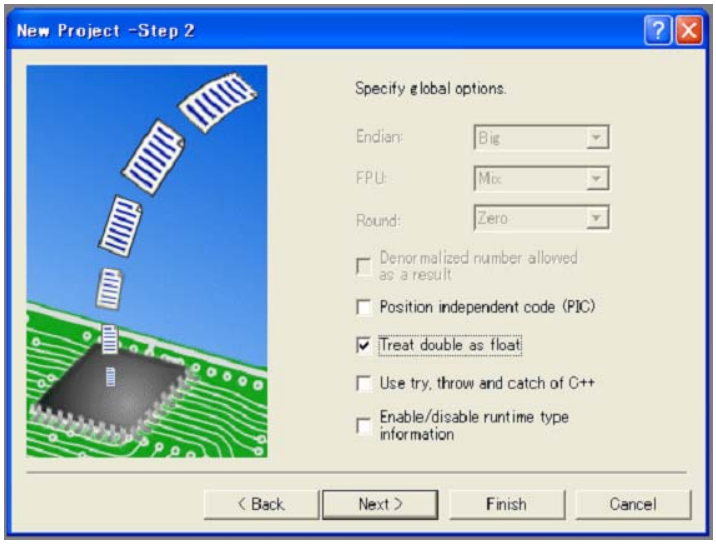


图 6.1 如何指定单精度模式（SH-1、SH-2、SH-2E、SH-2A、SH2-DSP、SH3、SH3-DSP 和 SH4AL-DSP）

(2) 对于 SH2A-FPU、SH-4 和 SH-4A

如图 6.2 所示，在 FPU 菜单中指定“单精度”(Single)。

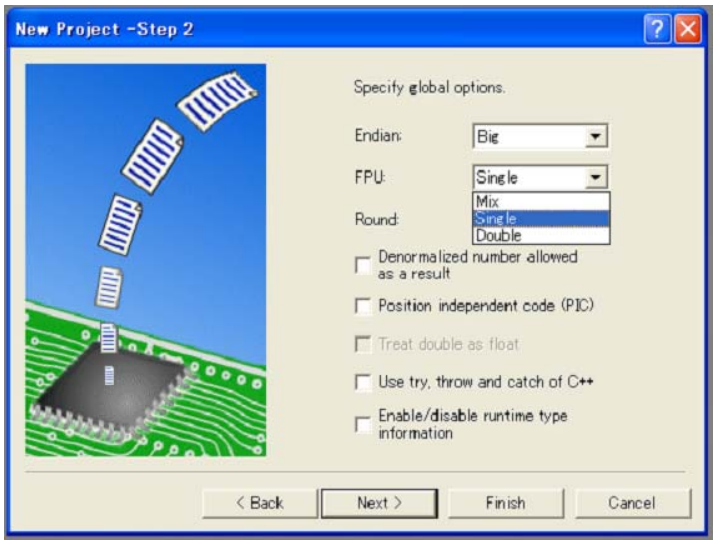


图 6.2 如何指定单精度模式（SH2A-FPU、SH-4 和 SH-4A）

在此画面上指定“双精度”(Double) 可以在双精度模式中执行所有运算。 指定“混合”(Mix) 可如程序中所描述，在单精度模式中计算浮点以及在双精度模式中计算双精度（此运算和不为 SH-1、SH-2、SH-2E、SH-2A、SH2-DSP、SH3、SH3-DSP 或 SH4AL-DSP 选定“将双精度当作浮点处理”(Treat double as float) 一样，但是，程序的性能可能会在指定双精度时较差，因为它会在 FPU 的单精度和双精度之间切换时执行。）

6.1.2 如何指定优化选项（速度和大小）

以下是主要的优化选项（图 6.3）。

(a) 优化大小和速度（默认）
(b) 优化大小
(c) 优化速度

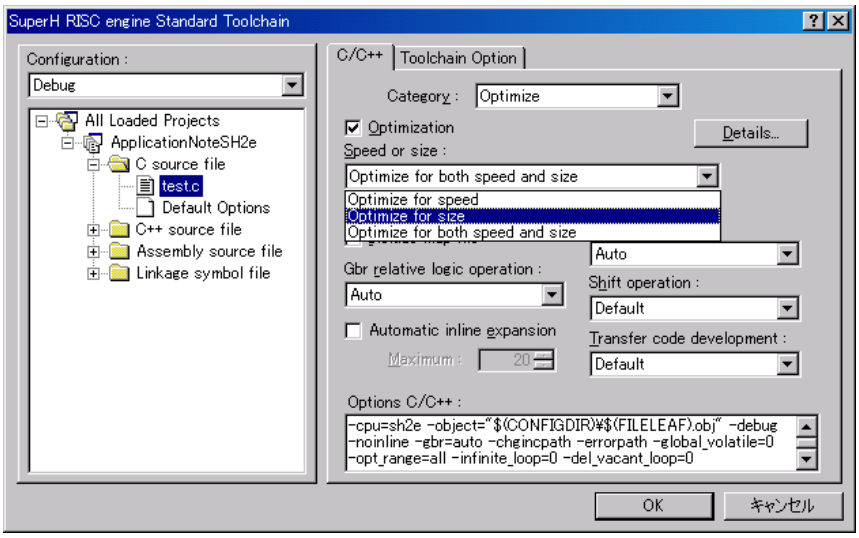


图 6.3 优化选项

“优化大小和速度”(Optimize for size and speed)（默认）仅执行可同时增进大小和速度的优化。“优化大小”(Optimize for size) 除了“优化大小和速度”所执行的优化之外，还执行会减弱速度而缩减大小的优化。“优化速度”(Optimize for speed) 除了“优化大小和速度”所执行的优化之外，还执行会影响缩减大小而增进速度的优化。

若要为大小或速度提供优先级，请个别指定“优化大小”(Optimize for size) 或“优化速度”(Optimize for speed) 。

在多数系统中，速度只对程序的有限部分占有重要性。在此情形下，对需要速度的文件使用“优化速度”(Optimize for speed) 以及对其他文件使用“优化大小”(Optimize for size) 执行优化会更有效率。您可以为每个文件指定不同的优化选项，因为它们不会更改文件之间的界面。

要为每个文件指定不同的选项，请从左边显示的目录树中选择一个目标源文件然后指定一个选项（图 6.4）。

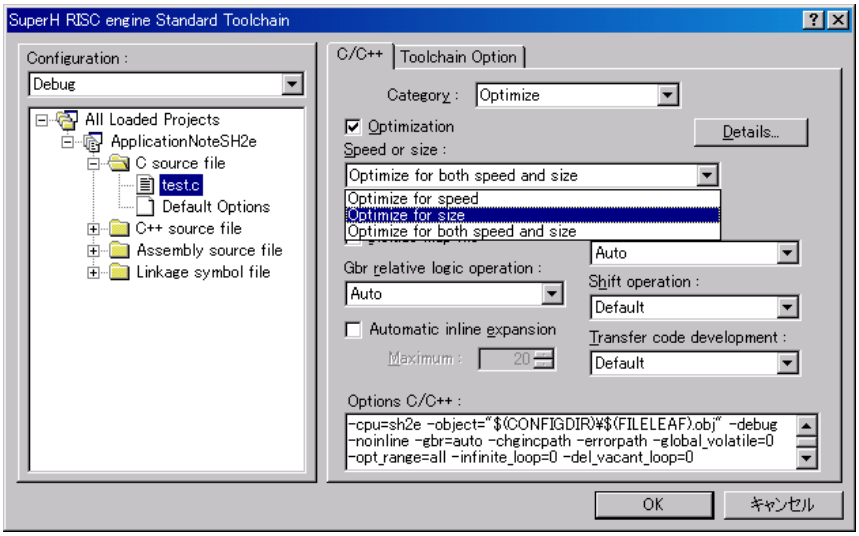


图 6.4 如何为每个文件指定不同的选项

6.1.3 实现程序兼容性所需注意的选项（函数界面）

一些会更改函数界面的选项可以提高程序的执行效率。这些选项在默认情形下可支持旧版本的编译程序以便实现其兼容性，它们可以在您更改整个工程时提高效率且不会出现任何兼容性问题。这些选项位于“编译程序”(Compiler) 选项的“其他”(Others) 菜单中。

注意： 如果工程包含使用汇编程序编写的程序，您必须检查创建工程时所使用的惯例。

(1) 如果使用 Callee 保存/恢复 MACH 和 MACL 寄存器 (Callee saves/restores MACH and MACL registers if used)

指定此选项可让函数使用 MACH 和 MACL 寄存器在函数入口及出口点保存和恢复这些寄存器。在默认情形下，这些寄存器会保存及恢复，并且多数作为工作寄存器使用。 因此，您可以选择不保存和恢复它们（取消选定复选框）来同时增进大小和速度（图 6.5）。

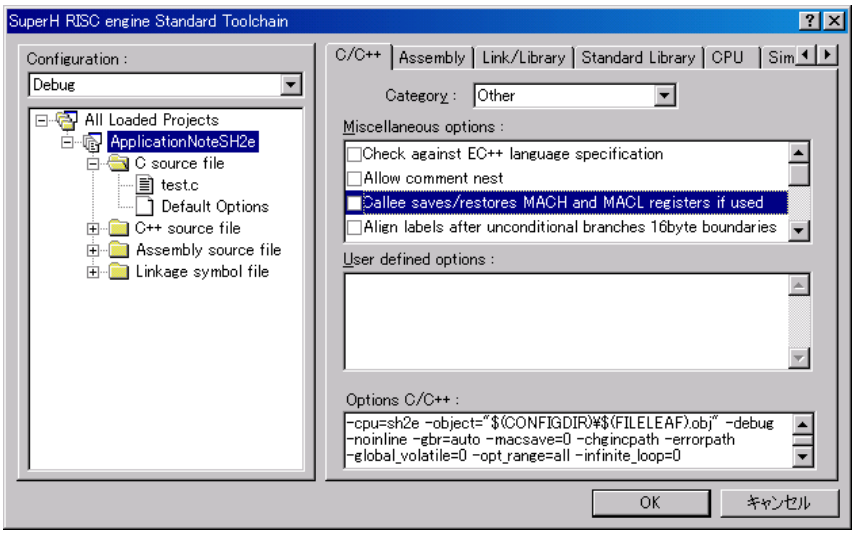


图 6.5 MACH 和 MACL 寄存器保存及恢复选项

(2) 扩展返回值到 4 字节

如果函数返回类型是字符、无符号字符、短，或无符号短，在默认情形下，符号扩展（或零扩展）不是通过被调用的函数而是调用的函数执行（此规格与旧版本的编译程序兼容）。

如果一个函数被调用超过一次，在缩减大小方面，让被调用的函数而不是调用的函数进行符号扩展一个返回值（选定复选框）会比较有利，因为扩展代码只须包括一次。

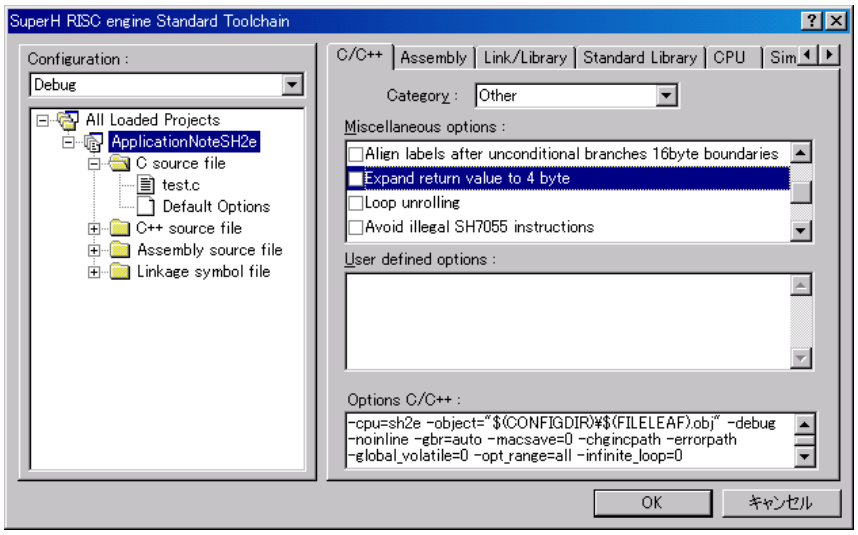


图 6.6 返回值扩展选项

注意： SH 寄存器大小是 4 字节。 您可以通过将函数和数据的返回值（如函数参数和本地变量）声明为任何一种四字节类型、整数、无符号、长和无符号长，放入寄存器来创建有效的程序，因为不需要进行任何符号扩展处理。

6.1.4 使用 volatile 声明处理变量的选项（volatile 变量）

指定 volatile 声明以禁止存取变量的优化。为要在程序中使用的外部变量声明 volatile，主要基于以下两个目的：

(a) 禁止存取外围输入-输出寄存器的优化。
(b) 禁止存取要在不同任务或中断处理中共享之变量的优化。
6.0 和以下版本的 SH C/C++ 编译程序几乎不执行任何存取外部变量的优化。然而，7.0 和以上版本则彻底执行增强优化。如果使用编译程序 6.0 或以下版本开发不具备符合下述条件 (1) 和 (2) 之变量的 volatile 声明的程序，建议您在将程序套接到编译程序 7.0 或以上版本时在程序中添加 volatile 声明。

如果您没有修改程序，请在“编译程序”(Compiler) 选项的“优化” (Optimize) 菜单中单击“详细资料...”(Details...) 按钮以设定“详细资料”选项。

使用这些将会禁止优化的选项，在不削弱于 6.0 或以下版本中开发之程序的功能下，禁止最小范围的优化。

(1) 如果没有为外围输入-输出寄存器声明 volatile
如果您将写入或读取的两次连续存取优化为一次存取，根据寄存器的规格，外围输入-输出寄存器可能具有不同的运算。

要禁止此类优化，请在“优化详细资料”(Optimize Details) 选项的“全局变量”(Global variables) 标签中指定“级别 1”(Level 1)（图 6.7）。

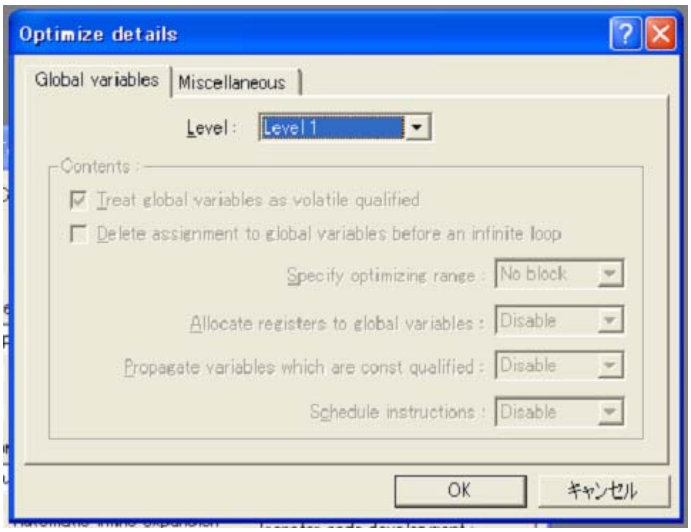


图 6.7 优化详细资料选项级别 1

但是“级别 1”也会禁止其他优化。 请按照以下方式保留 volatile 声明的效果以及执行最小范围的其他优化（图 6.8）。

(a) 将“级别”(Level) 设定为“级别 1”(Level 1)。
(b) 然后，将“级别”(Level) 设定为“定制”(Custom)（“级别 1”设定将会保留）。
(c) 将“指定优化的范围”(Specify optimizing range) 设定为“全部”(All)。
(d) 将“将寄存器分配到全局变量”(Allocate registers to global variables) 设定为“允许”(Enable)。
(e) 将“传播符合常数标准的变量”(Propagate variables which are const qualified) 设定为“允许”(Enable)。
(f) 将“预设指令”(Schedule instructions) 设定为“允许”(Enable)。

实例：

源代码

```
extern int  x;

void f (void)
{
  x=1;
   x=2;
}
```

汇编程序扩展代码

```
_f:
            MOV.L       L11,R6
            MOV         #2,R2
            RTS
            MOV.L       R2,@R6
```

汇编程序扩展代码（指定该选项时）

```
_f:
            MOV.L       L11,R6
            MOV         #1,R2
            MOV.L       R2,@R6
            MOV         #2,R2
            RTS
            MOV.L       R2,@R6
```

在此实例中，不指定选项导致输入-输出寄存器的两次存取结合成一次。因此，您可能会获得外围输入-输出寄存器的不同效果。

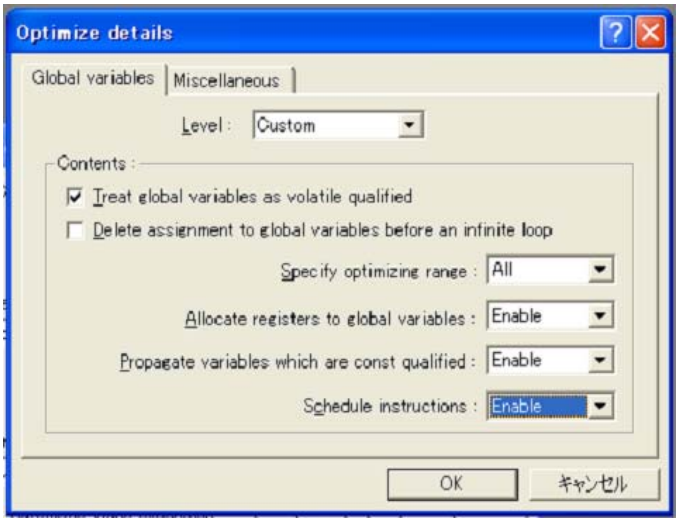


图 8 为外围输入-输出寄存器指定 volatile

注 1： 在通过每个型号的 Hew 创建的标题文件中，将会为外围输入-输出寄存器声明 volatile。如果外部变量不具备 volatile 声明将只是在任务和中断中共享的变量，并如 (2) 所述指定选项。

注 2： 如果在一个表达式中对一个外部变量进行两次参考，其顺序将无法确保。如果为外部变量声明 volatile，结果将会是一样。如果必须对外围输入-输出寄存器进行超过一次参考，请在不同的表达式中进行参考。

(2) 如果没有为要在任务和中断中共享的外部变量声明 volatile，该外部变量就会被放在存储器上。
结合连续存取并不会更改程序的效果。 但是，如果要在循环中参考的外部变量被分配到寄存器，在其他任务和中断处理中更改变量，将不会影响循环处理并且会因此更改运算。

要禁止此类优化，请在“优化详细资料”(Optimize Details) 选项的“全局变量”(Global variables) 标签中指定“级别 2”(Level 2)（图 6.9）。

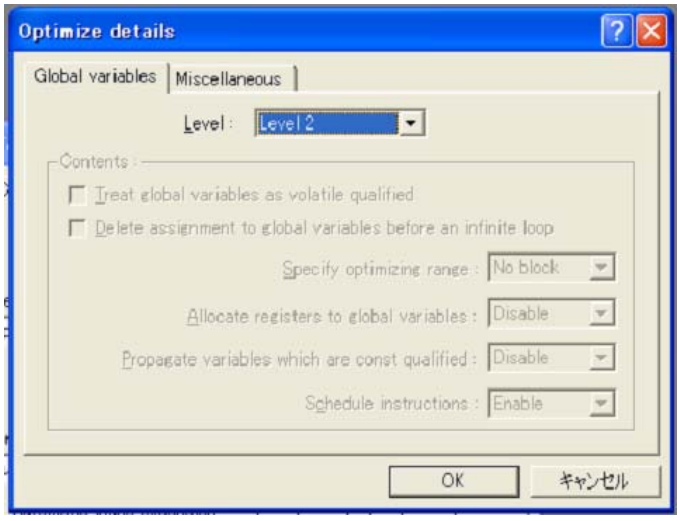


图 6.9 优化详细资料选项级别 2

但是“级别 2”也会禁止其他优化。 请按照以下方式保留 volatile 声明的效果以及执行最小范围的其他优化（图 6.10）。

(a) 将“级别”(Level) 设定为“级别 2”(Level 2)。

(b) 然后，将“级别”(Level) 设定为“定制”(Custom)（“级别 2”设定将会保留）。

(c) 将“指定优化的范围”(Specify optimizing range) 设定为“没有循环”(No loop)（即，禁止循环中的外部变量优化。如果保留默认设定“没有块”(No block)，将会禁止包含循环的所有结构中的外部变量优化。）

(d) 将“将寄存器分配到全局变量”(Allocate registers to global variables) 设定为“允许”(Enable)。

(e) 将“传播符合常数标准的变量”(Propagate variables which are const qualified) 设定为“允许”(Enable)。

实例：

源代码

```
extern int  x; /* 这可能
                  会因为
                  中断而更改。*/

void f (void)
{
  x=1;
  while (1){
    if (x!=1) break;
  }
}
```

汇编程序扩展代码

```
_f
L10:
        BRA     L10
        NOP
        RTS
        NOP
```

汇编程序扩展代码（指定该选项时）

```
_f
        MOV.L   L13,
        MOV     #1,R2
        MOV.L   R2,@R1
        MOV.L   @R1,R0
        CMP/EQ  #1,R0
        BT      L11
        RTS
        NOP
```

源代码将会假设，如果未指定该选项，循环将会在中断更改外部变量 x 时断开，但是因为并没有对 x 进行 volatile 声明，优化可能会使它形成无穷循环。

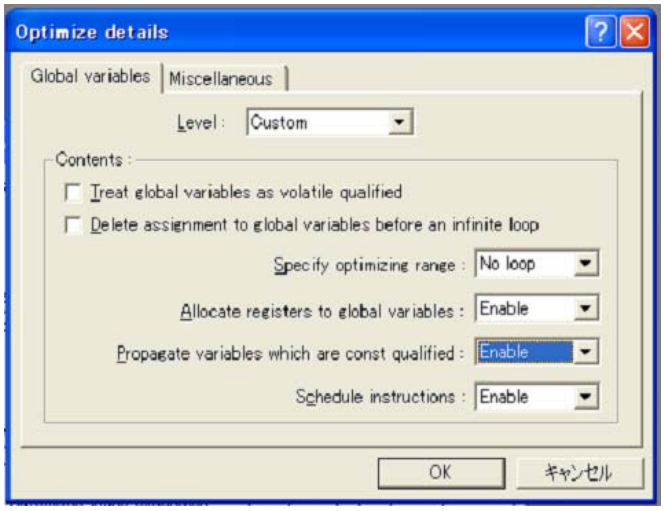


图 6.10 为要在任务和中断中共享的变量指定 Volatile

如果要在任务和中断中共享的变量之参考限制为循环条件表达式，请将“优化详细资料”(Optimize Details) 保持为默认（级别 3），然后在“其他”(Other) 选项中指定“将循环条件视为符合易失性标准来处理”(Treat loop condition as volatile qualified)，从而达到上述情况的同等效果以及最小化将禁止的优化范围（图 6.11）。

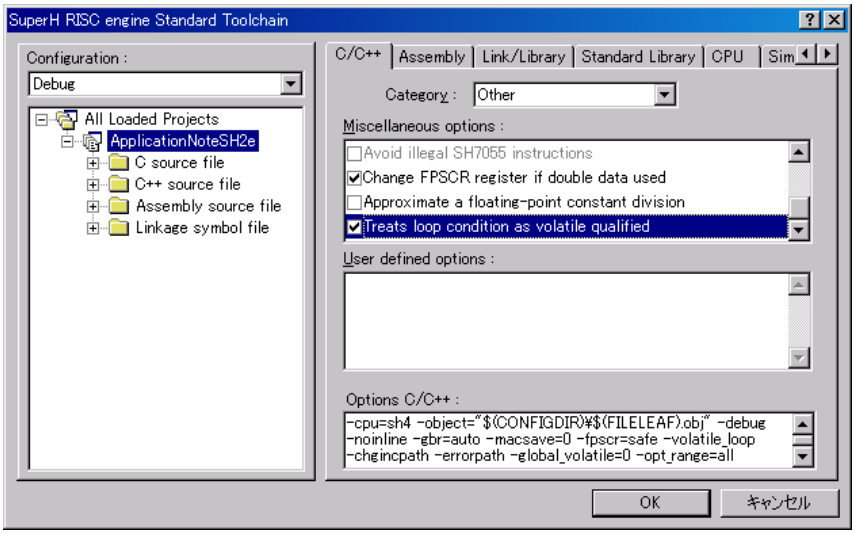


图 6.11 为循环条件声明 volatile

实例：

源代码

```
extern int  x; /* 这可能
                  会因为
                  中断而更改。 */

void f (void)
{
  x=1;
  while (x){     /* 共享的变量
                    在条件表达式
       中存取。 */
  }
}
```

汇编程序扩展代码

```
_f:
L10:
          BRA         L10
          NOP
          RTS
          NOP
```

汇编程序扩展代码（指定该选项时）

```
_f:
          MOV.L       L13,R1
          MOV         #1,R2
          MOV.L       R2,@R1
L11:
          MOV.L       @R1,R2
          TST         R2,R2
          BF          L11
          RTS
          NOP
```

在此实例中，要在任务中共享的变量将会在循环条件表达式中求值。 因此，请将循环条件定义为 volatile 以避免无穷循环。

注意： 本节项目 (2) 中说明的方法可确保要在任务和中断中共享的变量会在每次进入循环时被参考。这将使变量在每个循环迭代中被参考，从而使在其他任务或中断中更改之变量的值可以正确反映出来。
无论如何，在不包含循环的段中对一个变量进行的两次参考可以在此设定中优化。如果您具有这类参考并且需要正确反映在其他任务或中断中更改之变量的值，请指定项目 (1) “如果没有为外围输入-输出寄存器声明 volatile”中说明的一个选项。

实例：

```
extern int x;

void f(void){

  int a;

  a=x;

  /* 不含循环的长处理 */

  a=x;

}
```

如果您需要在这类程序中检测任务或中断在两次参考 x 之间对 x 的更改，请指定项目 (1) “如果没有为外围输入-输出寄存器声明 volatile”中说明的一个选项。

6.1.5 禁止删除空的循环

您在程序中编写以提供时序之空的循环可能会在 7.0 或以上版本中因为优化而被删除。若要禁止删除，请在“编译程序”(Compiler) 选项的“优化” (Optimize) 菜单中单击“详细资料...”(Details...) 按钮。 在“详细资料”(Details) 选项对话框中，选取“杂项”(Miscellaneous) 标签然后确定“删除空的循环”(Delete vacant loop) 复选框处于“关闭”状态（它在默认情况下为“关闭”）（图 6.12）。

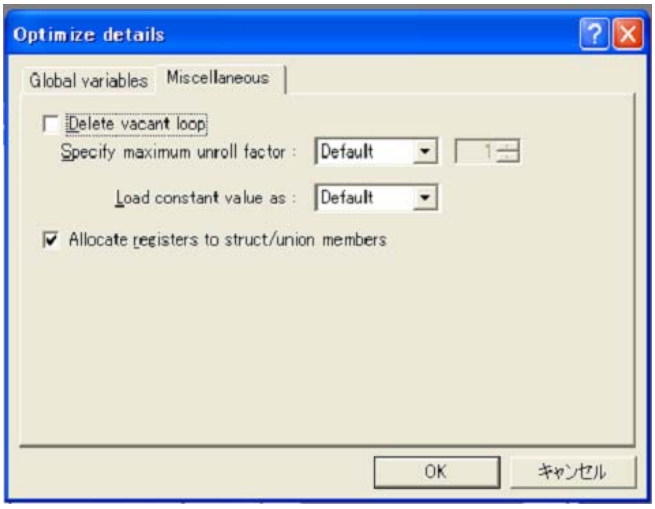


图 6.12 禁止删除空的循环

实例：

源代码

```
void f (void)
{
  int x;
  for (x=0; x<100; x++){
    /* 定时循环 */
  }
}
```

汇编程序扩展代码

```
_f:
            RTS
            NOP
```

汇编程序扩展代码（指定该选项时）

```
_f:
            MOV         #100,R2
L11:
            DT          R2
            BF          L11
            RTS
            NOP
```

在此实例中，由于定时循环中没有任何处理，代码可能会被删除，除非您禁止删除空的循环。

注意： 若要避免删除循环，您可以在循环中存取具有 volatile 声明的变量，或在循环中调用内建函数 nop()。在这些情况下，选定此选项以增强循环优化。

6.1.6 禁止常数变量的优化

优化处理将会优化 const 声明的变量，将它替换为常数。此过程将不会更改程序的运算。例如，在调式期间更改具有 const 声明之变量的值，将不会影响程序。
由于优化处理会用其初始值来替换具有 const 声明的变量 *a*，因此在调式期间更改值将不会更改程序的运算。

若要禁止此类优化，请在“编译程序”(Compiler) 选项的“优化”(Optimize) 菜单中单击“详细资料...”(Details...) 按钮。 在“详细资料”(Details) 选项对话框中，选取“全局变量”(Global variables) 标签然后按照以下方式指定（图 13）。

(a) 将“级别”(Level) 设定为“级别 3”(Level 3)（默认）。
(b) 然后，将“级别”(Level) 设定为“定制”(Custom)（“级别 3”设定将会保留）。
(c) 将“传播符合常数标准的变量”(Propagate variables which are const qualified) 设定为“禁止”(Disable)。

实例：

源代码

```
extern int x;
const int a=1;

void f (void)
{
  x=a;
}
```

汇编程序扩展代码

```
_f:
            MOV.L       L11,R6
            MOV         #1,R2
            RTS
            MOV.L       R2,@R6
```

汇编程序扩展代码（指定该选项时）

```
_f:
            MOV.L       L11+2,R6
            MOV.L       @R6,R2
            MOV.L       L11+6,R6
            RTS
            MOV.L       R2,@R6
```

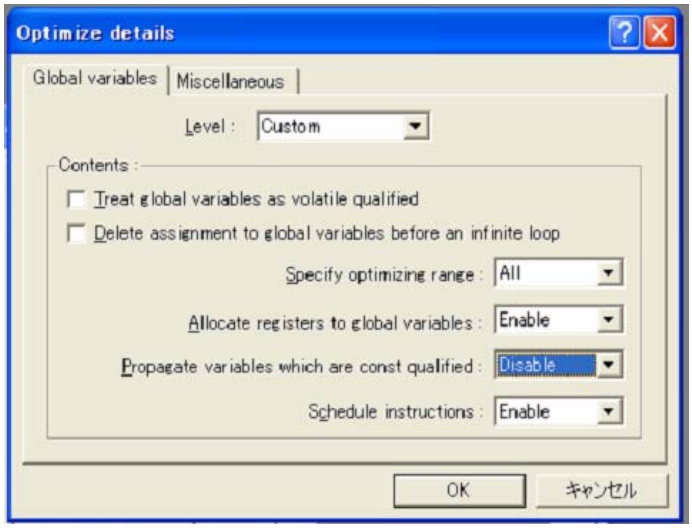


图 6.13 禁止常数变量的优化

6.1.7 增强浮点执行效率的有效选项

(1) 用乘法替换浮点常数的除法之优化
此优化会使用常数的倒数之乘法，替换浮点常数的除法。

此优化不在“优化”(Optimize) 菜单中提供，而是在 C/C++ 编译程序的“其他”(Other) 菜单中提供，因为所得的值可能会不同（虽然它处于错误范围内）。在此菜单中，选取“近似浮点常数除法”(Approximate floating-point constant division)（图 14）。

实例：

源代码

```
float x;

void f (float y)
{
  x=y/3.0;
}
```

汇编程序扩展代码

```
_f:
            MOVA        L11,R0
            MOV.L       L11+4,R2
            FMOV.S      @R0,FR8
            FDIV        FR8,FR4
            RTS
            FMOV.S      FR4,@R2
```

汇编程序扩展代码（指定该选项时）

```
_f:
            MOVA        L11,R0
            MOV.L       L11+4,R2
            FMOV.S      @R0,FR8
            FMUL        FR8,FR4
            RTS
            FMOV.S      FR4,@R2
```

3.0 的除法运算将会替换为更快的乘法运算（结果可能会不同，虽然它处于错误范围内。）

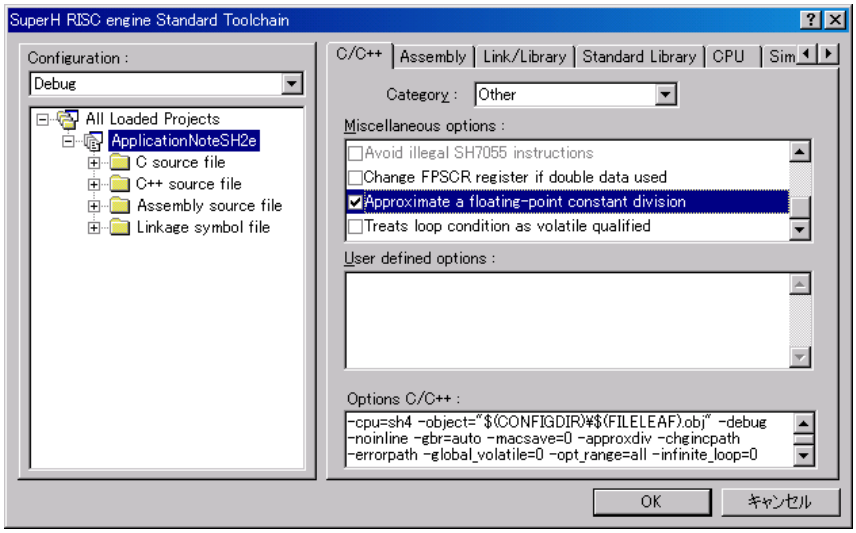


图 6.14 浮点常数的除法之优化

(2) 在 SH2A-FPU、SH-4 和 SH-4A 的浮点设定中选取“混合”(Mix) 时的警惕

如果您在 SH2A-FPU、SH-4 和 SH-4A 的浮点设定中 指定“混合”(Mix)，将会使用与旧版本编译程序的相同调用界面以实现其兼容性。在这个界面上，由于浮点设定将会在从函数返回时变成未定义，因此必须在每次发生此情况时复位 FPSCR。

在 C/C++ 编译程序的“其他”(Other) 选项中指定“使用双精度数据时更改 FPSCR 寄存器”(Change FPSCR register if double data is used)，限制在双精度运算之前和之后的 FPSCR 切换，从而增进程序的大小和速度。

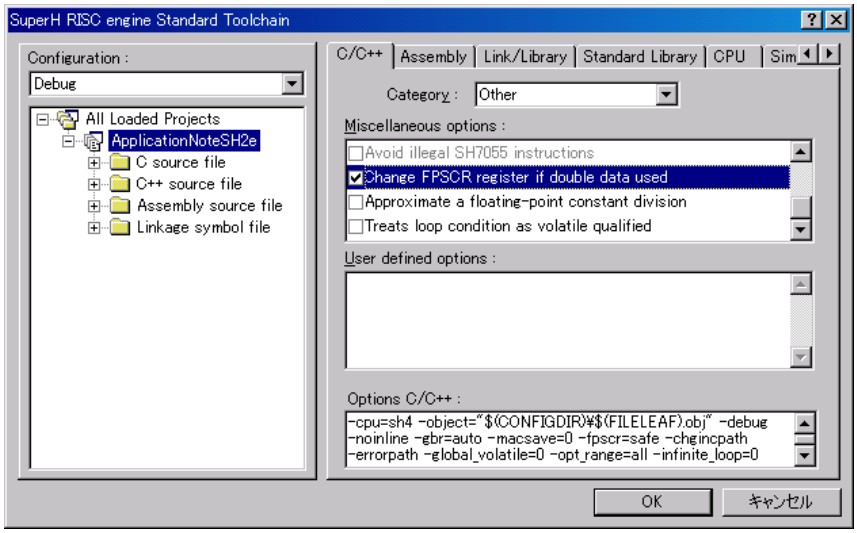


图 6.15 在 SH-4 的浮点设定中选取“混合”(Mix) 时的建议选项

注意： 由于此功能会更改函数的界面，这项更改必须同时在所有的文件中进行。

6.2 优化常数除法

◆重点：

优化处理会将常数除法转变成除法以外的运算。 因此，尽可能使用常数除法。

◆描述：

优化处理会将常数除法转变成常数倒数近似值的乘法然后微调结果。 这将大大增进在除法中调用子例程的执行速度。

◆使用的实例：

在以下的增进实例中，将常数用作除数可使指令字符串直接取得 3 的商数而无需调用除法例程。 类似代码也会为其他常数除法生成。

优化前的源代码

```
int x;
int z=3;
void f (int y){
  x=y/z;
}
```

优化前的汇编代码

```
_f:
            STS.L       PR,@-R15
            MOV.L       L11,R5
            MOV         R4,R1
            MOV.L       L11+4,R2
            JSR         @R2
            MOV.L       @R5,R0
            MOV.L       L11+8,R6
            LDS.L       @R15+,PR
            RTS
            MOV.L       R0,@R6
L11:
            .DATA.L     _z
            .DATA.L     __divls
            .DATA.L     _x
```

优化后的源代码

```
int x;
void f (int y){
  x=y/3;
}
```

优化后的汇编代码

```
_f:
            MOV.L       L11,R2
            DMULS.L     R4,R2
            STS         MACH,R6
            MOV         R6,R0
            ROTL        R0
            AND         #1,R0
            ADD         R6,R0
            MOV.L       L11+4,R6
            RTS
            MOV.L       R0,@R6
L11:
            .DATA.L     H'55555556
            .DATA.L     _x
```

注意： 这个可大大增进速度的优化并不应用于优化大小，因为所扩展的代码会变得太大。

6.3 整数除法的大小

♦重点：

以数据类型（字符或短）执行的整数除法会比以整数类型（32 位）来得快，因为前者比较短。

♦描述：

对于除法，每一个 32 位、16 位和 8 位大小都可使用不同的运行时例程。如果值的范围有限，以较小大小类型执行除法将会比较快。

♦使用的实例：

在以下的增进实例中，如果除数、被除数和结果是在 16 位范围内，将除法操作数和结果声明为短类型，仍然会调用 16 位除法例程 (divws)，而不是 32 位除法例程 (divls)。

优化前的源代码

```
int x;
int y;
int z;
void f(){
  x=y/z;
}
```

优化前的汇编代码

```
_f:
            STS.L       PR,@-R15
            MOV.L       L11+2,R6
            MOV.L       L11+6,R4
            MOV.L       L11+10,R2
            MOV.L       @R6,R0
            JSR         @R2
            MOV.L       @R4,R1
            MOV.L       L11+14,R6
            LDS.L       @R15+,PR
            RTS
            MOV.L       R0,@R6
L11:
            .RES.W      1
            .DATA.L     _z
```

优化后的源代码

```
short x;
short y;
short z;
void f(){
  x=y/z;
}
```

优化后的汇编代码

```
_f
            STS.L       PR,@-R15
            MOV.L       L11+2,R6
            MOV.L       L11+6,R4
            MOV.L       L11+10,R2
            MOV.W       @R6,R0
            JSR         @R2
            MOV.W       @R4,R1
            MOV.L       L11+14,R6
            LDS.L       @R15+,PR
            RTS
            MOV.W       R0,@R6
L11:
            .RES.W      1
            .DATA.L     _z
```

.DATA.L _y .DATA.L __divls .DATA.L _x	.DATA.L _y .DATA.L __divws .DATA.L _x

6.4 寄存器声明

♦重点：

SH C/C++ 编译程序 7.1 版本会在不论是否存在寄存器声明的情形下，以相同的方式分配变量。

♦描述：

编译程序会衡量本地变量的使用频率（通过给予循环中出现的变量较高的优先级，等等）然后依此分配寄存器。变量的寄存器声明将会被忽略。

♦使用的实例：

下图显示在具备和不具备寄存器声明时编译程序的实例。 两个程序都会使用相同的寄存器分配来生成代码。

不具备寄存器声明的源代码

```
int a[10];
int f(){
  int i;
  int s=0;
  for (i=0; i<10; i++)
    s+=a[i];
  return s;
}
```

汇编代码

```
_f:
          MOV       #0,R2
          MOV.L     L13,R5
          MOV       #5,R4
L11:
          MOV.L     @R5,R6
          DT        R4
          ADD       R6,R2
          MOV.L     @(4,R5),R6
          ADD       #8,R5
          BF/S      L11
          ADD       R6,R2
```

具备寄存器声明的源代码

```
int a[10];
int f(){
  register int i;
  register int s=0;
  for (i=0; i<10; i++)
    s+=a[i];
  return s;
}
```

汇编代码

```
_f
          MOV       #0,R2
          MOV.L     L13,R5
          MOV       #5,R4
L11:
          MOV.L     @R5,R6
          DT        R4
          ADD       R6,R2
          MOV.L     @(4,R5),R6
          ADD       #8,R5
          BF/S      L11
          ADD       R6,R2
```

```
            RTS
            MOV         R2,R0
L13:
            .DATA.L     _a
```

```
            RTS
            MOV         R2,R0
L13:
            .DATA.L     _a
```

注意： 当编译程序自动将变量分配到寄存器时，如果在函数中使用过多本地变量，将会很难执行有效的寄存器分配。建议您适当拆分函数直到循环中仅使用八个或更少的本地变量。

6.5 偏移结构声明中的成员

♦重点：

在代码的开头部分声明常用的结构成员可同时增进大小和速度。

♦描述：

程序通过将偏移添加到结构地址来存取结构成员。 偏移越小，速度和大小就越有利。 因此，请在代码的开头部分声明常用的成员。

在字符和无符号字符类型的开头部分为成员声明少于 16 字节，在短和无符号短类型的开头部分声明少于 32 字节，以及在整数、无符号、长和无符号长类型的开头部分声明少于 64 字节将最为有效。

♦使用的实例：

在下列实例中，结构的偏移将改变代码。

优化前的源代码

```
struct S{
  int a[100];
  int x;
};

int f(struct S *p){
  return p->x;
}
```

优化前的汇编代码

```
_f:
          MOV         #100,R0
          SHLL2       R0
          RTS
          MOV.L       @(R0,R4),R0
```

优化后的源代码

```
struct S{
  int x;
  int a[100];
};

int f(struct S *p){
  return p->x;
}
```

优化后的汇编代码

```
_f:
          RTS
          MOV.L       @R4,R0
```

6.6 分配位字段

◆重点：

要参考的具有相同表达式的位字段应该分配到同个结构中。

◆描述：

不同位字段中的成员每一次被参考时，必须加载包含位字段的数据。您可以通过将相关的位字段分配到同个结构，仅加载此数据一次。

◆使用的实例：

在下列实例中，相关的位字段将分配到同个结构，从而同时增进速度和大小。

优化前的源代码

```
struct bits{
  unsigned int b0: 1;
} f1, f2;

int f(void){
  if (f1.b0 && f2.b0) return 1;
  else return 0;
}
```

优化前的汇编代码

```
_f:
          MOV.L       L15,R6
          MOV.B       @R6,R0
          TST         #128,R0
          BT          L12
          MOV.L       L15+4,R6
          MOV.B       @R6,R0
          TST         #128,R0
          BT          L12
          RTS
          MOV         #1,R0
L12:
```

优化后的源代码

```
struct bits{
  unsigned int b0: 1;
  unsigned int b1: 1;
} f1;

int f(void){
  if (f1.b0 && f1.b1) return 1;
  else return 0;
}
```

优化后的汇编代码

```
_f:
          MOV.L       L11,R6
          MOV         #-64,R3
          EXTU.B      R3,R3
          MOV.B       @R6,R0
          AND         #192,R0
          CMP/EQ      R3,R0
          RTS
          MOVT        R0
L11:
          .DATA.L     _f1
```

```
            RTS
            MOV         #0,R0
L15:
            .DATA.L     _f1
            .DATA.L     _f2
```

6.7 软件流水线（浮点表搜索）

♦重点：

您可以通过将表搜索代码，设计为与载入之前的循环迭代的数据项目表参考的数据项目比较而不是立即比较它们，来增进表搜索代码的执行速度。

♦描述：

流水线优化不能仅使用很少的指令（如，表搜索）来充分优化循环，因为缺乏可以重新安排结构的空间。 例如，对于浮点表搜索，如果程序在从表加载数据后立即执行比较，FCMP 运算则必须等到加载完成后才能执行。 若要修正此程序，请将程序设计为将比较数据加载到一个循环迭代中的本地变量，然后在下一个迭代中比较它们。

♦使用的实例：

增进前的代码使用紧随加载的浮点数据 (FR8) 后面编写的 FCMP 指令来与之进行比较。 如果代码已获得增进，循环迭代中参考的数据将会在下一个迭代中比较，加载将会与循环的转移指令平行执行。

优化前的源代码

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  while (i<100){
    if (*p==b) return i;
   i++;
   p++;
  }
  return -1;
}
```

优化后的源代码

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  float tmp=*p;
  while (i<100){
    if (tmp==b) return i;
   i++;
   p++;
   tmp=*p;
  }
  return -1;
}
```

优化前的汇编代码

```
_f:
          MOV         #0,R5
          MOV.L       L16,R2
          MOV         #100,R6
L11:
```

优化后的汇编代码

```
_f:
          MOV.L       L16+2,R2
          MOV         #0,R5
          MOV         #100,R6
          FMOV.S      @R2,FR8
```

```
        FMOV.S      @R2,FR8
        FCMP/EQ     FR4,FR8
        BT          L12
        DT          R6
        ADD         #1,R5
        BF/S        L11
        ADD         #4,R2
        RTS
        MOV         #-1,R0
L12:
        RTS
        MOV         R5,R0
L16:
        .DATA.L     _a
```

```
L11:
        FCMP/EQ     FR4,FR8
        BT          L12
        ADD         #4,R2
        DT          R6
        FMOV.S      @R2,FR8
        BF/S        L11
        ADD         #1,R5
        RTS
        MOV         #-1,R0
L12:
        RTS
        MOV         R5,R0
L16:
        .RES.W      1
        .DATA.L     _a
```

6.8 确保数据存取大小

◆重点：

声明 volatile 可确保用于存取外围寄存器的存储器存取指令的大小（字节、字、长字）。

◆描述：

声明 volatile 以确保用于存取全局变量和指针的指令之大小。这将形成一个指令，用于根据数据类型的大小加载和存储数据。声明 volatile 以存取位字段，以便可以使用在声明位字段时所使用的数据类型来存取它们。除非 volatile 已声明，否则位字段的存取将会优化，可能导致使用所声明以外的其他类型来存取。

◆使用的实例：

如果未声明 volatile，成员 x 将会存取为字节存取。如果已声明 volatile，它将会存取为所声明的类型（字）。

不指定 volatile 的源代码

```
struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

汇编代码

```
_f:
          MOV.L     L11+2,R2
          MOV.L     @R2,R6
          MOV.B     @R6,R2
          RTS
          EXTS.W    R2,R0
L11:
          .RES.W    1
          .DATA.L   _p
```

指定 volatile 的源代码

```
volatile struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

汇编代码

```
_f:
          MOV.L     L11,R2
          MOV.L     @R2,R6
          MOV.W     @R6,R2
          SHLR8     R2
          RTS
          EXTS.B    R2,R0
L11:
          .DATA.L   _p
```

6.9 使用浮点指令

♦重点：

要使用单精度浮点指令 FABS（SH2-E、SH2A-FPU、SH-4 和 SH-4A）以及 FSQRT（SH2A-FPU、SH-4、SH-4A），请加入包含文件 <mathf.h> 然后调用单精度浮点函数 fabsf 和 sqrtf。

♦描述：

按照以下方式，使用单精度浮点指令 FABS（SH2-E、SH2A-FPU、 SH-4 和 SH-4A）以及 FSQRT（SH2A-FPU、SH-4、SH-4A）：

(a) 包含 <math.h>。

(b) 调用 fabsf 函数 (FABS) 和 sqrtf 函数 (FSQRT)。

♦使用的实例：

在增进前的实例中，<mathf.h> 尚未包含，因此编译程序会从程序库调用 *fabsf* 函数，而不会将它识别为一个标准函数。如果已包含 <mathf.h>，编译程序会将它识别为与 FABS 指令对应的函数，从而直接生成 FABS 指令。

优化前的源代码

```
float fabsf(float);

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

优化前的汇编代码

```
_f:
            STS.L       PR,@-R15
            FMOV.S      FR14,@-R15
            FMOV.S      FR15,@-R15
            MOV.L       L12+2,R2
            JSR         @R2
            FMOV.S      FR5,FR15
            MOV.L       L12+2,R2
            FMOV.S      FR0,FR14
            JSR         @R2
            FMOV.S      FR15,FR4
            FADD        FR0,FR14
            FMOV.S      FR14,FR0
```

优化后的源代码

```
#include <mathf.h>

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

优化后的汇编代码

```
_f:
            FABS        FR4
            FABS        FR5
            FADD        FR5,FR4
            RTS
            FMOV.S      FR4,FR0
```

```
            FMOV.S      @R15+,FR15
            FMOV.S      @R15+,FR14
            LDS.L       @R15+,PR
            RTS
            NOP
L12:
            .RES.W      1
            .DATA.L     _fabsf
```

注意：头文件<mathf.h> 不是 ANSI 的标准 C 程序库函数。

SuperH RISC Engine C/C++编译程序应用笔记

使用 HEW

第 7 节　使用 HEW

本章描述 HEW 在与创建和模拟相关的程序上的使用。
注意不同版本的 HEW 所支持的函数和方法将有所不同。
适当的版本将在每个主题的 [建议] 中表示。

下表显示与 HEW 的使用有关的项目列表。

编号	类别	项目	段
1	创建	再生成和编辑自动生成的文件	7.1.1
2		命令描述文件的输出	7.1.2
3		命令描述文件的输入	7.1.3
4		创建定制的工程类型	7.1.4
5		多个 CPU 功能	7.1.5
6		网络功能	7.1.6
7		从 HEW 的旧版本转换	7.1.7
8		将 HIM 工程转换为 HEW 工程	7.1.8
9		添加支持的 CPU	7.1.9
10	模拟	伪中断	7.2.1
11		方便断点函数	7.2.2
12		覆盖功能	7.2.3
13		文件输入/输出	7.2.4
14		调试程序目标同步	7.2.5
15		如何使用定时器	7.2.6
16		定时器的使用实例	7.2.7
17		重新配置调试程序目标	7.2.8
18	Call Walker	创建堆栈信息文件	7.3.1
19		启动 Call Walker	7.3.2
20		Call Walker 窗口和打开文件	7.3.3
21		编辑堆栈信息	7.3.4
22		汇编程序的堆栈区域大小	7.3.5
23		合并堆栈信息	7.3.6
24		其他功能	7.3.7

7.1 创建

7.1.1 再生成和编辑自动生成的文件

♦ 描述：

若您在创建新的工作空间时，将工程类型选取为应用程序 (Application)，HEW 将自动生成 I/O 寄存器定义、中断函数，及其他各种文件。

然而，当创建新的工程时，您可能偶尔跳过自动生成文件这一步，因为您在当时认为那些文件是不必要的。

您也可能忘了编辑或设定这类文件。

若的确如此，您可以使用此功能在创建工程后自动生成和编辑文件。

然而，此功能仅在您创建新工作空间时，将工程类型选取为应用程序 (Application) 之后可用。

♦ 使用：

HEW 菜单： **工程 (Project) > 编辑工程配置... (Edit Project Configuration...)**

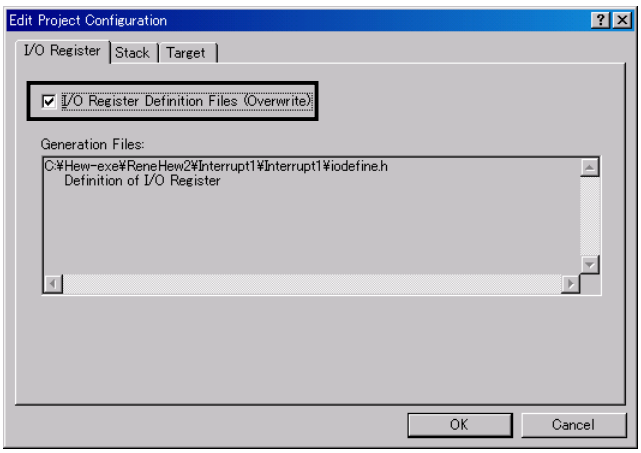
♦ 可被再生成的文件包括：

I/O 寄存器定义文件： iodefine.h

[生成方法]

您可通过在 [编辑工程配置 (Edit Project Configuration)] 对话框内，选中 [I/O 寄存器 (I/O Register)] 标签上的 [I/O 寄存器定义文件（覆盖）(I/O Register Definition Files (overwrite))] 来再生成 iodefine.h。

若您不小心修改了 iodefine.h，您可以再生成该文件，并覆盖已修改的文件。

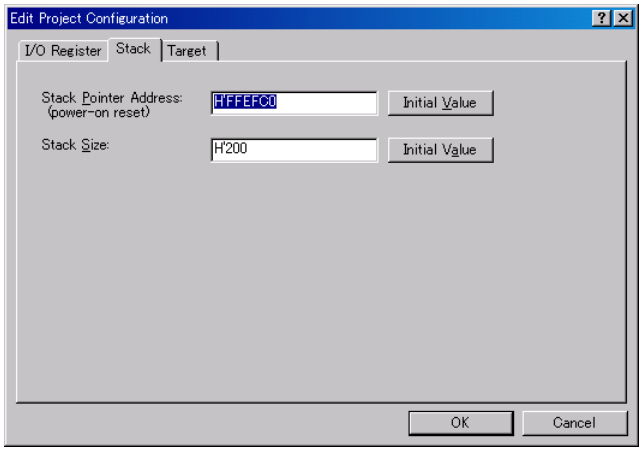


♦ 可被再编辑的文件包括：

堆栈大小设定文件：stacksct.h

［编辑方法］

您可以在 [编辑工程配置 (Edit Project Configuration)] 对话框内的 [堆栈 (Stack)] 标签上，编辑 [堆栈指针地址 (Stack Pointer Address)] 和 [堆栈大小 (Stack Size)] 的初始值。



♦ 注意：

再生成和再编辑文件受 HEW 2.0 或以上版本支持。

7.1.2 命令描述文件的输出

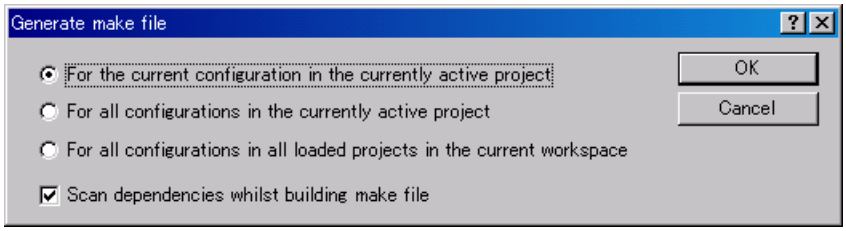
♦ 描述：

HEW 可以让您根据当前的选项设定来创建命令描述文件。

通过使用命令描述文件，您不须安装完整的 HEW，就可以创建当前工程。 当您要将工程传送给尚未安装 HEW 的人，或者要管理整个创建的版本（包括命令描述文件）时，就变得很方便。

♦ 命令描述文件的制作方法：

1. 确保生成命令描述文件的工程是当前工程。
2. 确保创建工程的创建配置是当前配置。
3. 选择 [创建 (Build) > 生成命令描述文件 (Generate make file)]。
4. 您将看到下列对话框。 在此对话框中，选取命令描述文件的其中一项生成方法。



♦ 命令描述文件生成目录：

HEW 在当前工作空间目录中创建 [命令描述 (make)] 子目录，并在此子目录内生成命令描述文件。命令描述文件的名称，是当前工程或配置的名称，配上 .mak 的扩展名（例如，debug.mak）。 HEW 生成的命令描述文件，能被 HEW 安装目录内的可执行文件 HMAKE.EXE 所执行。 然而，用户修改的命令描述文件将不能执行。

♦ 命令描述文件的执行方法：

1. 打开 [命令 (Command)] 窗口，然后前往包含该命令描述文件的 [命令描述 (make)] 子目录。
2. 执行 HMAKE。在命令行上，输入 HMAKE.EXE <命令描述文件名称>。

♦ 注意：

此功能受 HEW 1.1 或以上版本支持。

7.1.3 命令描述文件的输入

♦ 描述：

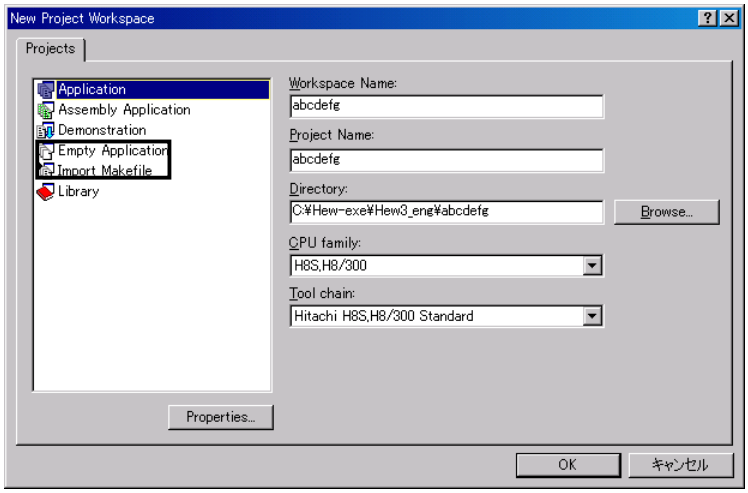
HEW 允许输入由 HEW 生成或在 UNIX 环境中使用的命令描述文件。

从命令描述文件，您可自动获取工程的**文件结构**。

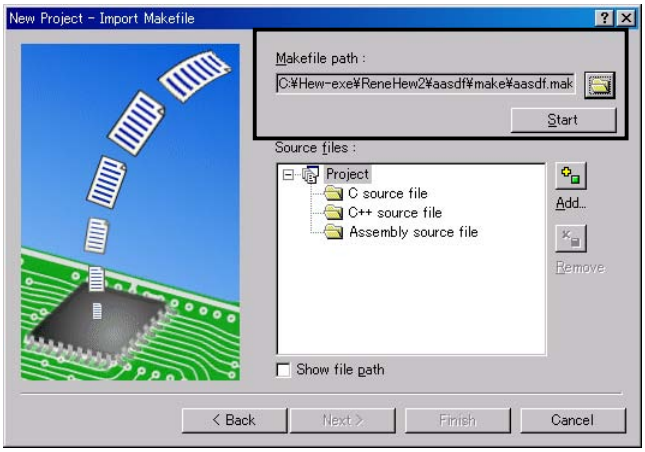
（不过，您不能获取选项设定或类似的规格。） 这方便了从命令行至 HEW 的转移。

♦ 命令描述文件的输入方法：

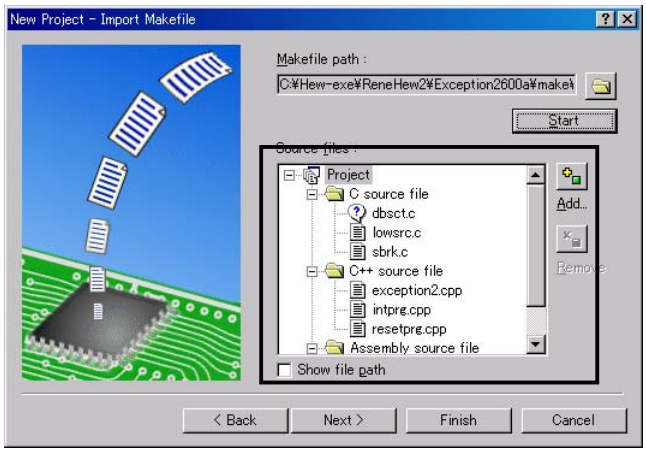
1. 当创建新的工作空间时，在 [新的工程工作空间 (New Project Workspace)] 对话框内，从工程类型选项选取 [导入命令描述文件 (Import Makefile)]。



2. 在 [新的工程－导入命令描述文件 (New Project-Import Makefile)] 对话框内的 [命令描述文件的路径 (Makefile path)] 字段中，指定命令描述文件的路径，然后单击 [开始 (Start)] 按钮。



3. [源文件 (Source files)] 窗格显示命令描述文件的源文件结构。在此结构图解中，任何含有标志的文件均为透过分析被证实为不含实体的文件。此文件将不被添加到工程（被忽略）。



4. 跟随向导，指定 CPU 和其他选项，并打开工作空间。 之后您就可以开始进行开发工作。

♦ 注意：

此功能受 HEW 3.0 或以上版本支持。

7.1.4 创建定制的工程类型

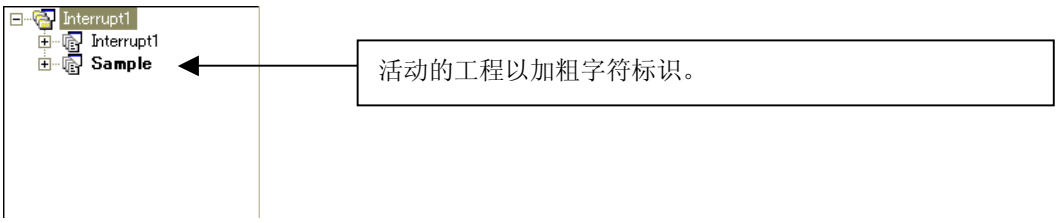
♦ 描述：

此功能允许用户创建的工程，被另一用户在其他机器上用作程序开发的模板。

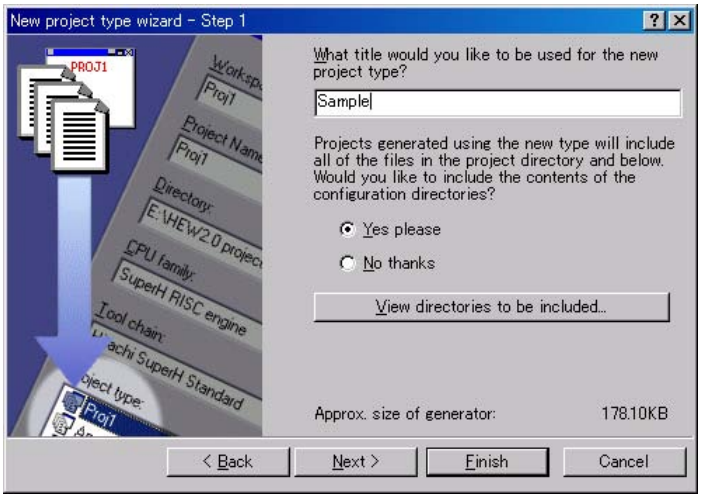
可从模板中获取的信息包括工程文件结构、创建选项、调试程序设定，及其他和工程有关的信息。

♦ 工程类型的存储方法：

1. 启动您要用以存储工程信息的工程，因为活动的工程将在工作空间打开时接收工程信息。要启动工程，通过选择 [工程 (Project) -> 设定当前工程 (Set Current Project)] 来选取工程。

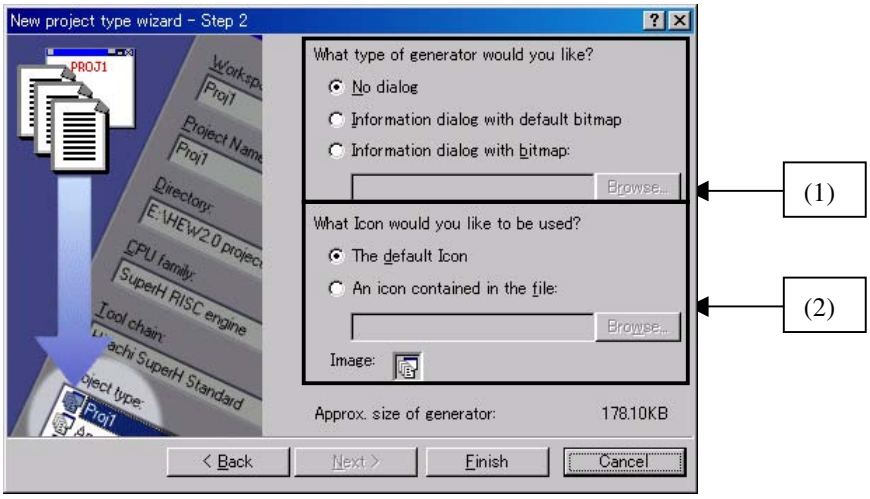


2. 通过选择 [工程 (Project) -> 创建工程类型 (Create Project Type)…] 来打开下列的工程类型向导，为您将用作模板的工程类型指定名称，同时指定是否要将含有后创建可执行文件及其他资源的配置目录，包含在模板内。
   您可通过单击 [完成 (Finish)] 按钮，在此退出工程类型向导。



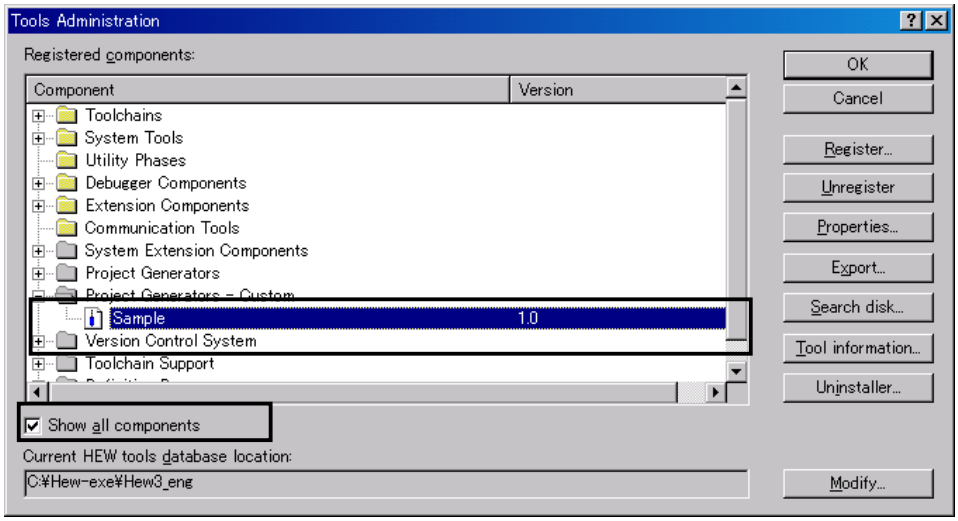
3. 在 [新的工程类型向导 – 步骤 1 (New project type wizard – Step 1)] 中，单击 [下一步 (Next)] 按钮以打开下列向导：当在步骤（1）中打开工程类型模板时，指定是否显示工程信息和点阵图。
   在步骤（2）中，您可将工程类型图标更改为用户指定的图标。 单击 [完成 (Finish)] 按钮。

   这些设定并不是强制性的。



4. 一个命名为“定制工程生成程序 (Custom Project Generator)”的工程类型模板因此创建。要在其他机器上使用此模板，请选择 [工具 (Tools) -> 管理 (Administration)…] 以打开下列对话框：
   在您选中了下列 [显示全部组件 (Show all components)] 复选框时，您将看到 [工程生成程序 – 定制 (Project Generators – Custom)]。

   单击已创建的工程类型，然后单击 [导出 (Export)…] 按钮。

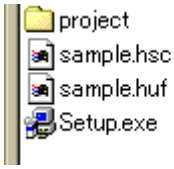


5. 下列对话框将打开。选取用以存储定制工程生成程序模板的目录。该目录必须是空的。
   工程类型的存储过程至此完成。

◆ 安装定制工程生成程序：

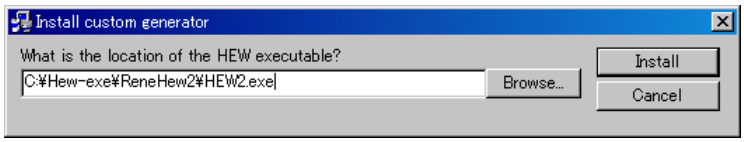
使用下列步骤，以在其他机器上安装由以上工程类型存储方法所创建的定制工程生成程序模板。

1. 下列安装环境为工程类型存储方法步骤 5 中所创建的目录而建：
   （安装环境目录）



2. 复制上述安装环境，再将副本安装到其他机器上。
   当您运行 Setup.exe 时，下列对话框将打开。 指定 HEW2.exe 将被安装的位置，然后单击 [安装 (Install)] 按钮。

   （目录实例：c:¥Hew2¥HEW2.exe）

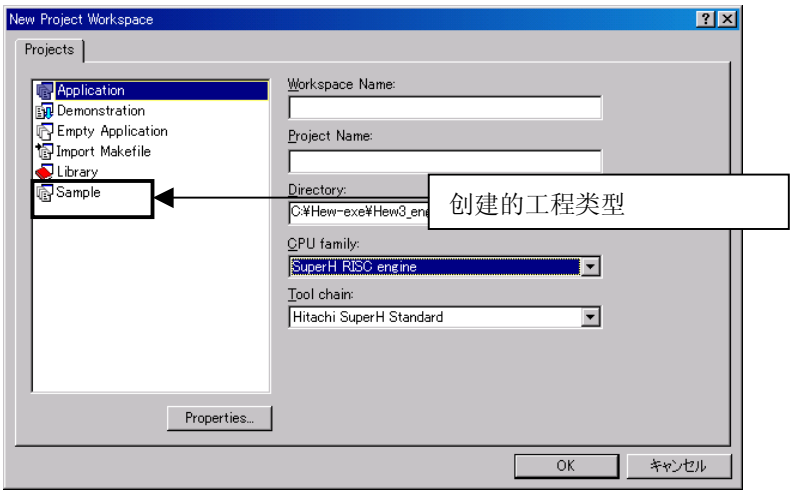


3. 环境已创建完成。

♦ 定制工程生成程序的使用实例：

下列实例显示已安装的定制工程生成程序模板的使用。

1. 启动 HEW，在 [欢迎使用！(Welcome!)] 对话框中选择 [创建新的工程工作空间 (Create a new project workspace)]。已安装的工程类型将会添加到 [工程 (Projects)] 列表。单击工程类型，然后单击 [确定 (OK)] 按钮。
   您可以开始使用已存储的工程模板，为任何新工程进行程序开发。



♦ 注意：

此功能受 HEW 2.0 或以上版本支持。

7.1.5 多个 CPU 功能

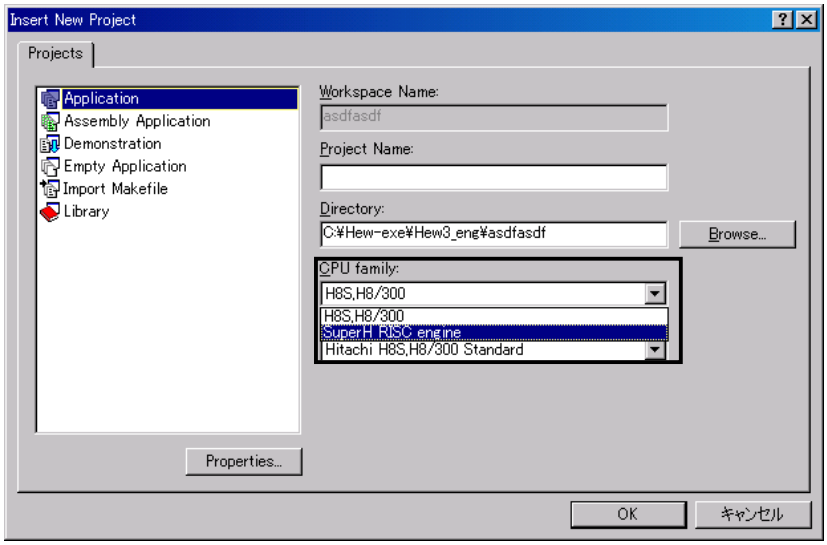
♦ 描述：

在工作空间中插入新工程时，您可以插入其他类型的 CPU。这将允许 SH 和 H8 的工程在单一工作空间内被管理。

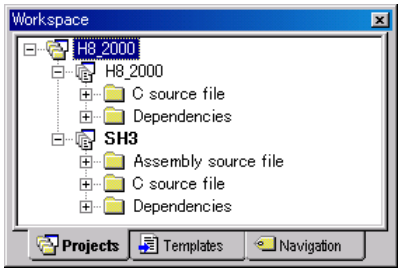
♦ 插入不同 CPU 家族的实例：

1. 在 H8 (SH) 工程打开时，单击 [工程 (Project) -> 插入工程 (Insert Project)…]。在 [插入工程 (Insert Project)] 对话框内，选取一个新工程，然后单击 [确定 (OK)] 按钮。

2. 将显示下列 [插入新工程 (Insert New Project)] 对话框：选取工程名称，将 CPU 类型选为 SH (H8)，然后单击 [确定 (OK)] 按钮。在工作空间内，除了当前的 CPU 类型，您还可以放置其他的 CPU 类型。



3. 通过以上步骤，您可以在单一工作空间内混合 SH 和 H8 工程。



♦ 注意：

此功能受 HEW 3.0 或以上版本支持。

7.1.6 网络功能

♦ 描述：

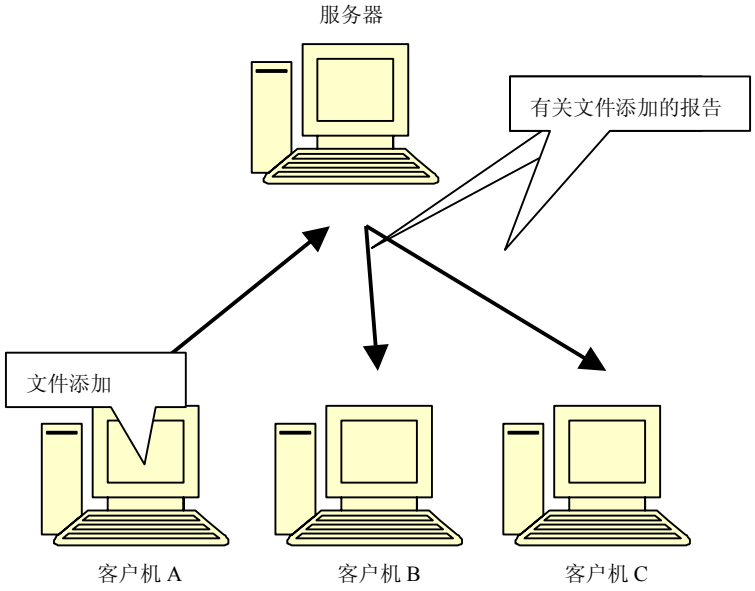
HEW 允许将工作空间和工程，通过网络被不同用户所共享。

由此，用户可通过同时操控共享的工程，获知其他用户所做出的更改。

此系统将一台计算机用作其服务器。

例如，若客户为工程添加了新文件，服务器机器将接获通知，然后再通知其他客户。

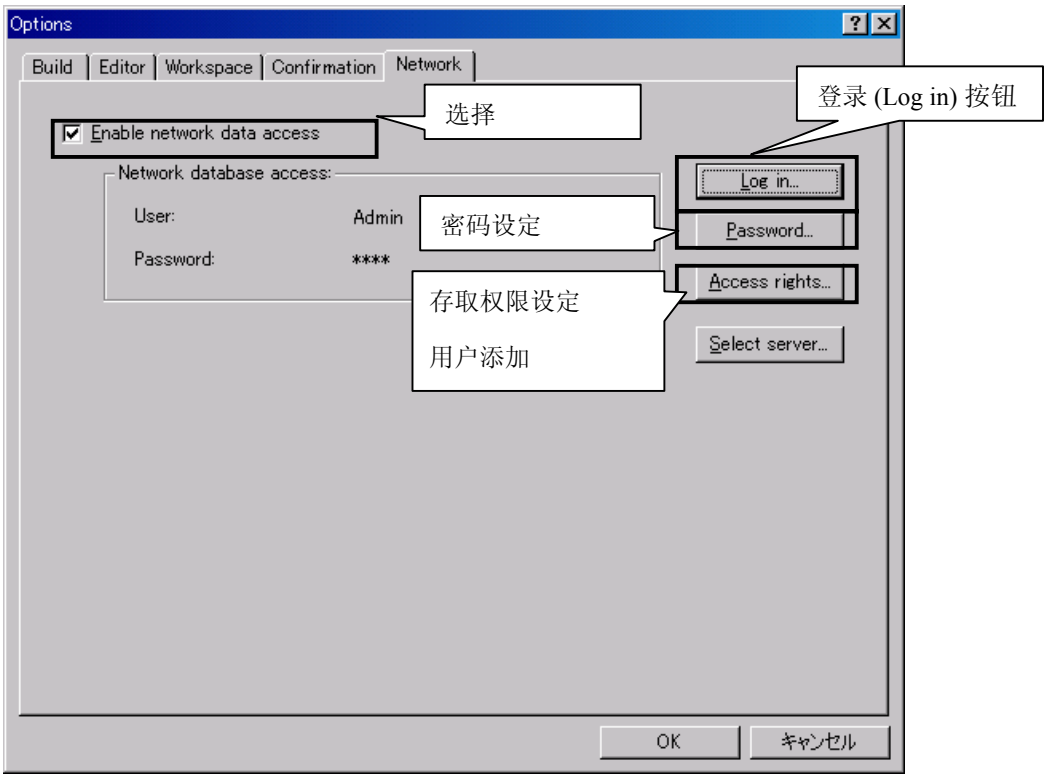
此外，用户可被授予特定工程或文件的存取权限。



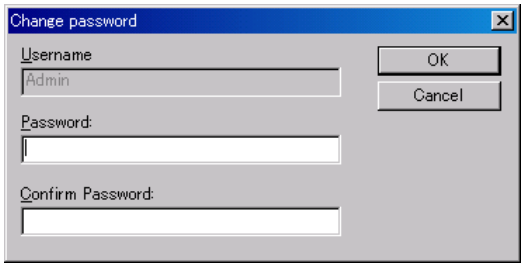
♦ 网络存取设置：

1. 选择 [工具 (Tools) -> 选项 (Options)]，然后选取 [网络 (Network)] 标签。 选中 [允许网络数据存取 (Enable network data access)] 复选框。
2. 一位管理员将会添加。 由于管理员开始时没有密码，您将需要指定密码。 管理员将被授予最高的存取权限。
3. 单击 [密码 (Password)…] 按钮，然后为管理员指定密码。
4. 单击 [确定 (OK)] 按钮。 这将让管理员存取网络。

[选项 (Options)] 对话框的 [网络 (Network)] 标签



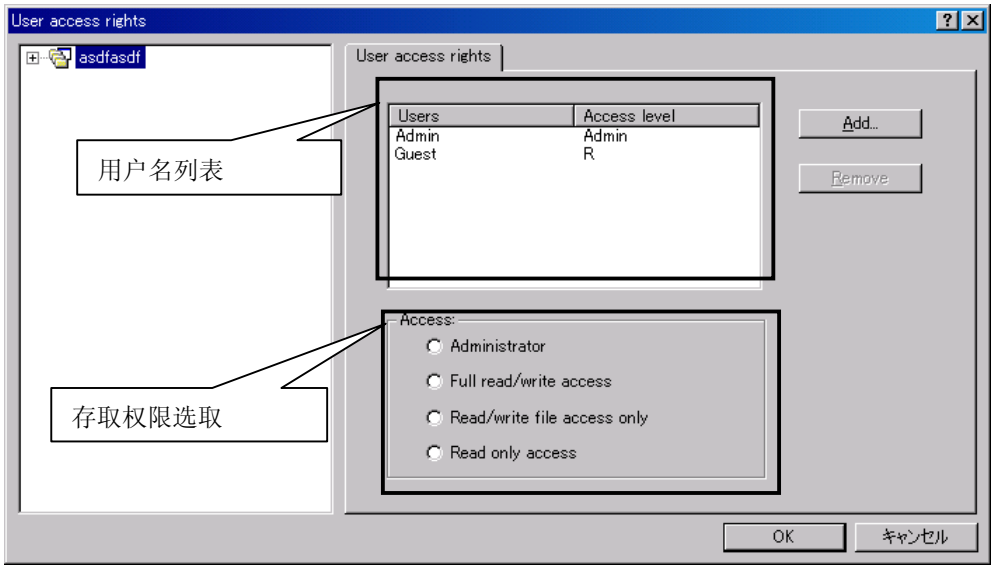
[更改密码 (Change password)] 对话框



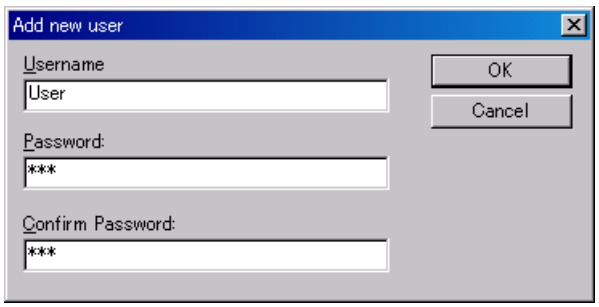
◆ 添加新用户：

默认情形下，将会添加一位管理员和一位访客。 您可以注册新用户。

1. 单击前页所示的 [登录 (Log in) …] 按钮。 登录为被授予管理员存取权限的用户。
2. 单击 [存取权限 (Access rights) …] 按钮，以打开下列 [用户存取权限 (User access rights)] 对话框。



3. 单击 [添加 (Add) …] 按钮，以打开 [添加新用户 (Add new user)] 对话框。
4. 输入新用户名称和密码（密码指定是强制性的）。



5. 接着，新用户名就会添加到用户列表中。 选取用户名和指定该用户的存取权限。
6. 单击 [确定 (OK)] 按钮。 您的指定即将生效。

♦ 选取服务器机器

选取将被用作服务器的机器。 如果您希望以自己的机器作为服务器，您不需要执行任何操作。

若您希望将其他机器指定为服务器，请单击 [选项 (Options)] 对话框内的 [选取服务器 (Select server) …] 按钮。 在下列对话框内选择 [远程 (Remote)]，然后指定计算机名称。

单击 [确定 (OK)] 按钮。 您的指定即将生效。

♦ 注意：

此功能受 HEW 3.0 或以上版本支持。

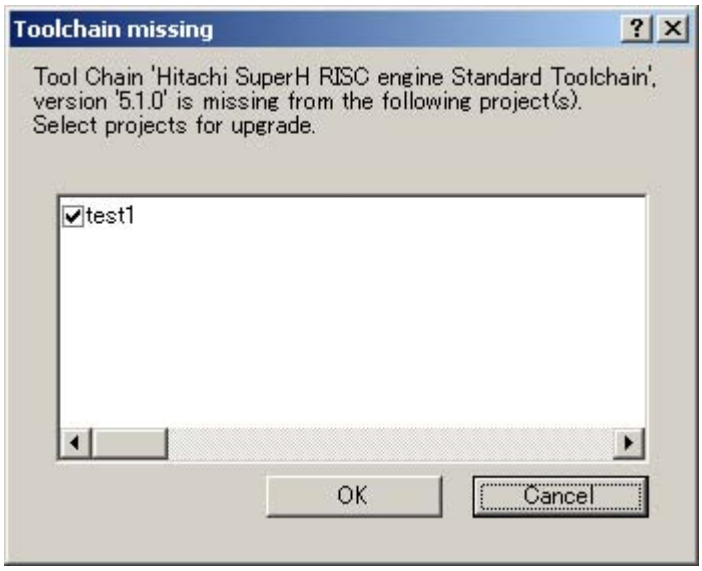
使用此功能将降低 HEW 的性能。

7.1.7 从 HEW 的旧版本转换

这里为您说明在“瑞萨集成开发环境”(Renesas Integrated Development Environment) 中指定编译程序版本的方法。 编译程序版本可以通过升级“瑞萨集成开发环境”(Renesas Integrated Development Environment) 来指定。

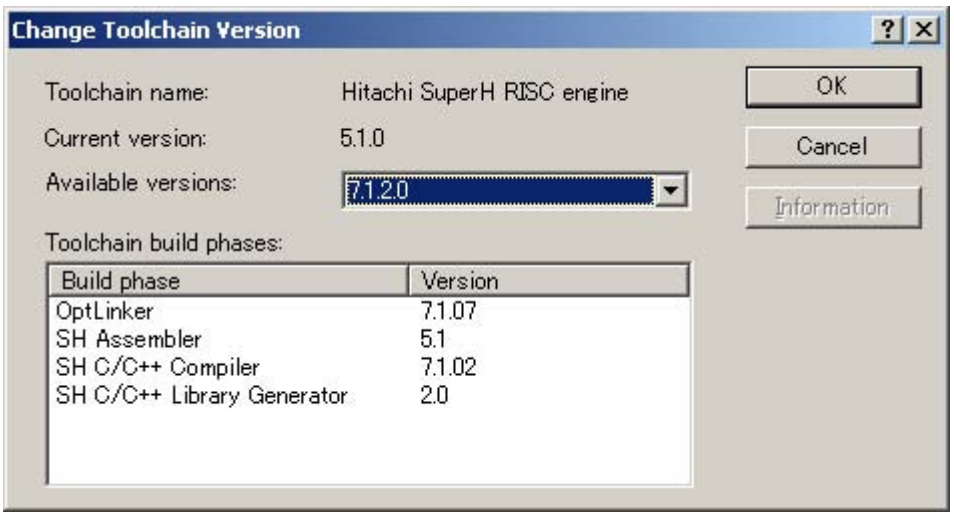
如果将旧版本（如 HEW1.1 或 SHC5.1B）中创建的工作空间在新版本（如 HEW3.0 或 SHC8.0）中打开，下列对话框将会出现。

(1) 检查要升级的工程。
检查要升级的工程的名称。



高性能嵌入式工作区 (High-performance Embedded Workshop)

(2) 指定编译程序版本
选取可以升级的编译程序版本。

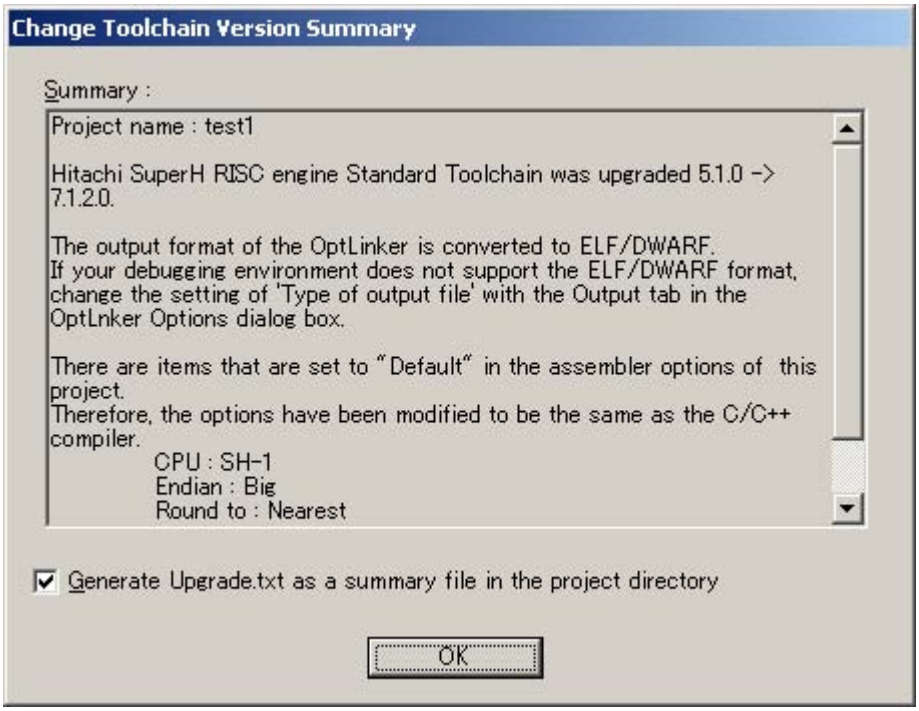


“更改工具链版本”(Change Toolchain Version) 对话框

(3) 确认信息

C/C++ 编译程序 6.0 或以上版本仅支持要输出之目标的 ELF/DWARF 文件格式。

文件格式将会在升级时更改为 ELF/DWARF 格式。如果当前的调试环境不支持 ELF/DWARF 格式，请将 ELF/DWARF 格式转换为升级后调试环境所支持的格式。



“确认信息”对话框

(4) 标准程序库生成程序选项

升级后，**“SuperH RISC engine 标准工具链”**（**SuperH RISC engine Standard Toolchain**）对话框中的**“标准程序库**标签类别：”(**Standard Library** Tab Category:) [**模式 (Mode)**] 将会更改为**“创建程序库文件（随时）”(Build a library file(anytime))**，因此，请务必小心。

7.1.8 将 HIM 工程转换为 HEW 工程

通过使用 HEW 系统随附的 HimToHew 工具，您可以将 HIM 工程转换为 HEW 工程。

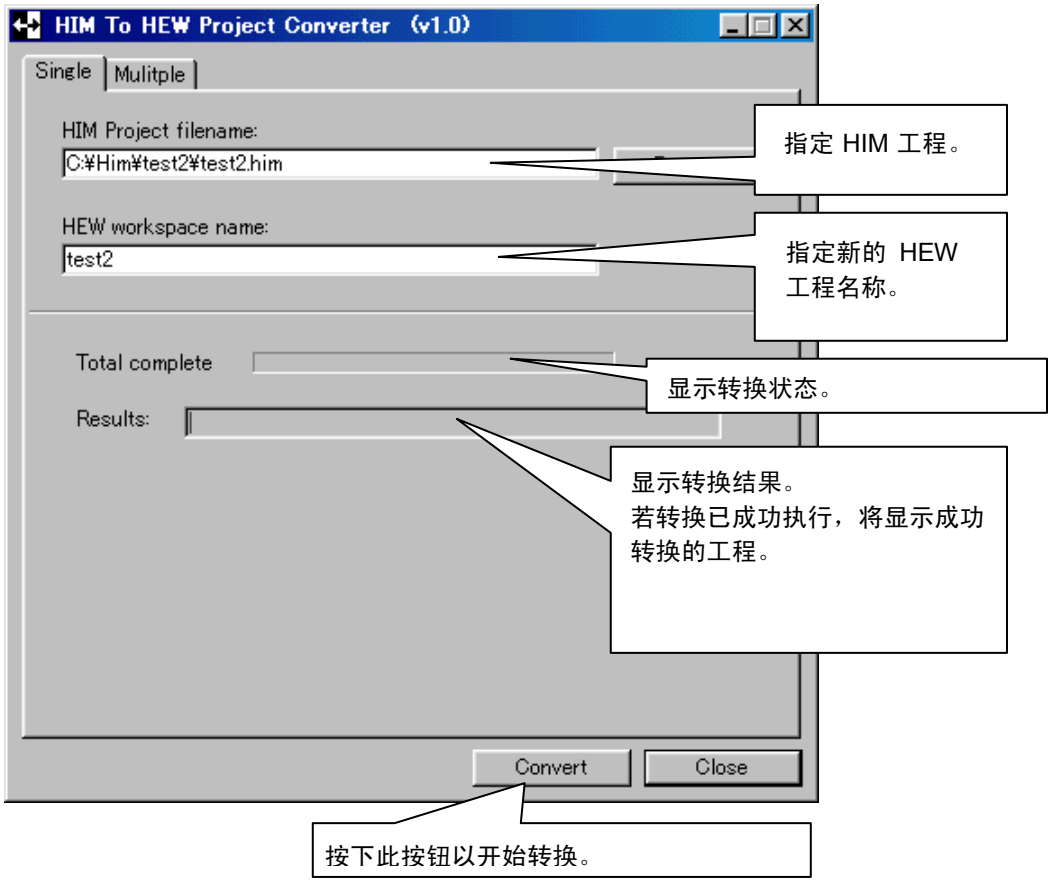
在 Windows® [开始 (Start) 菜单] 上的 [程序(P) (Programs(P))] 中，从 [瑞萨高性能嵌入式工作区 (Renesas High-performance Embedded Workshop)] 选取 [Him 到 Hew 的工程转换器 (Him To Hew Project Converter)]。

您将找到单一 (Single) 和多个 (Multiple) 标签。

若要从一个 HIM 工程生成一个 HEW 工作空间和一个 HEW 工程时，请选择单一 (Single) 标签。

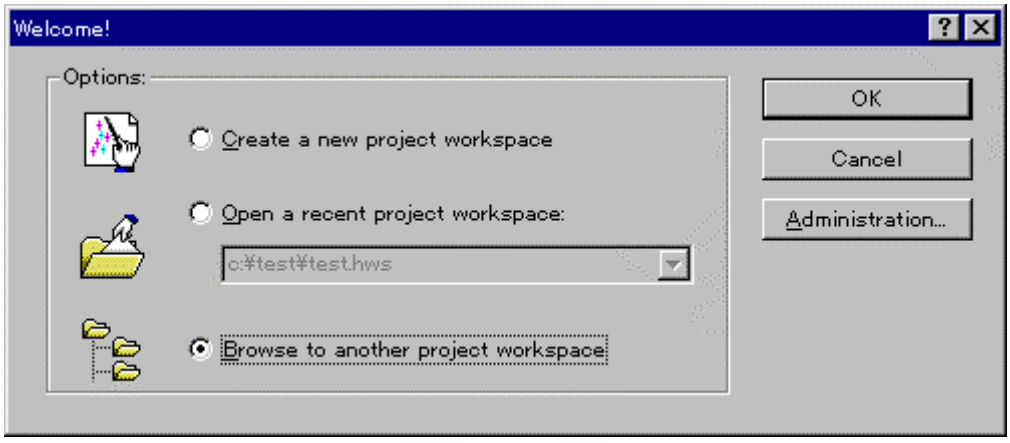
若将多个 HIM 工程转换为 HEW 工程，并将它们成批注册在 HEW 工作空间内时，则请选择多个 (Multiple) 标签。

(1) 单一 (Single) 标签

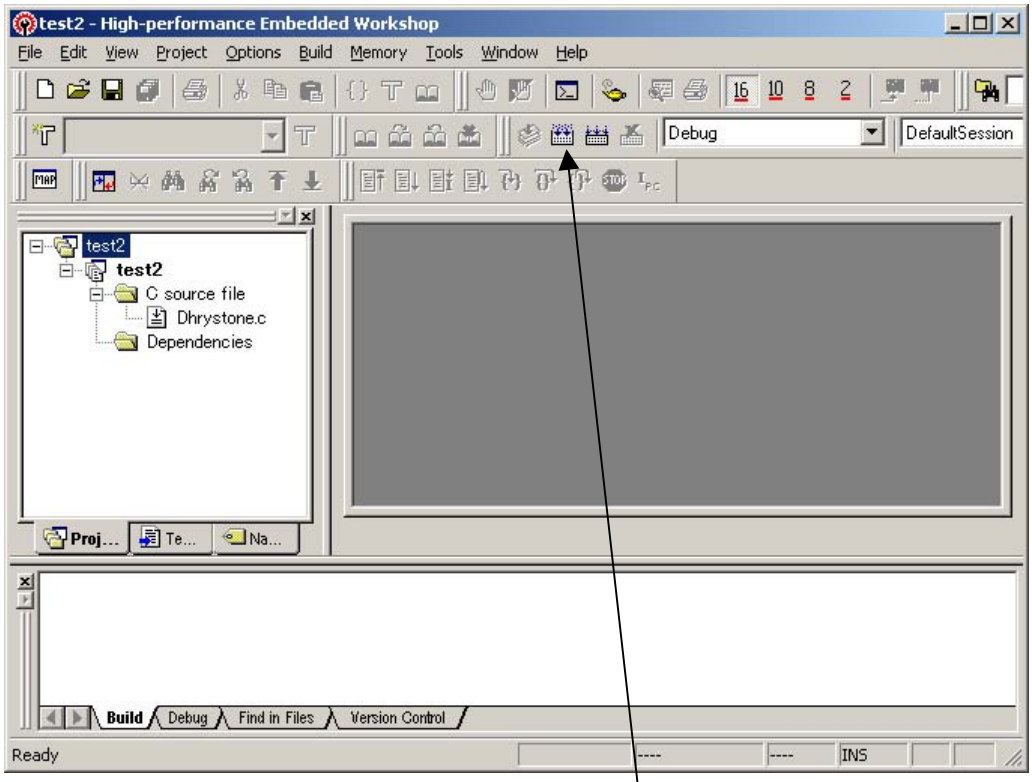


在下一个步骤中，启动 HEW。

选取**浏览至另一个工程工作空间 (Browse to another project workspace)**，单击 [确定 (OK)] 按钮，并指定已被转换的 HEW 工程。



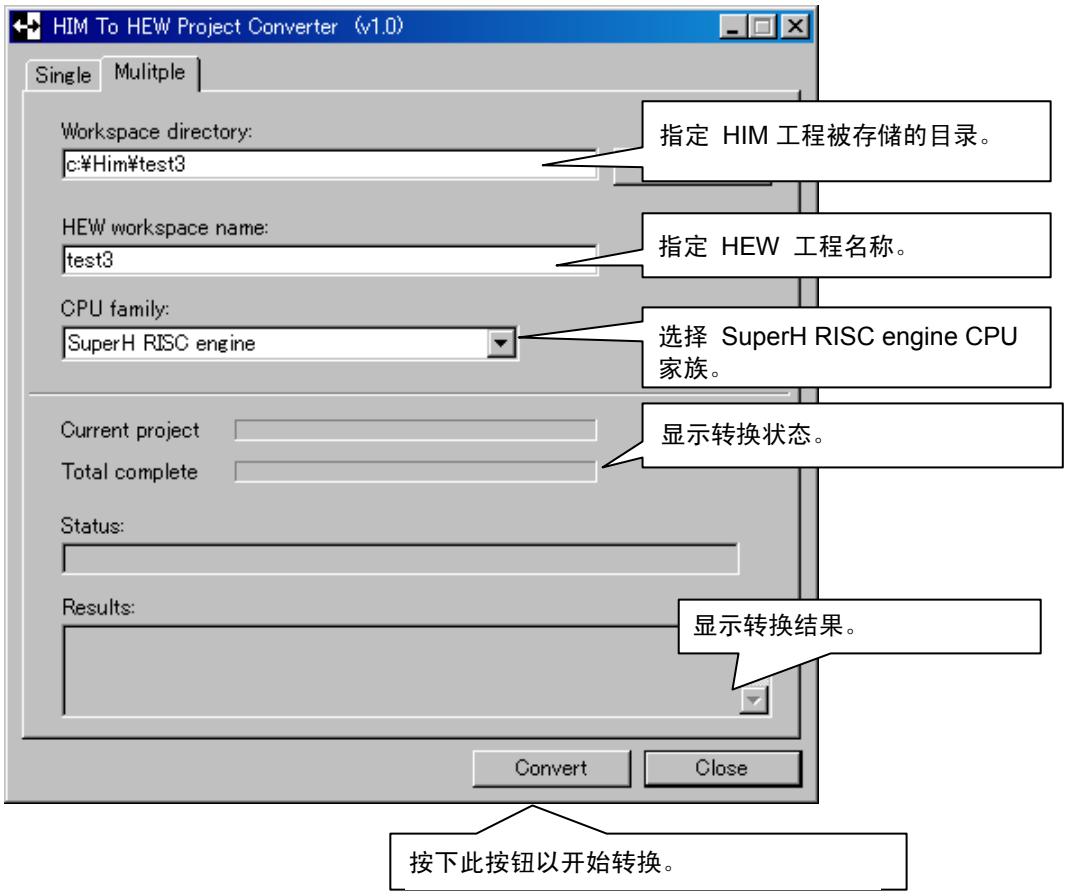
HEW 工程将如以下所示般打开：



指定 [创建 (Build) → 创建 (Build)] 以执行创建过程。 在命令菜单上，单击这里。

(2) 多个 (Multiple) 标签

此标签将多个 HIM 工程转换为 HEW 工程。



在转换后，请如单一标签的情形般启动 HEW，以创建已转换的 HEW 工作空间。

7.1.9 添加支持的 CPU

♦ 描述：

HEW 可以自动生成 I/O 寄存器定义和向量表文件，但 HEW 不能支持发布 HEW 后发布的新 CPU。

在此情形下，**设备更新程序 (DeviceUpdater)** 工具将会使 HEW 支持新的 CPU。

此外，此工具也可以更新生成的文件以进行版本错误修复。

♦ 如何获取**设备更新程序 (DeviceUpdater)**

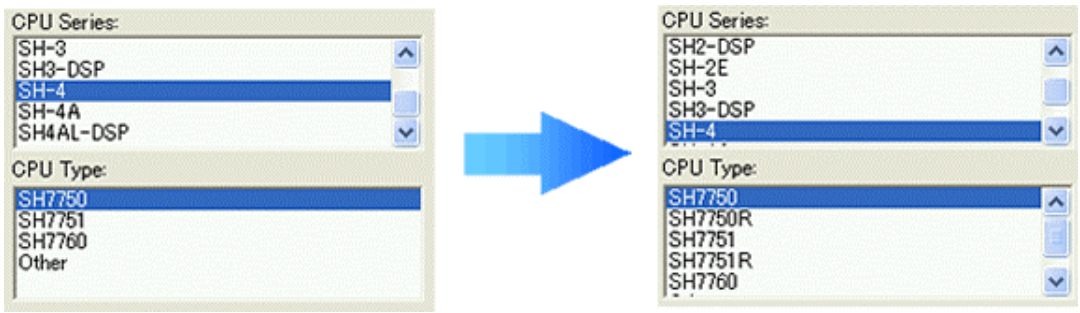
从 Renesas Technology Corp 的下列 URL 下载。

请参考此页的“注意”。

http://www.renesas.com/eng/products/mpumcu/tool/crosstool/support_tool/device_updater.html

♦ **设备更新程序 (DeviceUpdater)** 的执行结果

CPU 类型将会如下所示添加。



♦ 注意

此功能受 HEW 2.2 或以上版本支持。

7.2 模拟

7.2.1 伪中断

♦ 描述：

当单击模拟特定中断导因的伪中断按钮时，将造成人为的伪中断。

可以为每个按钮指定一项中断优先级和中断条件。

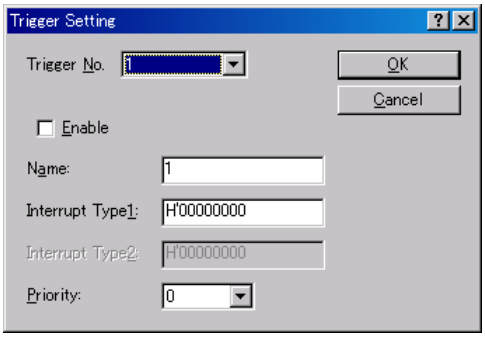
♦ 使用：

1. 当您选择 [视图 (View) -> CPU -> 触发器 (Trigger)] 时，将显示下列视图：

2. 在此视图上单击鼠标右键然后选择 [设定… (Setting…)]。 [触发器设定 (Trigger Setting)] 对话框将会显示。
若您选中 [允许 (Enable)] 复选框，由 1 号触发器所标识的中断将被允许。

另外，再指定中断名称、中断优先级，和中断条件（向量号）。

由 1 号触发器所标识的中断按钮已经启动。



3. 设定至此完成。 当单击以上步骤所设定的其中一个按钮时，程序将如适当的向量表所指定般停止。

♦ 注意：

此功能受 HEW 2.1 或以上版本支持。

7.2.2 方便断点函数

♦ 描述：

HEW 的断点设施包含下列便利功能，不单只在普通中断发生时，也可在中断条件被满足时启动。

文件输入

文件输出

中断

♦ 如何显示断点视图：

HEW 2.2 或以下版本： 选择 [视图 (View) -> 代码 (Code) -> 断点 (Breakpoints)]

HEW 3.0 或以上版本： 选择 [视图 (View) -> 代码 (Code) -> 事件点 (Eventpoints)]

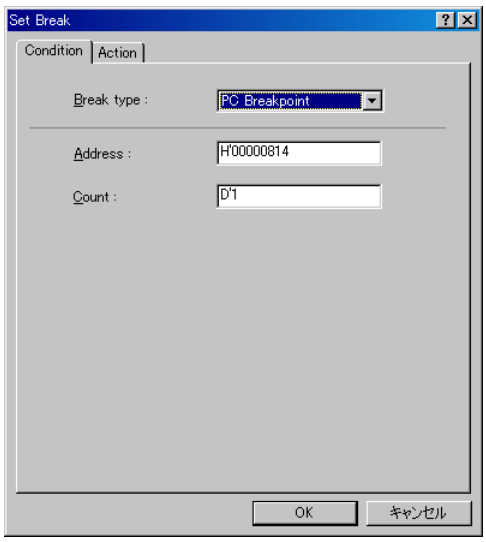
注意： 在 HEW 3.0 或以上版本内，前往 [断点 (Breakpoints)] 视图，然后单击 [软件事件 (Software Event)] 标签。

♦ 文件输入的设定实例：

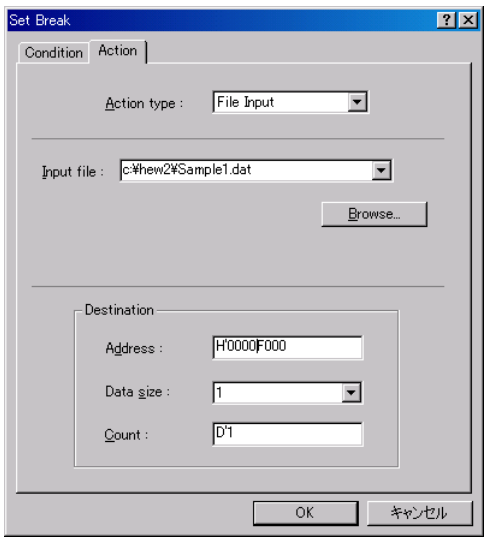
在 [断点 (Breakpoints)] 视图上右击，然后选择 [设定 (Setting) …]，以打开下列 [设定中断 (Set Break)] 对话框。如下所示，使用 PC 断点，以使中断条件在 PC 到达下列地址时获得满足。 其他断点类型的设定方法大致相同。

单击 [操作 (Action)] 标签，在 [操作类型 (Action type)] 字段中选取 [文件输入 (File Input)]，指定输入文件名称、输入地址，及其他项目，然后单击 [确定 (OK)] 按钮。

（[条件 (Condition)] 标签）



（[动作 (Action)] 标签）



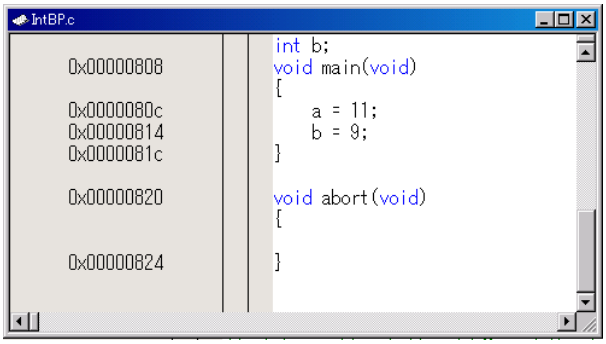
♦ 文件输入的操作实例：

让我们看看下列实际的操作实例：

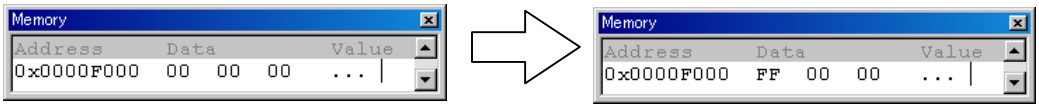
由于以上设定的结果，断点是在 [H’00000814]，同时输入文件包含 [H’FF]。

使用 Go 命令或类似的方法来运行程序。

（源代码片段）



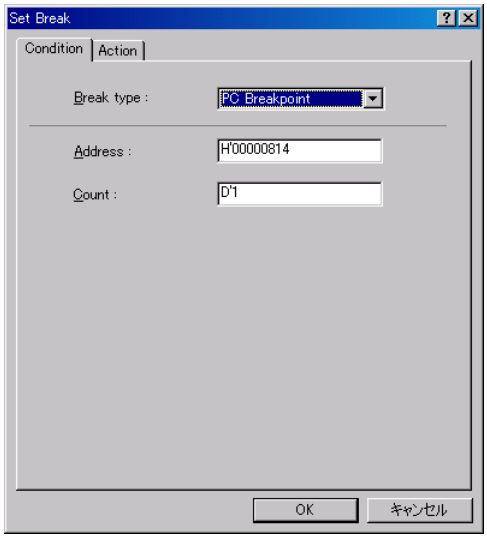
您将看到，当 PC 到达 [H’00000814] 时，由于中断条件被满足，因此地址 H’F000 的存储内容更改。



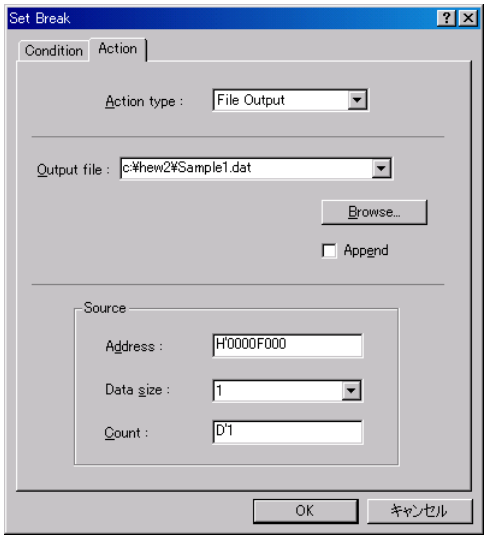
♦ **文件输出**的设定实例：

在 [设定中断 (Set Break)] 对话框中设定文件输出的方法，与设定文件输入的方法雷同。 文件输出断点也使用 PC 断点，以使 PC 在到达下列地址时，满足中断条件。 单击 [操作 (Action)] 标签，在 [操作类型 (Action type)] 字段中选取 [文件输出 (File Output)]，指定输出文件名称、输出地址，及其他项目，然后单击 [确定 (OK)] 按钮。

（[条件 (Condition)] 标签）



（[动作 (Action)] 标签）



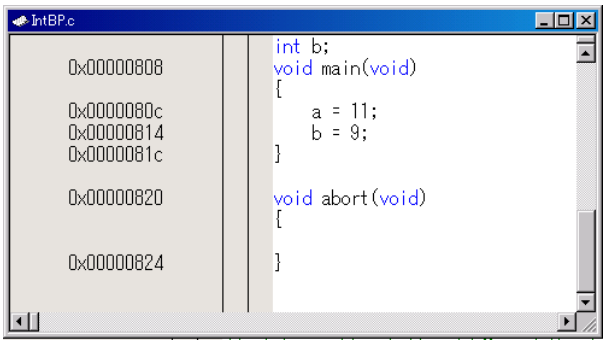
♦ **文件输出**的操作实例：

让我们看看下列实际的操作实例：

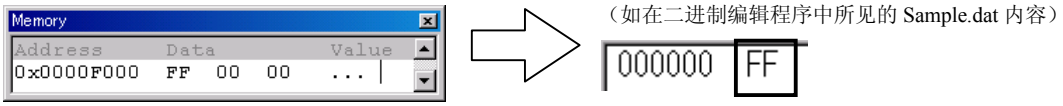
由于以上设定的结果，断点是在 [H'00000814]，同时地址 H'F000 的内容是 [H'FF]。

使用 Go 命令或类似的方法来运行程序。

（源代码片段）



您将看到，当 PC 到达 [H'00000814] 时，由于中断条件被满足，因此地址 H'F000 的内容被输出到文件。



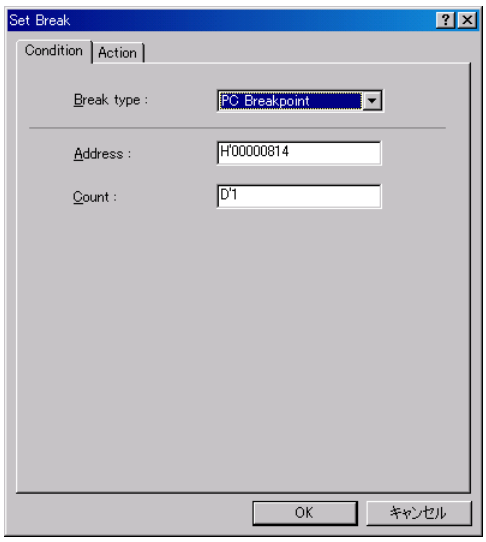
（如在二进制编辑程序中所见的 Sample.dat 内容）

♦ **中断**的设定实例：

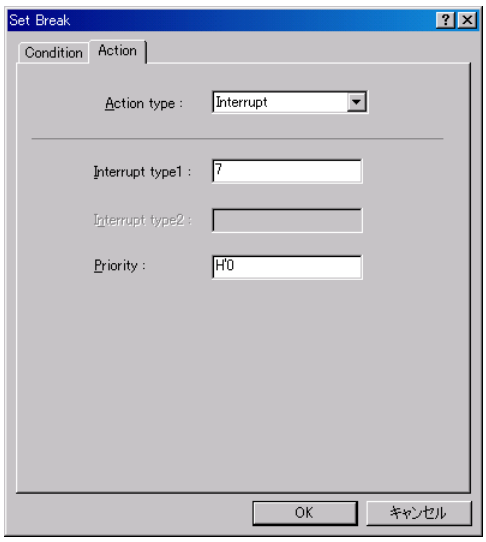
在 [设定中断 (Set Break)] 对话框中设定文件输出的方法，与设定文件输入的方法雷同。 如下所示，使用 PC 断点，以使中断条件在 PC 到达下列地址时获得满足。其他断点类型的设定方法大致相同。

单击 [操作 (Action)] 标签，在 [操作类型 (Action type)] 字段中选取 [中断 (Interrupt)]，指定中断优先级，及中断类型（向量号 7），然后单击 [确定 (OK)] 按钮。

（[条件 (Condition)] 标签）



（[动作 (Action)] 标签）



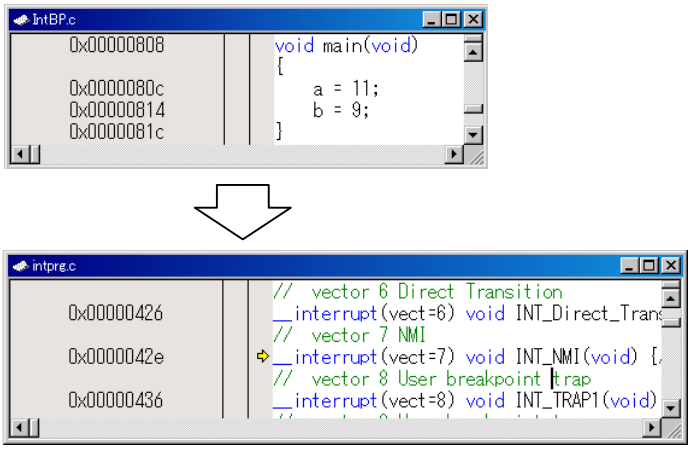
◆ **中断**的操作实例：

让我们看看下列实际的操作实例：

在断点因以上设定结果而被设定在 [H'00000814] 时，通过 Go 命令或类似方法运行程序。

您可以看到，当 PC 到达 [H'00000814] 时，将发生向量号 7 的非屏蔽中断 (NMI)。

（源代码片段）



7.2.3 覆盖功能

■ 描述：

HEW 允许用户在程序执行期间，在用户指定的地址范围内收集语句覆盖信息。 通过使用语句覆盖信息，您可以观察到每个语句被执行的方式。 另一方面，您可以轻易的识别未被执行的程序代码。

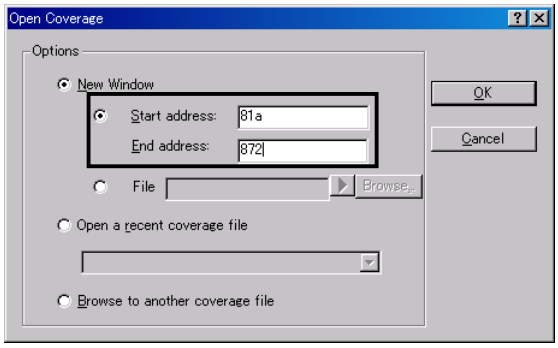
■ 如何开启 [打开覆盖 (Open Coverage)] 对话框：

[视图 (View) -> 代码 (Code) -> 覆盖 (Coverage) …]

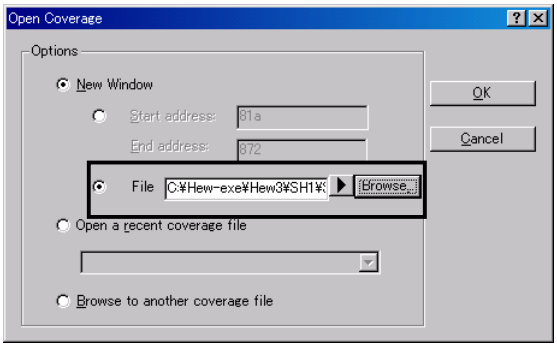
■ 如何收集新的覆盖信息：

1. 开启 [打开覆盖 (Open Coverage)] 对话框，选择 [新窗口 (New Window)]，然后输入起始和终止地址，以标识您要获取覆盖信息的范围。 若 HEW 的版本为 3.0 或以上，您可以指定 C 或 C++ 源文件名称，以标识您所要收集的信息。
   完成以上指定后，单击 [确定 (OK)] 按钮。

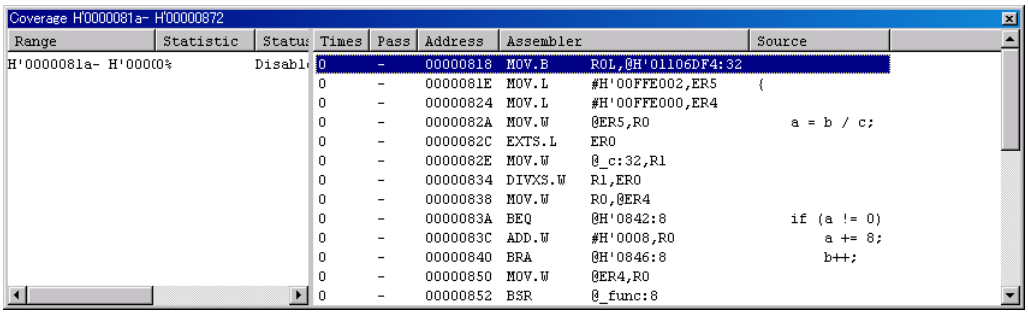
   （地址指定）



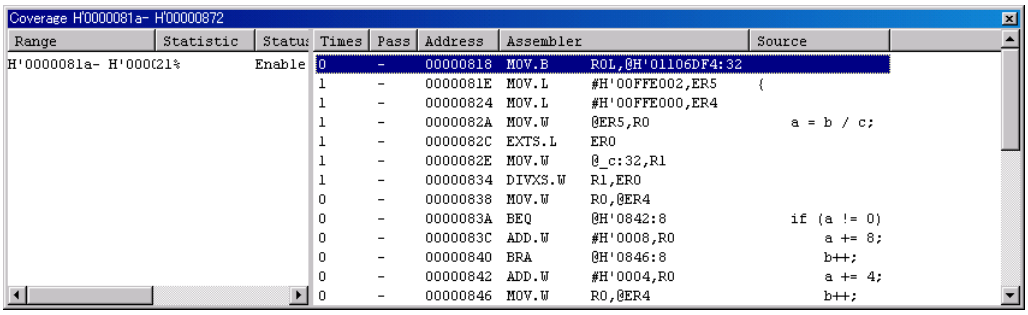
   （文件名指定） * 受 HEW 3.0 或以上版本支持



2. 在您单击 [确定(OK)] 按钮之后，将显示下列覆盖视图：
   在视图的右边，单击鼠标右键，并选择 [允许 (Enable)]。覆盖已被允许。

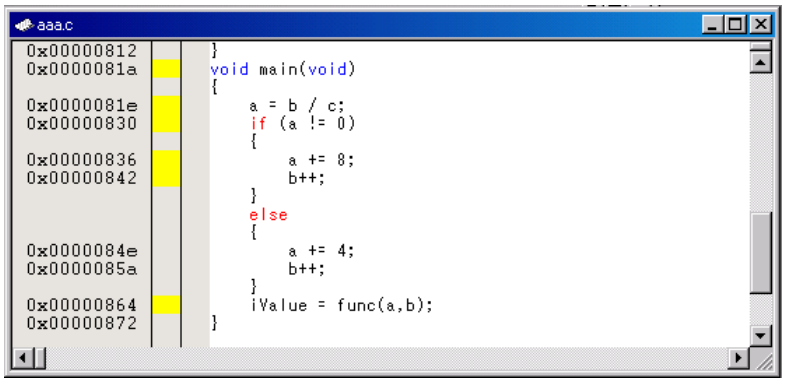


3. 让我们来运行程序。 在覆盖视图右边，留意到 [时间 (Times)] 列更改为 1 的行。这表示与此行对应的地址上的语句已被执行。
   在视图左边，显示了地址范围内的 C0 覆盖值。



注意： 覆盖的左视图仅存在于 HEW 3.0 或以上的版本。

4. 除了覆盖视图外，您可以使用其他方法来查看覆盖信息。 编辑程序画面左边的一个列中，显示程序执行有否传递特定源行。



♦ 保存数据：

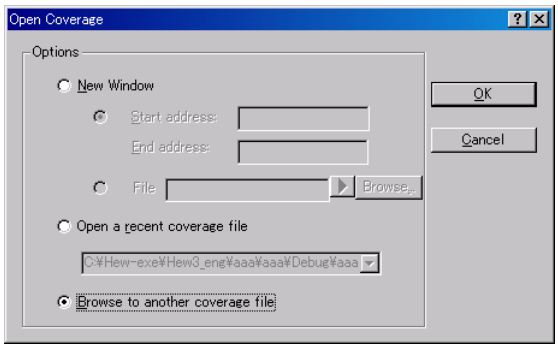
要保存覆盖信息，在覆盖视图右边单击鼠标右键，然后输入具备 *.cov 的扩展名的文件名称。

◆ 使用现有覆盖信息来收集信息：

您很少可以获取覆盖整个程序的单一覆盖信息集。

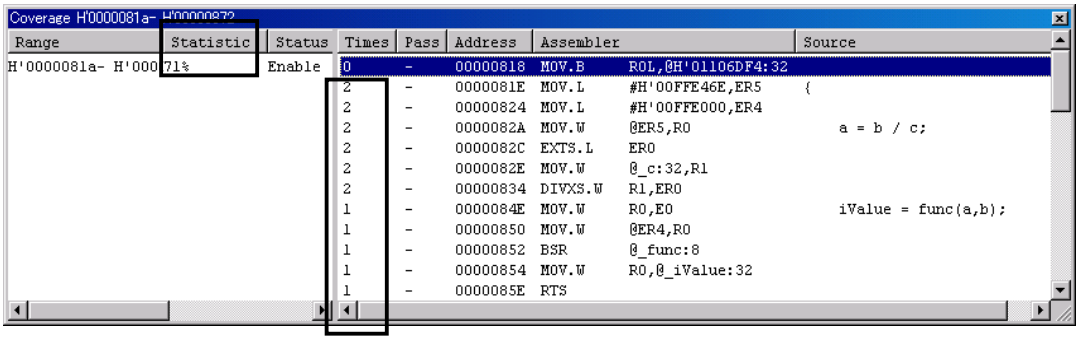
您可能要在重复收集覆盖的步骤时，增加覆盖的比率，并在不同的测试条件下执行。

为了这项目的，指定已保存在 [保存数据 (Save Data)] 中的一个文件，并在 [打开覆盖 (Open Coverage)] 对话框内选取 [打开最新的覆盖文件 (Open a recent coverage file)] 或 [浏览至其他覆盖文件 (Browse to another coverage file)]。 然后单击 [确定 (OK)] 按钮。



覆盖视图将打开。在新条件下再次运行程序。

如下所示，覆盖视图和编辑程序将显示反映当前运行的新信息，如运行的次数及新的 C0 覆盖值。



7.2.4 文件输入/输出

♦ 描述：

HEW 曾一度依赖 I/O 模拟功能，以模拟文件输入/输出的操作，而非实际执行文件输入/输出。

然而，现在 HEW 允许在替换下列文件后执行实际的文件输入或输出。

♦ 如何获取文件：

从下列 URL 下载文件：

http://www.renesas.com/jpn/products/mpumcu/tool/crosstool/support_tool/file_io.html

♦ 如何创建环境：

(1) 使用 HEW 创建工程。
将工程类型选取为 [应用程序 (Application)] 或 [演示 (Demonstration)]。
一些文件在创建的工程下被自动创建。
（若您将工程类型选取为 [应用程序 (Application)]，请在创建工程的步骤 3 选中 [使用 I/O 程序库 (Use I/O Library)] 复选框。
[I/O 流的数量 (Number of I/O Stream)] 字段中所指定的值，必须是实际处理的文件数 + 3（标准的 I/O 文件数）。
(2) 在已创建的文件中，替换“lowsrc.c”和“lowlvl.src”。（*1）
(3) 创建“C:\Hew2\stdio”目录。（*2）
(4) 执行再创建，以创建支持文件输入/输出的模拟程序/调试程序环境。

注意： 1. -lowsrc.c-

这些文件在 SH 和 H8 是公用的。

用包含在工程中的“lowsrc.c”文件将此文件替换。

-lowlvl.src-

此文件因 CPU 而异。

依据创建工程的 CPU，用文件夹内的“lowlvl.src”文件将此文件替换。

2. 在创建的环境中，当遇到文件 I/O 处理的程序代码时，标准 I/O 文件将实际打开，而不像一般执行惯例 — 模拟文件的打开。
因此，命名为“stdin”、“stdout”和“stderr”且会实际打开以进行标准 I/O 处理的文件，将会在首次执行程序时自动生成。
由于这些文件被定义为在“C:\Hew2\stdio”中创建，您必须创建项目（3）所述目录。若此目录不存在，HEW 将无法正常工作。
当运行模拟程序时，这些文件通过包含在工程中的“lowsrc.c”文件的 INIT_IOLIB() 打开。

stdin = 0
stdout = 1
stderr = 2

♦ 使用的实例：

如下列所示，考虑使用 printf 或类似方法以输出字符至标准输出 (stdout)：

```
(Sample program code)

void main(void)
{
        printf("***** ID-1 OK *****\n");
}
```

当您运行此程序时，它在您所创建的“c:\Hew2\stdio”目录中创建命名为 stdout 的文件。 文件的内容如下：

```
(Contents of stdout)

***** ID-1 OK *****
```

♦ 如何重定向 I/O：

要重定向 I/O，在 lowsrc.c 文件中的 _INIT_IOLIB 函数内更改它。

```
void _INIT_IOLIB(void)
{
FILE *fp;

    for( fp = _iob; fp < _iob + _nfiles; fp++ ) /* ファイル型デー
     {
        fp->_bufptr = NULL;                     /* バッファへのポ
        fp->_bufcnt = 0;                        /* バッファカウン
        fp->_buflen = 0;                        /* バッファ長
        fp->_bufbase = NULL;                    /* バッファへのベ
        fp->_ioflag1 = 0;                       /* I/Oフラグ
        fp->_ioflag2 = 0;                       /* I/Oフラグ
        fp->_iofd = 0;                          /* ファイル番号
     }

    // 標準入出力用ファイルをオープン。
    // "stdin"・"stdout"・"stderr"は、ファイルが存在しなくても自動生
    // "stdin"は、実際は"r"でオープンしなくてはならないが、
    // 自動生成するために"w"でオープンした後I/Oフラグに"r"を設定して
   if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdin",  "w", stdin )==NULL)   /
        stdin->_ioflag1 = 0xff;                                 /
    stdin->_ioflag1  = _IOREAD;                                 /
    stdin->_ioflag1 |= _IOUNBUF;                                /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdout", "w", stdout )==NULL) /
        stdout->_ioflag1 = 0xff;                                /
    stdout->_ioflag1 |= _IOUNBUF;                               /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stderr", "w", stderr )==NULL) /
        stderr->_ioflag1 = 0xff;                                /
    stderr->_ioflag1 |= _IOUNBUF;
}
```

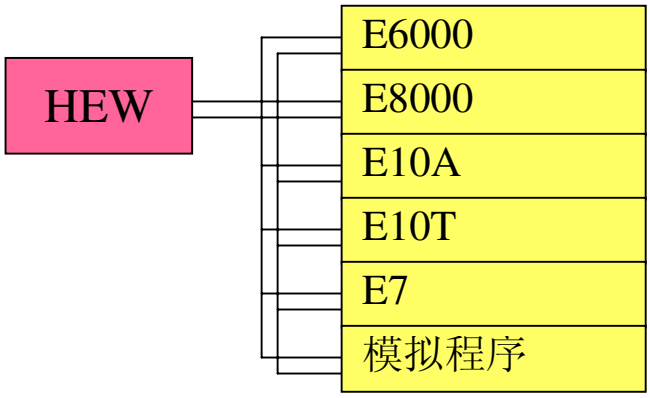
7.2.5 调试程序目标同步

♦ 描述：

HEW 允许您在单个 HEW 示例上，调试多个目标。

这意味着您可以同时调试多个目标，并使它们相互同步。

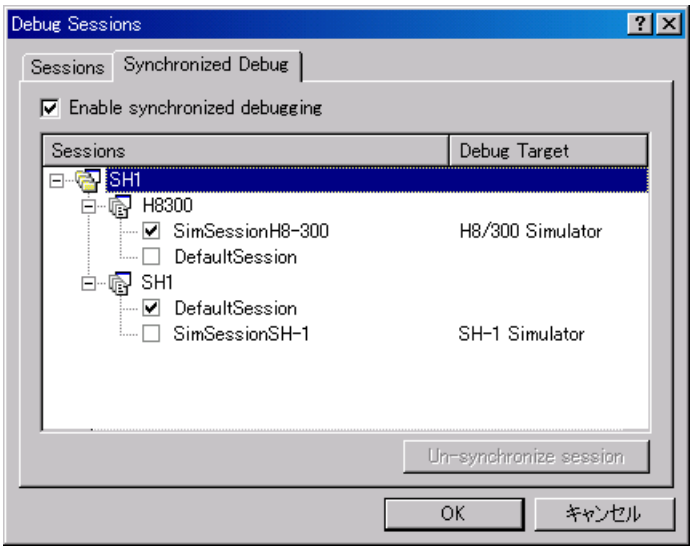
另外，您可以使会话中的一个事件（如一个步骤或 Go）与其他会话中的相同事件同步。



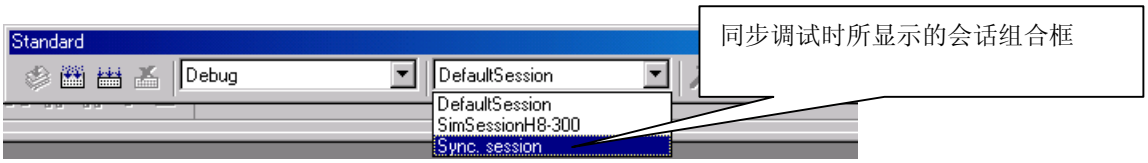
♦ 如何同步化调试程序目标：

1. 选择 [选项 (Options) -> 调试会话 (Debug sessions) …] 以打开下列对话框，然后单击 [同步化调试 (Synchronized Debug)] 标签。

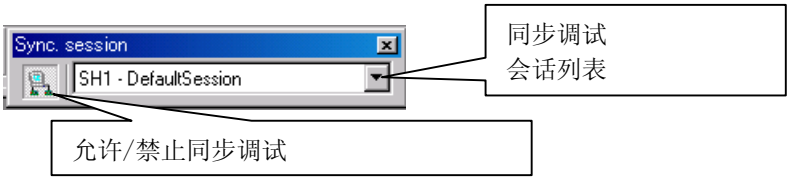
选中您要同步化的任何会话，然后选中 [允许同步化调试 (Enable synchronized debugging)] 复选框。



2. 在 [标准 (Standard)] 工具栏上，从会话组合框选取 [同步会话 (Sync. session)]。



3. [同步会话 (Sync. session)] 工具栏显示在工具栏中。 设定至此完成。



♦ 可用命令：

当允许了同步化调试时，您可以在同步化模式中执行下列操作：

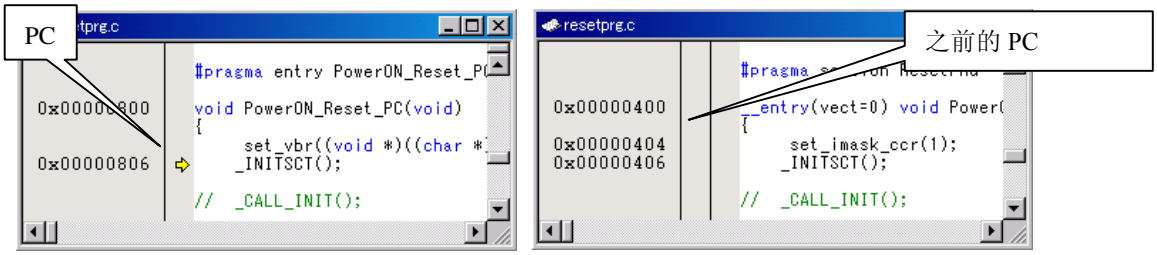
用户操作	目标调试程序会话 1	目标调试程序会话 2
在其中一个会话期间 [运行 (Run)]	“Run”	“Run”
在其中一个会话期间 [逐步执行 (Step)]	"Step"	"Step"
在其中一个会话期间按下 ESC	"Stop"	"Stop"
-	因为断点或用户程序错误而 “Stop”	Stop（和按下 ESC 时相同）
-	Stop（和按下 ESC 时相同）	因为断点或用户程序错误而 “Stop”
在其中一个会话期间 [复位 CPU (CPU reset)]	"CPU reset"	"CPU reset"

♦ 同步化调试实例

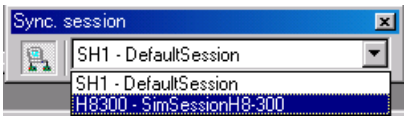
下面提供一个执行逐步命令的实例。

1. 在 [SH1 – SimSessionSH-1] 期间执行逐步。将形成下列条件：

SH – SimSessionSH-1 state　　H8300 - SimSessionH8-300 state

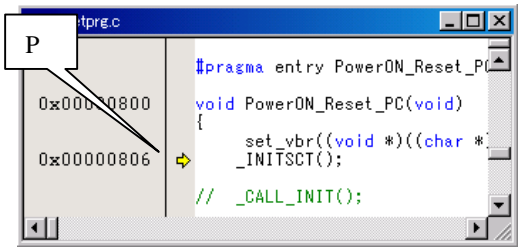


2. 使用 [同步会话 (Sync. session)] 工具栏更改会话。

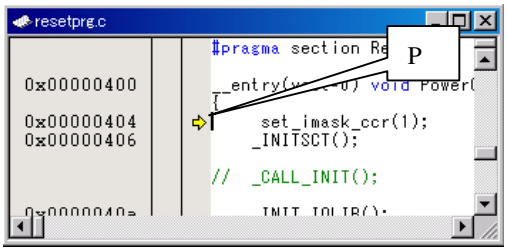


3. 如下所示，您可以看到 PC 也在 [H8300 – SimSessionH8-300] 会话期间移动到下一行。

SH – SimSessionSH-1 state



H8300 - SimSessionH8-300 state



♦ 注意：

此功能受 HEW 3.0 或以上版本支持。

7.2.6 如何使用定时器

♦ 描述：

HEW 支持定时器和中断的优先顺序设定。

对于每个定时器，仅支持通道 0。

HEW 对溢出、下溢和比较匹配中断的支持有限。 HEW 不支持涉及终端 I/O 的中断，如输入捕捉中断。

♦ 每个 CPU 中受支持的定时器控制寄存器

下表中“受支持的”列，○ 标示该寄存器受支持，而 Δ 标示只有与 [描述] 下的段落中所描述之功能关联的位受支持。

调试平台名称	定时器名称	支持的控制寄存器	受支持的
SH-1	ITU0	TSTR	Δ
		TCR	Δ
		TIER	○
		TSR	○
		TCNT	○
		GRA	○
		GRB	○
SH-2/SH-2E/ SH2-DSP (SH7065)	CMT0	CMSTR	○
		CMCSR	○
		CMCNT	○
		CMCOR	Δ
SH-3/SH3-DSP/ SH3-DSP（核心） SH-4/SH-4BSC/ SH-4(SH7750R)	TMU0	TCR	Δ
		TCNT	○
		TSTR	○
		TCOR	○
SH2-DSP（核心）	FRT0	TIER	Δ
		FTCSR	Δ
		FRC	○
		OCRA	○
		OCRB	○
		TCR	Δ
		TOCR	Δ

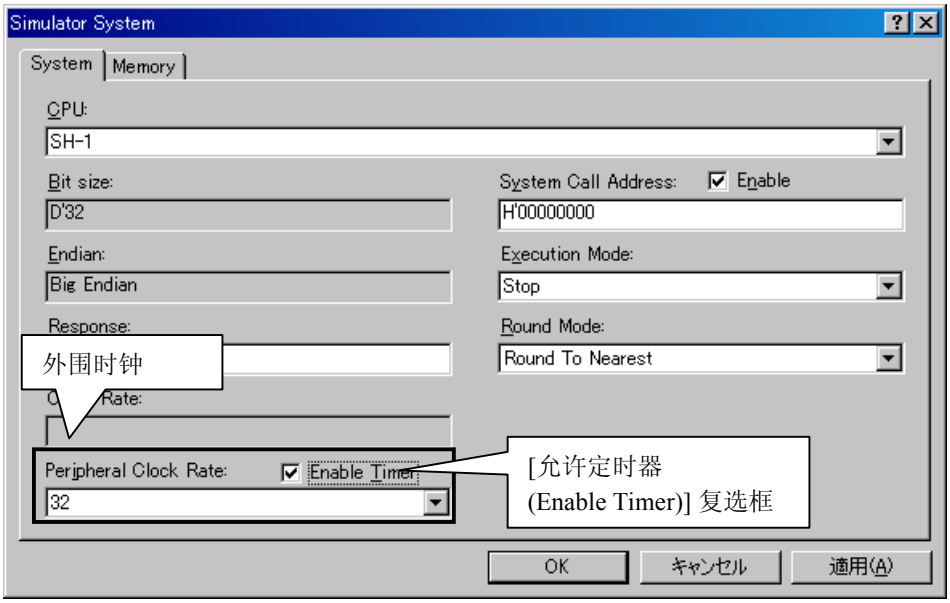
♦ 每个 CPU 中受支持的中断优先级设定寄存器

下表中“受支持的”列，○ 标示该寄存器受支持，而 Δ 标示只有与 [描述] 下的段落中所描述之功能关联的位受支持。

调试平台名称	支持的控制寄存器	受支持的
SH-1	IPRC	Δ
SH-2	IPRG	Δ
SH-2E	IPRJ	Δ
SH2-DSP (SH7065)	IPRL	Δ
SH-3/SH3-DSP/ SH3-DSP（核心）/ SH-4/SH-4BSC/ SH-4(SH7750R)	IPRA	Δ
SH2-DSP（核心）	INTPRI0B	Δ

♦ 定时器模拟方法

选择 [选项 (Options) -> 模拟程序 (Simulator) -> 系统 (System)...] 以打开下列 [模拟程序系统 (Simulator System)] 对话框，选中 [允许定时器 (Enable Timer)] 复选框，然后指定外部时钟和外围模块时钟之间的比率。



另外，您可以使用定时器控制寄存器和写入程序代码以允许它们，如下所示。

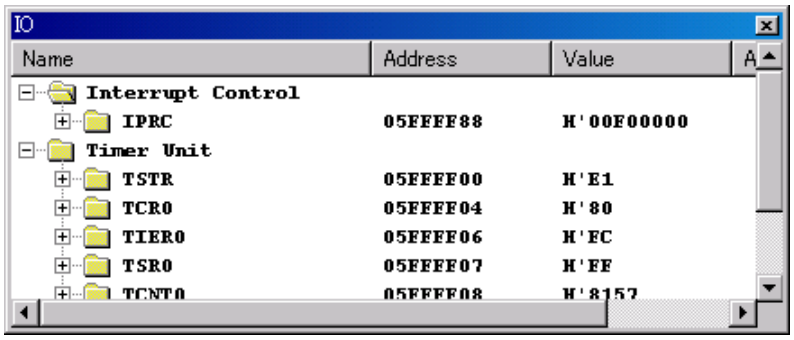
如果您创建通过外围模块驱动定时器的时钟，请使用适当的定时器控制寄存器来指定分频比。

```
// ITU0 start
P_ITU.TSTR.BIT.STR0 = 1;
// ITU0 OverFlow interrupt enable
P_ITU0.TIER.BIT.OVIE = 1;
while(1);
```

允许定时器 ITU0。

♦ 如何查看定时器寄存器设定：

要查看定时器寄存器和中断优先级设定寄存器上的设定，请选择 [视图 (View) -> CPU -> I/O] 以打开下列 I/O 窗口。



♦ 注意：

此功能受 HEW 3.0 或以上版本支持。

7.2.7 定时器的使用实例

♦ 描述：

本小节使用 SH7034 (SH-1) 中的 ITU 作为例子，概述如何使用比较匹配和循环控制程序中断。

♦ HEW 设置：

参考第 7.2.6 小节“如何使用定时器”中名为“定时器模拟方法”的段落，允许定时器。

♦ 样品程序包含可增加比较匹配中断的代码：

下列样品程序包含可增加比较匹配中断的代码。

在比较匹配中断出现之前，由 IPRC（中断优先级寄存器）指定的中断优先级，必须等于或高于由中断屏蔽位在 SR（状态寄存器）中指定的值。

[设定 SR 中断屏蔽位]
使用包含下列复位例程的文件，将 SR 中的位 4-7 设定为 0 至 15 的其中一个值。

resetprg.c

```
#define SR_Init     0x00000000
#define INT_OFFSET 0x10
```

屏蔽位 4-7（0 至 15）

[中断生成程序的说明]

SH1.c

```
#include "iodefine.h"
void main(void)
{
    P_ITU0.TCR.BYTE = 0xA1;              /* TCR  is B'*0100001     */
    P_ITU0.TIOR.BYTE = 0x88;             /* TIOR is B'*000*000     */
    P_ITU0.TIER.BYTE = 0xFF;             /* TIER is B'*****111     */

    P_INTC.IPRC.BIT.LU = 0x01;           /* INT priority = 1       */
    P_ITU0.GRA = 19999;                  /* GRA Value = 19,999     */

    P_ITU.TSTR.BIT.STR0 = 1;             /* ITU0 Start             */

    while(1)
    {
        while( !P_ITU0.TSR.BIT.IMFA ) ;  /* Wait IMFA = B'1        */

        P_ITU0.TSR.BIT.IMFA = 0;         /* Clear IMFA             */
    }
}
```

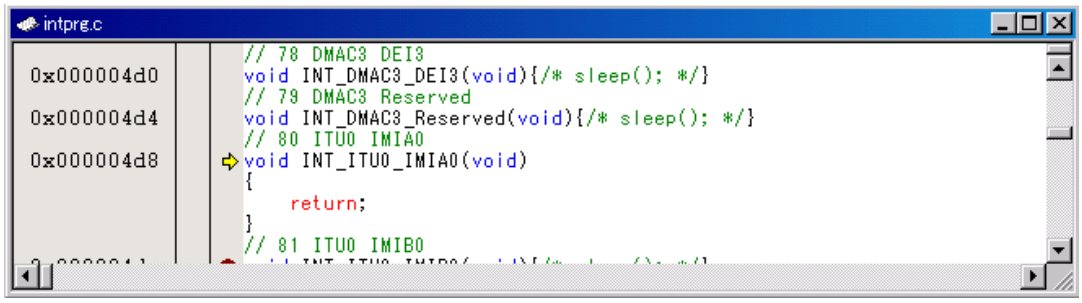
1
2
3
4

1. 当 TIER（定时器中断允许寄存器）中的 IMFA（比较匹配标志 A）位变成 1 时，中断将会被允许。
2. 设定 IMFA 中的中断优先级。
3. 启动 ITU0 定时器。
4. 等待直到 IMFA 位变成 1。（等待比较匹配）

♦ 程序执行：

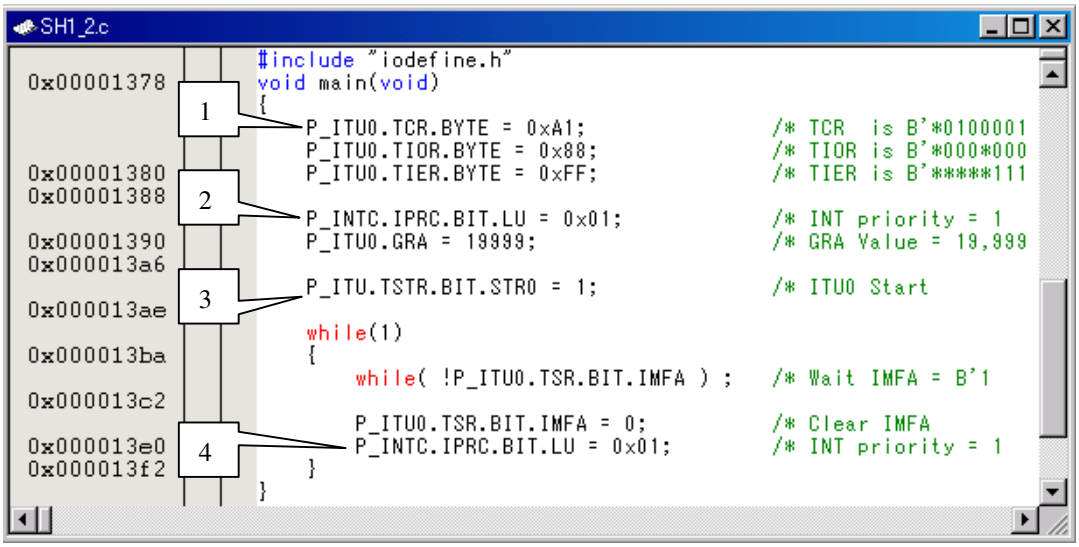
等待直到 TCNT0（定时器计数器 0）和 GRA（常规寄存器 A）如名为“中断生成程序的说明”段落中步骤 4 的匹配（比较匹配出现）。
两者匹配时，比较匹配中断将会出现，并且具备下列中断例程的调用结果：

如需详细信息，请参考有关的硬件手册。



♦ 包含用于循环处理程序的代码之样品程序

下列样品程序包含用于循环处理程序的代码。
出现比较匹配时，程序将会清除定时器，然后将控制转移到中断处理程序。
中断完成执行后，程序将会降低 IPRC（中断优先级寄存器） 中的中断优先级。
接着，控制将会回到导致中断的代码。 程序会提高中断优先级以确保 IMFA 位可以设定。
有关设定 SR 中断屏蔽位的信息，请参考比较匹配样品程序。



1. 设定 TCR（定时器控制寄存器）以确保定时器计数器 (TCNT) 可以在 IMFA（比较匹配标志 A）位变成 1 时被清除。
2. 设定 IMFA 中的中断优先级。
3. 启动 ITU0 定时器。
4. 出现比较匹配后，中断优先级将会被提高。

♦ 程序执行：

样品程序将会等待直到比较匹配出现。 出现比较匹配时，程序会将控制传递给下列中断例程。

中断例程会执行中断，降低 IMFA 中的中断优先级，然后将控制传回给程序。

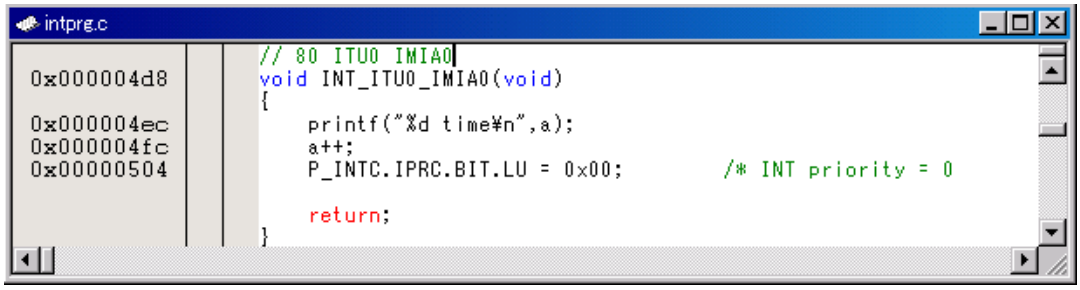
中断处理可以在这种方式中完成。

接着，程序将可以准备好接受下一个比较匹配中断。

如需详细信息，请参考有关的硬件手册。

根据 HEW 规格，出现中断时，计算机将会在导致中断之函数的起始处停止。

模拟循环处理程序时，您需要使用 Go 命令或类似方法，在每一个循环前进计算机。



7.2.8 重新配置调试程序目标

♦ 描述：

如果您在创建新工作空间时，将工程类型选取为应用程序 (Application)，HEW 将可以配置“调试程序目标”。

然而，在创建新的工程时，您可能偶尔不会进行此配置，因为您在当时认为这是不必要的。

若的确如此，您可以在创建工程后使用此功能再配置“调试程序目标”。

然而，此功能仅在您创建新工作空间时，将工程类型选取为应用程序 (Application) 之后可用。

♦ 使用：

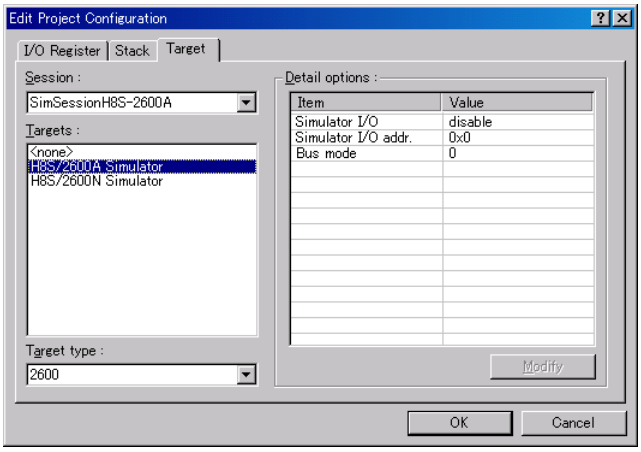
HEW 菜单： **工程 (Project) > 编辑工程配置... (Edit Project Configuration...)**

♦ 可被再配置的函数包括：

[设定方法]

您可以在 [编辑工程配置 (Edit Project Configuration)] 对话框内的 [目标 (Target)] 标签上，设定模拟程序和其他调试程序目标。

若会话已和调试程序连接，您将看到显示“此目标已存在。它无法支持复制目标 (This target has already existed. It does not support duplicated targets)”的信息，同时无法连接到调试程序目标。



♦ 注意：

再配置文件受 HEW 2.1 或以上版本支持。

7.3 Call Walker

♦ 描述：

Call Walker 会读取由优化连接编辑程序输出的堆栈信息文件 (*.sni)，或由模拟程序调试程序输出的配置文件信息文件 (*.pro) 。 Call Walker 也会显示静态使用之堆栈的大小。

虽然汇编语言程序使用之堆栈的大小不能输出至堆栈信息文件，您可以使用编辑功能来添加信息然后获取整个系统中所使用的堆栈之大小。

编辑关于所使用的堆栈之大小信息后，您可以将修改的信息保存在调用信息文件 (*.cal) 中，或从该文件读取修改的信息。

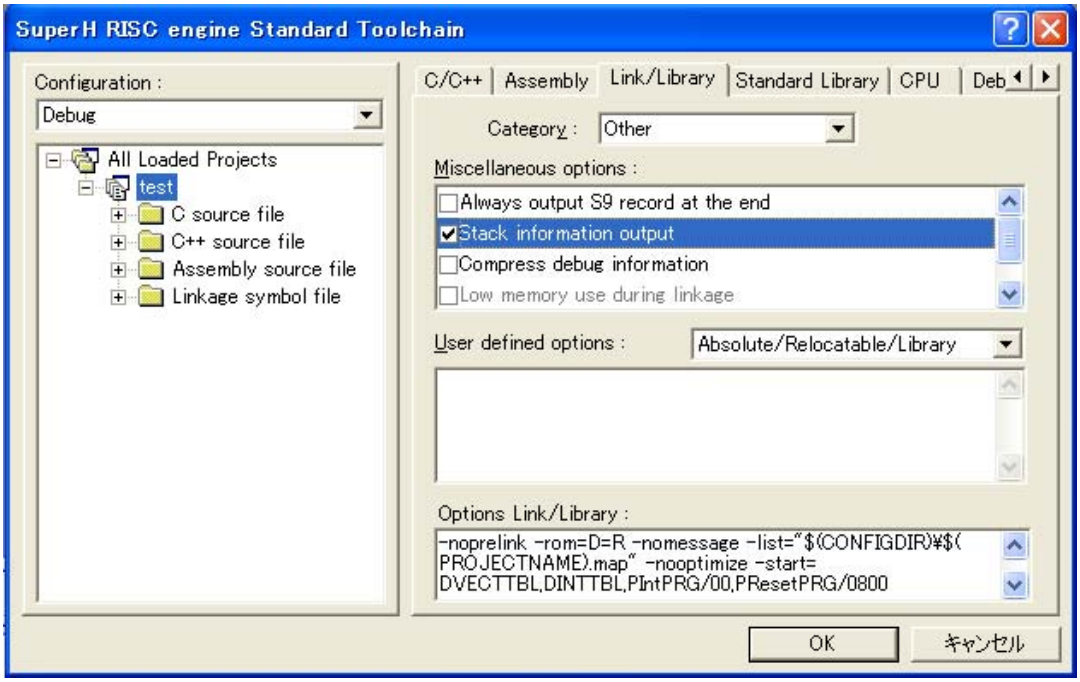
您也可以合并多个调用信息文件。

7.3.1 创建堆栈信息文件

遵循以下步骤创建堆栈信息文件或配置文件信息文件。

- 如何创建堆栈信息文件 （*. sni）

要创建堆栈信息文件，请在 [连接/程序库 (Link/Library)] 页中选择下列选项。



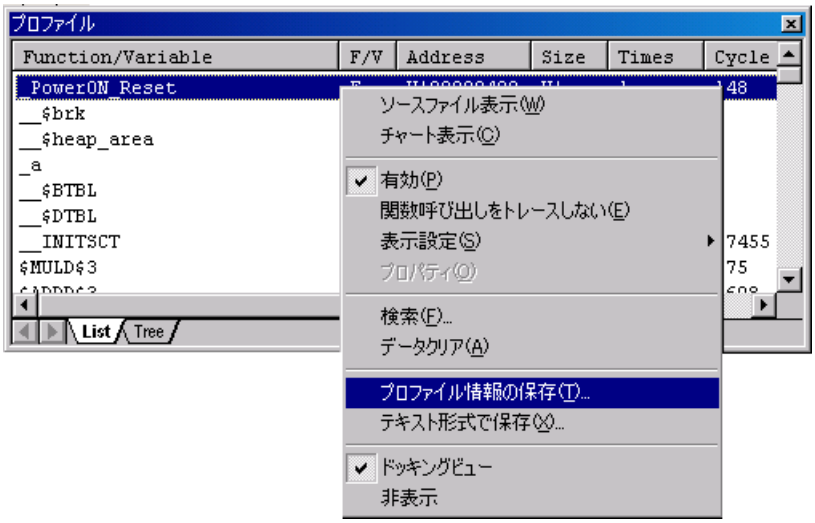
在此对话框中： 选择 [连接/程序库 (Link/Library)] 标签。 然后在 [类别 (Category)] 文本框中选择 [其他 (Other)] 并从 [杂项 (Miscellaneous options)] 列表中选择 [堆栈信息输出 (Stack information output)]。

命令行： STACk

♦ 如何创建配置文件信息文件 (*.pro)

使用配置文件功能执行所需的用户程序。
完成执行用户程序后，在 [配置文件 (Profile)] 窗口上单击鼠标右键以保存配置文件信息和创建配置文件信息文件 (*.pro)。有关如何创建配置文件的详细信息，请参考“模拟程序调试程序的高性能嵌入式工作区 3 用户指南”(High-performance Embedded Workshop 3 User's Manual for the Simulator Debugger) 中的第 4.14 节“查看配置文件信息”。

[配置文件 (Profile) 窗口]
[查看 (View)]->[性能 (Performance)]->[配置文件 (Profile)]



7.3.2 启动 Call Walker

您可以使用两种方式启动 Call Walker

- 从 [开始 (Start)] 菜单

选择 [程序 (Program)]->[瑞萨高性能嵌入式工作区 (Renesas High-performance Embedded Workshop)]->[Call Walker]。

- 从 HEW

选择 [工具 (Tools)]->[Call Walker]。

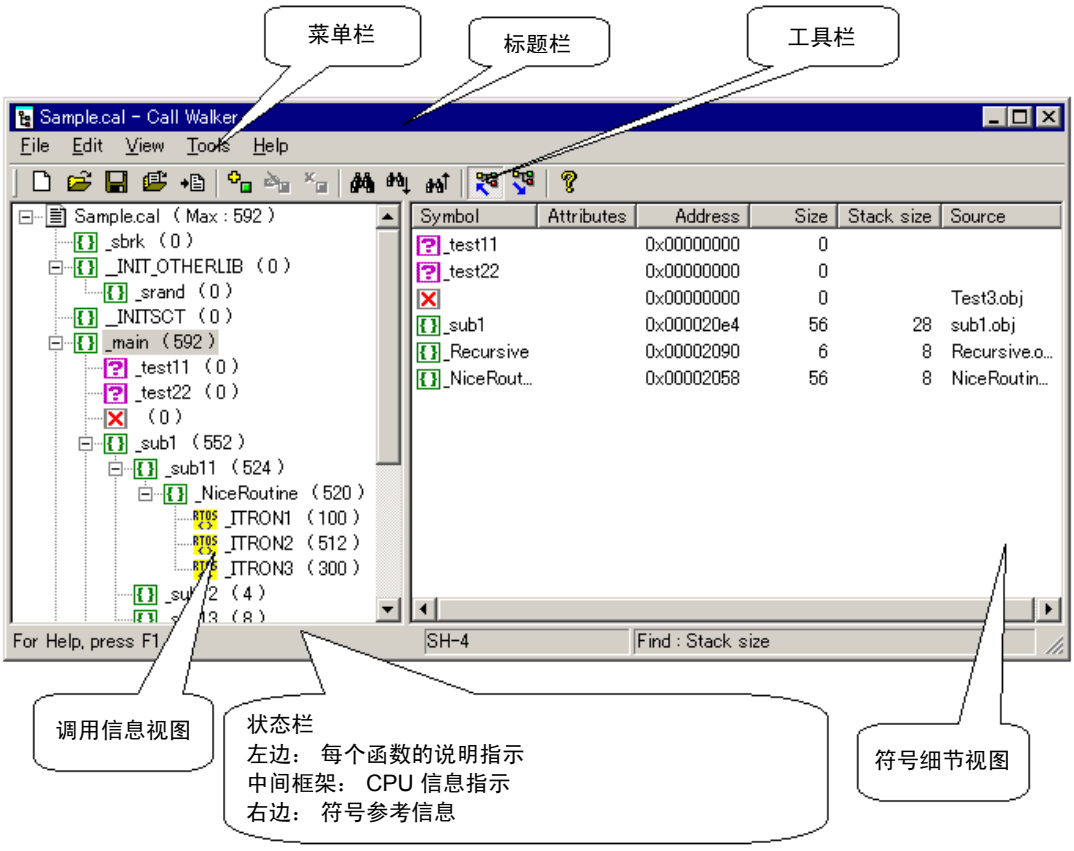
7.3.3 Call Walker 窗口和打开文件

启动 Call Walker 时，您可以通过选择 **[文件 (File)]->[导入堆栈文件 (Import Stack File)...]** 来打开所需的堆栈信息文件 (*.sni) 或配置文件信息文件 (*.pro)。

您也可以选择 **[文件 (File)]->[打开 (Open)...]** 来打开现有的已编辑文件 (*.cal)。

打开文件时，以下窗口将会显示。

注意： 对于不在标准程序库中的汇编程序函数，堆栈大小将显示为 0。请参考第 7.3.4 节“编辑堆栈信息”然后设定适当的堆栈大小。



♦ 调用信息视图

此视图显示符号的连接分层。
每个符号名称右边的数字表示所需的堆栈大小。

(1) 关于符号的详细信息
每个符号名称左边的图标表示符号的类型。
提供的类型包括：

File being edited

Assembler

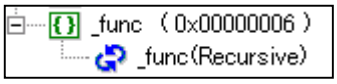
C/C++ function

直接或间接递归函数

(a) 直接递归函数
此图标表示所标示的函数将会直接调用自身。

[实例]

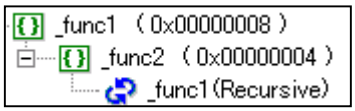
```
void func(int x)
{
    x++;
    if(x != OFF)
        func(x);

    if(x == MAX)
        return;
}
```



(b) 间接递归函数
此图标也表示所标示的函数将会间接调用自身。

[实例]

```
void func1(int a)
{
    func2(10);
}
void func2(int b)
{
    func1(9);
}
```



RTOS 函数（实时操作系统的函数，如 ITRON)

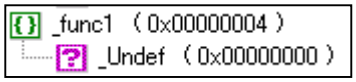
未知参考源函数

在下例中，func1() 函数调用 Undef() 函数。但是，
如果 Undef() 函数真的不存在，此图标将为 Undef() 函数显示。

调用不存在的函数将会导致连接错误。但是，通过使用 change_
message 连接选项，您可以将错误信息改为警告信息。
即使存在警告信息，您仍然可以创建加载模块。因此，您也可以创建
堆栈信息文件。

［实例］

```
void func1(void)
{
    Undef();
}
```

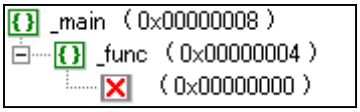


具备未分解地址的函数

此图标会在所标示的函数从下表调用时显示。

[实例]

```
static int (*key[3])()=
                {nop, stop, play};
void func(int x)
{
    (*key[a])();
}
```

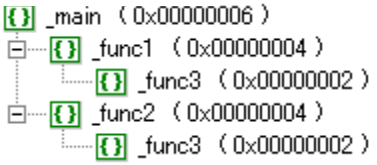


缩短图标

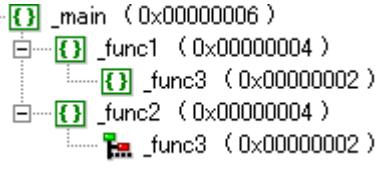
此工具显示所有的连接级。 如果用户应用程序很大，将显示的连接级
也很庞大。
因此，只有第一个符号会显示，其他的相同符号则使用缩短图标来简短显示。
要显示所有符号，请选择 [视图 (View)]->[显示全部符号 (Show All Symbols)]。
要显示部分符号，请选择 [视图 (View)]->[显示简单符号 (Show Simple Symbols)]。

[实例]

显示全部符号



显示简单符号

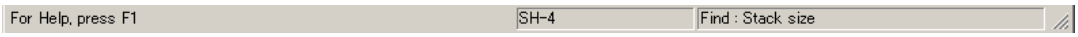


♦ 详细的符号视图

Symbol	Attri...	Address	Size	Stack size	Source
_INT_TXI1_...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_abort		0x000008...	0x00000002	0x00000004	CallWalker2....
_sbrk		0x000008...	0x0000002c	0x00000008	sbrk.obj
_sub		0x000008...	0x00000002	0x00000004	CallWalker2....
_nop		0x000008...	0x00000002	0x00000004	CallWalker2....
PowerON...		0x000004...	0x00000016	0x00000004	resetprg.obj
_play		0x000008...	0x00000002	0x00000004	CallWalker2....
_stop		0x000008...	0x00000002	0x00000004	CallWalker2....
_INT_TGI1...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0...	I	0x000004...	0x00000002	0x00000004	intprg.obj

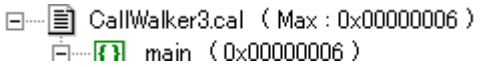
此视图显示关于每个符号的地址、属性、堆栈大小和其他详细信息。
单击一个符号然后单击鼠标右键可执行编辑命令。

♦ 状态栏



状态栏显示 CPU 类型以及关于当前打开的堆栈信息文件（在创建时）的其他信息。

♦ 最大堆栈大小



“最大(Max)”表示当前打开的堆栈信息文件中静态使用之堆栈的最大大小。

♦ 选取标准程序库版本

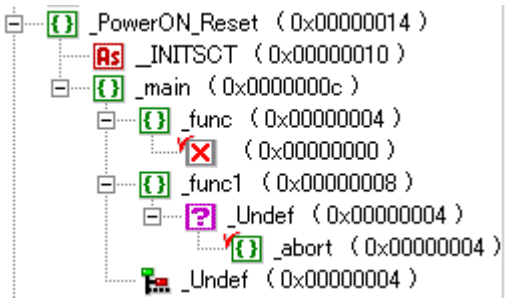
选择在您创建当前打开的堆栈信息文件时所使用的标准程序库版本。
标准程序库中汇编程序函数所使用的堆栈大小由标准程序库的版本决定。
如果您只安装一个 HEW 封装，您将不需要选择任何版本。

7.3.4 编辑堆栈信息

打开文件时，您可以从右边的详细符号视图选择所需的符号名称，然后使用编辑 (Edit) 菜单中的添加 (Add) …、修改 (Modify) …，或删除 (Delete) … 命令来添加、更改或删除符号。
您也可以在详细的符号视图中单击鼠标右键来执行相同的操作。

虽然此工具会计算静态使用之堆栈的最大大小，但因为多个中断和其他原因，用户仍需要编辑信息文件以确定动态使用之堆栈的最大大小。

您可以通过拖放左边调用信息视图中的所需符号来更改符号的位置。
移动或编辑符号时，左边调用信息视图中的相应符号旁边将会出现一个选中标记。

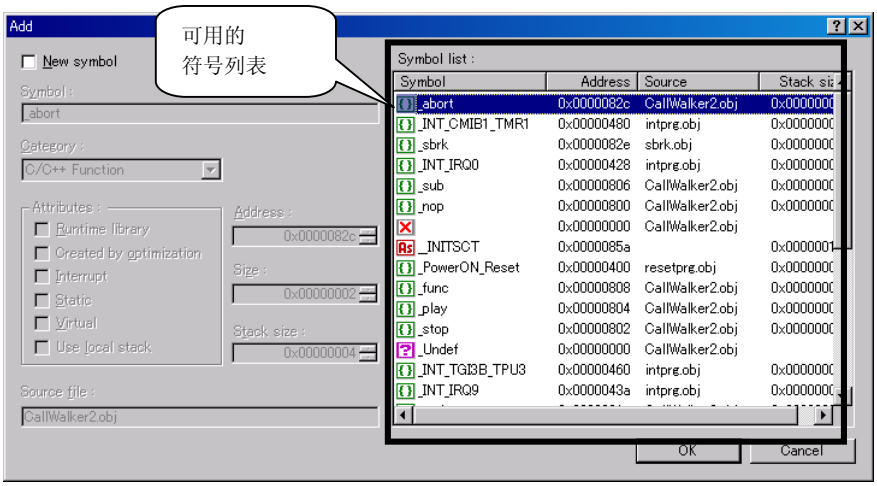


以下小节描述可用的命令。

♦ 添加 (Add)... 命令

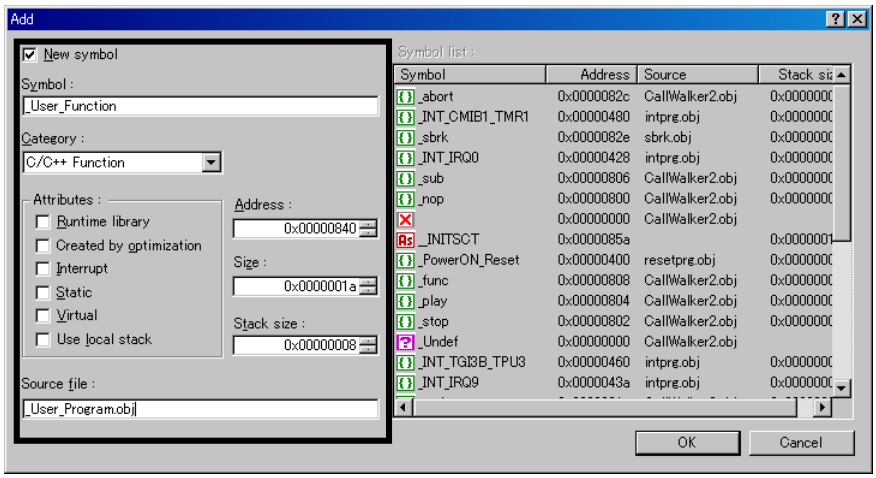
(1) 添加现有的符号

单击添加 (Add)... 命令时，以下对话框将会出现。 右边的列表显示当前文件中的符号。 要添加现有符号，请在列表中选择所需符号，然后单击 [确定 (OK)] 按钮。



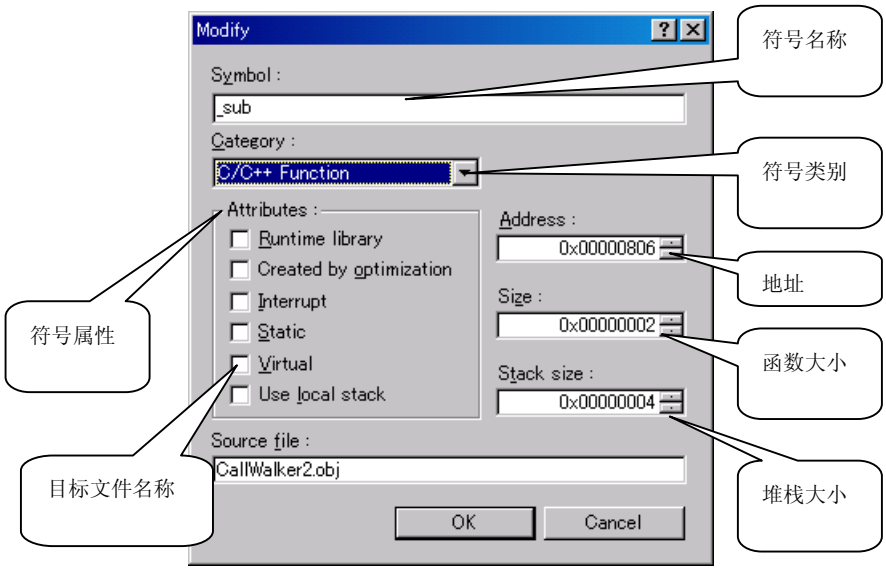
(2) 添加新的符号

选择左边的 [新符号 (New symbol)] 复选框时，您可以创建新的符号。
同时，您可以定义符号名称、符号类别、属性、地址、堆栈大小和其他详细信息。



♦ 修改 (Modify)... 命令

选择您要更改其信息的符号，然后单击 [修改 (Modify)...] 命令。 以下对话框将会显示。 您可以修改多个信息项目。



♦ 删除 (Delete)... 命令

要删除不需要用于确定堆栈大小的符号，请选择这些符号（在左边或右边视图中）然后单击 [删除 (Delete) …] 命令。

7.3.5 汇编程序的堆栈区域大小

和 C/C++ 程序所使用的不同，汇编程序使用的堆栈区大小不能在汇编中自动计算。 因此，汇编函数所使用的堆栈区大小，应该使用 Call Walker 编辑。
但是，堆栈区大小使用 **.STACK** 指令在汇编函数中指定。 Call Walker 显示由 **.STACK** 指令指定的值。

• **.STACK** 指令的描述

使用 Call Walker 定义被参考之指定的符号的堆栈数量。

符号的堆栈值只能定义一次；相同符号的第二次和之后的指定将会被忽略。 从 H'00000000 至 H'FFFFFFFE 范围中 2 的倍数可指定为堆栈值，任何其他值将为无效。

堆栈值必须如下指定：
- 必须指定一个常数值。
- 不能使用向前参考符号、外部参考符号和相对地址符号。

• **.STACK** 汇编程序指令的指定方法

Δ .STACK Δ <符号> = <堆栈值>

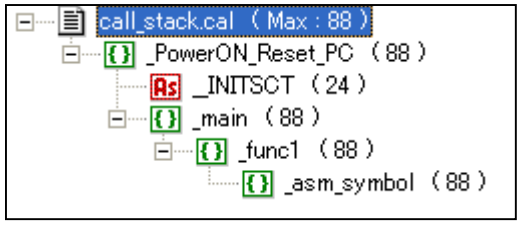
• 汇编程序的实例

```
        .EXPORT     _asm_symbol
        .SECTION    P,CODE,ALIGN=4
_asm_symbol:
        .STACK      _asm_symbol=88
        :
        RTS
        NOP
        .END
```

← **_asm_symbol** 函数的堆栈大小

• Call Walker 显示的实例

如以下实例所示，_asm_symbol 函数所使用的堆栈区大小在 Call Walker 中显示为“88”。



• 说明

(1) **.STACK** 汇编程序指令只能使 Call Walker 显示堆栈大小，而不会影响程序的行为。
(2) 此汇编程序指令受 SuperH RISC engine 汇编程序 7.00 或以上版本支持。

7.3.6 合并堆栈信息

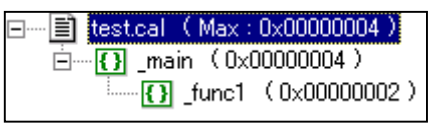
您可以将保存或编辑的堆栈信息文件，与其他堆栈信息文件合并。通过执行此操作，编辑的堆栈信息将不会被后创建的堆栈信息覆盖。

♦ 合并实例

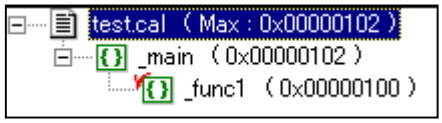
(1) test.c 的内容

```
void main(void)
{
    func1();
}
```

(2) 从 Call Walker 打开堆栈信息文件。



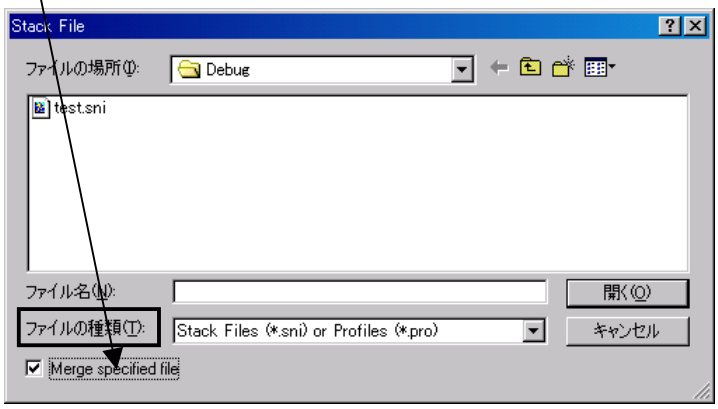
(3) 更改文件的内容（将 func1 的堆栈大小更改为 100）。



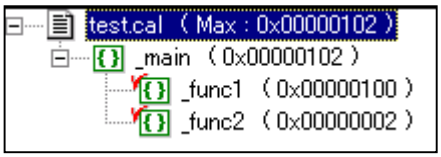
(4) 更改 test.c 的内容然后执行创建（**为 func2 添加一个调用**）。

```
void main(void)
{
    func1();
    func2();
}
```

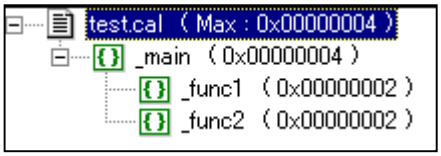
(5) 在 Call Walker 中打开 test.cal 的同时，打开 test.sni 。
选取此处然后选择 [确定 (OK)] 按钮。



(6) func2 的信息将会添加，同时保存于步骤 (3) 中更改之 func1 的堆栈大小。 这是堆栈信息的合并。

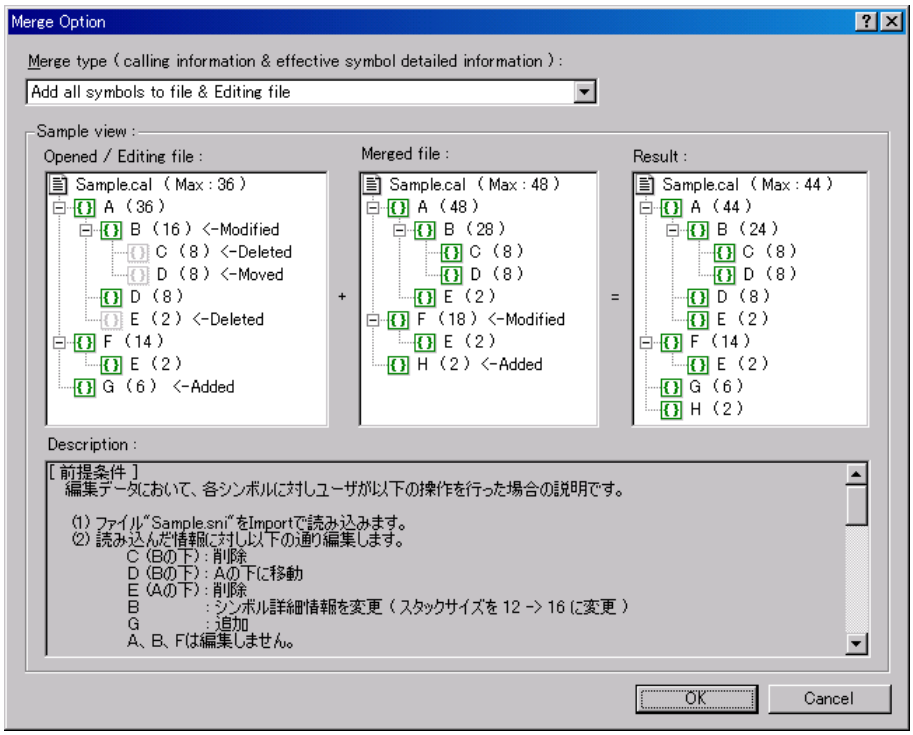


如果您在步骤 (5) 中没有选择 [合并指定的文件 (Merge specified file)] 复选框，步骤 (3) 中更改之 func1 的堆栈大小将会返回之前的值。



♦ 详细的合并选项

您可以更改合并的方法。 可用的方法共有五种。
有关合并方法的详细信息，请参阅以下对话框中的 [描述 (Description)]。
如何指定合并方法
[工具 (Tools)]-> [合并选项 (Merge Option)...]



♦ 注意

合并功能在 Call Walker 1.3 或以上版本中提供。

7.3.7 其他功能

- 实时操作系统图标

您可以在窗口左边的调用信息视图中，将实时操作系统图标显示为 RTOS。

[如何指定]
[工具 (Tools)]-> [实时操作系统选项 (Realtime OS Option)...]

此具备 csv 扩展名的文件随附在每个实时操作系统产品中。

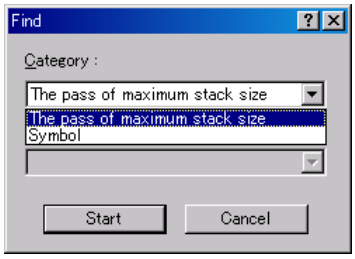
- 输出列表

  您可以以文件中的文本格式输出堆栈信息。

[如何输出]
[文件 (File)]->[输出列表 (Output List)...]

- 搜索功能

您可以通过在以下对话框中指定所需的目标，从调用信息视图查找下列两个项目。



(1) 使用最大堆栈大小传递
(2) 符号名称

[如何指定]|
[编辑 (Edit)]->[查找 (Find)...]
[编辑 (Edit)]->[查找下一个 (Find Next)...] （查找下一个项目）
[编辑 (Edit)]->[查找上一个 (Find Previous)...] （查找上一个项目）

- 设定调用信息视图的显示格式

您可以使用以下两个命令，选择用于显示堆栈大小的格式：

(1) 显示所需的堆栈
最大的堆栈大小将显示在最上面，而最小的堆栈大小则显示在最下面。

(2) 显示所用的堆栈
最小的堆栈大小将显示在最上面，而最大的堆栈大小则显示在最下面。

[如何指定]
[视图 (View)]->[显示所需的堆栈 (Show Required Stack)] 或 [显示所用的堆栈 (Show Used Stack)]

SuperH RISC Engine C/C++编译程序应用笔记

有效的 C++ 编程技术

第 8 节　有效的 C++ 编程技术

编译程序支持 C++ 和 C 语言。
本章将详细描述面向目标的语言 C++ 的选项，及如何使用各种 C++ 函数。

谨慎地为嵌入式系统的 C++ 程序编码。 否则，程序的目标大小或处理速度将比预期的来得大或低。
因此，本章呈现了一些 C++ 程序的性能比 C 逊色的情形，及您可用以修正性能降低的代码。

下表显示有效的 C++ 编程技术列表：

编号	类别	项目	节
1	初始化处理/后处理	全局类目标的初始化处理和后处理	8.1.1
2	C++ 函数简介	如何参考 C 目标	8.2.1
3		如何执行*新建*和*删除*	8.2.2
4		静态成员变量	8.2.3
4	如何使用选项	用于嵌入式应用程序的 C++ 语言	8.3.1
5		运行时类型信息	8.3.2
6		异常处理函数	8.3.3
7		禁止预连接程序的启动	8.3.4
8	C++ 编码的优缺点	构造函数 (1)	8.4.1
9		构造函数 (2)	8.4.2
10		默认参数	8.4.3
11		内联扩展	8.4.4
12		类成员函数	8.4.5
13		*operator* 运算符	8.4.6
14		函数超载	8.4.7
15		参考类型	8.4.8
16		静态函数	8.4.9
17		静态成员变量	8.4.10
18		匿名的*联合(union)*	8.4.11
19		虚拟函数	8.4.12

8.1 初始化处理/后处理

8.1.1 全局类目标的初始化处理和后处理

♦ 重点：

要在 C++ 中使用全局类目标，您必须在 *main* 函数之前和之后，分别调用初始化处理函数 (_CALL_INIT) 及后处理函数 (_CALL_END) 。

♦ 什么是全局类目标？

全局类目标是在函数外部声明的类目标。

（函数内部的类目标声明）

```
void main(void)
{
    X XSample(10);
    X* P = &XSample;

    P->Sample2();
}
```

（全局类目标声明）

```
X XSample(10);
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

在函数外部声明

♦ 为什么需要执行初始化处理/后处理？

若类目标如以上所示在函数内部被声明，类 *X* 的构造函数将在执行函数 *main* 时被调用。
相对的，全局类目标的声明将不在执行函数时被执行。
于是，您必须在调用 *main* 函数前调用 _CALL_INIT，以明确的调用类 *X* 的构造函数。同样的，在调用 *main* 函数后调用 _CALL_END，以调用类 *X* 的析构函数。

♦ 使用和不使用 _CALL_INIT/_CALL_END 时的操作：

下面显示当类 *X* 的成员变量 *x* 的值被参考时所得到的值。
不使用 _CALL_INIT/_CALL_END 时，将无法得到正确的值，且 *while* 语句中的表达式不能执行如下：

（成员变量 x 的值）
使用**_CALL_INIT --> 10**

不使用**_CALL_INIT -->**

```
class X{
    int x;
public:
     X(int n){x = n}; // 构造函数
     ~X(){}           // 析构函数
    void Sample2(void);
};
X XSample(10); // 全局类目标
void X::Sample2(void)
{
    while(x == 10)
    {
    }
}
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

<-- 参考位置

♦ 如何调用 _CALL_INIT/_CALL_END：

在调用 *main* 函数之前和之后，提供下列代码。

```
void INIT(void)
{
    _INITSCT();
    _CALL_INIT();
    main();
    _CALL_END();
}
```

若使用了 HEW，请在调用 *resetprg.c* 的 _CALL_INIT/_CALL_END 的段中，移除注解字符。

（*resetprg.c* 的 *PowerON_Reset* 函数）

```
__entry(vect=0) void PowerON_Reset(void)
{
    set_imask_ccr(1);
    _INITSCT();

// _CALL_INIT();        // 在您使用全局类目标时移除注解

// _INIT_IOLIB();      // 在您使用 SIM I/O 时移除注解

// errno=0;             // 在您使用 errno 时移除注解
// srand(1);            // 在您使用 rand() 时移除注解
// _s1ptr=NULL;         // 在您使用 strtok() 时移除注解

    HardwareSetup();   // 使用硬件设置
    set_imask_ccr(0);

    main();

// _CLOSEALL();         // 在您使用 SIM I/O 时移除注解

// _CALL_END();         // 在您使用全局类目标时移除注解

    sleep();
}
```

8.2 C++ 函数简介

8.2.1 如何参考 C 目标

♦ 重点：

使用 *'extern "C"'* 声明，以直接在 C++ 程序中使用现有 C 目标程序的资源。
同样的，C++ 目标程序的资源可在 C 程序中使用。

♦ 使用的实例：

1. 使用 *'extern "C"'* 声明，以参考 C 目标程序中的函数。

（C++ 程序）

```
extern "C" void CFUNC();
void main(void)
{
    X XCLASS;
    XCLASS.SetValue(10);

    CFUNC();
}
```

（C 程序）

```
extern void CFUNC();
void CFUNC()
{
    while(1)
    {
        a++;
    }
}
```

2. 使用 *'extern "C"'* 声明，以参考 C++ 目标程序中的函数。

（C 程序）

```
void CFUNC()
{
    CPPFUNC();
}
```

（C++ 程序）

```
extern "C" void CPPFUNC();
void CPPFUNC(void)
{
while(1)
{
        a++;
}
}
```

♦ 重要信息：

1. 无法连接旧版本（版本 5）编译程序所生成的 C++ 目标，因为编码和执行方法被更改了。
   确保在使用前重新编译。
2. 以上述方法调用的函数无法被超载。

8.2.2 如何执行 new 和 delete

♦ 重点：

要使用 *new*，请执行低层函数。

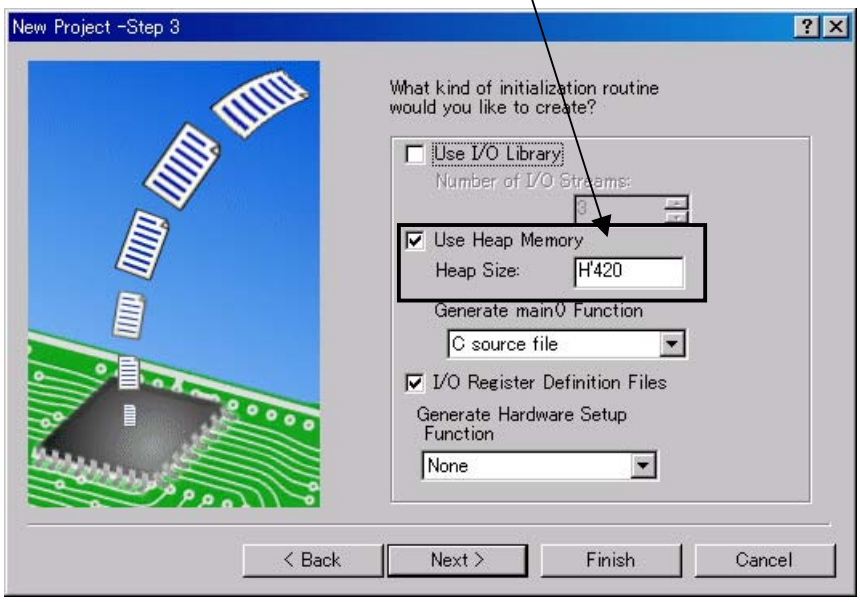
♦ 描述：

若在嵌入式系统中使用了*新建 (new)*，实际堆存储器的动态分配将由使用 *malloc* 实现。
于是，如使用 *malloc* 般，执行低层界面例程 (*sbrk*) 以指定将被分配的堆存储器大小。

♦ 执行方法：

要使用 HEW，确保在创建工作空间时选中了[使用堆存储器 (Use Heap Memory)]。
若选中了此选项，下一页显示的 *sbrk.c* 和 *sbrk.h* 将自动被创建。
指定要在堆大小 (Heap Size) 中分配的堆存储器大小。
要在创建工作空间后更改大小，请在 *sbrk.h* 中的 HEAPSIZE 更改定义的值。

若不使用 HEW，创建显示在下一页的文件，然后在工程中执行它。



```
(sbrk.c)

#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=   /* 指定定义的堆区域中的    */
                             /* 最小单位               */

static  union  {
    long  dummy ;           /* 4 字节边界的虚设        */
    char heap[HEAPSIZE];    /* 由 sbrk 所管理的区域    */
                            /*                 声明 */
 }heap_area ;

static  char  *brk=(char *)&heap_area;/* 分配的区域的终止地址                  */

/************************************************************************/
/*      sbrk: 数据写入                                                  */
/*      返回值: 分配的区域的起始地址（传递）                            */
/*                    -1                （失败）                        */
/************************************************************************/
char  *sbrk(size_t size)     /* 分配的区域大小                 */
{
    char  *p;

    if(brk+size>heap_area.heap+HEAPSIZE) /* 空区域大小        */
      return (char *)-1 ;

    p=brk ;                  /* 区域分配                       */
    brk += size ;            /* 终止地址更新                   */
    return p ;
}
```

```
(sbrk.h)

/* 由 sbrk 管理的区域大小 */
#define HEAPSIZE 0x420
```

8.2.3 静态成员变量

♦ 描述：

在 C++ 中，拥有*静态*属性的类成员变量，可被类的多个目标所共享。
于是，静态成员变量变得有用，例如，因为它可被相同的类的多个目标用作公用标志。

♦ 使用的实例：

在 main 函数内创建五个 A 类目标。
静态成员变量 *num* 具有 0 的初始值。此值将在每次创建目标时，被构造函数所增加。
目标共享的静态成员变量 num，可拥有的最大值为 5。

♦ FAQ：

下面列出一些有关使用静态成员变量的常见问题集。
[出现 L2310 错误]
使用静态成员变量时，将在连接时输出“** L2310 (E) 未定义的外部符号“类名称：：静态成员变量名称”在“文件名称”中被参考”(“** L2310 (E) Undefined external symbol “class-name::static-member-variable-name” referenced in “file-name”””) 的信息。

[解决方法]

此错误是因未定义静态成员变量而发生。
如下一页所示，添加下列其中一项定义：

```
若有初始值： int A::num = 0;
若没有初始值： int A::num = 0;
```

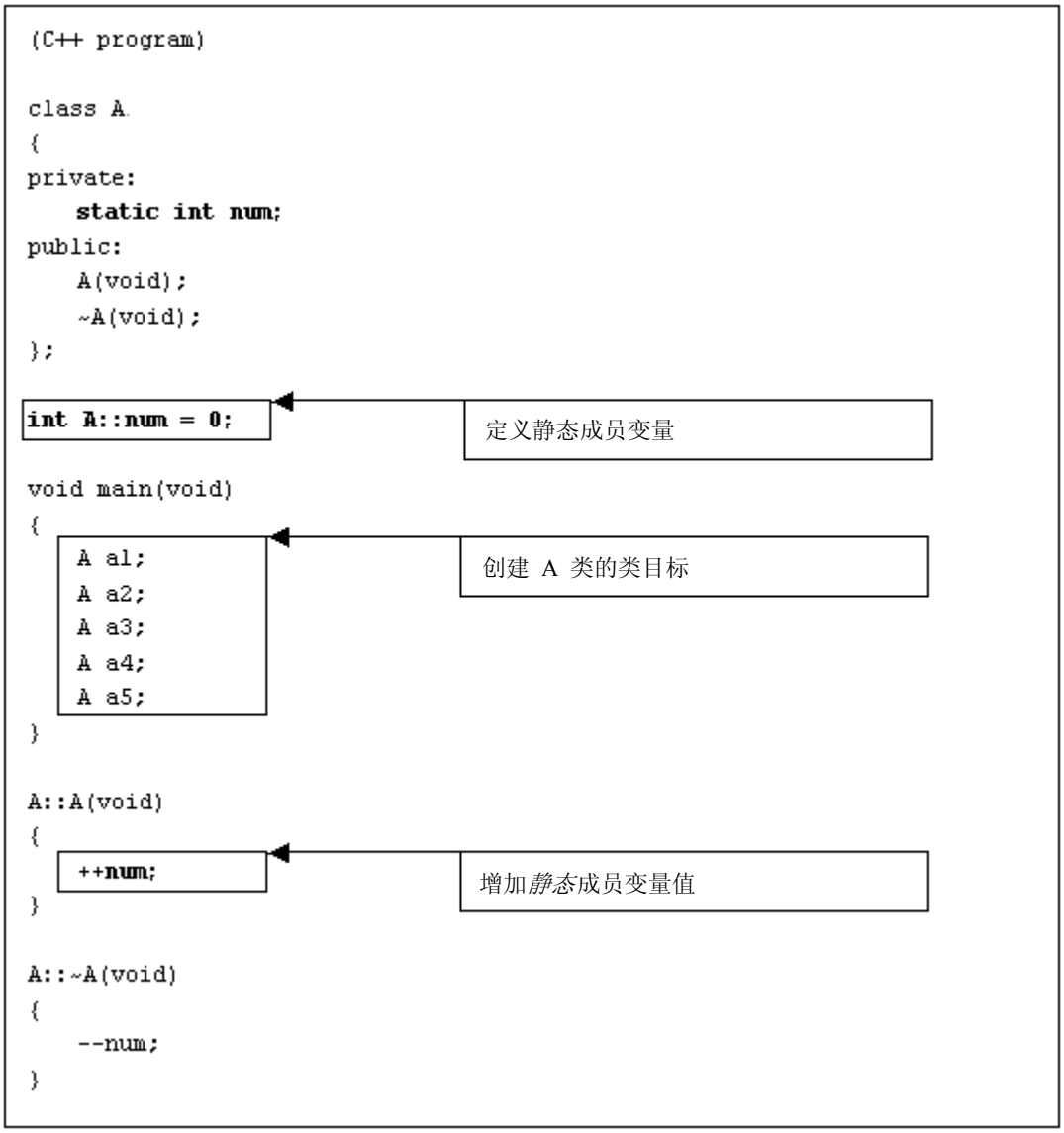
[无法分配初始值]

没有初始值被分配到将被初始化的*静态 (static)* 成员变量。

[解决方法]

默认设定在 D 段中创建了被当作具有初始值的变量来初始化、处理的*静态*成员变量。 因此，指定优化连接编辑程序的 ROM 执行支持选项，并在初始例程中，使用 *_INITSCT* 函数来将 D 段从 ROM 复制到 RAM。

注意： * 若 HEW 自动创建初始值，则不需要执行此解决方法。


(C++ program)
class A
{
private:
static int num;
public:
A(void);
~A(void);
};
int A::num = 0;
定义静态成员变量
void main(void)
{
A a1;
A a2;
A a3;
A a4;
A a5;
创建 A 类的类目标
}
A::A(void)
{
++num;
增加静态成员变量值
}
A::~A(void)
{
--num;
}

8.3 如何使用选项

8.3.1 用于嵌入式应用程序的 C++ 语言

♦ 描述：

ROM/RAM 的大小和执行速度对嵌入式系统很重要。
用于嵌入式应用程序的 C++ 语言 (EC++) 是 C++ 语言的一个子集。 对于 EC++，一些不适合嵌入式系统的 C++ 函数已被移除。
使用 EC++，您可以创建适用于嵌入式系统的目标。

♦ 指定方法：

对话框菜单： C/C++ 标签 类别 (Category)： 其他 (Other) 标签，对照 EC++ 语言规格 (Check against EC++ language specification)
命令行： *eccp*

♦ 不支持的关键字：

若包含了下列任何关键字，则会输出一则错误信息。

catch, const_cast, dynamic_cast, explicit, mutable, namespace, reinterpret_cast, static_cast, template, throw, try, typeid, typename, using

♦ 不支持的语言规格：

若包含了下列任何语言规格，则会输出一则警告信息。

多个继承，虚拟基本类

8.3.2 运行时类型信息

♦ 描述：

在 C++ 中，拥有虚拟函数的类目标，可能具备仅在运行时可识别的类型。
可用的运行时识别函数，能够为这类情况提供支持。
要在 C++ 中使用此函数，请使用 *type_info* 类、*typeid* 运算符，及 *dynamic_cast* 运算符。
对于编译程序，指定下列选项以使用运行时类型信息。
另外，在连接时指定下列选项，以启动预连接程序。

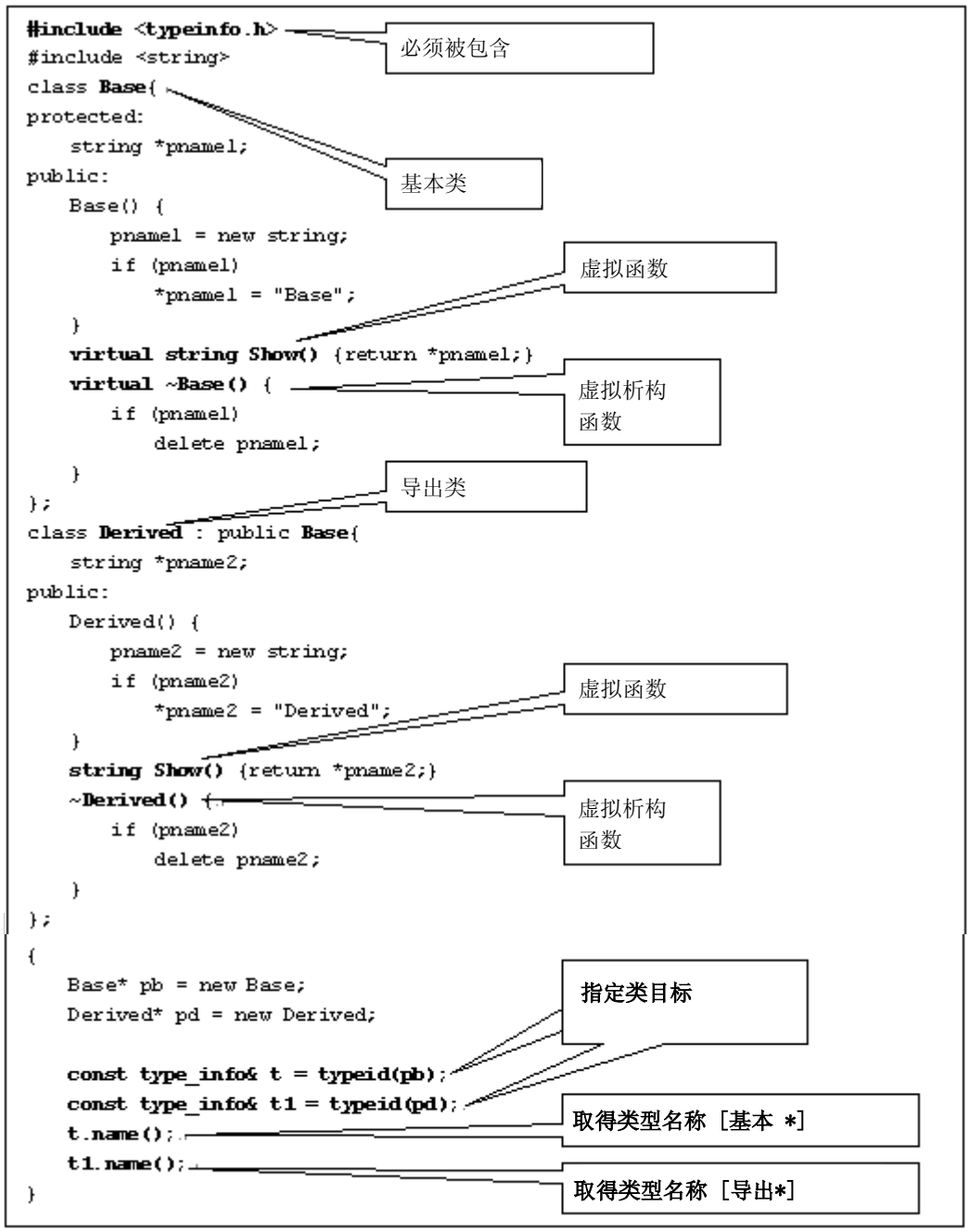
♦ 指定方法：

对话框菜单：CPU 标签，允许/禁止运行时类型信息 (Enable/disable runtime type information)
命令行： ***rtti=on | off***

对话框菜单： 连接/程序库 (Link/Library) 标签 类别： 输入 (Input) 标签，预连接程序控制项 (Prelinker control)
然后，选取自动 (Auto) 或运行 (Run) 预连接程序。
命令行：*Do not specify noprelink (default).*

♦ 使用 *type_info* 类及 *typeid* 运算符的实例：

type_info 类被用以识别目标的运行时类型。
使用 *type_info* 类，在执行时程序比较类型或取得类。
要使用 *type_info* 类，请使用 typeid 运算符来指定具备虚拟函数的类目标。



♦ 使用 *dynamic_cast* 运算符的实例：

例如，在包含虚拟函数的类及其导出类之间使用 *dynamic_cast* 运算符，以在运行时将导出类的指针或参考，转型至基本类的指针或参考。

```
#include <string>
class Base{
protected:
    string *pname1;
public:
    Base() {
        pname1 = new string;
        if (pname1)
            *pname1 = "Base";
    }
    virtual string Show() {return *pname1;}
    virtual ~Base() {
        if (pname1)
            delete pname1;
    }
};
class Derived : public Base{
    string *pname2;
public:
    Derived() {
        pname2 = new string;
        if (pname2)
            *pname2 = "Derived";
    }
    string Show() {return *pname2;}
    ~Derived() {
        if (pname2)
            delete pname2;
    }
};
void main(void)
{
    Derived *pderived = new Derived;
    Base *pbase = dynamic_cast<Base *> (pderived);

    string ddd;
    ddd = pbase-> Show();

    delete pbase;
}
```

8.3.3 异常处理函数

♦ 描述：

不像 C，C++具有被称为异常 (Exception) 的错误处理机制。
异常是将程序中的错误位置与错误处理代码连接的方法。
使用异常机制，以将错误处理代码集中到一个位置。
对于编译程序，指定下列选项以使用异常机制。

♦ 指定方法：

对话框菜单： CPU 标签，使用 C++ 的 try、throw 和 catch (Use try, throw and catch of C++)
命令行： *exception*

♦ 使用的实例：

若打开文件“INPUT.DAT”失败，请启动异常处理，并在标准错误输出中显示错误。

```
（异常处理的 C++ 程序实例）

void main(void)
{
    try
    {
        if ((fopen("INPUT.DAT","r"))==NULL){
            char * cp = "cannot open input file\n";
            throw cp;
        }
    }
    catch(char *pstrError)
    {
        fprintf(stderr,pstrError);
        abort();
    }
    return;
}
```

♦ 重要信息：

编码性能可能降低。

8.3.4 禁止预连接程序的启动

♦ 描述：

启动预连接程序将降低连接速度。 除非 C++ 的模板函数或运行时类型转换被使用，否则无须运行预连接程序。
要从命令行使用连接程序，请指定下列 *noprelink* 选项。
若使用了 Hew，且*预连接程序控制项 (Prelinker control)* 列表框被设定为自动 (Auto)，*noprelink* 选项的输出将被自动控制。

♦ 指定方法：

对话框菜单： 连接/程序库 (Link/Library) 标签 类别： 输入 (Input) 标签，预连接程序控制项 (Prelinker control)
命令行： *noprelink*

8.4 C++ 编码的优缺点

在编译 C++ 程序时，编译程序将 C++ 程序从内部转换为 C 程序，以创建目标。
本章将比较转换后的 C++ 程序和 C 程序，并描述对各个函数编码效率的影响。

编号	函数	开发与维护	大小缩减	速度	节
1	构造函数 (1)	◎	Δ	Δ	8.4.1
2	构造函数 (2)	◎	Δ	Δ	8.4.2
3	默认参数	◎	○	○	8.4.3
4	内联扩展	○	Δ	○	8.4.4
5	类成员函数	◎	Δ	Δ	8.4.5
6	*operator* 运算符	◎	Δ	Δ	8.4.6
7	函数超载	◎	○	○	8.4.7
8	参考类型	◎	○	○	8.4.8
9	静态函数	◎	○	○	8.4.9
10	静态成员变量	◎	○	○	8.4.10
11	匿名的*联合(union)*	◎	○	○	8.4.11
12	虚拟函数	◎	Δ	Δ	8.4.12

◎和 C 一样
○使用时须谨慎
Δ性能降低

8.4.1 构造函数 (1)

开发与维护	◎	大小缩减	Δ	速度	Δ

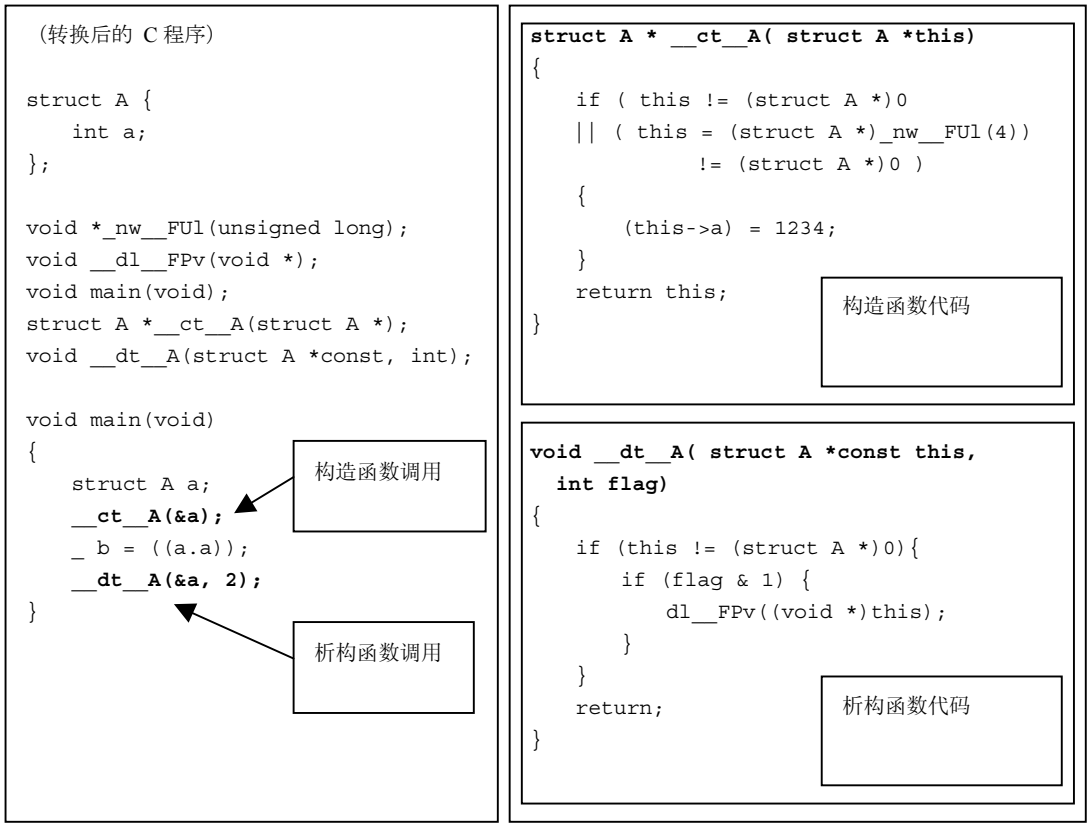
♦ 重点：

使用构造函数，以自动初始化类目标。 然而，请谨慎使用，因为它也将影响目标大小和处理速度，如下所示：

♦ 使用的实例：

创建 A 类构造函数与析构函数，并编译它们。大小和处理速度将受到影响，因为构造函数和析构函数将在类声明中被调用，同时判定也将在构造函数和析构函数的代码中作出。

```
（C++ 程序）
class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a;
    b = a.getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```


（转换后的 C 程序）
struct A {
int a;
};
void *_nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
void main(void)
{
struct A a;
__ct__A(&a);
_ b = ((a.a));
__dt__A(&a, 2);
}
构造函数调用
析构函数调用
struct A * __ct__A(struct A *this)
{
if (this != (struct A *)0
|| (this = (struct A *)_nw__FUl(4))
!= (struct A *)0)
{
(this->a) = 1234;
}
return this;
}
构造函数代码
void __dt__A(struct A *const this,
int flag)
{
if (this != (struct A *)0){
if (flag & 1) {
dl__FPv((void *)this);
}
}
return;
}
析构函数代码

8.4.2 构造函数 (2)

开发与维护	◎	大小缩减	Δ	速度	Δ

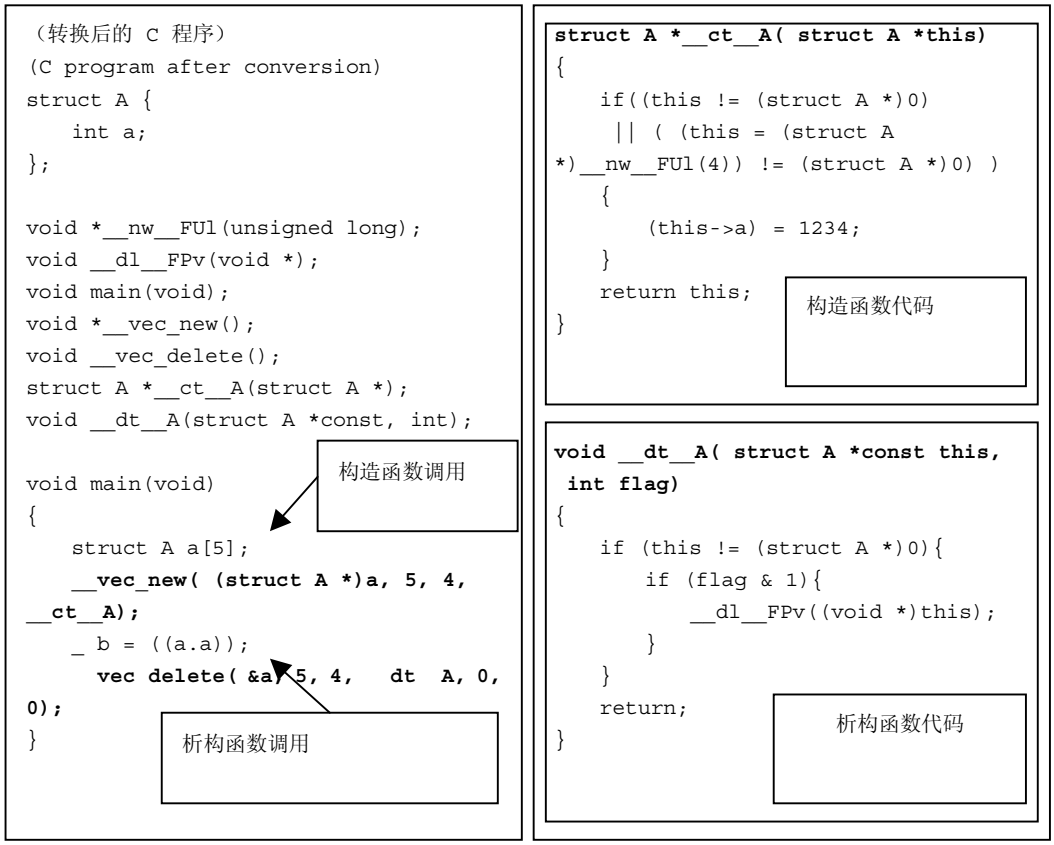
♦ 重点：

要在**数组**中声明类，请使用构造函数以自动初始化类目标。 然而，请谨慎使用，因为它也将影响目标大小和处理速度，如下所示：

♦ 使用的实例：

创建 A 类构造函数与析构函数，并编译它们。存储器需要被动态分配和释放，因为构造函数和析构函数在类声明中被调用，但在数组中被声明。
使用*新建 (new)* 和*删除 (delete)* 以动态分配和释放存储器。
这将需要执行低层函数。（要获取有关执行方法的详细资料，请参考第 8.1.2 节。）
大小和处理速度将受影响，因为判定和低层函数处理被添加在构造函数与析构函数的代码中。

```
（C++ 程序）
class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a[5];
    b = a[0].getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```


（转换后的 C 程序）
(C program after conversion)
struct A {
int a;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
void *__vec_new();
void __vec_delete();
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
void main(void)
{
struct A a[5];
__vec_new((struct A *)a, 5, 4,
__ct__A);
_ b = ((a.a));
vec delete(&a, 5, 4, dt A, 0,
0);
}
构造函数调用
析构函数调用
struct A *__ct__A(struct A *this)
{
if((this != (struct A *)0)
|| ((this = (struct A
*)__nw__FUl(4)) != (struct A *)0))
{
(this->a) = 1234;
}
return this;
}
构造函数代码
void __dt__A(struct A *const this,
int flag)
{
if (this != (struct A *)0){
if (flag & 1){
__dl__FPv((void *)this);
}
}
return;
}
析构函数代码

8.4.3 默认参数

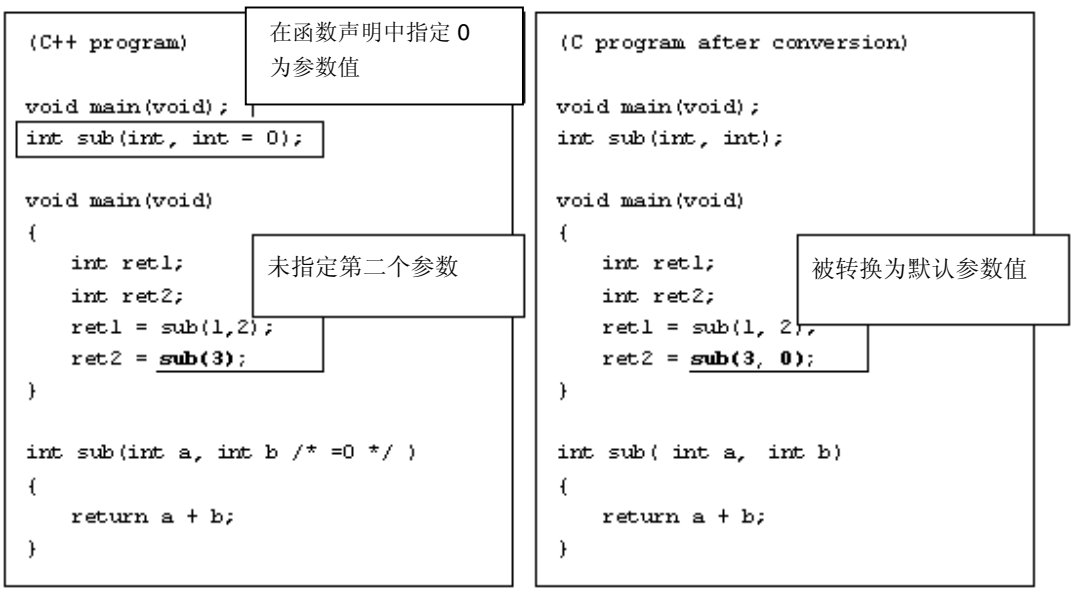
开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

在 C++ 中，默认参数可被用以设定调用函数时所使用的默认值。
要使用默认参数，在声明函数时为函数参数指定一个默认值。
这将消除在许多函数调用中指定参数的需要，并允许使用默认参数作为替代，进而使开发效率获得增进。
若指定了参数，参数值可被更改。

♦ 使用的实例：

下面显示在声明函数 *sub* 时将 0 指定为默认参数值时调用函数 *sub* 的实例。
如下所示，若调用函数 *sub* 时，默认参数值是可接受的，则不需指定任何参数。
而且，当被转换为 C 时，程序的效率也不会降低。
总之，默认参数确保更好的开发与维护效率，同时和 C 相较起来没有缺点。



8.4.4 内联扩展

开发与维护	○	大小缩减	Δ	速度	○

♦ 重点：

当编码函数的定义时，在起始处指定 *inline* 以形成函数的内联扩展。 这将消除函数调用的内务操作，并提高处理速度。

♦ 使用的实例：

将函数 *sub* 指定为内联函数，并在 main 函数内将它内联扩展。然后，移除函数 *sub* 的代码。
然而，函数 *sub* 无法从其他文件被参考。
请慎用内联扩展，因为虽然处理速度取得了一定的增进，但程序大小可能会变得过于庞大，除非仅使用小型函数。

（C++ 程序）

```
int a;

inline int sub(int x, int y)
{
    return (x+y);
}

void main(void)
{
    a = sub(1,2);
}
```

（转换后的 C 程序）

```
int a;
void main(void)
{
    a = 3;
    return;
}
```

扩展函数 *sub* 的内容

(1+2=3)

8.4.5 类成员函数

开发与维护	◎	大小缩减	Δ	速度	Δ

♦ 重点：

对类进行定义将允许信息的隐藏，并增进开发与维护效率。
然而，请慎用此技术，因为它将影响大小和处理速度。

♦ 使用的实例：

在下例中，类成员函数 *设定 (set)* 和 *添加 (add)* 被用以存取 *专用 (private)* 类成员变量 *a*、*b* 和 *c*。
当调用类成员函数时，C++ 程序中的参数规格仅有一个值或没有参数。
不过，如在转换后的 C 程序中所示，A 类（结构 A）的地址也被当作参数传递。
另外，*专用 (private)* 类成员变量 *a*、*b* 和 *c* 在类成员功能码中被存取。
然而，*this* 指针被用于对它们进行存取。
总的来说，慎用类成员函数，因为它将影响大小和处理速度。

（C++ 程序）

```
class A
{
private:
    int a;
    int b;
    int c;
public:
    void set(int, int, int);
    int add();
};

int main(void)
{
    A a;
    int ret;

    a.set(1,2,3);
    ret = a.add();

    return ret;
}
void A::set(int x, int y, int z)
{
    a = x;
    b = y;
    c = z;
}
int A::add()
{
    return (a += b + c);
}
```

（转换后的 C 程序）

```
struct A {
    int a;
    int b;
    int c;
};
void set__A_int_int(struct A *const, int, int, int);
int add__A(struct A *const);

int main(void)
{
    struct A a;
    int ret;

    set__A_int_int(&a, 1, 2, 3);
    ret = add__A(&a);

    return ret;
}
void set__A_int_int(struct A *const this, int x, int y, int z)
{
    this->a = x;
    this->b = y;
    this->c = z;
    return;
}
int add__A(struct A *const this)
{
    return (this->a += this->b + this->c);
}
```

8.4.6 *operator* 运算符

开发与维护	○	大小缩减	Δ	速度	Δ

♦ 重点：

在 C++ 中，使用关键字，*operator* 以超载运算符。
这将允许对用户操作的简单编码，如矩阵操作和向量计算。
然而，请慎用 *operator*，因为它将影响大小和处理速度。

♦ 使用的实例：

在下例中，使用 *operator* 关键字使一元运算符“＋”被超载。
若*向量 (Vector)* 类被声明如下，一元运算符“＋”可被更改到用户操作。
然而，大小和处理速度将受影响，因为如转换后的 C 程序所示，做出了使用 *this* 指针的参考。

（C++ 程序）

```
class Vector
{
private:
    int x;
    int y;
    int z;
public:
    Vector & operator+ (Vector &);
};

void main(void)
{
    Vector a,b,c;

    a = b + c;
}

Vector & Vector::operator+ (Vector & vec)
{
    static Vector ret;

    ret.x = x + vec.x;
    ret.y = y + vec.y;
    ret.z = z + vec.z;

    return ret;
}
```

用户操作（加法）

（转换后的 C 程序）

```
struct Vector {
    int x;
    int y;
    int z;
};

void main(void);
struct Vector *__plus__Vector_Vector(struct Vector *const, struct Vector *);

void main(void)
{
    struct Vector a;
    struct Vector b;
    struct Vector c;

    a = *__plus__Vector_Vector(&b, &c);
    return;
}

struct Vector *  plus  Vector Vector( struct Vector *const this,  struct Vector
*vec)
{
    static struct Vector ret;

    ret.x = this->x + vec->x;
    ret.y = this->y + vec->y;
    ret.z = this->z + vec->z;

    return &ret;
}
```

使用 this 指针参考

8.4.7 函数的超载

开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

在 C++ 中，您可以“超载”函数，即给予不同函数相同的名称。
此功能在您使用具有相同处理，但不同参数类型的函数时，尤其有效。
小心别对不存在共同性的函数使用相同的名称，因为那肯定会造成故障。
此函数的使用将不影响大小或处理速度。

♦ 使用的实例：

在下例中，添加了第一个和第二个参数，且所得的值被用作返回值。
所有函数具有相同名称，*添加 (add)*，但有不同的参数和返回值类型。
如在转换后的 C 程序中所示，添加 (add) 函数的调用或添加 (add) 函数的代码并不增加代码大小。
因此，此功能的使用将不影响大小或处理速度。

```
（C++ 程序）
void main(void);
int add(int,int);
float add(float,float);
double add(double,double);
void main(void)
{
    int    ret_i = add(1, 2);
    float  ret_f = add(1.0f, 2.0f);
    double ret_d = add(1.0, 2.0);
}

int add(int x,int y)
{
    return x+y;
}

float add(float x,float y)
{
    return x+y;
}

double add(double x,double y)
{
    return x+y;
}
```

（转换后的 C 程序）

```
void main(void);
int add__int_int(int, int);
float add__float_float(float, float);
double add__double_double(double, double);

void main(void)
{
    auto int ret_i;
    auto float ret_f;
    auto double ret_d;

    ret_i = add__int_int(1, 2);
    ret_f = add__float_float(1.0f, 2.0f);
    ret_d = add__double_double(1.0, 2.0);
}

int add__int_int( int x,  int y)
{
    return x + y;
}

float add__float_float( float x,  float y)
{
    return x + y;
}

double add__double_double( double x,  double y)
{
    return x + y;
}
```

8.4.8 参考类型

开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

使用参考类型的参数将允许对程序的简单编码，并增进开发与维护效率。
另外，参考类型的使用将不影响大小或处理速度。

♦ 使用的实例：

如下所示，以参考类型传递代替指针传递将允许简单编码。
在参考类型中，被传递的不是 *a* 和 *b* 的值，而是它们的地址。
参考类型的使用，如在转换后的 C 程序中所示，将不影响大小和处理速度。

（C++ 程序）

```
void main(void);
void swap(int&, int&);

void main(void)
{
    int a=100;
    int b=256;

    swap(a,b);
}

void swap(int &x, int &y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

（转换后的 C 程序）

```
void main(void);
void swap(int *, int *);

void main(void)
{
    int a=100;
    int b=256;

    swap(&a, &b);
}

void swap(int *x,  int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

8.4.9 静态函数

开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

若类的配置由于导出类等而变得复杂，具有 *专用 (private)* 属性的 *静态 (static)* 类成员变量将变得更加难以存取，以至必须将它们的属性更改为 *公用 （public）*。
要在不更改 *专用 (private)* 属性的情况下存取 *静态 (static)* 类成员变量，创建被用作界面的成员函数，并在函数中指定 *静态 (static)* 变量。
静态 (static) 函数于是仅用来存取静态类成员变量。

♦ 使用的实例：

如在下一页中所示，使用静态函数以存取静态成员变量。
虽然使用类将影响代码效率，但使用静态函数并不会影响大小和处理速度。

♦ 注意：

要获取有关 *静态 (static)* 成员变量的详细资料，请参考第 8.2.3 节“静态成员变量”。

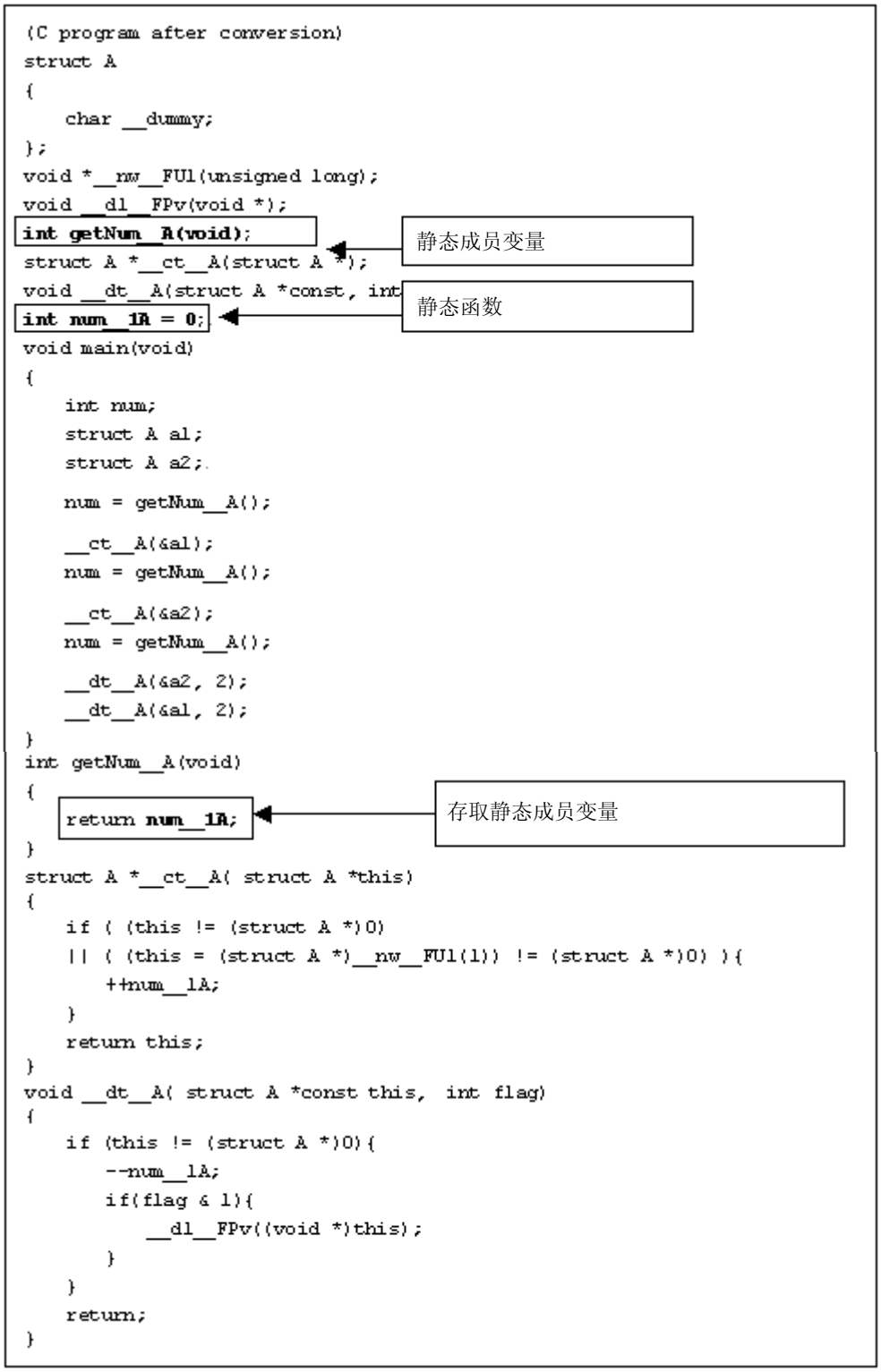
（C++ 程序）

```
class A
{
private:
    static int num;
public:
    static int getNum(void);
    A(void);
    ~A(void);
};

int A::num = 0;
void main(void)
{
    int num;

    num = A::getNum();

    A a1;
    num = a1.getNum();

    A a2;
    num = a2.getNum();
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}

int A::getNum(void)
{
    return num;
}
```

静态成员变量

静态函数

存取静态成员变量


(C program after conversion)
struct A
{
char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
int getNum__A(void);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int
int num__1A = 0;
void main(void)
{
int num;
struct A a1;
struct A a2;
num = getNum__A();
__ct__A(&a1);
num = getNum__A();
__ct__A(&a2);
num = getNum__A();
__dt__A(&a2, 2);
__dt__A(&a1, 2);
}
int getNum__A(void)
{
return num__1A;
}
struct A *__ct__A(struct A *this)
{
if ((this != (struct A *)0)
|| ((this = (struct A *)__nw__FUl(1)) != (struct A *)0)){
++num__1A;
}
return this;
}
void __dt__A(struct A *const this, int flag)
{
if (this != (struct A *)0){
--num__1A;
if(flag & 1){
__dl__FPv((void *)this);
}
}
return;
}
静态成员变量
静态函数
存取静态成员变量

8.4.10 静态成员变量

开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

在 C++ 中，拥有静态属性的类成员变量，可被类的多个目标所共享。
于是，静态成员变量变得有用，例如，因为它可被相同的类的多个目标用作公用标志。

♦ 使用的实例：

在 *main* 函数内创建五个 A 类目标。
静态成员变量 *num* 具有 0 的初始值。此值将在每次创建目标时，被构造函数所增加。
目标共享的静态成员变量 num，可拥有的最大值为 5。
另外，使用类将影响代码效率。
然而，使用静态成员变量并不影响大小和处理速度，因为编译程序将成员变量 *num* 当作普通全局变量来进行内部处理。

♦ 注意：

要获取有关*静态 (static)* 成员变量的详细资料，请参考第 8.2.3 节“静态成员变量”。

（C++ 程序）

```
class A
{
private:
    static int num;
public:
    A(void);
    ~A(void);
};

int A::num = 0;

void main(void)
{
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}
```

创建 A 类的类目标

增加*静态*成员变量值

（转换后的 C 程序）

```
struct A
{
    char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;
void main(void)
{
    struct A a1;
    struct A a2;
    struct A a3;
    struct A a4;
    struct A a5;
    __ct__A(&a1);
    __ct__A(&a2);
    __ct__A(&a3);
    __ct__A(&a4);
    __ct__A(&a5);
    __dt__A(&a5, 2);
    __dt__A(&a4, 2);
    __dt__A(&a3, 2);
    __dt__A(&a2, 2);
    __dt__A(&a1, 2);
}
struct A *__ct__A( struct A *this)
{
    if( (this != (struct A *)0)
    || ( (this = (struct A *)__nw__FUl(1)) != (struct A *)0) ){
        ++num__1A;
    }
    return this;
}
void __dt__A( struct A *const this,  int flag)
{
    if(this != (struct A *)0){
        --num__1A;
        if (flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

被编译程序当作普通全局变量来处理

建立 A 类的类目标

调用构造函数

调用析构函数

增加静态成员变量值

8.4.11 匿名的*联合 (union)*

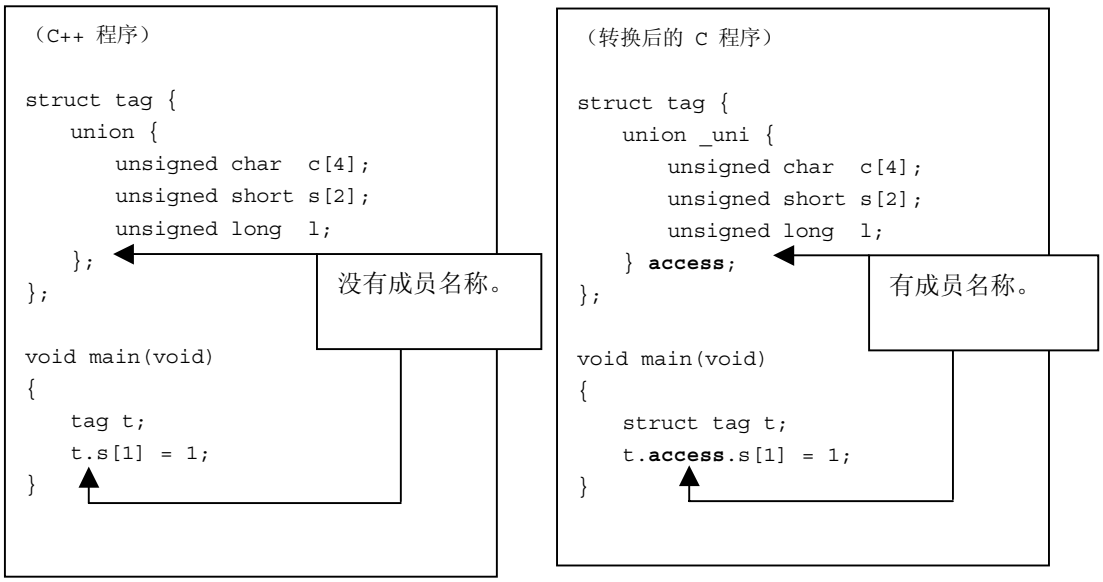
开发与维护	◎	大小缩减	○	速度	○

♦ 重点：

在 C++ 中，使用匿名的*联合 (union)* 以直接存取成员，而不需像在 C 中一样，需要指定成员名称。
这将增进开发的效率。 另外，它也不会影响大小和处理速度。

♦ 使用的实例：

在下例中，函数 main 被用以存取*联合 (union)* 成员变量 *s*。
在 C++ 程序中，成员变量 *s* 被直接存取。 在转换后的 C 程序中，它通过使用编译程序自动创建的成员名称被存取。
此简单编码的使用允许对成员变量的存取，而不影响目标效率。



8.4.12 虚拟函数

开发与维护	◎	大小缩减	Δ	速度	Δ

♦ 重点：

如下列程序所示，若每个基本类和导出类中都具有同名的函数，就必须使用虚拟函数。否则，不能如预期般正常进行函数调用。
若声明了虚拟函数，这些调用就可以如预期般正常进行。
使用虚拟函数以增进开发效率。然而，请慎用此函数，因为它将影响大小和处理速度。

♦ 使用的实例：

在 *main3* 函数调用中，两个指针存储 B 类地址。
若声明了 *virtual*，B 类 *foo* 函数将被正常调用。
若未声明 *virtual*，其中一个指针调用 A 类 *foo* 函数。
如下一页所示，虚拟函数的使用将创建表等，并将影响大小和速度。

```
（C++ 程序）

class A
{
private:
    int a;
public:
    virtual void foo(void);
};

class B : public A
{
private:
    int b;
public:
    virtual void foo(void);
};

void A::foo(void)
{
}

void B::foo(void)
{
}
```

虚拟函数声明

虚拟函数声明

```
void main1(void)
{
    A a;
    a.foo();
}

void main2(void)
{
    B b;
    b.foo();
}

void main3(void)
{
    B b;
    A * pa = &b;
    B * pb = &b;

    pa->foo();
    pb->foo();
}
```

虚拟函数调用

♦ 转换后的 C 程序（虚拟函数的表等）：

```
struct __T5585724;
struct __type_info;
struct __T5584740;
struct __T5579436;
struct A;
struct B;
extern void main1__Fv(void);
extern void main2__Fv(void);
extern void main3__Fv(void);
extern void foo__1AFv(struct A *const);
extern void foo__1BFv(struct B *const);
struct __T5585724
{
    struct __T5584740 *tinfo;
    long offset;
    unsigned char flags;
};
struct __type_info
{
    struct __T5579436 *__vptr;
};
struct __T5584740
{
    struct __type_info tinfo;
    const char *name;
    char *id;
    struct __T5585724 *bc;
};
struct __T5579436
{
    long d;                             // this 指针偏移
    long i;                             // 未被分配
    void (*f)();                        // 用于虚拟函数调用
};
struct A {                              // A 类声明
    int a;
    struct __T5579436 *__vptr;          // 虚拟函数表的指针
};
struct B {                              // B 类声明
    struct A __b_A;
    int b;
};
static struct __T5585724 __T5591360[1];
#pragma section $VTBL
extern const struct __T5579436 __vtbl__1A[2];
extern const struct __T5579436 __vtbl__1B[2];
extern const struct __T5579436 __vtbl__Q2_3std9type_info[];
#pragma section
extern struct __T5584740 __T_1A;
extern struct __T5584740 __T_1B;
```

```
static char __TID_1A;                          // 未被分配
static char __TID_1B;                          // 未被分配
static struct __T5585724 __T5591360[1] =       // 未被分配
{
    {
        &__T_1A,
        0L,
        ((unsigned char)22U)
    }
};
#pragma section $VTBL
const struct __T5579436 __vtbl__1A[2] =        // A 类的虚拟函数表
{
    {
        0L,                                    // 未被分配的区域
        0L,                                    // 未被分配的区域
        ((void (*)())&__T_1A)                  // 未被分配的区域
    },
    {
        0L,                                    // this 指针偏移
        0L,                                    // 未被分配的区域
        ((void (*)())foo__1AFv)                // ((void (*)())foo__1AFv) // A::foo() 的指针
    }
};
const struct __T5579436 __vtbl__1B[2] =        // B 类的虚拟函数表
{
    {
        0L,                                    // 未被分配的区域
        0L,                                    // 未被分配的区域
        ((void (*)())&__T_1B)                  // 未被分配的区域
    },
    {
        0L,                                    // this 指针偏移
        0L,                                    // 未被分配的区域
        ((void (*)())foo__1BFv)                // ((void (*)())foo__1BFv) // B::foo() 的指针
    }
};
#pragma section
struct __T5584740 __T_1A =                     // A 类的类型信息（未被分配）
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"A",
    &__TID_1A,
    (struct __T5585724 *)0
};
struct __T5584740 __T_1B =                     // B 类的类型信息（未被分配）
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"B",
    &__TID_1B,
    __T5591360
};
```

■ 转换后的 C 程序（虚拟函数调用）：

```
void main1__Fv(void)
{
    struct A _a;
    _a.__vptr = __vtbl__1A;
    foo__1AFv( &_a );                                   // foo_ 1AFv( &_a ); // 调用至 A::foo()
    return;
}
void main2__Fv(void)
{
    struct B _b;
    _b.__b_A.__vptr = __vtbl__1A;
    _b.__b_A.__vptr = __vtbl__1B;
    foo__1BFv( &_b );                                   // foo__1BFv( &_b ); // 调用至 B::foo()
    return;
}
void main3__Fv(void)
{
    struct __T5579436 *_tmp;
    struct B _b;
    struct A *__pa;
    struct B *__pb;

    (*((struct A*)(&_b))).__vptr = __vtbl__1A;
    (*((struct A*)(&_b))).__vptr = __vtbl__lB;
    _pa = (struct A *)&_b;
    _pb = &_b;

    _tmp = _pa->__vptr + 1;
    (  (void (*)(struct A *const))  _tmp->f  )  (  (struct A *)_pa + _tmp->d );
    // 调用至 B::foo()（被 _pa 指向的目标是  B）

    _tmp = _pb->__b_A.__vptr + 1;
    (  (void (*)(struct B *const))  _tmp->f  )  (  (struct B *)_pb + _tmp->d );
    // 调用至 B::foo()

    return;
}
```

SuperH RISC Engine C/C++编译程序应用笔记

优化连接编辑程序

第 9 节　优化连接编辑程序

本章描述在连接时有效选项的使用，以及在连接时模块间的优化。

下表显示与优化连接编辑程序之使用有关的项目列表。

编号	类别	项目	节
1	输入/输出选项	输入选项	9.1.1
		输出选项	9.1.2
2	列表选项	符号信息	9.2.1
3		参考数量	9.2.2
4		**交叉参考信息**	**9.2.3**
5	有效选项	输出至未使用区	9.3.1
6		S 类型文件的终止代码	9.3.2
7		调试信息压缩	9.3.3
8		连接时间缩减	9.3.4
9		**未被参考之符号的通知**	**9.3.5**
10		**缩小边界对齐的空区域**	**9.3.6**
11	优化选项	连接时的优化	9.4.1
12		优化选项的子选项	
13		统一常数/字符串	9.4.2
14		删除未被参考的变量/函数	9.4.3
15		优化寄存器保存/恢复代码	9.4.5
16		统一公用代码	9.4.6
17		优化转移指令	9.4.8
18		禁止部分优化	9.4.10
19		确认优化结果	9.4.11

9.1 输入/输出选项

9.1.1 输入选项

♦ **描述**

优化连接编辑程序可以根据用户的使用方式输入下列四种文件。
这是其中一种可方便您操作的特性。

♦ **指定方法**

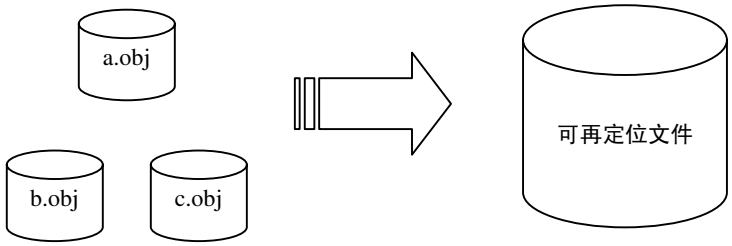
对话框菜单： **连接/程序库 (Link/Library) 标签类别： [输入] 显示有关项目([Input] Show entries for):**
命令行： *Input <子选项>:<文件名>*
Library<文件名>
Binary<子选项>:<文件名>

♦ **可用的输入文件**

文件种类	命令行
目标文件	input
可再定位文件	input
程序库文件	library
二进制文件	binary

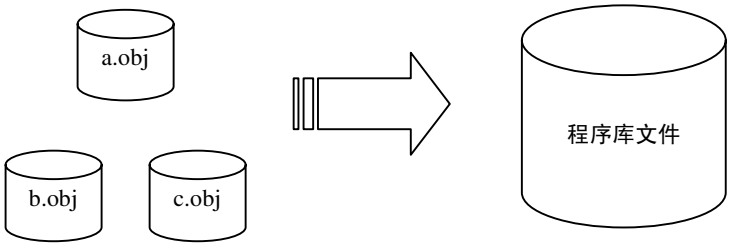
(1) 目标文件
从编译程序或汇编程序输出的普通文件。

(2) 可再定位文件
可再定位（地址未解析）的文件。
此文件包含一个或多个目标文件，以及使用输出选项从优化连接编辑程序生成。
可再定位文件中的符号**将会连接**，即使其他文件**并没有参考**它们。
因此请小心不要因为连接不必要的文件而增加 ROM 的大小。



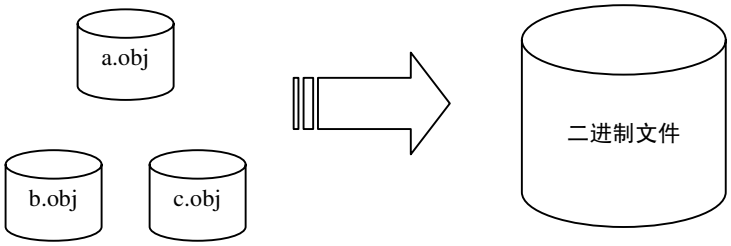
(3) 程序库文件

可再定位（地址未解析）的文件。
此文件包含一个或多个目标文件，以及使用输出选项从优化连接编辑程序生成。
可再定位文件中的符号**将不会连接**，如果其他文件**并没有参考**它们。



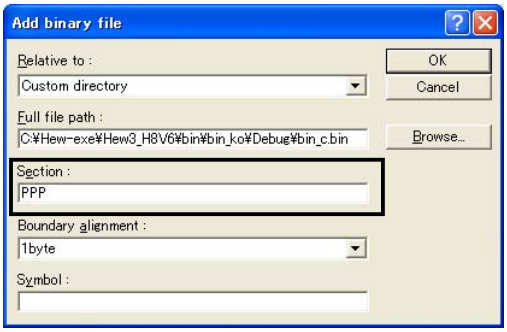
(4) 二进制文件

二进制文件可用于输入。
此文件包含一个或多个目标文件，以及使用输出选项从优化连接编辑程序生成。
输入二进制文件时，必须指定段名称。 此段名称使用**起始 (start)** 选项定位。
由于二进制文件没有调试信息，因此不能使用 C/C++ 源代码级调试程序。



[指定方法 1]

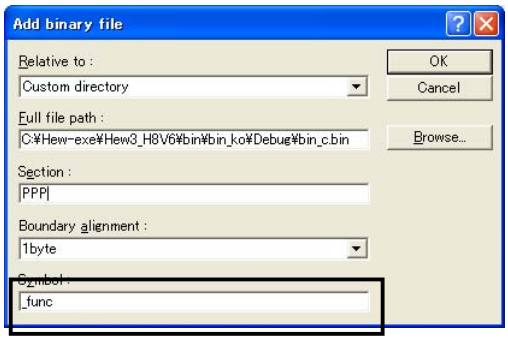
必须指定段名称。
对话框菜单： **连接/程序库 (Link/Library) 标签类别： [输入] 显示有关项目([Input] Show entries for)：**
二进制文件 (Binary files)
命令行： ***binary=bin_c.bin(PPP)***



[指定方法 2]

符号可以在二进制文件的开头处定义。
使用段名称指定符号名称，要执行此操作：
对于由 C/C++ 程序参考的变量名称，在符号名称的开头处添加一条下划线 (_)。

对话框菜单： **连接/程序库 (Link/Library) 标签类别： [输入] 显示有关项目([Input] Show entries for)：**
二进制文件 (Binary files)
命令行： ***binary=bin_c.bin(PPP,_func)***



[指定方法 3]

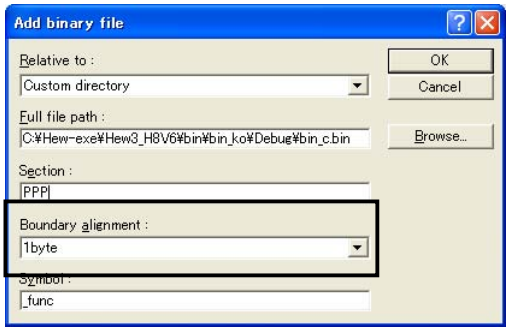
输入二进制文件时，可以指定边界对齐值。

如果边界对齐的指定被省略，将使用 1 作为默认值以便与旧版本兼容。
此边界对齐指定在优化连接编辑程序 9.0 或以上版本中有效。

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[输入] 显示有关项目([Input] Show entries for)：二进制文件 (Binary files)**

命令行： ***binary=bin_c.bin(PPP:<边界对齐>,_func)***

<边界对齐>: 1 | 2 | 4 | 8 | 16 | 32 (默认值：1)



9.1.2 输出选项

♦ 描述

一些类型的 ROM 写入程序只能输入 HEX 文件或 S 类型文件。
优化连接编辑程序可以根据用户的使用方式输出下列八种文件。
如有必要，用户可以更改输出文件的种类。

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[输出] 输出文件的类型 ([Output] Type of output file)：**
命令行： *form{ absolute | relocate | object | library=s | library=u |*
hexadecimal | stype | binary }

♦ **可用的输出文件**

编号	文件种类	命令行
1	绝对文件	form absolute
2	可再定位文件	form relocate
3	目标文件	form object
4	用户程序库文件	form library=s
5	系统程序库文件	form library=u
6	HEX 文件	form hexadecimal
7	S 类型文件	form stype
8	二进制文件	form binary

(1) 绝对文件

由优化连接编辑程序解析地址的文件。
由于此文件具有调试信息，因此可以使用 C/C++ 源代码级调试程序。
写入 ROM 时，必须将此文件转换为 S 类型格式或 HEX 或二进制。

(2) 可再定位文件

可再定位（地址未解析）的文件。
由于此文件具有调试信息，因此可以使用 C/C++ 源代码级调试程序。
要执行此文件，必须通过再次连接将此文件转换为绝对文件。

(3) 目标文件

此文件在使用提取选项将模块（目标）作为目标文件提取时使用。

使用命令行指定时，所需的目标文件可以通过此选项从指定的程序库文件提取。
使用 HEW 时，请在**连接/程序库 (Link/Library) 标签类别：[其他] 用户定义的选项 ([Other] User defined options)：**指定下列选项
[提取选项]
form=object
extract=<模块名称>

(4) 用户程序库/系统程序库

输出程序库文件。

(5) HEX 文件

输出 HEX 文件。
由于此文件没有调试信息，因此不能使用 C/C++ 源代码级调试程序。
要获取有关 HEX 文件的详细资料，请参考《SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册》(SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 的 18.1.2 HEX 文件格式。

(6) S 类型文件

输出 S 类型文件。
由于此文件没有调试信息，因此不能使用 C/C++ 源代码级调试程序。
要获取有关 S 类型文件的详细资料，请参考《SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册》(SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 的 18.1.1 S 类型文件格式。

(7) 二进制文件

输出二进制文件。

由于二进制文件没有调试信息，因此不能使用 C/C++ 源代码级调试程序。

9.2 列表选项

9.2.1 符号信息列表

♦ **描述**

除了连接映像信息，优化连接编辑程序还可以通过指定附加子选项来输出符号地址、大小和优化信息。

— 符号地址 **-ADDR**

— 大小 **-SIZE**

— 优化 **-OPT**（**ch-** 已改变，**cr-** 已创建，**mv-** 已移动）

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：|列表 (List)| 内容 (Contents)：显示符号(Show symbol)**

命令行：*list [=<文件名>]*

Show symbol

<*.map 文件>

```
*** Options ***
        :
*** Error information ***
        :
*** Mapping List ***
        :
*** Symbol List ***

SECTION=
FILE=                   START      END    SIZE
 SYMBOL                 (1)ADDR    (2)SIZE  INFO  COUNTS  (3)OPT

SECTION=P
FILE=C:\Hew-exe\Hew3_SHV9\bin\bin\Debug\bin.obj

                        00000800  00000821       22

    _main                          00000800      6      func ,g      *    ch
    _abort                         00000806      4      func ,g      *    ch
    _com_opt1                      0000080a      18     func ,g      *    cr ch

*** Delete Symbols ***
        :
*** Variable Accessible with Abs8 ***
        :
*** Variable Accessible with Abs16 ***
        :
*** Function Call ***
      :
```

9.2.2 符号参考计数

♦ **描述**

除了连接映像信息，优化连接编辑程序还可以通过指定附加子选项来输出静态符号参考计数。

— 符号参考计数 -COUNTS

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[列表 (List)] 内容 (Contents)：显示参考(Show reference)**
命令行： *list [=<文件名>]*
Show reference

<*.map>
*** Options ***
:
*** Error information ***
:
*** Mapping List ***
:
*** **Symbol List** ***

SECTION=
FILE= START END SIZE
SYMBOL ADDR SIZE INFO **(1)COUNTS** OPT

SECTION=P
FILE=C:\Hew-exe\Hew3_SHV9\bin\bin\Debug\bin.obj

00000800 00000821 22
_main
00000800 6 func ,g **1** ch
_abort
00000806 4 func ,g **0** ch
_com_opt1
0000080a 18 func ,g **2** cr ch
*** Delete Symbols ***
:
*** Variable Accessible with Abs8 ***
:
*** Variable Accessible with Abs16 ***
:
*** Function Call ***

9.2.3 交叉参考信息

• **描述**

除了连接映像信息，优化连接编辑程序还可以通过指定附加子选项来输出交叉参考信息。交叉参考信息可以让程序搜索全局符号被参考之处。
本地符号和静态符号将不会输出。

• 指定方法

对话框菜单：**连接/程序库（Link/Library）标签类别：[列表 (List)] 内容 (Contents)：显示交叉参考(Show cross reference)**

命令行： *list [=<文件名>]*
Show xreference

```
<*.map 文件>
*** Cross Reference List ***

No    Unit Name    Global Symbol    Location    External Information
(1)      (2)            (3)           (4)              (5)
---- ----------- --------------- -------- ---------------------
0001 test1
     SECTION=P
       _main
                                  00000100
  SECTION=B
       _sl1
                                  00007000 0001(0000011a:P)
       _sl2
                                  00007004 0001(0000010e:P)
                 _ret
                                  00007008 0001(00000128:P)
     SECTION=D
0002 test2
     SECTION=P
                 _func1
                                  0000015c 0001(00000124:P)
                 _func2
                                  00000164 0001(0000013c:P)
                 _func3
                                  00000170 0001(00000150:P)
```

• 每个项目的描述

(1) 单元号码是目标单元中的识别号码，在“外部信息”(External Information) (5) 中显示。
(2) 目标名称指定连接时的输入顺序。
(3) 符号名称以每一个段的递升顺序输出。
(4) 符号分配地址是将可再定位格式指定为输出文件格式 (form=relocate) 时，段起始处的相对值。
(5) 外部符号被参考时的地址
输出格式： <单元号码> (<段中的地址或偏移>:<段名称>)。

• 说明

此选项在优化连接编辑程序 9.0 或以上版本中有效。

9.3 有效选项

9.3.1 输出至未使用区

♦ 描述

优化连接编辑程序可以将任何数据输出至未使用区。
这对 ROM 转移非常有用，而且在程序异常中止时，通过不使用数据执行未使用区来检测异常中断也很有用。
1、2 或 4 字节值在输出数据大小中有效。 如果指定奇数数位，高位将使用 0 扩展，以便将它作为偶数数位使用。
输出数据的最大大小是 4 字节。 如果指定的值超过 4 字节，将使用底层的 4 字节。
此选项只在输出文件为 S 类型文件、二进制或 HEX 时可用。

♦ 指定方法

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[输出] 显示有关项目([Output] Show entries for)：**
指定要填入未使用区的值 (Specify value filled in unused area)
命令行：*space [=<数值>]*

♦ 实例：

(1) 通过以下方式分隔文件并指定使用数据填入未使用区的范围

连接/程序库 (Link/Library) 标签类别：[输出] 显示有关项目([Output] Show entries for)：分隔的输出文件 (Divided output files)
-output="C:\bin\Debug\a.bin"=00-0FFFF

(2) 通过以下方式指定数据的填入

连接/程序库 (Link/Library) 标签类别：[输出] 显示有关项目([Output] Show entries for)：指定要填入未使用区的值 (Specify value filled in unused area)
-space=FF

下页中的 <指定要填入未使用区的值（Specify value filled in unused area） [H’FF]> 实例显示在未使用区中填入数据的方式。

♦ **S 类型文件的实例**

如下例所示，0xFF 记录将添加到有数据的范围中的未使用区。
如果未指定此选项，没有数据的范围中的记录将不会输出。
如果指定了此选项，0xFF 记录将根据**分隔的输出文件 (Divided output files)** 输出选项中的输出范围指定，添加到没有数据的范围中的区域。

<未指定要填入未使用区的值 (NOT Specify value filled in unused area)>


```
                S00E000062696E20202020206D6F74C8
                S1070000000040DF4
                S1070014000041AC6
                S107001C000041CBC
                S10700200000041EB6
                S10700240000420B0
                S10700280000422AA
                S107002C000004424A4
                S107004000004268E
                S10700440000042888
                S10700480000042A82
                S107004C000042C7C
…               S1070050000042E76
                S10700540000043070

S11308901F9045EC7A00000008C67A01000008D2D7
S11308A04018010D6D0401006D0501006D0640064D
S11308B06C4A68EA0B061FD445F61F9045E40120F4
S10908C06D766D725470A8
S10F08C6000008DA000008DE00FFE42A4D
S10B08D200FFE00000FFE42A2E
S10708DA00FFE00A2D
S10F08DE7900000A6BA00000200C54708C
S9030400F8
```

<指定要填入未使用区的值 (Specify value filled in unused area [H'FF])>


```
S00E000062696E20202020206D6F74C8
S1070000000040DF4
S1130004FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8
S1070014000041AC6
S1070018FFFFFFFE4
S1070001C000041CBC
S10700200000041EB6
S10700240000420B0
S10700280000422AA
S1070002C000004424A4
S1130030FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCC
S107004000004268E
S10700440000042888
S10700480000042A82


…


S113FF8AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF73
S113FF9AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF63
S113FFAAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF53
S113FFBAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S113FFCAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF33
S113FFDAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S113FFEAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF13
S109FFFAFFFFFFFFFFFF03
S9030400F8
```

♦ **二进制文件的实例**

如下例所示，有数据的范围中的未使用区从 0x00 更改为 0xFF。
如果未指定此选项，没有数据的范围中的记录将不会输出。
如果指定了此选项，0xFF 记录将根据**分隔的输出文件 (Divided output files)** 输出选项中的输出范围指定，添加到没有数据的范围中的区域。

<未指定要填入未使用区的值 (NOT Specify value filled in unused area)>


```
000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 00 00 00 00  ................
0001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

…

```
0008a0  40 18 01 00 6D 04 01 00  6D 05 01 00 6D 06 40 06  @...m...m...m.@.
0008b0  6C 4A 68 EA 0B 06 1F D4  45 F6 1F 90 45 E4 01 20  lJh.....E...E..
0008c0  6D 76 6D 72 54 70 00 00  08 DA 00 00 08 DE 00 FF  mvmrTp..........
0008d0  E4 2A 00 FF E0 00 00 FF  E4 2A 00 FF E0 0A 79 00  .*.......*....y.
0008e0  00 0A 6B A0 00 00 20 0C  54 70                    ..k... .Tp
```

<指定要填入未使用区的值 (Specify value filled in unused area [H'FF])>


```
000100  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000110  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000120  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000130  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000180  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000190  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 FF FF FF FF  ................
0001c0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001d0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001e0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

…

```
00ffc0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffd0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffe0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00fff0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
010000
```

♦ **HEX 文件的实例**

如下例所示，0xFF 记录将添加到有数据的范围中的未使用区。
如果未指定此选项，没有数据的范围中的记录将不会输出。
如果指定了此选项，0xFF 记录将根据**分隔的输出文件 (Divided output files)** 输出选项中的输出范围指定，添加到没有数据的范围中的区域。

<未指定要填入未使用区的值 (NOT Specify value filled in unused area)>

```
:0400000000000400F8
:040014000000041ACA
:04001C000000041CC0
:040020000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C0000000424A8
:04004000000042692
:04004400000004288C
:040048000000042A86
:04004C000000042C80
```

…

```
:0C08C600000008DA000008DE00FFE42A51
:0808D20000FFE00000FFE42A32
:0408DA0000FFE00A31
:0C08DE007900000A6BA00000200C547090
:00000001FF
:0400000300000400F5
```

<指定要填入未使用区的值 (Specify value filled in unused area [H’FF])>

```
:0400000000000400F8
:10000400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
:040014000000041ACA
:04001800FFFFFFFFE8
:04001C000000041CC0
:040020000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C0000000424A8
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:04004000000042692
```

…

```
:FFFCF500FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:FFFDF400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:FFFEF300FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:0EFFF200FFFFFFFFFFFFFFFFFFFFFFFFFFFF0F
:00000001FF
:0400000300000400F5
```

♦ **说明**

此选项在优化连接编辑程序 8 或以上版本中有效。

9.3.2 S 类型文件的终止代码

♦ **描述**

在一些类型的 ROM 写入程序中，如果 S 类型文件的终止代码不是 s9 记录，运行时错误可能会在输入至 ROM 写入程序期间发生。
这是因为如果入口地址超过 0x10000，终止代码将会是 s7 或 s8。
通过指定此选项，终止代码将会永远是 s9。

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[其他 (Other)] 杂项 (Miscellaneous options)：**
始终在结束部分输出 S9 记录 (Always output S9 record at the end)
命令行： *s9*

♦ **说明**

要获取有关 S 类型文件的详细资料，请参考《SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册》(SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 的 18.1.1 S 类型文件格式。

9.3.3 调试信息压缩

♦ **描述**

通过指定此选项，装入文件到调试程序时的装入时间将会缩减。
但相反的，连接时间将会增加。

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[其他 (Other)] 杂项 (Miscellaneous options)：**
压缩调试信息 (Compress debug information)
命令行： *compress*
uncompress

♦ **说明**

此功能只在输出文件为绝对文件时有效。

9.3.4 连接时间缩减

♦ **描述**

指定此选项时，连接编辑程序会在连接时以较小的单元装入必要信息以缩减存储器的占用率。
因此，连接时间可以缩减。
在由于连接大型工程而导致处理缓慢，以及被连接编辑程序占用的存储大小超过所使用机器中的可用存储时，请尝试使用此选项。

♦ **指定方法**

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[其他 (Other)] 杂项 (Miscellaneous options)：**
连接期间使用低存储 (Low memory use during linkage)
命令行： *memory={high | low}*

♦ **实例：**

以下实例是指定或未指定此选项时的连接时间比较。
在以下例子中，连接时间缩减了 34 %。

<衡量条件>
♦ 1,000 个文件
♦ 每个文件 100 个符号
♦ 1,000 个函数符号
♦ 指定相同选项，除了此选项。

<memory = high>
111 秒

<memory = low>
73 秒

■ **说明**

此选项在优化连接编辑程序 8 或以上版本中有效。

9.3.5 未被参考之符号的通知

• 描述

当工程很大时，很难查找已经定义但未被参考的外部定义符号。

指定此选项时，未被参考的外部符号可以通过连接时输出消息予以通知。

要输出通知消息，消息选项 (*1) 也必须指定。

(*1) **连接/程序库 (Link/Library) 标签类别：[输出] [显示有关项目：] ([Output] [Show entries for :]) [输出消息] ([Output messages]) 压制的信息级消息 (Repressed information level messages)：**

• 指定方法

对话框菜单：**连接/程序库 (Link/Library) 标签类别：[输出] [显示有关项目：] ([Output] [Show entries for :]) [输出消息] ([Output messages]) 通知未使用的符号 (Notify unused symbol)**

命令行：*msg_unused*

• 输出消息

L0400 (I) Unused symbol “file”-“symbol”

file 中名为 **symbol** 的符号未被使用。

• 说明

(1) 此选项在优化连接编辑程序 9 或以上版本中有效。
(2) 在以下任何情形下，参考不会被正确分析，因此输出消息中显示的信息将会不正确。
- **–goptimize** 在汇编时未指定，而且具有转移到同个文件中的相同段。
- 具有同个文件中常数符号的参考。
- 在编译时指定优化时，具有即时从属函数的转移。
- 优化是在连接时指定且常数是统一的。

9.3.6 缩小边界对齐的空区域

• 描述

指定此选项时，为每个目标文件生成的段边界对齐之空区域，将会在连接时填写。

因此，由边界对齐生成的不必要空区域将会填写，缩小数据段的整体大小。

此选项会影响常数区域（C 段）、初始化的数据区域（D 段）以及未初始化的数据区域（B 段）。

• 指定方法

对话框菜单： **连接/程序库 (Link/Library) 标签类别：[输出] [显示有关项目：] ([Output] [Show entries for :])**

缩小边界对齐的空区域 (Reduce empty areas of boundary alignment)

命令行：*data_stuff*

• 实例：

下例显示边界对齐的空区域如何缩小

```
(file1.c)
short s1;
char c1;
```

```
(file2.c)
char c2;
```

<不指定 **data_stuff** 时>

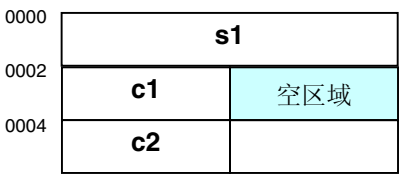
在不指定 **data_stuff** 时，边界对齐的一个字节空区域将在 **file1.c** 和 **file2.c** 之间生成，因为边界对齐的 SH CPU 指定的值是 4。

在此实例中，如果下一个要连接的顶端数据的大小是一个字节，就不需要此边界对齐。

但是，下一个文件的顶端数据是 2 个字节或更大，就应该在此文件 (**file1.c**) 的末端执行边界对齐。

因此，数据对齐和数据大小是

s1（2 字节）+ c1（1 字节）+ 空区域（1 字节）+ c2（1 字节）= 5 字节



<指定 **data_stuff** 时>

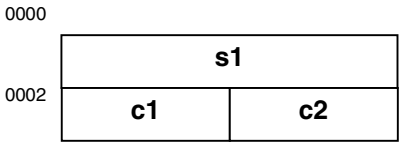
指定 **data_stuff** 时，如此例所示，如果下一个要连接的顶端数据的大小是一个字节，将不会生成边界对齐的空区域。

因此，数据对齐和数据大小是

s1（2 字节）+ c1（1 字节）+ c2（1 字节）= 4 字节

在这里，数据大小缩小为 4 字节。

在此程序实例中，空区域会在连接时填入段的边界对齐时生成。 然而，数据分配的顺序将不会改变。



• 说明

(1) 此选项在优化连接编辑程序 8.00.06 或以上版本中有效。
(2) 此选项的函数不适用于汇编程序生成的目标文件。
(3) 此选项的指定在下列任何情况下无效：
  • **library** 或 **object** 指定为优化连接编辑程序的输出格式
  • **absolute** 指定为优化连接编辑程序的输入格式
  • **memory=low** 已指定
  • 连接时的优化 (**optimize**) 已指定
(4) 优化将不会应用到指定此选项时所生成的可再定位文件的连接。

9.4 优化选项

9.4.1 连接时的优化

♦ **描述**

编译程序会在生成目标文件时输出补遗信息到每个模块。
优化连接编辑程序会根据这些补遗信息，执行在编译和连接时不可能执行的模块间优化。
因此，ROM 大小和执行速度都会获得增进。

♦ **指定方法**

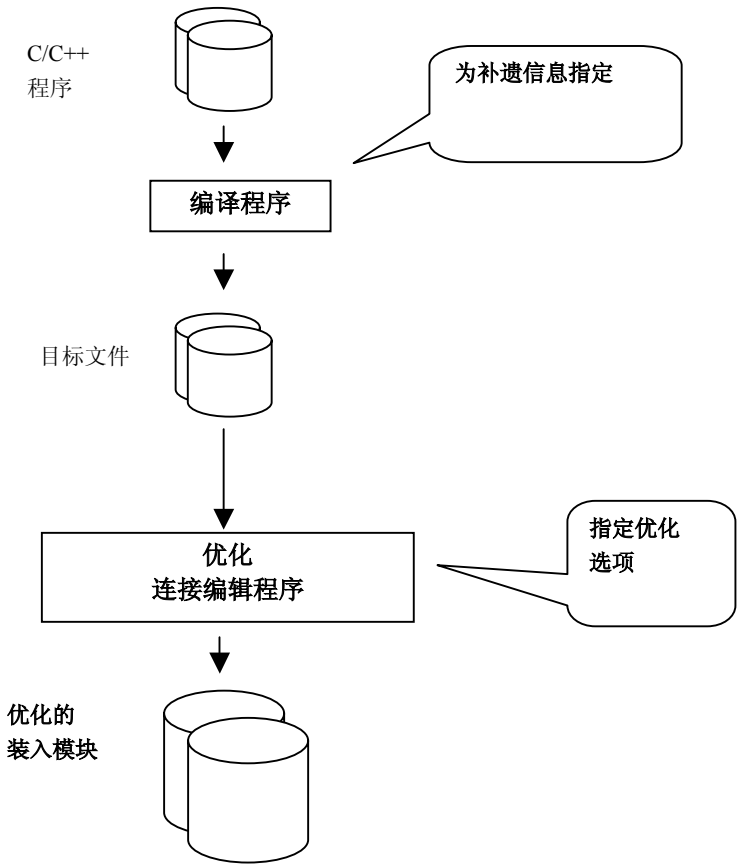
对话框菜单：**连接/程序库 (Link/Library) 标签类别： [优化] 优化项目 ([Optimize] Optimize items)**
命令行：*optimize=<子选项>*
<子选项> 在第 9.4.2 节到第 9.4.6 节中描述。

下列补遗信息的指定在编译时是必要的，即使并未指定连接时的优化。 不进行下列指定，连接时的优化将不可用。

♦ **补遗信息的指定方法**

对话框菜单：**C/C++ 标签类别：(C/C++ Tab Category:) [优化] 为模块间优化生成文件 ([Optimize] Generate file for inter-module optimization)**
命令行：*goptimize*

♦ 模块间优化流程



9.4.2 统一常数/字符串

大小	O	速度	-

♦ **描述**

相同值的常数和具有常数属性的相同字符串被跨模块统一。
此选项将删除常数段以增进大小。
速度将不会改变。

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) |优化| 优化项目(|Optimize| Optimize items)**
统一字符串 (Unify strings)
命令行：*optimize=string_unify*

♦ **相同值常数的实例**

const long 变量 **cl1**、**cl2** 具有相同常数值，因而被统一为一个常数。
这将缩小 ROM 大小的 4 字节。


```
(file1.c)
#include <machine.h>
const long cl1=100;
void main(void);
void func01(long);
long g_max;
void main(void)
{
        func01(cl1+1);
        func02(cl1+2);
        func03(cl1+3);
}
void func01(long c_litr)
{
        g_max = c_litr++;
}
```

```
(file2.c)
#include <machine.h>
const long cl2=100;
void main(void);
void func02(long);
void func03(long);
extern long g_max;

void func02(long c_litr)
{
        func03(cl2+c_litr);
        nop();
}
void func03(long c_litr)
{
        g_max = c_litr;
}
```

9.4.3 删除未被参考的变量/函数

大小	O	速度	-

♦ **描述**

从未被参考的变量/函数将会删除。指定此优化时，必须指定一个入口函数。没有指定入口函数，此优化将不会执行。
这是因为 CPU 会从向量表跳转至入口函数，而入口函数的优化或地址位于入口函数前面的函数将会更改跳转地址。

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) |优化| 优化项目([Optimize] Optimize items)**
删除死码 (Eliminate dead code)
命令行：*optimize=symbol_delete*

♦**入口函数的指定方法**

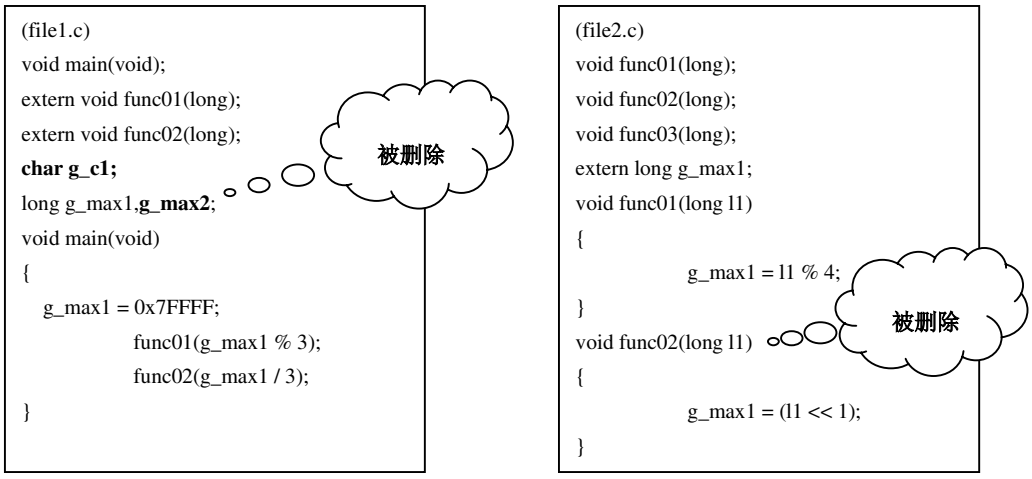
对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) |输入| 使用入口点 ([Input] Use entry point)**
命令行：*entry=<符号名称> | <地址>*

指定符号名称时，在符号名称的开头处添加一条下划线 (_)。

实例： main -> _main

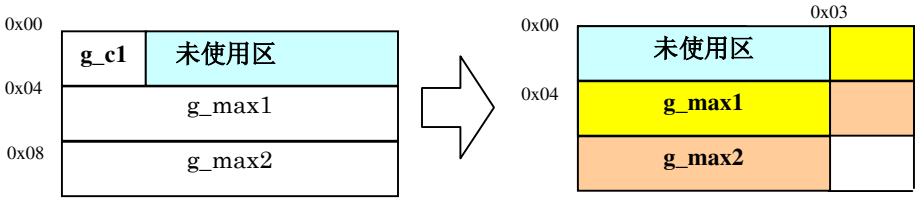
■ **删除未被参考的变量/函数的实例**

从未被参考的变量 **g_max2** 和函数 **func03** 将会删除。



char 类型变量 **g_c1** 从未被参考，但不会删除。
这是因为 SH 是 4 字节边界对齐，如果删除 **g_c1**，下一个变量的地址将不是 4 的倍数。
存取旧地址符号将会出现地址错误因为 CPU 的指定

[若 1 字节变量被删除]



若优化被执行，4 字节变量 **g_max1** 将会通过地址 0x03 存取。

9.4.4 优化寄存器保存/恢复代码

大小	○	速度	○

♦ **描述**

函数调用间的关系将被分析，且冗余的寄存器保存/恢复代码将通过此规格被删除。另外，根据函数调用之前和之后的寄存器状态，对所要使用的寄存器数量加以修改。

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [优化] 优化项目([Optimize] Optimize items)**
再分配寄存器 (Reallocate registers)
命令行：*optimize=register*

优化寄存器保存/恢复代码的实例

函数 **func1** 调用函数 **func2** 和 **func3**。

```
(file1.c)
void func1(int i1,int i2,int i3,int i4,int i5,long *l6)
{
   a = 0 * i1;
   b = 1 * i2;
   d = 4 * i4;
   h = 8 * i5;
   i = 9
   *l6 = b;
   func2(i,h,3000,200,100,l6);
   func3(i,h,3000,200,100,l6);
}
```

```
(file2.c)
void func2(int i1,int i2,int i3,int i4,int i5,long *l6)
{
   a = 0 * i1;
   b *= 1 * i2;
   d *= 4 * i4;
   f *= 6 / i2;
   g = 7 / i3;
   h = 8 * i5;
   i *= 9 / b ;
   *l6 = b * g;
}
```

```
(file3.c)
void func3(int i1,int i2,int i3,int i4,int i5,long *l6)
{
   a = 0 * i1;
   b *= 1 * i2;
   c = 2 * i3;
   d *= 4 * i4;
   e *= 5 * i1;
   f *= 6 / i2;
   g *= i2 / i3;
   h *= 8 * i5;
   i *= (9 / b) * ((*l6)++);
   *l6 *= b * g;
}
```

♦ 通过优化寄存器保存/恢复代码之代码的实例

优化前和优化后的代码实例如下所示。
由于父函数中的寄存器保存/恢复代码会添加，因此子函数中的寄存器保存/恢复代码将会减少。

在以下实例中，是 SH-1，
ROM 大小： 532 字节到 524 字节
执行速度： 718 周期到 711 周期

（优化前）
保存/恢复 R13-R14（**2 个寄存器**）

```
_func1:
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
              …
        MOV.L     @R15+,R14
        RTS
        MOV.L     @R15+,R13
```

（优化后）
保存/恢复 ER2-ER4（**7 个寄存器**）

```
_func1:
        MOV.L     R8,@-R15
        MOV.L     R9,@-R15
        MOV.L     R10,@-R15
        MOV.L     R11,@-R15
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
              …
        MOV.L     @R15+,R14
        MOV.L     @R15+,R13
        MOV.L     @R15+,R12
        MOV.L     @R15+,R11
        MOV.L     @R15+,R10
        MOV.L     @R15+,R9
        RTS
        MOV.L     @R15+,R8
```

保存/恢复 R10、R11、R12、R14
（**4 个寄存器**）

```
_func2:
        MOV.L     R10,@-R15
        MOV.L     R11,@-R15
        MOV.L     R12,@-R15
        MOV.L     R14,@-R15
              …
        MOV.L     @R15+,R14
        MOV.L     @R15+,R12
        MOV.L     @R15+,R11
        RTS
        MOV.L     @R15+,R10
```

无保存/恢复（**0 个寄存器**）

```
_func2:
        STS.L     PR,@-R15
              …
        LDS.L     @R15+,PR
        NOP
        RTS
        NOP
```

保存/恢复 R8-R14（**7 个寄存器**）

```
_func3:
        MOV.L     R8,@-R15
        MOV.L     R9,@-R15
        MOV.L     R10,@-R15
        MOV.L     R11,@-R15
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
              …
        MOV.L     @R15+,R14
        MOV.L     @R15+,R13
        MOV.L     @R15+,R12
        MOV.L     @R15+,R11
        MOV.L     @R15+,R10
        MOV.L     @R15+,R9
        RTS
        MOV.L     @R15+,R8
```

保存/恢复 R13、R14（**2 个寄存器**）

```
_func3:
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
              …
        MOV.L     @R15+,R14
        RTS
        MOV.L     @R15+,R13
```

9.4.5 统一公用代码

大小	O	速度	-

♦ **描述**

多个代表相同指令的字符串将被统一入一个子例程内，同时代码大小通过此规格获得缩减。
此优化将提高函数调用的内务操作并降低执行速度，因此需要小心处理。

具备相同统一代码的优化之最小代码大小可以指定。

若在编译时指定函数的内联扩展，此优化将不会执行，因为执行速度已经降低。

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：（Link/Library Tab Category:）[优化] 优化项目（[Optimize] Optimize items）**
删除相同代码（Eliminate same code）
命令行：*optimize=same_code*

♦ **统一大小的指定方法**

对话框菜单：**连接/程序库标签类别：（Link/Library Tab Category:）[优化] 删除的大小 ([Optimize] Eliminated size)**
命令行：*samesize=<size>*

♦ **实例： C 源程序**

函数 **func00** 和 **func01** 具有相同的表达式行。

```
(file1.c)
void main(void);
int func00(int,int,int);
extern int func01(int,int,int);
int ret;
void main(void)
{
        ret =  func00(10,11,12);
        ret += func01(20,21,22);
}
int func00(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

```
(file2.c)
void func01(void);
int func01(int,int,int);
int func01(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

♦ **实例：代码**

优化前和优化后的代码实例如下所示。
公用代码将会统一到新的函数 **_com_opt1** 中，该代码从原始位置调用。

在以下实例中，是 SH-1，
ROM 大小： 40 字节到 24 字节
执行速度： 46 周期到 60 周期

（优化前）

```
(file1.c)
_main:
        STS.L   PR,@-R15
        MOV     #12,R6
        MOV     #11,R5
        BSR     _func00
        MOV     #10,R4
        MOV.L   L13,R2
        MOV.L   L13+4,R1
        MOV.L   R0,@R2
        MOV     #22,R6
        MOV     #21,R5
        JSR     @R1
        MOV     #20,R4
        MOV.L   L13,R7
        MOV.L   @R7,R2
        ADD     R0,R2
        LDS.L   @R15+,PR
        RTS
        MOV.L   R2,@R7
_func00:
        ADD     #1,R6
        MOV     R6,R0
        ADD     #1,R5
        AND     R5,R0
        MOV     R0,R2
        AND     R6,R2
        ADD     R2,R0
        AND     R6,R2
        RTS
        ADD     R2,R0
```

公用代码

```
(file2.c)
_func01:
        ADD     #1,R6
        MOV     R6,R0
        ADD     #1,R5
        AND     R5,R0
        MOV     R0,R2
        AND     R6,R2
        ADD     R2,R0
        AND     R6,R2
        RTS
        ADD     R2,R0
```

（优化后）

```
(file1.c)
_main:
        STS.L   PR,@-R15
        MOV     #H'0C,R6
        MOV     #H'0C,R5
        BSR     _func00
        MOV     #H'0A,R4
        MOV.L   P_00001034,R2
        MOV.L   P_00001038,R1
        MOV.L   R0,@R2
        MOV     #H'16,R6
        MOV     #H'15,R5
        JSR     @R1
        MOV     #H'14,R4
        MOV.L   P_00001034,R7
        MOV.L   @R7,R2
        ADD     R0,R2
        LDS.L   @R15+,PR
        RTS
        MOV.L   R2,@R7
_func00:
        STS.L   PR,@-R15
        BSR     _com_opt1
        NOP
        LDS.L   @R15+,PR
        RTS
        ADD     R2,R0
_com_opt1:
        ADD     #1,R6
        MOV     R6,R0
        ADD     #1,R5
        and     R5,R0
        MOV     R0,R2
        AND     R6,R2
        ADD     R2,R0
        AND     R6,R2
        RTS
        NOP
```

新函数

```
(file2.c)
_func01:
        STS.L   PR,@-R15
        BSR     _com_opt1
        NOP
        LDS.L   R15+,PR
        RTS
        ADD     2,R0
```

9.4.6 优化转移指令

大小	○	速度	○

■ **描述**

存取其他文件中的函数时，以及在可通过 PC 相对寻址模式 (BSR) 存取的地址范围 (*1) 中存取时，C/C++ 编译程序通过绝对寻址模式 (JSR) 调用函数。
因为优化连接编辑程序执行连接时的优化，它可以重新计算其他文件中的转移目标的转移范围。
若有可能，可以将转移指令更改为 PC 相对寻址模式 (BSR)。

虽然原始转移范围超出可通过 PC 相对寻址模式存取的地址范围，如果转移范围通过其他优化缩减，仍然可以将转移指令更改为 BSR。

若任何其他优化项目被执行，不管此优化是否被指定，它将始终被执行。

注意：1. 可通过 PC 相对寻址模式存取的地址范围： -4096 至 4094 字节

■ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [优化] 优化项目([Optimize] Optimize items)**
优化转移 (Optimize branches)
命令行：*optimize=branch*

■ **实例：C 源程序**

函数 **main** 调用函数 **func01**。

```
(file1.c)
long func01(long,long);
void main(void);
long g_l1,g_l2;
void main(void)
{
   g_l1 = 100;
   g_l2 = 200;
            g_l1 = func01(g_l1,g_l2);
}
                        :
                        :
long func01(long l1,long l2)
{
            return l1 + l2;
}
```

♦ **实例：代码**

优化前和优化后的代码实例如下所示。
函数 **func01** 通过 **BSR** 调用。

在以下实例中，是 SH-1，
ROM 大小： 46 字节到 42 字节
执行速度： 22 周期到 21 周期

（优化前）

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #-56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        MOV.L     L13+8,R3
                  JSR       @R3
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
        .DATA.L   _func01
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```

（优化后）

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #--56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        NOP
        BSR       _func01
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```

9.4.7 禁止部分优化

♦ **描述**

不要优化连接编辑程序优化某些变量或函数时，可以将该变量或函数如下指定。
可以通过符号名称和通过地址范围禁止。

♦ **禁止删除未被参考的符号**

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [优化] 禁止项目([Optimize] Forbid item)**
死码的删除 (Elimination of dead code)
命令行：*symbol_forbid=<符号名称>*

♦ **禁止公用代码的统一**

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [优化] 禁止项目([Optimize] Forbid item)**
相同代码的统一 (Elimination of same code)
命令行：*samecode_forbid=<函数名称>*

♦ **禁止优化时的地址范围**

♦ **指定方法**

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [优化] 禁止项目([Optimize] Forbid item)**
存储器分配在 (Memory allocation in)
命令行：*absolute_forbid=<地址>[+大小]*

9.4.8 确认优化结果

♦ **描述**

优化连接编辑程序的优化结果可以如下确认。

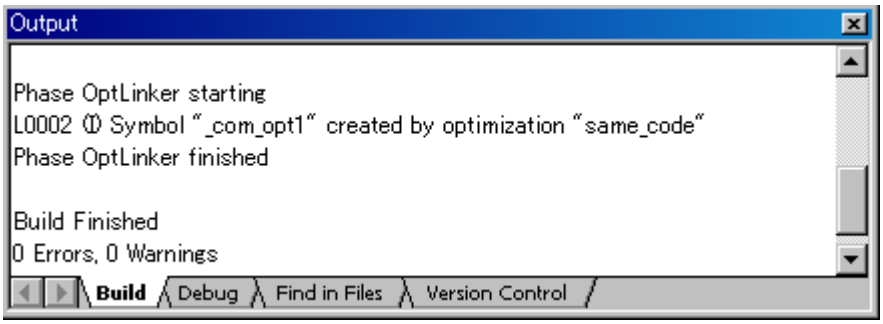
♦ **通过消息确认**

使用 HEW 时，可以通过不在以下对话框中核选，来输出优化结果。

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [输出] 显示有关项目([Output] Show entries for)：**
压制的信息级消息(Repressed information level messages)
命令行：*message[=<错误编号>]>*
: *nomessage*

♦ **消息输出的实例**

以下实例显示新函数已通过统一公用代码创建。



♦ **通过列表确认**

优化结果通过指定下列选项确认。
有关详情，请参考第 9.2.1 节“符号信息列表”。

对话框菜单：**连接/程序库标签类别：(Link/Library Tab Category:) [列表] 内容：([List] Contents :) 符号 (Symbol)**
命令行：*list [=<文件名>]*
Show symbol

SuperH RISC Engine C/C++编译程序应用笔记

MISRA C

第 10 节　MISRA C

10.1　MISRA C

10.1.1　什么是 MISRA C?

MISRA C 是指 Motor Industry Software Reliability Association (MISRA) 于 1998 年发行的 C 语言之使用指导，以及由这些指导标准化的 C 编写规则。 C 语言本身非常有用，但存在一些特殊问题。 MISRA C 指导将这些问题划分成五种类型：程序设计员错误、关于语言的误解、意外的编译程序运算、执行时的错误，以及编译程序本身的错误。 MISRA C 的目的是克服这些问题的同时，提高 C 语言的安全使用。 MISRA C 包含两种类型的 127 条规则： *必需*和 *咨询*。 代码开发应该致力于符合所有的这些规则，但因为这有时候难以实现，同时在不遵守规则时也需要确认过程和证明时间。 不同事项的遵从也需要从这些规则中划分，例如需要测量软件公制时。

10.1.2　规则实例

本小节将介绍一些实际的 MISRA C 规则。图 10.1 显示规则 62：所有 switch 语句应该包含最终默认子句。 它被分类为程序设计员错误。 在 switch 语句中，如果“default”标签被错误拼写为“defalt”，编译程序将不会把它当着是一个错误。如果程序设计员没有注意到这个错误，预期的默认操作将永远不会执行。这个问题可以通过应用规则 62 予以避免。

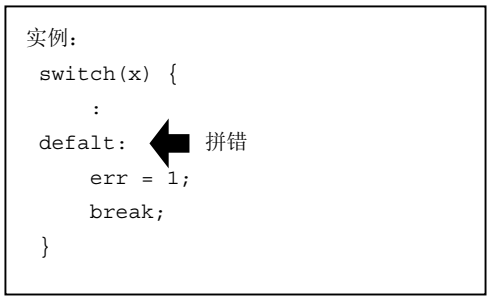


图 10.1 规则 62

图 10.2 显示规则 46：表达式的值在标准许可的任何求值顺序下应该是一样的。它被分类为关于语言的误解。换句话说，如果 ++i 先求值，表达式将变成 2+2，但如果是 i 先求值，表达式将变成 2+1，同样的，由于不具备函数参数之求值顺序的规定，如果 ++j 先求值，表达式将变成 f(2,2)，但如果是 j 先求值，表达式将变成 f(1,2)。 这个问题可以通过应用规则 46 予以避免。

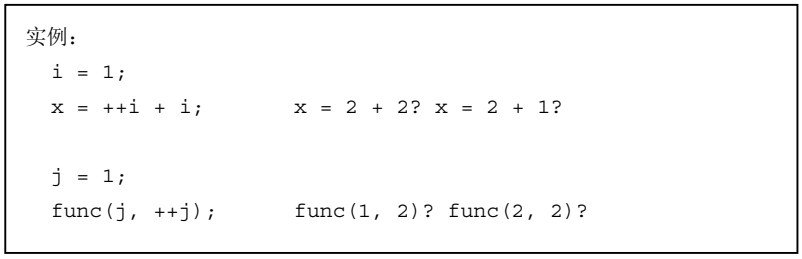


图 10.2 规则 46

图 10.3 显示规则 38：移位运算符的右操作数应该在 "0" 和（左操作数的位宽度 -1）之间。它被分类为意外的编译程序运算。在 ANSI 中，如果位移运算符的移位数字是一个负数或大于要移位的对象之大小，计算结果将不明确。在图 10.3 中，如果 us 在移位时的移位数字不在 0 和 15 之间，结果将不明确，而且值会根据编译程序而有所不同。这个问题可以通过应用规则 38 予以避免。

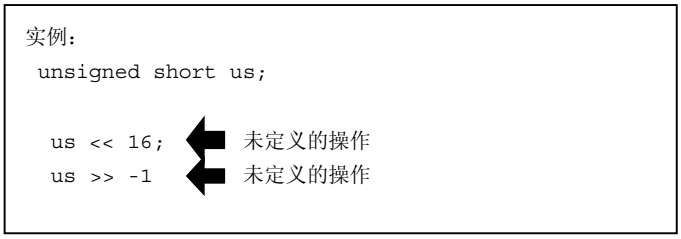


图 10.3 规则 38

图 10.4 显示规则 51：常数无符号整数表达式的计算法不应该导致环绕式。它被分类为执行时的错误。当无符号整数计算的结果是理论上的负数时，很难确定是否应该预期一个理论上的负数，或是一个无符号的计算结果就足够了。此情形将会导致故障。此外，加法计算的结果也可能导致溢出，形成一个非常小的值。这个问题可以通过应用规则 51 予以避免。

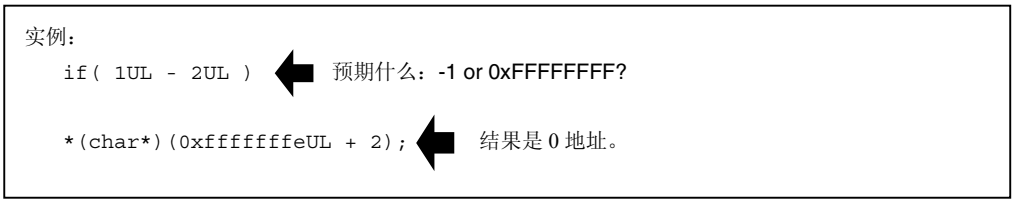


图 10.4 规则 51

10.1.3 遵从矩阵

使用 MISRA C，将会检查源代码是否符合所有 127 条规则。此外，需要制作如表 10.1 所示的表，显示是否维持每项规则。它称为*遵从矩阵*。由于视觉检查所有规则存在一定困难，我们建议您使用静态检查工具。MISRA C 指导也指出，使用工具来遵守规则是非常重要的。由于不是每个规则都可以使用此类工具检查，您需要执行视觉审查来以视觉的方式检查这些规则。

表 10.1 遵从矩阵

规则编号	编译程序	工具 1	工具 2	审查（视觉）
1	警告 347			
2		违例 38		
3			警告 97	
4				通过
...	...	...	...	...

10.1.4 规则违例

规则违例包含那些已知是安全的，以及那些可能具有更多影响的。诸如之前的违例应该可以接受，但太轻易接受规则违例将导致遗失某个程度的安全性。这就是为什么 MISRA C 需要规定接受规则违例的特殊程序。这些违例需要有效理由，以及验证该违例是否安全。 因此，所有可接受规则的位置和有效理由都清楚记录。 这样一来，违例将不会如此轻易被接受，组织中拥有适当权限的人员将会在咨询专家后在这类记录文档上签名。这意味着当一个和已接受之规则一样的规则违例时，它将被视为“已接受的规则违例”，并且可以当作是已接受的，而且不需要再次执行上述程序。 当然，这类违例需要定时审查。

10.1.5 MISRA C 的遵从

为了鼓励 MISRA C 的遵从，需要将代码开发为遵从规则，而规则违例问题也需要解决。 若要显示代码使用规则编译，需要具足遵从矩阵和已接受规则违例的文档，以及每个规则违例都有附带签名。为了预防将来发生问题，您应该训练程序设计员充分利用 C 语言和所使用的工具、执行关于编写样式的政策、选择适当的工具，以及测量各类软件公制。 这些工作应该正式标准化，以及附带适当文档。 MISRA C 的遵从比只是根据指导开发个别产品所需要注意的事项更多，它还关系组织本身的利益。

10.2 SQMlint

10.2.1 什么是 SQMlint？

SQMlint 是一个套件，为 Renesas C 编译程序提供附加功能用于检查它是否遵从 MISRA C 规则。 SQMlint 会静态检查 C 源代码，以及报告违反规则的区域。 在 Renesas 产品开发环境中，SQMlint 作为 C 编译程序的一部分运行。 SQMlint 可以通过在编译时添加选项简易启动，如图 10.5 所示。 它不会影响编译程序所生成的代码。

表 10.2 列出 SQMlint 支持的规则。

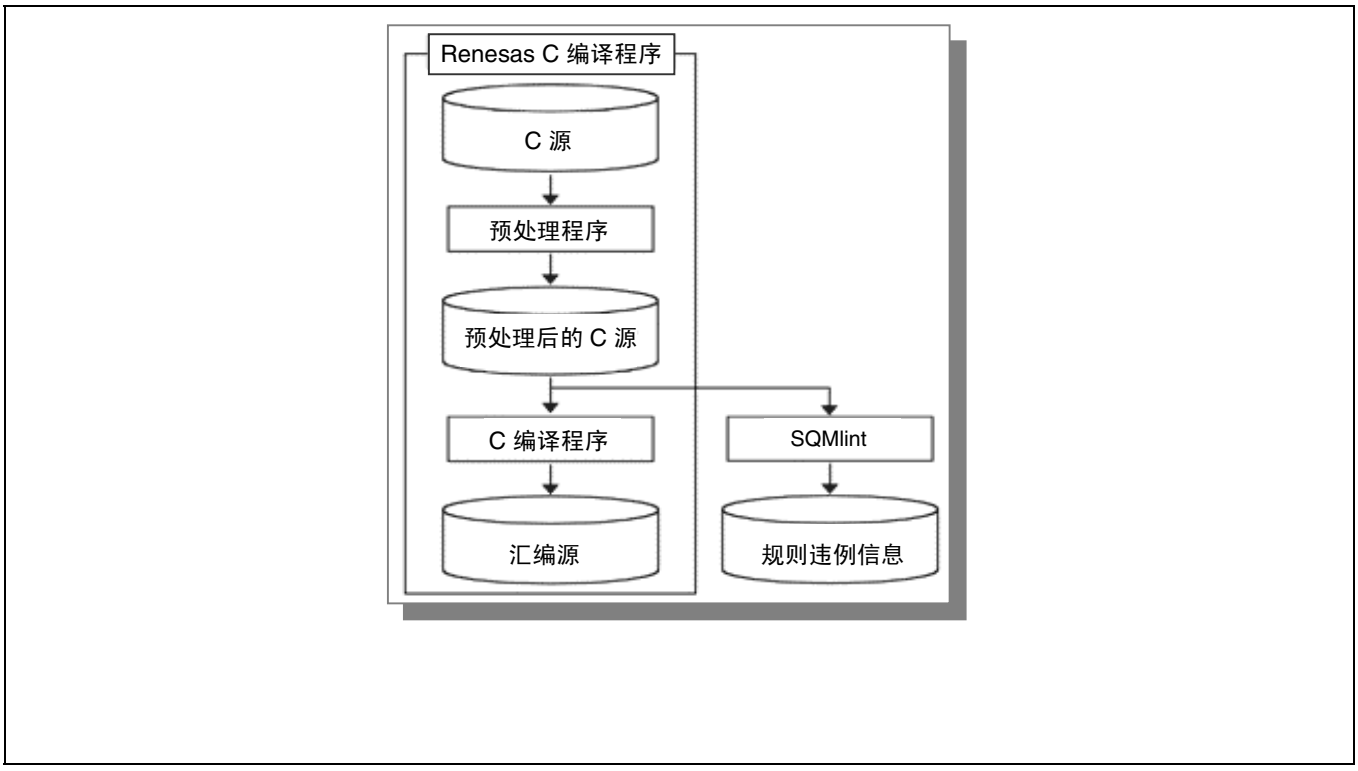


图 10.5 SQMlint 的定位

表 10.2　SQMlint 支持的规则

规则	测试	规则	测试	规则	测试	规则	测试	规则	测试	规则	测试
1	○	26	×	51	○*	76	○	101	○	126	○
2	×	27	×	52	×	77	○	102	○	127	○
3	×	28	○	53	○	78	○	103	○		
4	×	29	○	54	○*	79	○	104	○		
5	○	30	×	55	○	80	○	105	○		
6	×	31	○	56	○	81	×	106	○*		
7	×	32	○	57	○	82	○	107	×		
8	○	33	○	58	○	83	○	108	○		
9	×	34	○	59	○	84	○	109	×		
10	×	35	○	60	○	85	○	110	○		
11	×	36	○	61	○	86	×	111	○		
12	○	37	○	62	○	87	×	112	○		
13	○	38	○	63	○	88	×	113	○		
14	○	39	○	64	○	89	×	114	×		
15	×	40	○	65	○	90	×	115	○		
16	×	41	×	66	×	91	×	116	×		
17	○*	42	○	67	×	92	×	117	×		
18	○	43	○	68	○	93	×	118	○		
19	○	44	○	69	○	94	×	119	○		
20	○	45	○	70	○*	95	×	120	×		
21	○*	46	○*	71	○	96	×	121	○		
22	○*	47	×	72	○*	97	×	122	○		
23	×	48	○	73	○	98	×	123	○		
24	○	49	○	74	○	99	○	124	○		
25	×	50	○	75	○	100	×	125	○*		

○: 可测试　×: 不可测试　*: 具有限制的可测试

表 10.3　SQMlint 支持的规则数量

规则类别	可测试规则的数量（SQMlint 支持/总计）
必需	67/93
咨询	19/34
总计	86/127

10.2.2 使用 SQMlint

SQMlint 启动选项可以从设定“HEW 编译程序选项”的窗口简易设定。 图 10.6 显示用于指定 HEW 选项的对话框，其中 [MISRA C 规则检查 (MISRA C rule check)] 应该从 [类别 (Category)] 选取。

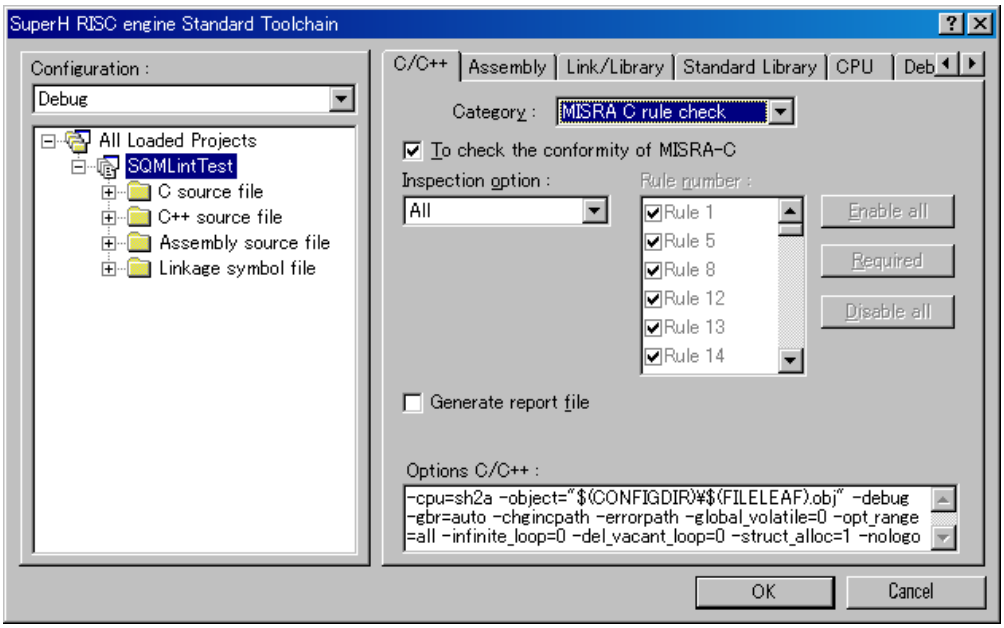


图 10.6 HEW 选项窗口

如此，SQMlint 将会在编译时间启动。 此对话框中 [检查选项] ([Inspection Option]) 的意义是:

- [全部] ([All])：对所有规则执行测试。
- [必需] ([Required])：根据 MISRA C 规则，仅对需要的规则执行测试。
- [定制] ([Custom])：对用户所指定的规则执行测试。 请使用右边的复选框和按钮选择规则。

10.2.3 查看测试结果

测试结果可以通过下列三种方式输出：

(a) 标准错误输出

信息以 HEW 编译错误相同的方式输出。 您可以通过双击信息，或右击信息并选择 [跳转] ([Jump]) 来执行标签跳转。源代码可以使用改正编译错误相同的操作来简易地进行修正。

请注意，右击信息并选择 [帮助] ([Help])时将可以显示说明。

(b) CSV 文件

电子表格软件可以读取的文件格式，使审查工作更简易执行。

(c) SQMmerger

SQMmerger 是一个用来将 C 源文件和 SQMlint 生成的 CSV 格式化的报告文件，合并到包含 C 源行及其关联报告信息的文件的工具。

要执行 SQMmerger，请使用下列命令输入格式：

```
sqmmerger -src <c 源文件名> -r <报告文件名> -o <输出文件名>
```

同时显示源文件和测试结果，如图 10.7 所示。

```
1 : void func(void);
2 : void func(void)
3 : {
4 : LABEL:
  [MISRA(55) Complain] label ('LABEL') should not be used
5 :
6 : goto LABEL;
    [MISRA(56) Complain] the 'goto' statement shall not be used
7 : }
```

图 10.7 SQMmerger

10.2.4 开发程序

图 10.8 显示如何使用 SQMlint 执行开发。

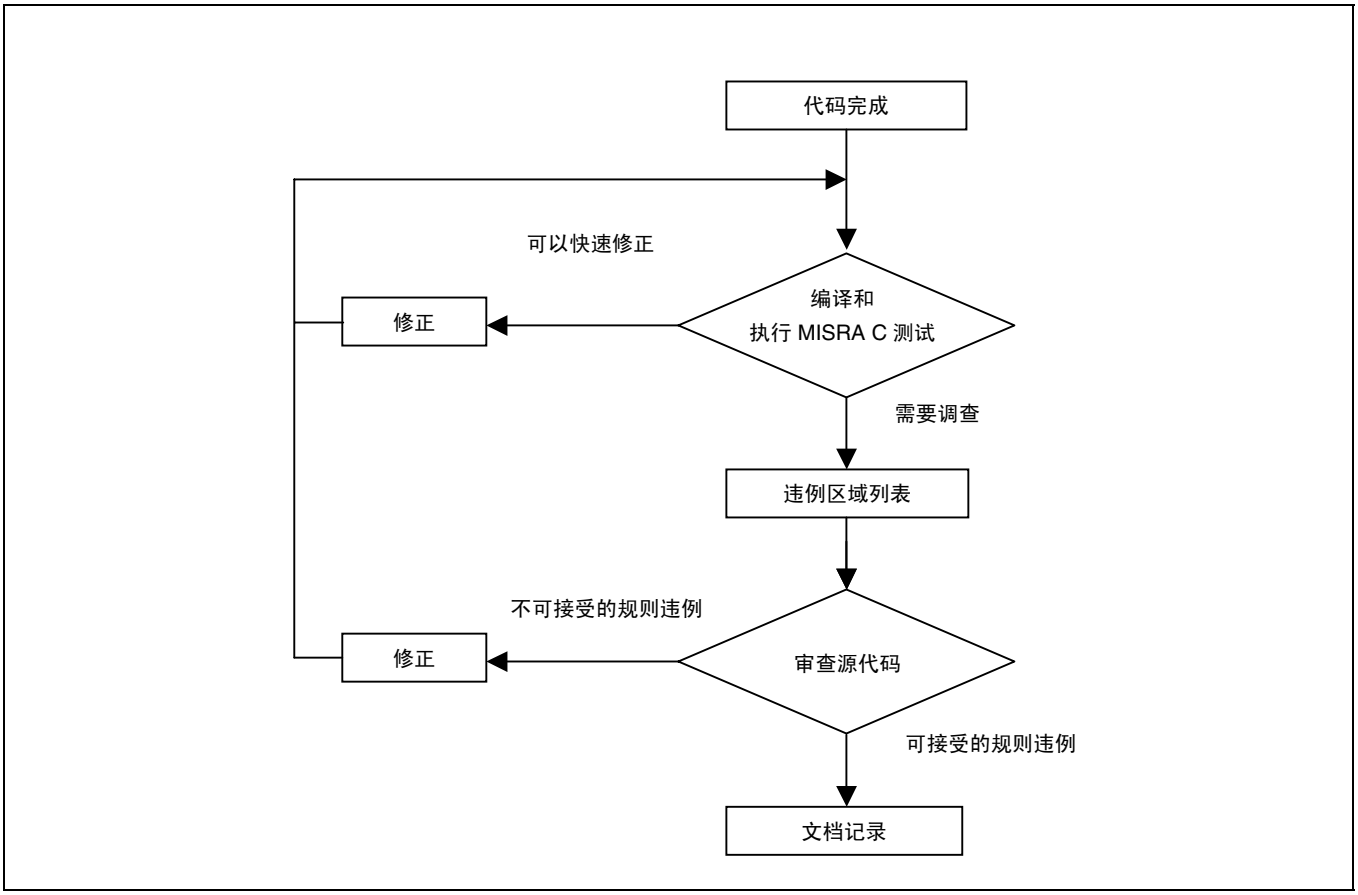


图 10.8 使用 SQMlint 的开发程序

– 收集所有编译错误。 SQMlint 假设 C 源代码是有效的。
– 查找 SQMlint 所检测的错误。
– 修正可简易修正的错误。
– 创建需要调查的规则违例的位置列表，以及执行审查。
– 对审查时判断为不可接受的规则执行修正。
– 记下审查时判断为可接受的规则，保留一份记录。

10.2.5 支持的编译程序

SQMlint 支持下列编译程序：

- SHC/C++ 编译程序封装 9.00 版本 00 和以上版本

10.2.6 可以通过 SHC/C++ 编译程序检查的规则

下列规则不能通过 SQMlint 检查，但这些规则的违例可以通过 SHC/C++ 编译程序信息检测。

表 10.4 可以使用 SHC/C++ 编译程序检查的规则

规则编号	规则描述	SHC/C++ 编译程序信息
9	注解将不会嵌套。	C5009 (I) 不允许已嵌套的注解(Nested comment is not allowed) 存在已嵌套的注解。
26	当某个目标或函数声明超过一次时，这些声明将会兼容。	C2136 (E) 类型不符(Type mismatch) 具有 extern 或 static 存储器类的变量或函数已声明超过一次，但类型不符。
52	所有语句将可以到达。	C0003 (I) 无法到达的语句(Unreachable statement) 存在一个将不会执行的语句。

第 11 节　问题解答

本节提供用户常见问题的解答。

11.1　C/C++ 编译程序/汇编程序

11.1.1　const 声明

问题

我已执行 const 声明，但无法将它分配到常数区 (C) 段。

解答

使用 const 声明符号会产生下列结果。

(1) const char msg[]="sun";
C 段分配： 字符串 "sun"

(2) const char *msg[]=("sun", "moon");
C 段分配： 字符串 "sun" 和 "moon"
D 段分配： msg[0] 和 msg[1]
（*msg[0] 和 *msg[1] 的前导地址）

(3) const char *const msg[]=("sun", "moon");
C 段分配： 字符串 "sun" 和 "moon"，msg[0] 和 msg[1]
（*msg[0] 和 *msg[1] 的前导地址）

(4) char *const msg[]=("sun", "moon");
C 段分配： 字符串 "sun" 和 "moon"，msg[0] 和 msg[1]
（*msg[0] 和 *msg[1] 的前导地址）

11.1.2　一位数据的正确求值

问题

我尝试确定某位是否已在位字段中设定，但在一些情形下无法正确对位进行求值。

解答

若一位数据被声明为带符号，位本身会被解释为该符号位。

因此，使用单位表示的值是“0”和“-1”。

为了表示“0”和“1”值，请确定将该数据声明为无符号。

实例：

永远不正确的求值

```
struct{
        char  p7:1;
        char  p6:1;
        char  p5:1;
        char  p4:1;
        char  p3:1;
        char  p2:1;
        char  p1:1;
        char  p0:1;
}s1;

if(s1.p0 == 1){
    s1.p1 = 0;
}
```

正确的求值

```
struct{
        unsigned char  p7:1;
        unsigned char  p6:1;
        unsigned char  p5:1;
        unsigned char  p4:1;
        unsigned char  p3:1;
        unsigned char  p2:1;
        unsigned char  p1:1;
        unsigned char  p0:1;
}s1;

  if(s1.p0 == 1){
     s1.p1 = 0;
}
```

注意：编写 if 语句的条件时，若使用 0 比较，将会生成更有效率的代码。

11.1.3 安装

问题

我输入了编译程序、汇编程序或连接程序的命令，但它们无法启动。

解答

检查编译程序、汇编程序和连接程序的安装目录是否包含在“PATH”环境变量中。

要从 DOS 窗口启动编译程序，请先设定下列环境：

(1) 设定路径 (PATH)
将路径 (PATH) 选项设定到可用工具的所在位置。

实例：

```
c:\> PATH=%PATH%; C:\Hew3\Tools\Renesas\Sh\9_0_0\bin (RET)
```

这应被添加到现有路径(PATH)。

(2) 设定 SHC_LIB
表示保存 SuperH RISC engine C/C++ 编译程序之主要文件的地点。 此设定不能省略。

实例：

```
c:\> set SHC_LIB=C:\Hew3\Tools\Renesas\Sh\9_0_0\bin (RET)
```

(3) 设定 SHC_TMP
指定创建 C/C++ 编译程序使用的临时文件的路径。 此设定不能省略。

实例：

```
c:\> set SHC_TMP=C:\tmp
```

(4) 设定 SHC_INC
此环境变量在从指定路径读取 C/C++ 编译程序的标准标头文件时设定。 可在路径之间使用逗号（“,”）隔开来指定多个路径。 如果不设定此环境变量，标准标头文件将会从 SHC_LIB 读取。

实例：

```
c:\> set SHC_INC=C:\Hew3\Tools\Renesas\Sh\9_0_0\include
```

11.1.4 运行时例程指定和执行速度

问题

请告诉我关于编译程序所提供的运行时例程的速度。

解答

下表列出使用内部 ROM 和 RAM 时的运行时例程速度/FPL 速度。 有关运行时例程的命名规则，请参考附录 A“运行时例程的命名规则”。 用于创建程序库的选项如下：

表 11.1 程序库创建选项

	cpu	Pic	endian	denormaliaztion	round	fpu	double=float
SH-1	sh1	-	big	-	-	-	无
SH-2	sh2	1	big	-	-	-	无
SH-2A	sh2a	1	big	-	-	-	无
SH-3	sh3	1	big	-	-	-	无
SH-4	sh4	0	big	off	zero	无	-
SH-4A	sh4a	0	big	off	zero	无	-

表 11.2 运行时例程速度/FPL 速度列表 (1)

编号	类型		函数名称	堆栈大小	执行周期数					
					SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
1.1	整数运算	乘	_muli	12	25	-	-	-	-	-
2.1		除	_divbs	4	14	15	-	11	7	8
2.2			_divbu	0	10	10	-	7	6	6
2.3			_divws	4	18	15	-	9	7	9
2.4			_divwu	0	13	13	-	8	7	7
2.5			_divls	8	39	41	-	27	20	19
2.6			_divlsp	12	-	82	-	-	-	-
2.7			_divlspnm	8	-	55	-	-	-	-
2.8			_divlu	4	33	35	-	24	17	15
3.1		余数	_modbs	8	16	17	-	11	7	8
3.2			_modbu	4	11	11	-	7	6	6
3.3			_modws	8	17	14	-	9	7	9
3.4			_modwu	4	12	12	-	8	7	7
3.5			_modls	12	46	48	-	30	23	18
3.6			_modlsp	12	-	82	-	-	-	-
3.7			_modlspnm	8	-	55	-	-	-	-
3.8			_modlu	8	33	35	-	24	17	15

表 11.2 运行时例程速度/FPL 速度列表 (2)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
4.1	浮点运算	加	_adds	24	91	103	43	69	–	–
4.2			_addd_a	44	149	179	90	117	–	–
5.1		后增量	_poas	44	101	113	43	76	–	–
5.2			_poad	84	167	196	89	129	–	–
6.1		减	_subs	24	105	117	43	79	–	–
6.2			_subdr	44	168	198	86	129	–	–
7.1		后减量	_poss	44	112	123	43	82	–	–
7.2			_posd	84	195	222	120	146	–	–
8.1		乘	_muls	24	76	93	34	62	–	–
8.2			_muld_a	64	139	183	71	122	–	–
9.1		除	_divs	20	78	90	35	59	–	–
9.2			_divdr	60	169	205	83	133	–	–
10.1		比较	_eqs	20	67	76	10	51	–	–
10.2			_eqd_a	32	97	115	44	74	–	–
10.3			_nes	20	57	66	7	45	–	–
10.4			_ned_a	32	82	100	37	64	–	–
10.5			_gts	20	60	69	18	48	–	–
10.6			_gtd_a	32	88	106	39	69	–	–
10.7			_lts	20	60	69	18	48	–	–
10.8			_ltd_a	32	88	106	39	69	–	–
10.9			_ges	20	60	69	18	48	–	–
10.10			_ged_a	32	87	105	38	68	–	–
10.11			_les	20	60	69	18	48	–	–
10.12			_led_a	32	87	105	38	68	–	–

表 11.2 运行时例程速度/FPL 速度列表 (3)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
11.1	转换符号	_negs	0	8	9	5	6	-	-
11.2		_negd_a	12	30	39	15	26	-	-
12.1	转换	_stod_a	12	66	75	34	51	-	-
12.2		_dtos_a	20	120	126	59	81	-	-
12.3		_stoi	12	360	249	21	165	-	-
12.4		_dtoi_a	20	252	224	116	134	-	-
12.5		_stou	12	360	249	21	165	-	-
12.6		_dtou_a	20	252	224	116	134	-	-
12.7		_itos	12	299	273	38	161	-	-
12.8		_itod_a	12	309	286	155	170	-	-
12.9		_utos	8	287	259	33	152	-	-
12.10		_utod_a	8	293	268	146	159	-	-
12.11		_utof	16	-	-	-	-	-	-
12.12		_ftou	8	-	-	-	-	-	-
13.1	设定位字段	_bfsbs	24 (16*[1])	84	91	-	-	-	-
13.2		_bfsbu	20 (16*[1])	44～91	44～91	-	-	-	-
13.3		_bfsws	24 (16*[1])	92	97	-	-	-	-
13.4		_bfswu	20 (16*[1])	77	84	-	-	-	-
13.5		_bfsls	24 (16*[1])	108	61～524	-	-	-	-
13.6		_bfslu	20 (16*[1])	106	44～259	-	-	-	-
14.1	参考	_bfxbs	8	58	61	-	-	-	-
14.2	位字段	_bfxbu	8	53	58	-	-	-	-
14.3		_bfxws	8	67	67	-	-	-	-
14.4		_bfxwu	8	51	56	-	-	-	-
14.5		_bfxls	8	72	76	-	-	-	-
14.6		_bfxlu	8	58	62	-	-	-	-

注意：1. 仅限于 SH-3

表 11.2 运行时例程速度/FPL 速度列表 (4)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
15.1	移动区域	_quick_evn_mvn	4	12+3*(n/4)					
15.2		_quick_mvn	8	17+3*(n/4) (n<=64)					
				24+1.625*(n/4) (n>=68)					
15.3		_quick_odd_mvn	4	12+3*(n/4)					
15.4		_slow_mvn	12	21+5*n+3*((n-1)/4)					
16.1	比较字符串	_quick_strcmp1	0	26+7*(n/4)+5*((n-1)%4)					
16.2		_slow_strcmp1	0	35+7*n					
17.1	复制字符串	_quick_strcpy	16	30+6*(n/4)+4*((n-1)%4)					
17.2		_slow_strcpy	24	24+6*n+2*((n-1)/4)					
18.1	左移	_sftl	4	18	19	-	-	-	-
18.2		_sta_sftl0	0	4	5	-	-	-	-
18.3		_sta_sftl1	0	5	6	-	-	-	-
18.4		_sta_sftl2	0	4	5	-	-	-	-
18.5		_sta_sftl3	0	7	8	-	-	-	-
18.6		_sta_sftl4	0	6	7	-	-	-	-
18.7		_sta_sftl5	0	8	9	-	-	-	-
18.8		_sta_sftl6	0	8	9	-	-	-	-
18.9		_sta_sftl7	0	10	11	-	-	-	-
18.10		_sta_sftl8	0	5	6	-	-	-	-
18.11		_sta_sftl9	0	6	7	-	-	-	-
18.12		_sta_sftl10	0	7	8	-	-	-	-
18.13		_sta_sftl11	0	8	9	-	-	-	-
18.14		_sta_sftl12	0	8	9	-	-	-	-
18.15		_sta_sftl13	0	9	10	-	-	-	-
18.16		_sta_sftl14	0	10	11	-	-	-	-
18.17		_sta_sftl15	0	11	12	-	-	-	-
18.18		_sta_sftl16	0	5	6	-	-	-	-
18.19		_sta_sftl17	0	6	7	-	-	-	-
18.20		_sta_sftl18	0	7	8	-	-	-	-
18.21		_sta_sftl19	0	8	9	-	-	-	-
18.22		_sta_sftl20	0	8	9	-	-	-	-
18.23		_sta_sftl21	0	9	10	-	-	-	-
18.24		_sta_sftl22	0	10	11	-	-	-	-
18.25		_sta_sftl23	0	11	12	-	-	-	-
18.26		_sta_sftl24	0	7	8	-	-	-	-
18.27		_sta_sftl25	0	8	9	-	-	-	-
18.28		_sta_sftl26	0	8	9	-	-	-	-
18.29		_sta_sftl27	0	10	11	-	-	-	-

表 11.2 运行时例程速度/FPL 速度列表 (5)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
18.30	左移	_sta_sftl28	0	10	11	-	-	-	-
18.31		_sta_sftl29	0	10	11	-	-	-	-
18.32		_sta_sftl30	0	8	9	-	-	-	-
18.33		_sta_sftl31	0	7	8	-	-	-	-
19.1	右移	_sftrl	4	18	19	-	-	-	-
19.2		_sftra	4	19	20	-	-	-	-
19.3		_sta_sftrl0	0	4	5	-	-	-	-
19.4		_sta_sftrl1	0	5	6	-	-	-	-
19.5		_sta_sftrl2	0	4	5	-	-	-	-
19.6		_sta_sftrl3	0	7	8	-	-	-	-
19.7		_sta_sftrl4	0	7	8	-	-	-	-
19.8		_sta_sftrl5	0	9	10	-	-	-	-
19.9		_sta_sftrl6	0	8	9	-	-	-	-
19.10		_sta_sftrl7	0	10	11	-	-	-	-
19.11		_sta_sftrl8	0	5	6	-	-	-	-
19.12		_sta_sftrl9	0	7	8	-	-	-	-
19.13		_sta_sftrl10	0	7	8	-	-	-	-
19.14		_sta_sftrl11	0	8	9	-	-	-	-
19.15		_sta_sftrl12	0	9	10	-	-	-	-
19.16		_sta_sftrl13	0	10	11	-	-	-	-
19.17		_sta_sftrl14	0	10	11	-	-	-	-
19.18		_sta_sftrl15	0	11	12	-	-	-	-
19.19		_sta_sftrl16	0	5	6	-	-	-	-
19.20		_sta_sftrl17	0	7	8	-	-	-	-
19.21		_sta_sftrl18	0	7	8	-	-	-	-
19.22		_sta_sftrl19	0	8	9	-	-	-	-
19.23		_sta_sftrl20	0	9	10	-	-	-	-
19.24		_sta_sftrl21	0	10	11	-	-	-	-
19.25		_sta_sftrl22	0	10	11	-	-	-	-
19.26		_sta_sftrl23	0	11	12	-	-	-	-
19.27		_sta_sftrl24	0	7	8	-	-	-	-
19.28		_sta_sftrl25	0	9	10	-	-	-	-
19.29		_sta_sftrl26	0	9	10	-	-	-	-
19.30		_sta_sftrl27	0	10	11	-	-	-	-
19.31		_sta_sftrl28	0	10	11	-	-	-	-
19.32		_sta_sftrl29	0	10	11	-	-	-	-
19.33		_sta_sftrl30	0	8	9	-	-	-	-
19.34		_sta_sftrl31	0	7	8	-	-	-	-
19.35		_sta_sftra0	4	4	5	-	-	-	-

表 11.2 运行时例程速度/FPL 速度列表 (6)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
19.36	右移	_sta_sftra1	0	4	5	-	-	-	-
19.37		_sta_sftra2	0	7	8	-	-	-	-
19.38		_sta_sftra3	0	8	9	-	-	-	-
19.39		_sta_sftra4	0	10	11	-	-	-	-
19.40		_sta_sftra5	0	11	12	-	-	-	-
19.41		_sta_sftra6	0	13	14	-	-	-	-
19.42		_sta_sftra7	0	14	15	-	-	-	-
19.43		_sta_sftra8	0	13	15	-	-	-	-
19.44		_sta_sftra9	0	14	16	-	-	-	-
19.45		_sta_sftra10	0	16	18	-	-	-	-
19.46		_sta_sftra11	0	17	19	-	-	-	-
19.47		_sta_sftra12	0	17	19	-	-	-	-
19.48		_sta_sftra13	0	17	19	-	-	-	-
19.49		_sta_sftra14	0	14	16	-	-	-	-
19.50		_sta_sftra15	0	11	13	-	-	-	-
19.51		_sta_sftra16	0	7	8	-	-	-	-
19.52		_sta_sftra17	0	8	9	-	-	-	-
19.53		_sta_sftra18	0	10	11	-	-	-	-
19.54		_sta_sftra19	0	11	12	-	-	-	-
19.55		_sta_sftra20	0	11	12	-	-	-	-
19.56		_sta_sftra21	0	13	14	-	-	-	-
19.57		_sta_sftra22	0	11	12	-	-	-	-
19.58		_sta_sftra23	0	11	12	-	-	-	-
19.59		_sta_sftra24	0	9	10	-	-	-	-
19.60		_sta_sftra25	0	10	11	-	-	-	-
19.61		_sta_sftra26	0	12	13	-	-	-	-
19.62		_sta_sftra27	0	13	14	-	-	-	-
19.63		_sta_sftra28	0	13	14	-	-	-	-
19.64		_sta_sftra29	0	11	12	-	-	-	-
19.65		_sta_sftra30	0	10	11	-	-	-	-
19.67		_sta_sftra31	0	7	8	-	-	-	-
20.1	加长	_abs64	8	-	-	-	-	-	-
20.2		_add64	8	31	42	17	27	18	16
20.3		_sub64	8	31	42	17	27	18	16
20.4		_mul64	36	133	92	36	64	46	30
20.5		_div64s	64	36	45	14	31	21	17
20.6		_div64u	60	36	45	14	31	21	17
20.7		_mod64s	64	36	45	14	31	21	17
20.8		_mod64u	60	36	45	14	31	21	17

表 11.2 运行时例程速度/FPL 速度列表 (8)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
20.9	加长	_neg64	8	26	31	14	24	15	13
20.10		_not64	8	23	42	14	21	15	23
20.11		_and64	8	31	42	17	28	18	30
20.12		_or64	8	31	42	17	28	18	30
20.13		_xor64	8	31	42	17	28	18	30
20.14		_shlld64	20	94	105	30	44	27	54
20.15		_shlrd64	20	94	105	32	47	29	47
20.16		_shard64	24	111	127	30	46	28	45
20.17		_bfs64s	52	159	170	77	110	54	69
20.18		_bfs64u	52	-	-	77	-	-	-
20.19		_bfx64s	24	148	152	128	101	109	139
20.20		_bfx64u	24	139	143	121	96	102	134
20.21		_cmplt64	4	23	27	12	17	13	11
20.22		_cmplt64u	4	23	27	12	17	13	11
20.23		_cmpgt64	4	23	27	12	17	13	11
20.24		_cmpgt64u	4	23	27	12	17	13	11
20.25		_cmple64	4	23	27	12	17	13	11
20.26		_cmple64u	4	23	27	12	17	13	11
20.27		_cmpge64	4	23	27	12	17	13	11
20.28		_cmpge64u	4	23	27	12	17	13	11
20.29		_cmpeq64	4	20	22	11	14	11	8
20.30		_cmpne64	4	20	22	11	14	11	8
20.31		_convi64	8	21	28	10	20	11	11
20.32		_convu64	8	18	25	9	18	10	8
20.33		_convs64	20	284	273	155	177	136	115
20.34		_convs64u	20	293	281	155	182	140	115
20.35		_convf64	20	-	-	155	-	-	115
20.36		_convf64u	20	-	-	155	-	-	115
20.37		_convw64	20	63	77	33	52	35	38
20.38		_convw64u	20	63	77	33	52	35	38
20.39		_convd64	20	63	77	33	52	35	38
20.40		_convd64u	20	63	77	33	52	35	38
20.42		_conv64u	0	4	4	3	3	3	2
20.43		_conv64s	24	350	332	73	199	-	24
20.44		_conv64us	24	348	330	71	197	-	17
20.45		_conv64f	28	-	-	73	-	26	24
20.46		_conv64uf	28	-	-	71	-	25	17
20.47		_conv64w	20	219	216	111	136	98	102
20.48		_conv64uw	20	211	210	106	132	95	99

表 11.2 运行时例程速度/FPL 速度列表 (8)

编号	类型	函数名称	堆栈大小	执行周期数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
20.49	加长	_conv64d	20	219	216	111	136	98	102
20.50		_conv64ud	20	211	210	106	132	95	99
20.51		_bfs64sp1	60	312	346	161	236	3859	205
20.52		_bfs64up1	60	-	-	161	-	-	205
20.53		_bfx64sp1	36	298	323	155	224	3572	243
20.54		_bfx64up1	40	387	388	190	258	3990	243
21.1	Packed	_pack1_st16	4	13	15	10	12	159	9
21.2	结构	_pack1_st32	4	19	21	17	18	289	12
21.3		_pack1_st64	4	33	37	15	31	483	21
21.4		_pack1_ld16	4	17	18	10	13	232	16
21.5		_pack1_ld32	4	29	30	17	22	372	-
21.6		_pack1_ld64	8	67	75	38	54	819	56

注意：测量从进入运行时例程直到退出计算。

表 11.3 运行时例程速度/FPL 速度列表

编号	类型	函数名称	堆栈大小	执行周期数		
				SH2-DSP	SH3-DSP	SH4AL-DSP
1.1	DSP	_padd24	8	46	29	30
1.2		_padd40	8	58	36	35
1.3		_pdiv16	24	37	24	21
1.4		_pdiv32	36	37	23	22
1.5		_pdiv24	36	37	23	22
1.6		_pdiv40	36	46	31	24
1.7		_pmul32	16	51	35	32
1.8		_pmul24	24	129	84	79
1.9		_pmul40	44	187	134	110
1.10		_psub24	24	46	29	30
1.11		_psub40	8	58	36	35
1.12		_pconv16s	12	19	12	20
1.13		_pconv16w	16	57	37	41
1.14		_pconv32s	12	20	12	19
1.15		_pconv32w	16	53	34	39
1.16		_pconv24s	12	16	9	19
1.17		_pconv24w	16	56	36	36
1.18		_pconv40s	16	29	18	23
1.19		_pconv40w	16	39	27	28
1.20		_pconvs16	16	1610	944	465
1.21		_pconvs32	16	577	348	203
1.22		_pconvs24	16	1636	957	483
1.23		_pconvs40	16	528	325	210
1.24		_pconvw16	16	10331	6202	3158
1.25		_pconvw32	20	525	324	209
1.26		_pconvw24	20	10352	6214	3171
1.27		_pconvw40	20	531	329	212
1.28		_pcmplt40	4	36	22	23
1.29		_pcmple40	4	36	22	26
1.30		_pcmpgt40	4	36	22	26
1.31		_pcmpge40	4	36	22	26
1.32		_pcmpeq40	4	30	19	16
1.33		_pcmpne40	4	31	20	20
1.34		_pdiv16_sat	28	37	24	21
1.35		_pdiv32_sat	40	37	23	22
1.36		_pmul32_sat	16	67	45	43

注意：测量从进入运行时例程直到退出计算。

11.1.5 SH 系列目标兼容性

问题

对使用编译选项“-cpu=sh1”（或 sh2、sh2e、sh3、sh4）和“-pic=1”编译的目标进行连接会出现问题吗？

解答

基本上，只要微型计算机具备向上兼容性，SH-1 目标和 SH-3 目标就可以进行连接然后在 SH-3 上执行。这意味着可以继续使用之前的资源而无须修改。

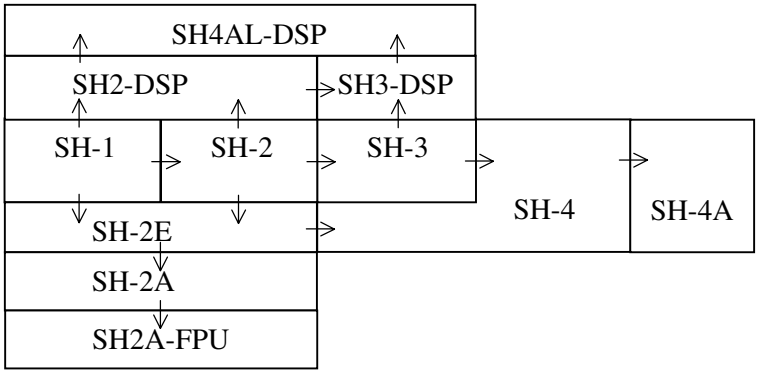


图 11.1 目标兼容性

注意： (1) SH-1、SH-2、SH-2E、SH2-DSP、SH-2A 和 SH2A-FPU 是 big-endian；将这些型号的目标与 SH-3、SH3-DSP、SH4AL-DSP、SH-4 和 SH-4A 配合使用时，它们必须作为 big-endian 使用。

(2) 使用“=pic=1”选项编译的目标和使用“=pic=0”选项编译的目标可以连接；但是，所生成的程序将不会是位置无关。

(3) SH-3、SH3-DSP、SH4AL-DSP、SH-4 和 SH-4A 的中断期间运算与 SH-1、SH-2、SH-2E、SH2-DSP、SH-2A 和 SH2A-FPU 的不一样，因此需要使用中断处理程序 。

要获取有关“-endian”选项的信息，请参考第 11.1.15 节“数据 Endian 赋值”。

11.1.6 执行宿主机和 OS

问题

什么是执行宿主机 和 OS?

解答

下表列出可以运行 SuperH RISC engine C/C++ 编译程序 9.0 版本的宿主机和 OS。

表 11.4 执行宿主机 和 OS 列表

系统名称	OS	注释
HP9000/700 HITACHI9000 HITACHI9000V	HP-UX 10.2 版本	
IBM-PC/AT	Windows98/Me/2000/XP/NT	Pentium 处理器
SPARC	Solaris 2.5 版本 Solaris 8 版本	

11.1.7 无法进行 C/C++ 源代码级调试。

问题

使用了“-debug”编译程序选项，但仍然无法在 C 源代码级执行调试。

解答 1

要在连接和编译期间输出调试信息，您需要指定适当的选项。

请注意，如果包含源程序的目录和编译期间存在的不同，将无法在 C 源代码级执行调试。 在此情形下，您可以将源程序返回其原始目录，或重新编译该程序。

对于连接程序 7 或以上版本：

若在连接期间指定输出范围，使输出划分到几个文件，调试信息将不会附加到每一个文件，而是只附加到一个单独的文件。 因此，除非将调试信息文件装入调试程序，否则将无法在 C 源代码级执行调试。

对于连接程序 6 版本：

在连接期间，您可以指定一组选项来输出各种类型的目标格式，但其中一些将不能被调试程序使用。

从下表选择适合调试程序使用的目标格式。

表 11.5 选项/子命令和兼容调试程序

兼容调试程序	选项/子命令	
	目标格式	调试信息输出
支持 ELF/DWARF 格式的第三方调试程序	ELF	DEBUG
Hitachi 集成管理器（版本 4），+E7000	SYSROFPLUS	SDEBUG
Hitachi 集成管理器（版本 3），+E7000	SYSROF	SDEBUG
Hitachi 调试界面（版本 2），+E6000	SYSROF	DEBUG
Hitachi 调试界面（版本 3），+E6000	ELF	SDEBUG

解答 2

指定 -code=asm 时，无法在 C 源代码级执行调试。

如果您使用内联汇编程序并指定 -code=asm。

要在 C 源代码级为使用内联汇编程序的工程执行调试，请仅对使用内联汇编程序的文件指定 -code=asm。

11.1.8 内联扩展中出现的警告

问题

(1) 尝试内联扩展时，出现“#pragma inline 中的函数（函数名称）未扩展”("Function (function name) in #pragma inline is not expanded") 警告。

(2) 尝试内联扩展时，出现“函数未被优化”("Function not optimized") 警告。

解答

这些警告消息并不影响程序的执行。

(1) 检查使用“#pragma inline”指定的函数是否符合内联扩展的条件。

具备使用“ #pragma inline”指定的函数名称的函数，以及使用函数说明符 inline（C++ 语言）指定的函数，将会在被调用时进行内联扩展。 但是，它们在下列情形下将不会被扩展。

- 函数在 #pragma inline 说明符之前被定义
- 函数具有变量参数
- 参数地址在函数中被参考
- 通过要扩展的函数地址执行调用
- 从条件/逻辑运算符的第二个运算符

实例：

```
#pragma inline(A,B)
int A(int a)
{
        if(a>10) return 1;
          else return 0;
}
int B(int a)
{
          if(a<25) return 1;
          else return 0;
}
void main()
{
        int a;
          if( A(a)==1 && B(a)==1 )
{
   .....

}
}
```

A() 被内联扩展，而 B() 则没有。
（因为出现不需要执行求值 B(a)==1 的情况）

(2) 这是因为存储器不足所导致。 当 SuperH RISC engine C/C++ 编译程序执行内联扩展时，函数大小会增加，因此在优化处理进行到中途时可能会出现存储器不足，进而使大于表达式单元的优化无法再执行。要纠正此状况，请尝试以下方法。

- 不要扩展大的函数
- 不要扩展在多个地点被调用的函数
- 减少被扩展的函数数量
- 增加可用的存储器数量

11.1.9 编译时出现“函数未被优化”("Function not optimized") 警告

问题

使用“-optimize=1”选项进行编译时，收到“函数未被优化”("Function not optimized") 警告。之前，我可以使用相同编译选项在相同系统环境下编译此程序而没有出现任何问题。 为什么我会收到此警告？

解答

这则警告并不影响程序的执行。

以下是导致此警告消息的可能原因。

(1) 超出了编译程序的限制。
优化期间，编译程序会生成新的内部变量，所以在某些情形下会超出编译程序限制。若出现此情形，应该将函数划分成更小的函数。

要获取有关编译程序限制的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)，第 16.1 节“编译程序的限制”。

(2) 存储器不足
若在优化处理期间存储器不足，SuperH RISC engine C/C++ 编译程序将发出警告并停止在表达式或以上的单元中执行优化。 编译将会继续，但最终结果将会和选取 optimize=0 优化级别时的一样。 若要避免此警告，请将 C 源程序中的大型函数划分成更小的函数。

如果此方法无效，唯一的其他方法是增加编译程序可用的存储器数量。

(3) 内联扩展的案例
请参考第 11.1.8 节“内联扩展中出现的警告”。

11.1.10 编译时出现“编译程序版本不符”("compiler version mismatch") 消息

问题

编译时，出现严重错误消息“编译程序版本不符”("compiler version mismatch")。 为什么？

解答

检查使用环境变量 “PATH” 和“SHC_LIB” 指定的目录是否正确无误。

实例：

如果环境变量设定如下，上述错误消息将会输出。

```
PATH =(SHC 8.0 版本的路径)
SHC_LIB = (SHC  6.0 版本的 C 编译程序的路径名)
```

11.1.11 编译时出现“存储器溢出”("memory overflow") 错误

问题

编译时，出现严重错误“存储器溢出”("memory overflow")。 为什么？

解答

以下是导致存储器溢出错误的可能原因。

(1) 存储器不足

(2) 使用“SHC_LIB”环境变量设定的路径名所指定的目录中，并不存在所有的 C/C++ 编译程序文件。

实例：

若使用以下设定，上述错误消息将会出现。
环境变量设定为 SHC_LIB=/SHC/BIN
文件同时保存在 /SHC/BIN 和 /SHC/MSG
在此情形下，所有文件必须存在于 /SHC/BIN 中。

(3) 环境变量未正确设定。

若使用 PC 版本，不应该将环境变量“SHC_LIB”设定到具有程序库的目录，而是设定到包含 SHC.EXE 的目录。安装编译程序时创建的批文件 SETSHC.BAT 会将“SHC_LIB”变量设定到包含“SHC.EXE”文件的“C:\SHC\BIN”目录。

11.1.12 “包含”(Include) 的指定步骤

问题

我不了解包含文件的所有各种选项。

请告诉我它们的使用方式以及优先顺序。

解答

包含文件的搜索路径使用选项或环境变量指定。

被“<”和“>”括起来的文件，从使用“-include”选项所指定的目录读取；若指定多个目录，它们将以被指定时的顺序搜索。 若文件无法在使用“-include”选项指定的目录中找到，将会搜索使用“SHC_INC”环境变量指定的目录，然后搜索系统目录 (SHC_LIB)。

被引号 (") 括起来的文件将会从当前目录开始搜索。 若在当前目录中没有找到这些文件，将按照上述规则进行搜索。

搜索包含文件之目录的步骤简述如下：

```
-inc > SHC_INC > SHC_LIB
```

除了上述规则外，也可使用“-preinclude”选项来强制读取文件。 使用此选项时，通过此选项指定的文件将会放置在所有要编译的文件的前面，然后执行编译。

通过使用此选项来读取只要暂时使用的文件，如包含 #pragma 语句和测试数据的文件，将可以重新编译而无须修改源文件。

11.1.13 编译批文件

问题

编译时需要设定的选项很多，而每一次都要重复设定它们也非常麻烦。

有没有一些更方便的方法？

解答

编译时，可以使用“-subcommand”选项（“-subcommand=<文件名>”）。

“-subcommand”选项可以在命令行上多次使用。 子命令文件可以包含命令行参数，这些参数使用空格、回车或制表符隔开。 子命令文件的内容会在指定子命令的位置，扩展到命令行参数中。

但是，“-subcommand”选项本身不能在子命令文件中指定。

实例：

在以下实例中，命令行扩展为相等于

shcΔ-optimize=1Δ-listfileΔ-debugΔ-cpu=sh2Δ-pic=1Δ-sizeΔ-euc

Δ-endian=bigΔtest.c

命令行

shcΔ-sub=test.subΔtest.c

test.sub 的内容

```
-optimize=1
-listfile
-debug
-cpu=sh2
-pic=1
-size
-euc
-endian=big
```

11.1.14 程序中的日文文本

问题

我在工作站和 PC 上开发了程序的源代码，但工作站和 PC 上的日文代码并不一样，因此很难管理源文件。 有没有更简易的方法可以执行这些操作？

解答

若在日文代码中使用移位 JIS 格式，如果在工作站（使用日文的 EUC 编码）上进行编译，应该使用“-sj”编译程序选项。相反的，如果将 EUC 代码用于要在 PC 上编译的程序中，则应该指定“-euc”编译程序选项。即使是在混合使用 EUC 和移位 JIS 代码的工作站网络环境中，通过设定适当的编译选项，就可以使用任何一种日文编码进行编译。

编译可以使用目标机器上采用的日文代码执行。

表 11.6 系统与日文代码对应表

主机	默认
SPARC	EUC
HP9000/700	移位 JIS
PC9800 系列	移位 JIS
IBM-PC	移位 JIS

实例：

如果源代码是在工作站 (SPARC) 上编写而在 PC (IBM PC) 上编译，则可以在编译中使用“-euc”选项，以防止字符串中日文代码的误译。

11.1.15 数据 Endian 赋值

问题

SH 型号是使用 big-endian 或 little-endian 数据？

解答

Renesas Technology SuperH RISC engine 家族是 big-endian 系统。

然而，SH-3、SH3-DSP、SH-4、SH-4A 和 SH4AL-DSP 支持“-endian=Big(Little) ”选项以允许 CPU big/little-endian 的切换。

注意：

(1) “-endian”选项 可以与“-cpu”选项的任何任意子选项组合，但 little-endian 目标程序不能在 SH-3、SH3-DSP、SH-4、SH-4A 或 SH4AL-DSP 以外的其他产品上执行。

(2) Big-endian 目标和 little-endian 目标不能一起使用。

(3) endian 类型的不同可能会影响程序的执行。

实例：受 endian 类型影响的代码

```
f( ){
        int a=0x12345678;
        char *p;

        p=((char *)(&a));

        if(*p==0x12){ (1) }
        else{ (2) }
   }
```

在此例子中，若数据是 big-endian 则 (1) 将会执行，但若是 little-endian，则 *p 是 0x78，因此将执行 (2)。

要获取有关数据赋值的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)，第 10.1.2 (4) 节“Little Endian 的存储器分配”。

(4) “-denormalize=on|off”选项可用于选择是否处理非正常化的数字或把它们作为 0 处理（只有在 -cpu=sh4 或 -cpu=sh4a 时）。

但是，在“-denormalize=on”时，若将非正常化数字输入 FPU，则会出现异常。 因此，必须在软件上编写异常处理以处理非正常化数字。

11.1.16 使用“#pragma inline_asm”汇编

问题

使用“#pragma inline_asm”汇编程序时，出现“非法数据区地址”("ILLEGAL DATA AREA ADDRESS")（错误编号 452）错误。

解答

(1) 检查您是否使用“-code=asmcode”选项编译。

(2) 检查汇编语言代码中是否存在数据表。

以下是导致此状况的一个可能原因。

```
#pragma inline_asm(bar)
int bar()
{
        MOV.L    #160,R9
}
```

在上述代码中，行

```
        MOV.L    #160,R9
```

并没有被 SuperH 微型计算机解释为将值“160”直接移到寄存器的指令。

一般上，必须创建和装入数据库。汇编程序会自动识别和创建数据库；但所生成的数据不具备编译程序输出的汇编语言源对齐，因此出现错误。汇编程序自动生成数据的此类情况已不可在当今的编译程序中预期，因此无法为导致汇编程序自动生成数据库的 inline_asm 函数编写汇编语言源中的代码。然而，可以按照以下方法修改上述例子中的代码以避免此问题。

已修改代码的实例

< 修改前>

```
        MOV.L    #160,R9
```

< 修改后 >

```
        MOV         #100,R9
        ADD         #60,R9
```

11.1.17 有权限的模式 (Privileged Mode)

问题

嵌入式函数“set_cr” 和“get_cr” 无法正常操作。

解答

上述嵌入式函数只能在 SH-3 和 SH-4A 中的有权限的模式中使用。

要获取一般信息，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)，第 10.2.3 节“固有函数”；要获取有关 SH-3 之有权限的模式的信息，请参考硬件手册。检查在这些嵌入式函数被调用时是否有设定有权限的模式。（在有权限的模式中，SR 寄存器 MD 位将会设定。） 要从非权限模式转变至有权限模式，必须发出 TRAPA 指令。

11.1.18 关于目标生成

问题

直接从编译程序生成目标时，以及通过汇编程序生成目标时，出现以下错误。

(1) 程序大小不一样。
(2) 符号类型 是 DAT 而不是 ENT。

解答

直接生成或使用汇编程序生成目标时，由于生成目标之方法的不同，所生成的装入模块一般上会不一样。这不是错误的操作。

由汇编程序输出的目标不会区别 ENT 和 DAT；这同样不是错误的行为。

11.1.19 关于 #pragma gbr_base 功能

问题

使用“#pragma gbr_base”功能和加载程序到仿真程序或尝试写入 ROM 时，出现错误。

解答

$G0、$G1 段应该作为初始化数据段处理。

一般上，变量的分配如下：

(1) 没有指定初始值的变量将分配到未初始化的数据段（默认情况下，段名称为“B”）。
(2) 有指定初始值的变量将分配到已初始化的数据段（默认情况下，段名称为“D”）。
(3) 指定“const”的变量，将分配到常数段（默认情况下，段名称为“C”）。

但是，对于使用“#pragma gbr_base”（或 gbr_base1）指定的变量将不会有这些区别，而且全部都会分配到 $G0（或 $G1）段；因此，编译程序会将 $G0、$G1 视为已初始化的数据区，并生成假设若变量没有指定初始值则指定“0”的目标。

11.1.20. 编译包含日文代码的程序

问题

在 PC 上编译已确定可以在 SPARC 工作站上正确编译的程序时，出现错误。

解答

检查日文代码 是否未包含在源程序中。 SuperH RISC engine C/C++ 编译程序同时支持 EUC 和移位 JIS 两种编码的日文代码，但不同宿主机上的默认编码将会不一样。 在 SPARC 工作站上，默认日文编码是 EUC，但在 PC 上则是移位 JIS。在 PC 上编译使用 EUC 日文代码的程序时，应该指定 -euc 选项。 要获得有关不同宿主机上的默认日文代码的更多信息，请参考第 9.1.19 节“程序中的日文文本”。

11.1.21 浮点运算的速度

问题

请告诉我关于浮点运算的执行速度。

解答

使用标准程序库的基本函数的执行速度在表 11.8（用于 SH-1、SH-2、SH-3）、表 11.9 (SH-2E)、表 11.10 (SH-4)、表 11.11 (SH-4A) 以及表 11.12（用于 SH-2A、SH2A-FPU）中显示。 要获得有关执行算术运算和其他浮点运算的信息，请参考第 11.1.4 节“运行时例程指定和执行速度”。

表 11.7 显示创建标准程序库的条件。

表 11.7 创建标准程序库的条件

条件	创建程序库的选项						
	Cpu	pic	endian	denormal	round	fpu	double=float
1	sh1	–	big	–	–	–	无
2	sh2	0	big	–	–	–	无
3	sh3	0	big	–	–	–	无
4	sh2e	0	big	–	–	–	无
5	sh4	0	big	off	zero	无	–
6	sh4	0	big	off	zero	single	–
7	sh4	0	big	off	zero	double	–
8	sh4a	0	big	off	zero	无	–
9	sh4a	0	big	off	zero	single	–
10	sh4a	0	big	off	zero	double	–
11	sh2a	0	big	–	–	–	无
12	sh2afpu	0	big	off	zero	无	–
13	sh2afpu	0	big	off	zero	single	–
14	sh2afpu	0	big	off	zero	double	–

表 11.8 浮点程序库函数的执行速度 （SH-1、SH-2、SH-3）

CPU		SH-1	SH-2	SH-3
创建程序库的条件		1	2	3
单精度	Sinf	2,438	2,497	1,632
	Cosf	2,384	2,434	1,599
	Tanf	3,120	3,196	2,091
	asinf	5,176	5,418	3,526
	acosf	5,355	5,622	3,659
	atanf	2,924	3,160	2,054
	logf	3,710	3,816	2,490
	sqrtf	3,252	1,018	661
	expf	4,327	4,432	2,873
	powf	4,649	4,824	3,139
双精度	sin	5,297	4,964	3,282
	cos	5,289	4,918	3,279
	tan	7,460	7,087	4,673
	asin	13,898	13,788	9,004
	acos	14,158	14,084	9,196
	atan	5,583	5,687	3,712
	log	8,756	8,368	5,535
	sqrt	2,903	2,946	1,803
	exp	9,501	8,952	5,912
	pow	9,337	8,943	5,918

注意：时钟周期个数

表 11.9 浮点程序库函数的执行速度 (SH-2E)

CPU		SH-2E
创建程序库的条件		4
单精度	sinf	307
	cosf	302
	tanf	343
	asinf	1,267
	acosf	1,289
	atanf	468
	logf	213
	sqrtf	648
	expf	299
	powf	472
双精度	sin	3,005
	cos	3,002
	tan	4,339
	asin	8,544
	acos	8,717
	atan	3,434
	log	5,144
	sqrt	1896
	exp	5,475
	pow	5,437

注意：时钟周期个数

表 11.10 浮点程序库函数的执行速度 (SH-4)

CPU		SH-4		
创建程序库的条件		5	6	7
单精度	Sinf	63	211	264
	Cosf	62	209	253
	Tanf	80	229	266
	Asinf	75	444	222
	Acosf	72	464	238
	Atanf	102	313	325
	Logf	86	174	203
	Sqrtf	--------*	--------*	--------*
	Expf	119	243	278
	Powf	386	377	213
双精度	Sin	336	231	139
	Cos	312	229	135
	Tan	408	229	186
	Asin	523	432	206
	Acos	616	452	253
	Atan	391	258	145
	Log	405	174	194
	Sqrt	--------*	--------*	--------*
	Exp	403	247	193
	Pow	1,031	377	213

注意：*SH-4 支持 sqrt 指令，因此 sqrt 函数将会被省略。
时钟周期个数

表 11.11 浮点程序库函数的执行速度 (SH-4A)

CPU		SH-4A		
创建程序库的条件		8	9	10
单精度	Sinf	100	228	312
	Cosf	108	231	307
	Tanf	141	258	335
	Asinf	118	501	271
	Acosf	126	525	288
	Atanf	148	357	385
	Logf	131	197	254
	Sqrtf	--------*	--------*	--------*
	Expf	169	272	221
	Powf	411	387	194
双精度	Sin	305	256	194
	Cos	287	258	187
	Tan	377	260	248
	Asin	466	494	267
	Acos	558	514	324
	Atan	331	290	191
	Log	348	197	235
	Sqrt	--------*	--------*	--------*
	Exp	386	277	257
	Pow	877	399	221

注意：*SH-4A 支持 sqrt 指令，因此 sqrt 函数将会被省略。
时钟周期个数

表 11.12 浮点程序库函数的执行速度（SH-2A、SH2A-FPU）

CPU		SH-2A	SH2A-FPU		
创建程序库的条件		11	12	13	14
单精度	Sinf	1,001	67	225	279
	Cosf	954	66	220	275
	Tanf	1,806	83	239	289
	Asinf	1,545	78	475	238
	Acosf	1,699	73	493	250
	Atanf	1,602	93	329	347
	Logf	1,720	87	162	214
	Sqrtf	562	--------*	--------*	--------*
	Expf	1,463	115	242	298
	Powf	2,140	392	376	228
双精度	Sin	3,431	275	247	136
	Cos	3,387	264	242	130
	Tan	4,425	356	239	180
	Asin	5,550	413	457	199
	Acos	5,949	478	475	245
	Atan	3,641	286	272	139
	Log	4,557	352	162	182
	Sqrt	1,622	--------*	--------*	--------*
	Exp	4,137	375	246	190
	Pow	4,086	772	376	228

注意：*SH2A-FPU 支持 sqrt 指令，因此 sqrt 函数将会被省略。
时钟周期个数

11.1.22 使用 PIC 选项

问题

我想让程序使用位置无关代码，我应该怎么做？

详细的问题：

(1) 我要将多个应用程序动态转移到可用的 RAM 以执行。
(2) 我想知道如何执行初始化操作。
(3) 请告诉我实际的限制以及需要注意的事项。

解答

若要将程序从 ROM 转移到 RAM 中的固定地址以执行，请不要使用 -PIC 选项；而是使用第 11.2.4 节“转移到 RAM 并执行程序”中所描述的步骤。

若要将代码动态转移到 RAM，则 -PIC 选项会比较方便使用，但此选项只在程序段有效，而且不会生成位置无关的数据。因此，数据区只能加载到固定地址。

因为此限制，若要使整个程序（包括数据）位置无关，编写程序时必须特别注意。

以下说明在不包括数据段时的编程步骤。

- 不包括数据段时的编程步骤

配置程序的实例

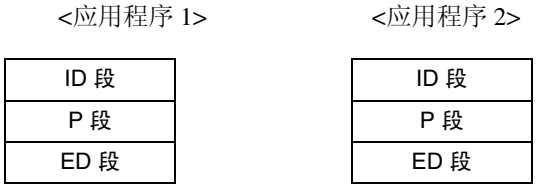


图 11.2 段

C 语言程序

```
<main.c>
main()
{
     int i;
     for (i=0;i<10;i++){
        sub(i);
     }
}
```

```
<sub.c>
sub(int p)
{
    int i;
    for (i=0;i<p;i++){
          ;
    }
}
```

汇编语言程序

```
<pic.src>
        .import     _main
        .section    ED,DATA,ALIGN=4   ; 生成 ED 终止段
        .section    ID,DATA,ALIGN=4   ; 标头的数据段
        .data.l     (STARTOF ED)
        .data.l     _main
        .end

<lnk.sub>
        input main
        input sub
        input pic
        start       ID,P,ED/0         ;  从 0 地址分配；ID 在起始，ED 在终端
        list  pic
        exit
```

标头（ID 段）将会添加到每个程序。

ID 段的内容如下：

偏移 0 地址　　程序大小

偏移 4 地址　　入口点（主地址）

在此方式下，程序将会生成，而控制这些的程序将会根据 ID 计算加载地址和执行地址。

以下显示控制程序的实例。

```
<control.c>
void load_program(int ID){
      char *p;
size=load_ID( ID);                /* 装入程序 ID 标头数据              */
                                  /* 返回值是程序大小                  */
     p=malloc(size);
      if(p!=NULL){
          mload(p,ID);            /* 将程序数据写入堆                  */
          go((*(long**)p+1)+(long*)p);
                                  /* 将 PC 设定在程序的前导地址        */
                                  /* 然后执行                          */
    }
    else {
       error("Insufficient Memory");
       }
}
```

这是程序图像；执行的方法将会根据所使用的 OS 而异。 上述实例应该作为当程序动态运行时的流程级实例。

11.1.23 优化删除了大量代码

问题

编译后，大量代码被删除。

解答

可能是执行了下列类型的优化。

(1) 删除空的循环
空的循环可使程序等待一段固定时间，它可能会在优化过程中被删除。

实例

```
set_param();                /* 设定参数                                */
   for(i=0;i<10000;i++);    /* 设定参数后，设定结果                    */
                            /* 空的循环使程序等待一段固定时间          */
                            /* 编译程序本身把它当作无意义              */
                            /* 而删除                                  */
   read_data();             /* 取得结果                                */
                            /* 由于循环已被删除，等待时间将会免除，    */
                            /* 然后在失败前尝试读取结果                */
```

(2) 删除局部变量的替换
不论是否将值替换到局部变量中，若该值不被参考，替换操作本身将被删除。

实例

```
int data1, data2, data3;
func()
{
    int   res1,res2,res3;

    res1=data1*data2;
    res2=data2*data3;       /* res2 在这之后不被参考，因此表达式本身将会被删除。   */
    res3=data3*data1;
    sub(res1,res1,res3);  /*     第二个参数的指定错误                          */
                          /* 若编写 res2 而不是 res1，上述表达式将不会被删除。   */
}
```

局部变量直到函数的结束部分均有效，因此一般上不会在函数内的局部变量中替换值，也不会被参考。所以出现上述编程错误时，就可能会删除。

11.1.24 局部变量的值无法在调试期间显示

问题

我看不到局部变量的值。

在调试期间，代码有参考局部变量，但它的值无法被参考，或不正确。

解答

可能是执行了下列类型的优化。

(1) 编译期间的常数操作
在编译期间，任何值确定为在编译时而不是在运行时计算，变量本身可能会被删除。

实例 1

```
int x;
func()
{
    int a;
    a=3;
    x=x+a;          /* 这里，此表达式在编译期间变成 x=x+3          */
                    /* 如果“a”未在别处使用，则无需将“a”视为变量。  */
                    /* 因此，它也将从调试信息中删除。              */
}
```

实例 2

```
func(int a,int b)
{
    int tmp;
    int len;

    tmp=a*a+b*b;
    len=sq(tmp);    /* 这将变成 len=sq(a*a+b*b)；且 tmp 被删除。   */

        :
}
```

这类情况是可能发生的，但它们不会影响实际的程序操作。

(2) 删除未被参考的变量

实例 3

```
int data1, data2, data3;
func()
{
     int   res1,res2,res3;

     res1=data1*data2;
     res2=data2*data3;          /* 此表达式会被删除，而 res2 本身也会被删除。   */
     res3=data3*data1;
     sub(res1,res1,res3);       /*      错误编写第二个参数                      */
                                /* 若将 res1 更改为 res2，就不会删除。          */
}
```

局部变量直到函数的结束部分均有效，因此一般上不会在局部变量中替换值，也不会被参考。所以出现上述编程错误时，就可能会删除。

11.1.25 中断禁止/允许宏

问题

我要将宏用于中断禁止/允许处理，我应该怎么做？

解答

使用嵌入式函数即可执行此操作，如下例所述。 要获取有关嵌入式函数的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)，第 10.2.3 节“固有函数”。

实例

```
#include <machine.h>
  #define disable() { save_cr=get_cr(); set_imask(0x0f); }
  #define enable()  { set_cr(save_cr); }

  function()
{
      int save_cr;

      disable();
      sub();
      enable();
  }
```

11.1.26 SH-3 和更新型号中的中断函数

问题

从 SH-3 开头的 SuperH 微型计算机中，编写中断函数的步骤是否有任何不同？

(1) 我要使用多个中断，但使用已指定“#pragma interrupt”的函数，
  (a) SSR、SPC 保存指令不可使用。
  (b) 清除 SR 的 RB 和 BL 位的指令无法使用。
  (a) SSR、SPC 恢复指令不可使用。
(2) 我要使用具有“#pragma interrupt”语句的 TRAP 数量指定，但 SR 的 BL 位保留为 1，因此在发出 TRAPA 指令时，出现指令异常。

解答

编译程序不会输出 SSR 或 SPC 保存/恢复指令。 可以使用“#pragma inline_asm”功能明确编写它们，或使用汇编程序编写程序。SR 设定可以使用嵌入式函数 set_cr、get_cr 编写。

SH-3 和更新型号的中断处理与 SH-1、SH-2 以及 SH-2E 中的处理有很大的不同。 在后者的微型计算机中，向量表将会在发生中断时被参考，并控制相应中断例程的转移。 然而，在 SH-3 和更新型号中将会转移到固定地址。 因此，一般上必须将中断处理程序放到中断转移目的地，以便禁止/允许多个中断、对中断因素求值，以及开始处理不同的中断因数。这类中断处理程序通常以汇编语言编写。

请参考第 2.2 节和 2.3 节“样品程序简介”。

参考材料：适当的 SH770 硬件手册

11.1.27 SH4 浮点的运算结果

问题

SH4 浮点的运算结果与预期的值不符。

解答

编译程序具有 FPU 选项。FPU 选项共有三种样式：FPU=single/double/No specification。其不同如下所示：

FPU=single：所有浮点表达式将视为单精度。

FPU=double：所有浮点表达式将视为双精度。

FPU=No specification：浮点表达式的精度将跟随 C 描述的类型。

根据 FPU 选项，在 FPSCR 寄存器的 PR 位设定将会有所不同。

(1) 此值（PR 位）在初始条件中设定为 [0(=single precision)]。
(2) 如果在 FPU 选项中没有指定，C 编译程序生成的代码会改变每次 FPU 运算时的 PR 位。但是在指定 FPU=double/single 时，将完全不会生成 PR 更改代码。
(3) 因此，在指定 FPU=No specification 和 FPU=single specification 时，选项将会正确运算而不考虑上述的位，但在指定 FPU=double 时，则需要在用户端将 [1=(double)] 明确指定到 PR 位。

11.1.28 关于优化选项

问题

优化选项会产生什么改变（速度、大小）？

解答

指定的优化选项会改变生成的代码。（请勿使用优化更改用户程序的运算法。）使用优化，可以优化如函数内联扩展和解开循环的代码，因此运行时循环的次数将会改变。 由此，运算的时序也会更改。 首先，请指定足够的运算时序。 而且，除了上述事项外，变量存取的优化也是重要考虑事项。 数据指令可以在寄存器之间达成而不需要存储器，以及相应于变量存取的优化之情况，称为 [时序验证] ([Timing verification])。 如果您选择 [不要优化] ([Do not want to optimize]) 变量，请确定包含必要的附加 volatile 声明。

11.1.29 函数的一个参数无法正确转移。

问题

函数的一个参数无法正确转移。

解答

请确定是否已声明函数的原型。

如果函数的原型未声明，参数（char、unsigned、char、float）将变成自动类型转变的目标。 这时，需要将要调用的函数方声明为更改的类型。

建议声明函数的原型。

函数原型的现有声明可通过编译程序的消息选项确认。

11.1.30 如何检查可能导致不正确操作的编码

问题

是否有任何可检查潜在问题代码（如：函数的原型声明缺失）的函数？

解答

在编码程序时，注意某些代码在语言规格中未出现错误，但却可能造成不正确的操作结果。这些代码可通过使用选项输出信息消息来检查。

MISRA-C 检查工具可以和 9 或以上版本配合使用。

[指定方法]

对话框菜单：**C/C++** 标签**类别：(C/C++** tab **Category:) [源 (Source)] 消息 (Message)，显示信息级消息 (Display information level message)**

命令行：***message***

说明

在对话框菜单中，移除消息左侧的复选标记将禁止消息的输出。 在命令行中，在无消息 (nomessage) 选项的子选项中指定错误编号将禁止消息的输出。 要获取有关错误编号的详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)，第 12 节“编译程序错误消息”。

生成信息消息后，编译程序将执行错误校正并生成目标程序。 检查由编译程序执行的错误校正是否符合程序的目的。

11.1.31 注解编码

问题

(1) 我该如何嵌套注解？

(2) 我该如何在 C 语言程序中编码 C++ 注解？

解答

(1) 有一个选项可让您嵌套注解而不会生成错误。 在此情形下，注意这些注解将被解释如下。

[指定方法]

对话框菜单：C/C++ 标签类别：(C/C++ tab Category:) [其他 (Other)] 杂项 (Miscellaneous options)：允许注解嵌套

命令行：***comment***

表 11.13 嵌套注解。

C/C++ 源代码	不允许已嵌套的注解	允许已嵌套的注解
/* comment */	识别为注解语句	识别为注解语句
/* /* comment */ */	编码错误	识别为注解语句
/* /* /* comment */	识别为注解语句	编码错误

(2) 可以使用 C++ 注解代码“//”。 “//”和 C 注解代码 (/* */) 之间具有下列关系。 可识别为注解的部分以下划线标识：

```
void func()
{
    abc=0;          //   /* 注解    */

    def=1;          /* 注解
    ghi=2;          // 注解        */
}
```

← // 之后的代码被识别为注解

← 以 /* */ 括起来的代码被识别为注解

11.1.32 如何在汇编程序被嵌入时创建程序

问题

使用 #pragma inline_asm 执行汇编程序固有代码，在编译时输出警告消息。

解答

汇编程序嵌入文件必须以汇编语言输出，然后被汇编。

若要在 HEW 上创建文件，请将包含汇编程序嵌入的文件指定到汇编输出（请参考“如何为每个文件指定选项”）。当以这种形式创建时，已汇编输出的文件将自动被汇编。

在下面的实例中，指定了包含汇编嵌入的 test.c 文件：

<HEW 2.0 或以上版本>

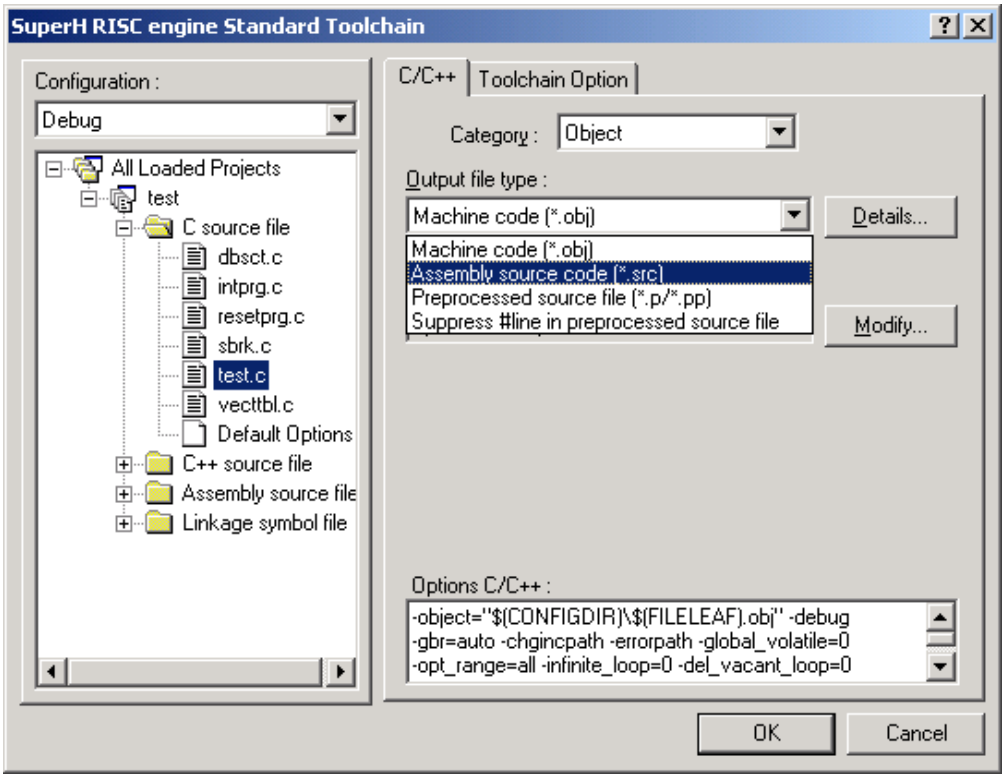


图 11.3 编译程序对话框

从 C/C++ 标签类别：(C/C++ Tab Category:) [目标 (Object)] 输出文件类型 (Output file type)：选取汇编源代码 (*.src)。

文件以此规格正常创建。

请注意，此指定将禁止 C 源代码调试。

11.1.33 C++ 语言规格

问题

是否有任何函数支持 C++ 语言中的程序开发？

解答

SuperH RISC engine C/C++ 编译程序支持下列函数，以支持 C++ 中的程序开发：

(1) 支持 EC++ 类程序库

由于支持 EC++ 类程序库，固有 C++ 类程序库可以从 C++ 程序使用而无需任何指定。

以下四种类型的程序库受支持：

- 流 I/O 类程序库
- 存储器操纵程序库
- 复数计算类程序库
- 字符串操纵类程序库

要获取详细资料，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)，第 10.4.2 节“EC++ 类程序库”。

(2) EC++ 语言规格语法检查函数

根据 EC++ 语言规格，在 C++ 程序上使用编译程序选项检查语法。

[指定方法]

对话框菜单：**C/C++** 标签**类别**：(**C/C++** Tab **Category**:) **[其他 (Other)] 杂项 (Miscellaneous options)： 对照 EC++ 语言规格 (Check against EC++ language specification)**

命令行：***ecpp***

(3) 其他函数

以下受支持的函数用于 C++ 程序的有效编码：

<较佳 C 函数>

- 函数的内联扩展
- 运算符，如 +、-、<< 的定制
- 使用多个定义函数的名称简化
- 注解的简单编码

<面向目标的函数>

- 类
- 构造函数
- 虚拟函数

要获取有关在 C++ 程序中使用程序库函数时，设定执行环境的描述，请参考 SuperH RISC engine C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)，第 9.2.2(4) 节“C/C++ 程序库函数初始设定 (_INILIB)”。

11.1.34 如何在预处理程序扩展后查看源程序

问题

我该如何在扩展宏之后复查程序？

解答

由预处理程序扩展的源程序输出可使用编译程序选项来指定。

若源程序在扩展前是一个 C 语言程序，它可以 <文件名>.p 的扩展名输出。对于 C++ 程序，扩展名是 <文件名>.pp。

在此情形下，将不会创建目标程序。 因此，任何优化选项规格都不可用。

[指定方法]
对话框菜单：**C/C++** 标签**类别：(C/C++** Tab **Category**:) **[目标 (Object)] 输出文件类型 (Output file type)：预处理源文件 (*.p/*.pp)**
命令行：***preprocessor***

11.1.35 程序在 ICE 上正确运行，但在实际芯片上安装后运行失败

问题

程序在 ICE 上调试时可正确运行，但在实际芯片上操作失败。

解答

若程序包含初始化数据区（D 段），它会在 ICE 上使用仿真存储器。 因此，读/写操作可在 ICE 上执行，然而，仅有读操作可在实际芯片上操作，因为实际芯片上的存储器是 ROM。 尝试写操作时，将导致程序执行出错。

初始化数据区必须在加电复位时从 ROM 区域复制到 RAM 区域。

使用 HEW2.0 或以上版本的优化连接编辑程序，及 HEW1.2 的模块间优化器的 ROM 实施支持选项，来分别为 ROM 和 RAM 保留区域。

要获取有关如何将数据从 ROM 区域复制到 RAM 区域的描述，请参考第 2.3.4 节“创建初始化部分”。

11.1.36 如何使用为 H8 微型计算机开发的 C 语言程序

问题

在 SH 微型计算机上使用为 H8S，H8/300 微型计算机开发的 C 语言程序时，我该确认哪些要点？

解答

必须谨慎处理有关程序的以下要点：

(1) int 类型数据被视为 4 字节数据。

在 H8S，H8/300 系列上，int 类型数据被视为 2 字节数据，但在 SH 系列上，它们被视为 4 字节数据。 确保值的范围不存在任何问题。

(2) 某些扩展函数无法使用。

SH 系列 C/C++ 编译程序和 H8S 及 H8/300 系列 C/C++ 编译程序上的函数可相互兼容，例如通过使用 #pragma 语句，然而，在扩展函数和指定上，它们之间仍有差异。

请注意，内建函数是 CPU 指定的。

(3) 汇编程序嵌入的注意事项

由于体系结构上的差异，SH 系列无法处理任何嵌入 H8S，H8/300 系列汇编源的代码。

如果您要在 SH 开发环境中使用于 M32R 开发环境中创建的 C 源文件，可以使用“转换助手”(Translation Helper)。

这是一个支持工具，可以将 M32R 开发环境中创建的所有 C 源文件，顺利转换到 SH 开发环境。

“转换助手”(Translation Helper) 可以从“瑞萨开发环境”(Renesas Development Environment) 网站免费下载。

11.1.37 导致无穷循环的优化

问题

为何无穷循环会在我升级编译程序，或打开优化时发生？

解答

无穷循环的发生可能会因为编译程序优化导致，例如，在下列公用源中，“a”的替换是从寄存器而不是从存储器读取，以防止“*d”的值在通过中断更改时被反映。 此优化是编译程序指定的一部分，并且可以使用易失性类型说明符防止。

实例

C 源

```
void f( int *d)
{
        int a;
        do
        {
                a=*d;
        }while(a!=0);
}
```

具备优化的汇编程序源

```
_f:                                     ; 函数：f
                                        ; 帧大小=0
          .STACK      _f=0
          MOV.L       @R4,R2
L11:
          TST         R2,R2             ; 不从存储器读取
          BF          L11
          RTS
          NOP
          .END
```

被修改的 C 源

```
void f( volatile int *d)
{
        int a;
        do
        {
                a=*d;
        }while(a!=0);
}
```

具备优化之被修改的汇编程序源

```
_f:                                     ; 函数: f
                                        ; 帧大小=0
           .STACK       _f=0
L10:
           MOV.L        @R4,R2          ; 从存储器读取
           TST          R2,R2
           BF           L10
           RTS
           NOP
.END
```

11.1.38 关于 DSP 程序库的警惕

问题

使用 DSP 程序库时，我有时会遇到异常终止，而且无法获得我预期的结果。 有哪些关于使用 DSP 程序库的警惕是我必须注意的？

解答

请检查下列事项：

1. 存储器损毁

因为 DSP 程序库使用堆存储器，如果存储器已经损毁，将无法获得正确的计算结果。

堆存储器损毁也会导致异常操作。

2. DSP 存储器（X 和 Y 存储器）的正确使用

有些 DSP 程序库函数需要输入和输出要放置在 X/Y 存储器中的数据。 对于这类 DSP 程序库函数，根据函数的描述，分配包含输入和输出数据到 X/Y 存储器的段。 您可以使用“#pragma section”更精细地隔开段。 有关详情，请参考*第 3.7.2 节“段切换”*。

另外，使用“-dspc”选项时，您可以使用 X/Y 存储器说明符简易地隔开 X/Y 存储器段。

请注意，使用过滤器函数时，必须将工作空间分配到 Y-RAM。 如果不指定“-dspc”选项，请在连接期间将 DY 和 BY 段分配到 Y-RAM。 如果指定“-dspc”选项，则在连接期间将 $YD 和 $YB 段分配到 Y-RAM。

3. DSP 程序库函数的使用方法

使用 DSP 程序库函数时，可能需要特殊的预处理和后处理。

检查相应的程序库函数，以及预处理和后处理是否正确使用。

要获得有关如何使用每个程序库函数的详情，请参考*第 3.13 节“DSP 程序库”*。

4. 缩放错误

由于 DSP 程序库函数会执行缩放处理，这类处理可能会导致错误的发生。

要获得有关缩放的详情，请参考*第 3.13 节“DSP 程序库”*。

11.1.39 DSP 程序库函数的最大取样数据计数

问题

DSP 程序库函数的最大取样数据计数是多少？

解答

DSP 程序库函数的最大取样数据计数主要取决于两个要素： DSP 存储器（X/Y 存储器）容量，以及函数是一个在位函数或是不在位函数。

对于在位函数：

maximum-sampling-count = *x-or-y-memory-size* / 2（short 类型大小）

对于不在位函数，最大取样计数应该是计算结果的一半，因为输入和输出区域需要隔开。

当 X-RAM 和 Y-RAM 是 8k：

- `FftComplex`

  最大取样计数：2048

  使用的堆大小：17334

- `FftReal`
  - 当输入数据放置在 X/Y 存储器外时

    最大取样计数：4096

    使用的堆大小：18358

  - 当输入数据放置在 X/Y 存储器内时

    最大取样计数：2048

    使用的堆大小：17334

- `IfftComplex`

  最大取样计数：2048

  使用的堆大小：17334

- `IfftReal`

  最大取样计数：2048

  使用的堆大小：19382 (17334 + 2048)

  （这是因为即使是在 `IfftReal()` 中，`malloc` 也用于分配区域。）

- FftInComplex

  最大取样计数：4096

  使用的堆大小：18358

- FftInReal

  最大取样计数：4096

  使用的堆大小：18358

- IfftInComplex

  最大取样计数：4096

  使用的堆大小：18358

- IfftInReal

  最大取样计数：4096

  使用的堆大小：18358

11.1.40 位字段的读/写指令

问题

```
struct bit{
 unsigned short int b0 : 1;
 unsigned short int b1 : 1;
 unsigned short int b2 : 1;
 unsigned short int b3 : 1;
 unsigned short int b4 : 1;
 unsigned short int b5 : 1;
 unsigned short int b6 : 1;
 unsigned short int b7 : 1;
 unsigned short int b8 : 1;
 unsigned short int b9 : 1;
 unsigned short int b10 : 1;
 unsigned short int b11 : 1;
 unsigned short int b12 : 1;
 unsigned short int b13 : 1;
 unsigned short int b14 : 1;
 unsigned short int b15 : 1;
}  ;
```

在上述代码中，我要定义一个位字段，并存取特定寄存器的 16 位宽的位，但我最后却使用字节和位运算指令执行存取。对于只能存取 16 位的寄存器，在生成字节存取或位运算指令时，我无法正确读取寄存器值。我该怎么做？

解答

只要程序中没有特殊的指定，位字段成员都通过编译程序优化的指令存取。 SH-2A 和 SH2A-FPU 生成位存取指令，而其他 CPU 则生成字节存取指令。 因此，存取可能通过意外的指令执行。 指定 volatile 以使用为成员变量设定的类型执行存取。

要防止编译程序改变存取方法以及多次存取，请明确地为您要防止这类改变的变量指定 volatile。

没有 volatile 的 C 源

```
struct bit reg;

void f()
{
        reg.b6=1;
}
```

没有 volatile 的汇编程序源
（除了 SH-2A 和 SH2A-FPU）

```
_f:                          ; 函数: f
                             ; 帧大小=0
           .STACK       _f=0
           MOV.L        L11+2,R6    ; _reg
           MOV.B        @R6,R0
           OR           #2,R0
           RTS
           MOV.B        R0,@R6
```

没有 volatile 的汇编程序源
（用于 SH-2A 和 SH2A-FPU）

```
f:                           ; 函数: f
.STACK           _f=0
      MOV.L          L11,R2      ; reg
      BSET.B         #1,@(0,R2)
      RTS/N
```

有 volatile 的 C 源

```
volatile struct bit reg;

void f()
{
         reg.b6=1;
}
```

有 volatile 的汇编程序源
（除了 SH-2A 和 SH2A-FPU）

```
_f:                                ; 函数: f
                                   ; 帧大小=0
           .STACK       _f=0
           MOV.L        L11+2,R6    ; _reg
           MOV          #2,R5       ; H'00000002
           MOV.W        @R6,R2
           SHLL8        R5
           OR           R5,R2
           RTS
           MOV.W        R2,@R6
```

有 volatile 的汇编程序源
（用于 SH-2A 和 SH2A-FPU）

```
f:                           ; 函数: f
                             ; 帧大小=0;
.STACK     _f=0
MOV.L      L11+2,R6;reg
MOV.W      @R6,R2
MOVI20     #512,R5   ;H'00000200
OR         R5,R2
RTS
MOV.W      R2,@R6
```

请注意，类型为 long long 的位字段，始终使用运行时例程存取。

C 源

```
struct bit{
 unsigned long long int b0 : 1;
 unsigned long long int b1 : 1;
 unsigned long long int b2 : 1;
 unsigned long long int b3 : 1;
 unsigned long long int b4 : 1;
 unsigned long long int b5 : 1;
 unsigned long long int b6 : 1;
 unsigned long long int b7 : 1;
};
```

```
struct bit reg;

void f()
{
        reg.b6=1;
}
```

汇编程序源

```
_f:                                 ; 函数: f
                                    ; 帧大小=12
          .STACK      _f=12
          STS.L       PR,@-R15
          MOV         #1,R1         ; H'00000001
          MOV.L       R1,@-R15
          MOV         #0,R4         ; H'00000000
          MOV.L       R4,@-R15
          MOV.L       L11+4,R1      ; _reg
          MOV.L       L11+8,R5      ; __bfs64u_p
          MOV.W       L11,R0        ; H'0601
          JSR         @R5
          MOV         R15,R2
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
           NOP
```

11.1.41 指定中断处理

问题

我要指定中断处理。 我该怎么做？

解答

要指定中断处理，请确定在设定 HEW 工程时先检查向量表定义。 由于包含中断处理函数模板的文件将会生成，请编辑此文件。 同时也需要注意 SH-1 和 SH-2 的中断处理格式与 SH-3 和 SH-4 的不同，而 HEW 生成的文件也不一样。

- 对于 SH-1 和 SH-2

中断处理需要：1) 中断处理函数，2) 向量表，以及 3) 初始化状态寄存器的中断屏蔽位。 以 SH-1 和 SH-2 为实例，IRQ0 中断导因的处理在 SH7020 工程中指定。

1. 中断处理函数

HEW 随附空的中断处理函数。intprg.c 文件包含 void INT_IRQ0(void) 的定义。您可以使用此函数来指定 IRQ0 处理。请注意，需要为中断处理函数指定“#pragma interrupt”。 此操作使用不需要更改的 vect.h 执行。

```
//intprg.c
// 64 中断 IRQ0
void INT_IRQ0(void)
{
/* 在这里指定处理 */
}
```

```
// vect.h
// 64 中断 IRQ0
#pragma interrupt INT_IRQ0
extern void INT_IRQ0(void);
```

2. 向量表

此表可在 HEW 生成时使用，并且不需要编辑。 向量表是 vecttbl.c 中的 void *INT_Vectors[]。在 SH-1 和 SH-2 中，发生中断时，控制将转移到在向量表中注册的其中一个函数。 IRQ0 的向量号是 64，可以对照硬件文档确认。 当中断的发生是因为 IRQ0 中断导因时，函数 INT_Vectors[60] 将会被调用 (60 = 64 - 4)。由于名为“INT_IRQ0”的函数在 INT_Vectors[60] 中注册，因此当中断是由 IRQ0 导致时，将会执行 INT_IRQ0。

```
void *INT_Vectors[] = {
// 4 非法代码
    (void*) INT_Illegal_code,
...
// 64 中断 IRQ0
    (void*) INT_IRQ0,
...
};
```

3. 初始化状态寄存器的中断屏蔽位

对于要使用的中断处理，必须正确初始化状态寄存器的中断屏蔽位。 在 resetprg.c 中，将 SR_Init 设定为适当的值，从 0x000000F0 开始。 在 PowerON_Reset_PC 中，set_cr 可用于设定状态寄存器的中断屏蔽位。

```
#define SR_Init     0x000000F0
```

- 对于 SH-3 和 SH-4

对于 SH-3 和 SH-4，中断处理也需要： 1) 中断处理函数，2) 向量表，以及 3) 初始化状态寄存器的中断屏蔽位。 以 SH-3 和 SH-4 为实例，IRQ0 中断导因的处理在 SH7705 工程中指定。

1. 中断处理函数

由于 HEW 随附空的中断处理函数，因此在定义新的函数需要将它删除。 因为 _INT_IRQ0 在 intprg.src 中定义，请将它删除，以及删除在 vect.inc 中的 .global INT_IRQ0 指定。然后，使用 C 语言如常定义 void INT_IRQ0(void)。 您不需要指定“#pragma interrupt”。

```
;intprg.src
...
;H'5E0 H-UDI
_INT_H_UDI
;H'600 IRQ0    ; 删除这个
_INT_IRQ0      ; 删除这个
;H'620 IRQ1
_INT_IRQ1
...
```

```
;vect.h
...
;H'5E0 H-UDI
.global          _INT_H_UDI
;H'600 IRQ0                    ; 删除这个
.global          _INT_IRQ0     ; 删除这个
;H'620 IRQ1
.global          _INT_IRQ1
...
```

2. 向量表

此表可在 HEW 生成时使用，并且不需要编辑。 向量表是 vecttbl.src 中的 _INT_Vectors。在 SH-3 和 SH-4 中，发生中断时，控制将转移到 vhandler.src 中的 IRQHandler。中断处理例程的地址从中断事件寄存器的值计算，然后控制将转移到该例程。IRQ0 的异常代码是 H'600，可以对照硬件文档确认。INT_Vectors 的偏移是 H'B8，从表达式取得：{(H'600 - H'40)} / 4 。由于 INT_Vectors 的元素大小是 4，INT_IRQ0，INT_Vectors 的第 46 个元素 (H'B6 / 4 = 46) 将会作为中断例程调用。IRQHandler 处理如下：

(1) 从中断事件寄存器取得异常代码。
(2) 取得 INT_Vectors 地址。
(3) 计算中断处理例程的地址。
(4) 取得中断屏蔽。
(5) 在 ssr 中设定中断屏蔽。
(6) 在 spc 中设定中断处理例程的地址。
(7) 使用 rte 对中断处理例程执行跳转。

```
      .org      H'500
_IRQHandler:
      PUSH_EXP_BASE_REG
;
      mov.l     #INTEVT,r0              ; 设定事件地址                    -(1)
      mov.l     @r0,r1                  ; 设定异常码
      mov.l     #_INT_Vectors,r0        ; 设定向量表地址                  -(2)
      add       #-(h'40),r1             ; 异常码 - h'40
      shlr2     r1
      shlr      r1
      mov.l     @(r0,r1),r3             ; 设定中断函数 addr               -(3)
;
      mov.l     #_INT_MASK,r0           ; 中断屏蔽表 addr
      shlr2     r1
      mov.b     @(r0,r1),r1             ; 中断屏蔽
      extu.b    r1,r1                                                     -(4)
;
      stc       sr,r0                   ; 保存 sr
      mov.l     #(RBBLclr&IMASKclr),r2  ; RB,BL, 屏蔽清除数据
      and       r2,r0                   ; 清除屏蔽数据
      or        r1,r0                   ; 设定中断屏蔽
      ldc       r0,ssr                  ; 设定当前状态                    -(5)
;
      ldc.l     r3,spc                                                    -(6)
      mov.l     #__int_term,r0          ; 设定中断终止
      lds       r0,pr
;
      rte                                                                 -(7)
      nop
;
      .pool
      .end
```

3. 初始化状态寄存器的中断屏蔽位

和 SH-1 及 SH-2 一样，必须为 SH-3 和 SH-4 正确初始化状态寄存器的中断屏蔽位。在 resetprg.c 中，将 SR_Init 设定为适当的值，从 0x000000F0 开始。 在 PowerON_Reset_PC 中，set_cr 可用于设定状态寄存器的中断屏蔽位。

```
#define SR_Init     0x000000F0
```

11.1.42 长时间运行程序时发生的一般无效指令异常

问题

设备运行 10 分钟到 2 小时后，发生一般无效指令异常，并且需要复位。 是否有方法可以分析那里出问题？

解答

基本上，这情况表示发生一般无效指令异常，但系统可能会失去控制并导致因为以下原因而引起的一般无效指令异常。如果系统在长时间操作后失去控制，很可能是出现 (2) 的情况。

(1) 执行了意外的中断。
(2) 堆栈溢出损毁有效的 RAM 数据。
(3) 板环境存在问题（如数据冲突或存储器软件错误）。
要找出问题的导因，请执行以下步骤和操作设备：

- 允许指令跟踪。
- 设定一般无效指令异常期间，中断函数跳转至的断点。

设备一旦操作和发生一般无效指令异常时，处理将会在为中断函数设定的断点停止。发生此情况时，分析指令跟踪的状态，然后确定问题的导因。

如果是堆栈溢出导致问题的发生，请使用以下分析方法：

- 为紧挨在堆栈区域起始地址之前的地址设定读取/写入中断存取。

设备一旦操作和发生会使堆栈溢出的存取时，处理将会在上面设定的断点停止。发生此情况时，如果存取指令是堆栈存取指令，导致问题的原因多数是堆栈溢出。

11.1.43 当整数计算的结果与预期的值不同时

问题

有时候，将整数乘法的结果替换为加长类型的变量时，会返回非预期的值。

将 60000*70000 更改为 60000*30000，则取得正确的值。

为何当乘法的结果超出 int 值时会取得不正确的值，即使替换是对加长类型的变量执行。

实例：

```
long long l_max;
        :
        l_max=60000*70000;
```

解答

尽管替换的变量是加长类型，所计算的整数会在指定为常数时被当作是 int 类型（4 字节）处理。

因此，在乘法期间，60000*70000 会变成 0xFA56EA00，但在对加长类型执行替换时，将会发生符号扩展，而它将变成 0xFFFFFFFFFA56EA00。

由于 60000*30000 变成 0x6B49D200，符号扩展将不会发生，它将变成 0x000000006B49D200，而正确的值也可以取得。

要取得预期的计算结果，您需在常数值后面指定 LL，以便使编译程序可以明确地识别该值为加长类型。

实例

```
long long l_max;
       :
    l_max = 60000LL * 70000LL;     // 在一个或两个常数后面指定 LL。
```

11.2 连接编辑程序

11.2.1 连接时出现“未定义符号”("Undefined symbol") 消息

问题

在连接时，出现“未定义符号”("Undefined symbol")消息。 为什么？

它表示什么？

解答

请检查以确定程序库已经连接。此外，也检查已经声明或使用的函数是否实际存在于代码中。有关详情，请参考第 3.15.2 (2) 节“有关连接的重要信息”。

11.2.2 连接时出现“再定位大小溢出”("RELOCATION SIZE OVERFLOW") 消息

问题

在连接时，我收到“再定位大小溢出”("RELOCATION SIZE OVERFLOW") 警告消息。 另外，我应该如何检查缺失的段地址指定？

解答

检查并确定 #pragma abs16、#pragma gbr_base 或 #pragma gbr_base1 的指定并没有超出限制。

段地址使用 START 命令以段名称指定；没有指定的段放置在已指定地址的最后一个段的后面。

编程中的此类错误会在具有大量段名称时特别频繁发生。

如果存在未使用 START 命令指定的段，此命令将会导致警告的输出。

(1) 消息实例

以下是连接编辑程序输出的选项和消息实例。

```
input sample.obj
input low/__main.obj
input low/__exit.obj
library lib/shclib.lib
library low/shclow.lib
output sample.abs
form a
entry _$main
start C,B,D,P/0400                    （缺失 $G0 和 $G1 的指定）
;start C,B,D,$G0,$G1,P(0400)          （正常终止的参数指定）

** L1120 (W) Section address is not assigned to "$G0"
** L1120 (W) Section address is not assigned to "$G1"

连接编辑程序已完成
```

因为 $G0 和 $G1 段名称未定义，警告消息将会输出。

11.2.3 连接时出现“段属性不符”("SECTION ATTRIBUTE MISMATCH") 消息

问题

在连接时，出现“段属性不符”("SECTION ATTRIBUTE MISMATCH") 警告消息。 我该怎么解决这个问题？

解答

此错误可能因为下列任何一项原因导致。

(1) 为同个段指定了不同的对齐。
检查是否没有为同个段名称指定不同的对齐。

(2) 尝试连接到使用“-cpu=sh4”选项编译的目标，以及使用不同 cpu 选项编译的目标。
在使用 cpu=sh4 选项（注释）的编译中，每个段将无条件地设定为 alignd ata8。 因此，对齐将与使用其他 cpu 选项编译的目标不同。 在此情形下，也可以使用连接编辑程序的 ALIGN_SECTION 选项/子命令来避免此问题。

(3) 第 11.2.4 节“转移到 RAM 并执行程序”中解答 2 所说明的修改，符合以下所有条件。 请注意，您可以忽略所有输出的警告。
(a) 程序段 P 的名称使用 C/C++ 编译程序的段选项或其他方法更改。
(b) 上面 (a) 中的段指定为转移源段。

注释： 在使用 cpu=sh4 选项的编译中，每个段将无条件地设定为 alignd ata8（5 或以下版本）。 存储器区域可能会因为八字节对齐而在段之间增加。

11.2.4 转移到 RAM 并执行程序

问题

我要将我的程序转移到执行比较快的 RAM，我该如何进行？

<操作环境>

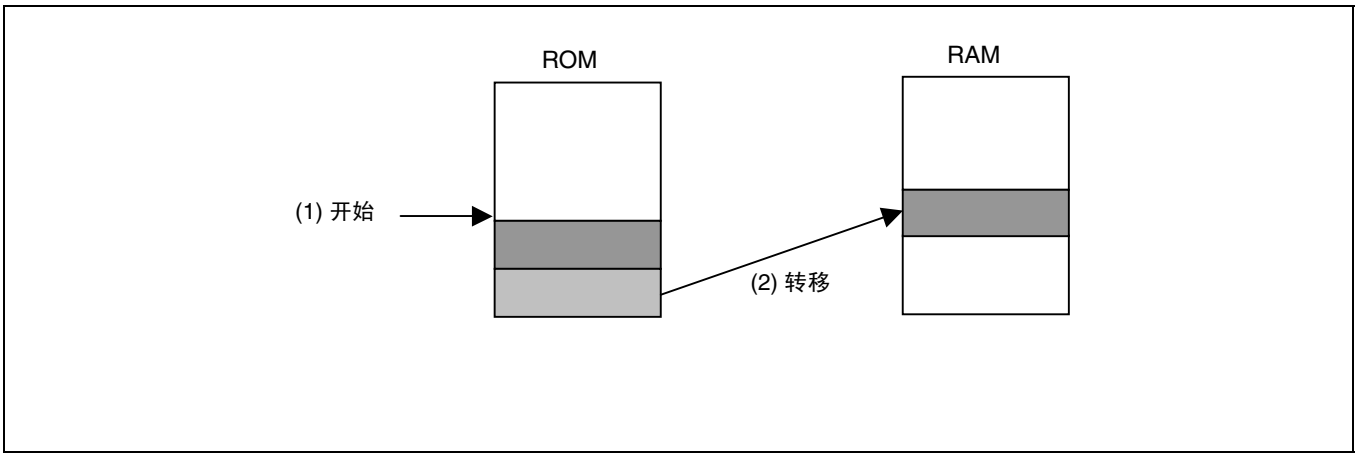


图 11.4 将程序从 ROM 转移到 RAM

<细节>

(1) 启动常驻在 ROM 中的程序。
(2) 将程序本身的代码的一些段转移到 RAM。

解答 1

必须将程序代码复制到 RAM 中的固定地址时，和初始化数据一样，ROM 支持 连接程序中可用来从 RAM 执行程序的函数（在连接时，将会出现地址解析，因此无法确定 RAM 中的地址并在运行时复制程序）。

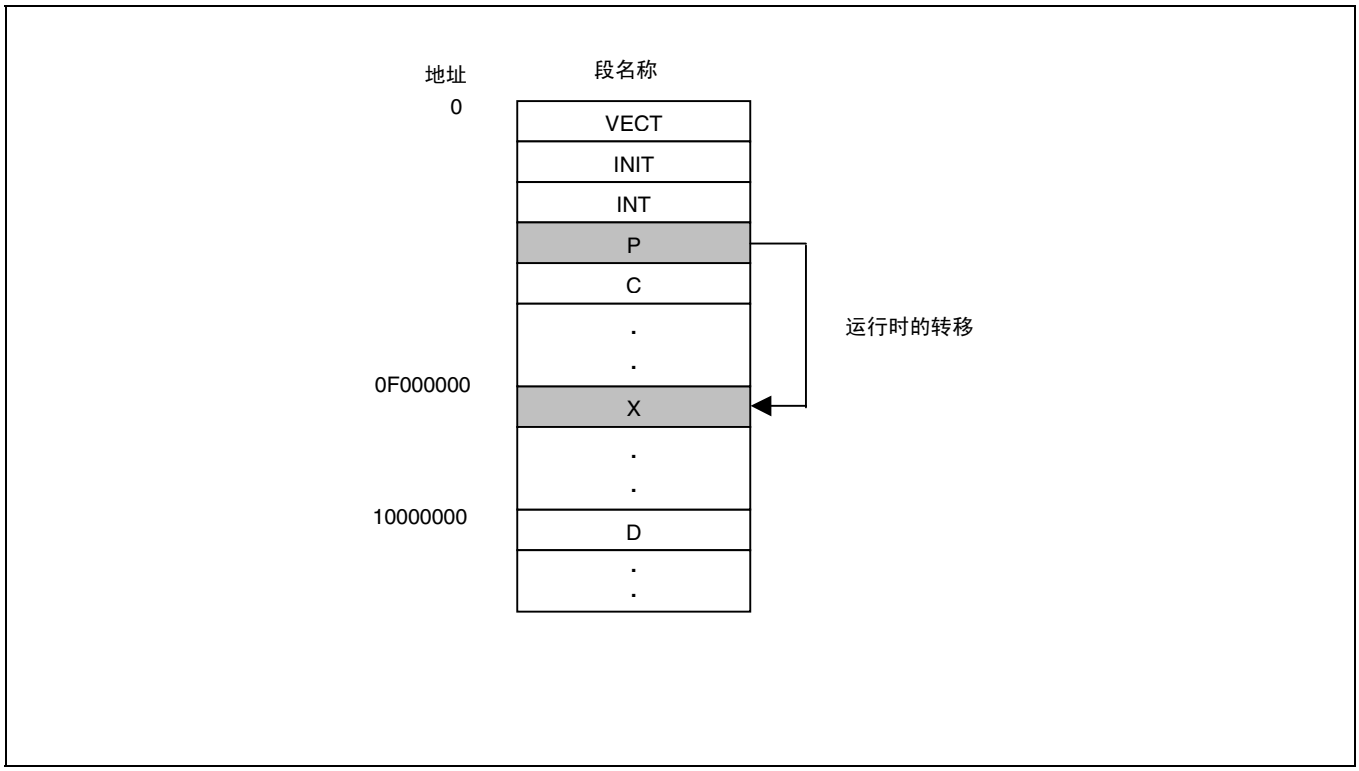


图 11.5 段配置的实例

下图 11.5 显示具有段配置的程序的实例。

C 语言部分

```
/*****************************************************************/
/*                      文件名 "init.c"                          */
/*---------------------------------------------------------------*/
/*       编译选项初始化程序段名称        */
/*****************************************************************/
#include "sample.h"             /* 包含第 2 节 的 sample.h 文件      */
extern int *_B_BGN,*_B_END;
extern int *_P_BGN;             /* P 段的起始地址                    */
extern int *_X_BGN;             /* X 段的起始地址                    */
extern int *_X_END;             /* X 段的终止地址                    */
extern void _INITSCT(void);
extern void _INIT();
extern void main();

void _INIT()
{
        _INITSCT();
        main();
        for ( ; ; )
            ;
}

void _INITSCT(void)
{
        int *p,*q;

        for ( p = _B_BGN; p < _B_END; p++ )
            *p = 0;

        /* 从 P 段复制到 X 段 */
        for ( p = _X_BGN, q = _P_BGN; p < _X_END; p++, q++ )
            *p = *q;
/*****************************************************************/
/*                      文件名 "main.c"                          */
/*---------------------------------------------------------------*/
/*            默认情况下，程序段名称为“P”。                      */
/*****************************************************************/
int a = 1;
int b;
```

```
const int c = 100;
void main(void)
{
        /* 此例程从复制目的地 (RAM) 执行 */
        for ( ; ; )
            ;
}

/*********************************************************************/
/*                        文件名“int.c”                              */
/*********************************************************************/
#include "sample.h"        /* 包含第 2 节 的 sample.h 文件           */
#include "7032.h"          /* 包含第 2 节 的 7032.h 文件             */
extern int a;              /* D 段代码                               */
extern int b;              /* B 段代码                               */
extern const int c;        /* C 段代码                               */
#pragma interrupt(IRQ0, inv_inst)

/*********************************************************************/
/*                      中断模块 IRQ0                                */
/*********************************************************************/
extern void IRQ0(void)
{
        a = PB.DR.WORD;
        PC.DR.BYTE = c;
}

/*********************************************************************/
/*                    中断模块 inv_inst                              */
/*********************************************************************/
extern void inv_inst(void)
{
        return;
}
```

汇编语言代码部分

```
;*****************************************************************
;*                     文件名“sct.src”                           *
;*****************************************************************
                .SECTION        P,CODE,ALIGN=4
                .SECTION        X,CODE,ALIGN=4
                .SECTION        B,DATA,ALIGN=4
                .SECTION        C,DATA,ALIGN=4

__P_BGN:      .DATA.L  (STARTOF P)                ; P 段的起始地址
__X_BGN:      .DATA.L  (STARTOF X)                ; P 段在 RAM 中的起始地址
__X_END:      .DATA.L  (STARTOF X)+(SIZEOF X)     ; P 段在 RAM 中的终止地址
__B_BGN:      .DATA.L  (STARTOF B)                ; BBS 段的起始地址
__B_END:      .DATA.L  (STARTOF B)+(SIZEOF B)     ; BBS 段的终止地址

                .EXPORT __P_BGN
                .EXPORT __X_BGN
                .EXPORT __X_END
                .EXPORT __B_BGN
                .EXPORT __B_END
                .END


;*****************************************************************
;*                     文件名“vect.src”                          *
;*****************************************************************
        .SECTION        VECT,DATA,ALIGN=4

        .IMPORT __INIT
        .IMPORT _inv_inst
        .IMPORT _IRQ0

        .DATA.L __INIT
        .DATA.L H'FFFFFFC
        .ORG     H'0080
        .DATA.L _inv_inst
        .ORG     H'0100
        .DATA.L _IRQ0
        .END
```

命令行的命令如下。

命令指定

shcΔ-debugΔ-section=P=INITΔinit.c
shcΔ-debugΔ-section=P=INTΔint.c
shcΔ-debugΔmain.c
asmshΔsct.srcΔ-debug
asmshΔvect.srcΔ-debug
optlnkΔ-nooptimizeΔ-sub=rom.sub

连接程序选项文件

```
;*********************************************************************
;*                      文件名“rom.sub”                              *
;*********************************************************************
sdebug
input vect, sct, init, int, main
ROM (P,X)                   ; 地址解析以将 P 段分配到 X
start VECT/0,INIT,INT,P,C,D/10000000,X/0f000000
                            ; VECT、INIT、INT、P、C、D 在 ROM 中，X 在 RAM 中。
output sample.abs
list sample.map
exit
```

通过以上代码，P 段的程序将复制到 X 段并执行。

INIT 段是执行复制的例程，因此必须与要复制的例程分开。 此处的主程序（P 段）从复制目的地运行。

解答 2

在 HEW2.0 或以上版本中，您可以使用优化连接编辑程序的 ROM 支持功能，在执行期间简易地将一个程序段复制到 RAM 中的固定地址（在连接时决定），然后从 RAM 执行该程序。

首先，在启动时转移要从 RAM 执行的程序段，指定该段的地址。 此处理将添加到 HEW 生成的 dbsct.c 文件中。 此时，PXX 段中的代码将会转移到 XX 段。 如下添加指定。

```
#pragma section $DSEC
static const struct {
              char *rom_s;        /* ROM 内已初始化数据段的起始地址        */
              char *rom_e;        /* ROM 内已初始化数据段的终止地址        */
              char *ram_s;        /* RAM 内已初始化数据段的起始地址        */
}DTBL[]= {{__sectop("D"), __secend("D"), __sectop("R")},
{__sectop("PXX"),__secend("PXX"),__sectop("XX")}};
#pragma section $BSEC
static const struct {
              char *b_s;          /* 未初始化数据段的起始地址              */
              char *b_e;          /* 未初始化数据段的终止地址              */
}BTBL[]= {__sectop("B"), __secend("B")};
```

以上是 PXX 段和 XX 段的设定

执行此处理后，副本将会在启动时从 PXX 段发送到 XX 段。

使用优化连接编辑程序指定转移目的地 XX 段的起始地址。

选择 [创建 (Build) -> SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain)... -> 优化连接程序 (Optimization Linker)]。 在打开的页中，选择类别 (category) 段，然后单击 [编辑 (Edit)] 按钮以显示段设定对话框。

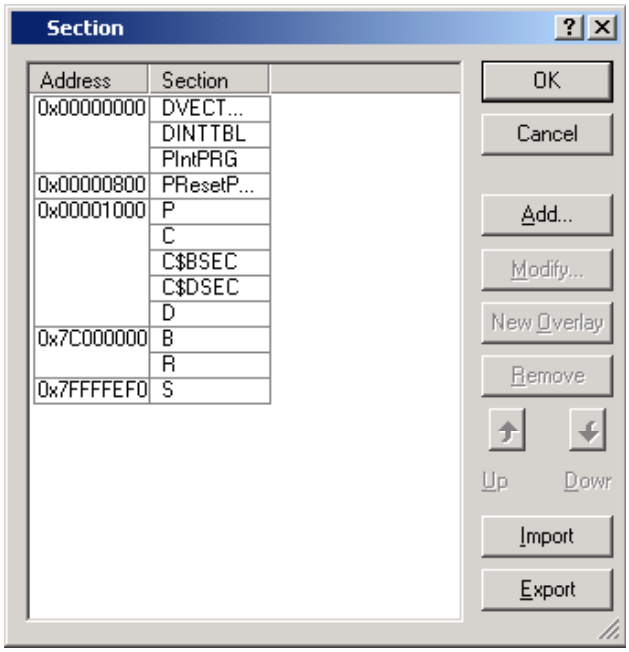


图 11.6“段设定”(Section Settings) 对话框

在此处，设定 PXX 段和 XX 段。

选择 [创建 (Build) -> SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain) -> 优化连接程序 (Optimization linker)]。在打开的页中，从 [类别 (category)] 选择 [输出 (Output)]，然后从 [选项项目 (Option item)] 选择 [从 ROM 到 RAM 映像的段 (Sections for mapping from ROM to RAM)]，以设定从 PXX 到 XX 的映像。

使用这些设定，程序将可以从 RAM 执行。

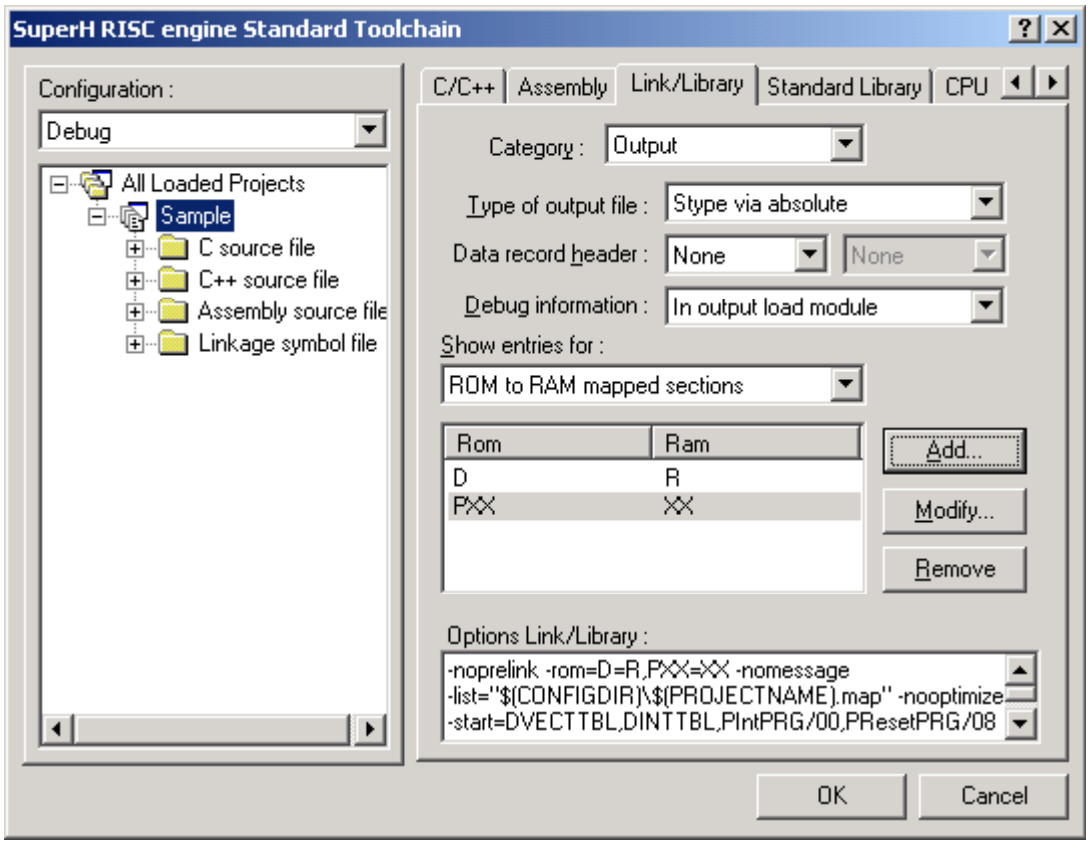


图 11.7“优化连接程序”(Optimization Linker) 对话框

注意：

通过 HEW2.0 或以上版本的改进，一般上消息将不会再输出，但在以下情形下，可能会输出与 HEW1.2 一样的警告消息“(L1323 (W) 段属性不符： "FXX"”(L1323 (W) Section attribute mismatch: "FXX")。请注意，这不是一个问题。

(1) 程序段 P 的名称使用 C/C++ 编译程序的段选项或其他方法更改。

(2) 上面 (1) 中的段指定为转移源段。

11.2.5 固定特定存储器中的符号地址以进行连接

问题

在内部 ROM 中固定一个程序后，我要为外部存储器开发一个程序，并且在以后只要更新外部存储器程序。

解答

在内部 ROM 中固定一个程序时，可使用连接命令 fsymbol 输出为内部 ROM 进行外部定义标签的定义文件。

定义文件由汇编程序 EQU 语句创建，因此在创建外部存储器程序时，可以汇编此文件并输入到 ROM 中的参考固定地址。

使用的实例：

图 11.8 说明产品 A 的功能 A 被修改为功能 B，以开发产品 B 的实例。使用此功能，通过解析共享 ROM 中的符号地址，即可使用公用 ROM。

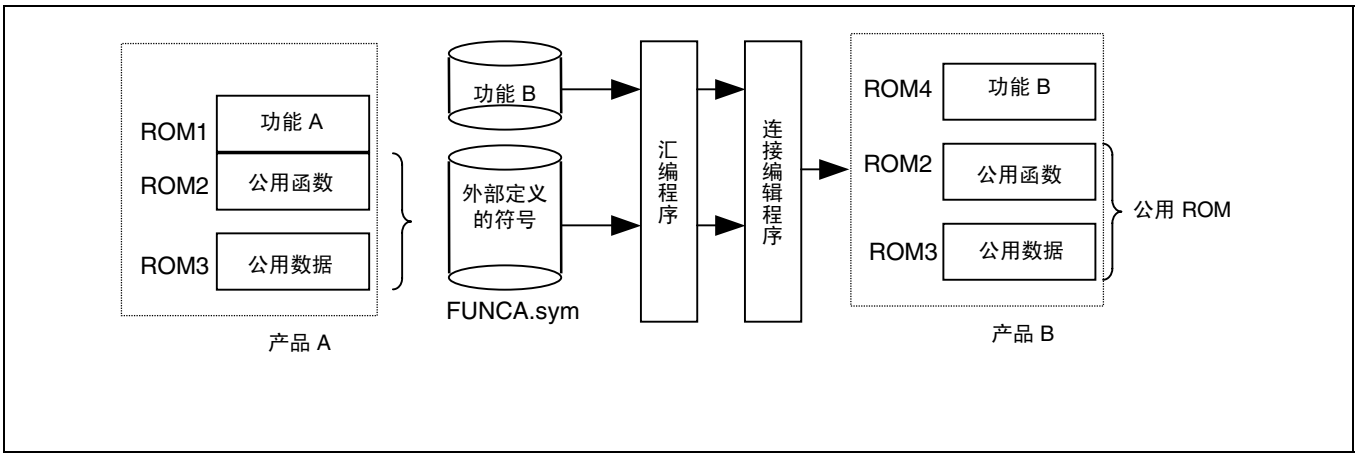


图 11.8 使用输出符号地址功能的实例

指定外部定义符号文件输出的实例

optlnkΔROM1,ROM2,ROM3Δ-output=FUNCAΔ-fsymbol=sct2,sct3

外部定义的符号 sct2 和 sct3 将输出到文件。

文件输出的实例 (FUNCA.sym)

```
;H 系列连接编辑程序生成的文件    1997.10.10
;fsymbol = sct2, sct3

;段名称 = sct1
.export sym1
sym1:  .equ     h'00FF0080
.export sym2
sym2:  .equ     h'00FF0100
;段名称 = sct2
.export sym3
```

```
sym3:  .equ     h'00FF0180
    .end
```

指定汇编和重新连接的实例

asmshΔROM4
asmshΔFUNCA.sym
optlnkΔROM4,FUNCA

ROM4 中外部参考的符号可以在无须连接目标文件 ROM2、ROM3 的情形下解析。

注意：使用此步骤时，功能 A 中的符号不能从公用函数参考。

11.2.6 使用覆盖

问题

我要在我的程序中使用覆盖。

在运行时，我要将一个程序从 ROM 转移到 RAM 以执行，但我要用同一 RAM 区域执行两个或以上不会同时执行的例程。

解答

要获得有关将程序从 ROM 转移到 RAM 以执行的信息，请参考第 11.2.4 节“转移到 RAM 并执行程序”。

程序的基本如下所示，但以下步骤是必需的。

- 实例

以下是将多个程序或数据集，从外部 ROM 转移到更快的内部 RAM 以执行的实例，它们不会同时存在。

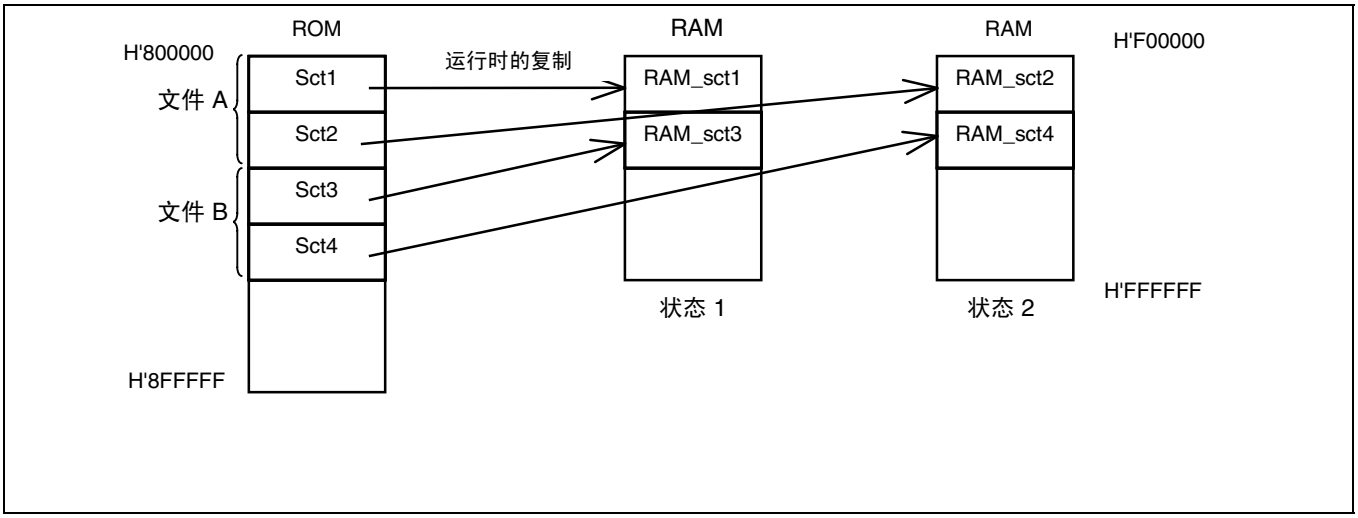


图 11.9 将多个段分配到同个地址

命令实例

optlnkΔ-subcommand=test.sub

test.sub 的内容

```
INPUT  A,B
ROM    Sct1=RAM_sct1
ROM    Sct3=RAM_sct3
ROM    Sct2=RAM_sct2
ROM    Sct4=RAM_sct4
START  Sct1,Sct2,Sct3,Sct4/800000
START  RAM_sct1,RAM_sct3:RAM_sct2,RAM_sct4/0F00000
```

说明

RAM_sct1 和 RAM_sct2 将会从同个地址分配。 RAM_sct3 与 RAM_sct1 连接，而 RAM_sct4 则与 RAM_sct2 连接。

11.2.7 为未定义的符号指定错误输出

问题

我要在连接期间若出现未定义的符号时输出错误消息，以及防止装入模块的输出。

解答

UDFCHECK 选项应该在连接时指定。

使用此方法，如果存在任何未定义的符号，错误消息 221 将会输出，而装入模块的输出也会被制止。

（如果没有指定 UDFCHECK 选项/子命令，警告消息 105 将会显示，而装入模块将会生成。）

但是，在连接编辑程序 7 或以上版本中，UDFCHECK 选项已被删除，而 UDFCHECK 将永远允许。

11.2.8 S 类型文件的统一输出格式

问题

我要统一 S 类型文件的 S1、S2、S3 混合输出格式。

解答

这可以通过指定数据记录（S1、S2、S3）的输出实现，不论选项的载入地址是什么。

实例： optlnk test.abs -form=stype -output=test.mot -record=s2 ； 所有数据将由 S2 输出。

11.2.9 输出文件的分隔

问题

我要将每个 ROM 设备的输出文件分隔成一些文件。

解答

如果在输出文件名称的末端指定一个起始地址和终止地址，将可以输出所指定区域的目标。输出文件名称可以指定超过两个。

实例：optlnk test.abs -form=stype -output=test1. mot=0-FFFF test2.mot=10000-1FFFF; 0x0-0xFFFF 的一个区域输出到 test1.mot，0x10000-0x1FFFF 的一个区域输出到 test2.mot。

11.2.10 在 Windows 2000 上执行 optlinksh.exe

问题

如果在 Windows2000 上执行“optlnksh.exe”，将会输出 [2020 语法错误 (2020 SYNTAX ERROR)]。

解答

请确定环境变量 SHC_TMP 中是否有空格。

即使 SHC_TMP 有空格，它也可以在 shc 中正确执行，但会在 optlnksh 中出现一个错误（2020 语法错误）。Windows 2000 中的临时目录是 C:\Documents and Settings\foo\Local Settings\Temp（foo 是用户名）。

11.2.11 优化连接编辑程序的输出文件格式

问题

请告诉我 ROM 写入程序可使用的装入模块文件格式。

解答

由优化连接编辑程序输出的装入模块如下所示：

为 ROM 写入程序创建装入模块时，以十六进制或 SType 格式将它输出。 在此情形下，将不会输出调试信息。

- 优化连接编辑程序支持 C/C++ 编译程序 7.1 版本、8.0 版本在调试时以 ELF/DWARF2 格式输出装入模块 。 由更早版本创建的装入模块，以 SYSROF 或 ELF/DWARF1 格式输出，因此必须使用 ELF/DWARF 格式转换器更改该格式以便在最新的版本中使用。

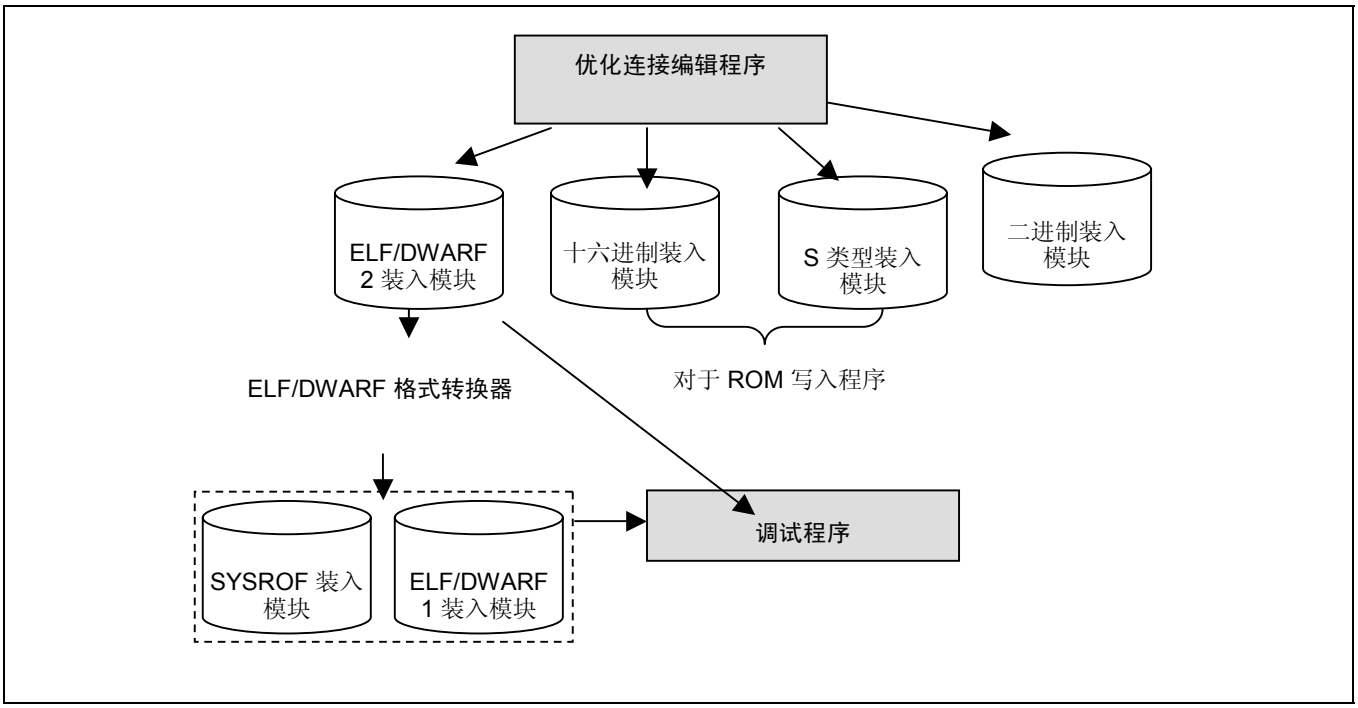


图 11.10 优化连接编辑程序输出装入模块

11.2.12 计算程序大小（ROM 和 RAM）的方法

问题

您可以告诉我如何正确测量 ROM 和 RAM 大小吗？

解答

您可以在优化连接编辑程序输出的列表文件中检查它们。

指定方法

从对话框菜单： [优化连接程序 (Optimization Linker)] 标签 -> [类别 (Category)]: [列表 (List)] 连接列表输出

从命令行：-list=*文件名*

检查方法

指定此选项以输出以下列表文件 (*.map)。

在此情形下，由于代码属性段来自 DVECTTBL、DINTTBL、PIntPRG、PResetPRG、P、C$BSEC、C$DSEC 和 D，ROM 大小是 0x00006a8。

RAM 区域来自 B、R 和 S，因此其大小是 0x52c。

*** 映像列表 ***

```
SECTION                          START      END         SIZE   ALIGN

DVECTTBL
                                 00000000   0000000f      10       4
DINTTBL
                                 00000010   000003ff     3f0       4
PIntPRG
                                 00000400   00000557     158       4
PResetPRG
                                 00000800   00000833      34       4
P
                                 00001000   000010db      dc       4
C$BSEC
                                 000010dc   000010e3       8       4
C$DSEC
                                 000010e4   000010ef       c       4
D
                                 000010f0   0000111b      2c       4
B
                                 7c000000   7c0003ff     400       4
R
                                 7c000400   7c00042b      2c       4
S
                                 7c000500   7c0005ff      100      4
```

11.2.13 输出段对齐不符时

问题

当我输入如下的二进制文件，以及通过段地址运算符参考二进制文件的段名称时，输出 L1322 警告。 我该如何避免这个问题？

[指定的选项]

```
binary=project.bin(BIN_SECTION)
```

[C/C++ 程序]

```
void main(void)
{
    unsigned char *s_ptr;
    s_ptr = __sectop("BIN_SECTION");
    dummy(s_ptr);
}
```

解答

使用段地址运算符（__sectop 和 __second）时，具备大小为 0 以及边界对齐号为 4 的段，将会在编译程序所生成的代码中创建，如下所示。

在此情形下，二进制段将会输入，但二进制段实体的边界对齐号将是 1。由于同个段名称具有超过一个边界对齐号，因此输出 L1322 警告消息。

请注意，尽管输出此警告消息，程序的操作将不会受影响。

此警告消息可以通过在使用优化连接程序输入二进制文件时指定边界对齐号来避免。

[使用 when __sectop 代码]

```
_main:                              ; 函数: main
                                    ; 帧大小 =0
          .STACK      _main=0
          MOV.L       L13+2,R4      ; STARTOF BIN_SECTION
          BRA         _dummy
          NOP
...
          .SECTION    . BIN_SECTION,DATA,ALIGN=4        ; 具备大小为 0，以及具备
                                                          边界对齐号为 4 的段
          .END
```

如何避免警告的实例

从对话框菜单： [优化连接程序 (Optimization Linker)] 标签 -> [类别 (Category)]: [输入 (Input)] 选项项目： 二进制文件

从命令行： binary=binary_data.bin(BIN_SECTION:4)

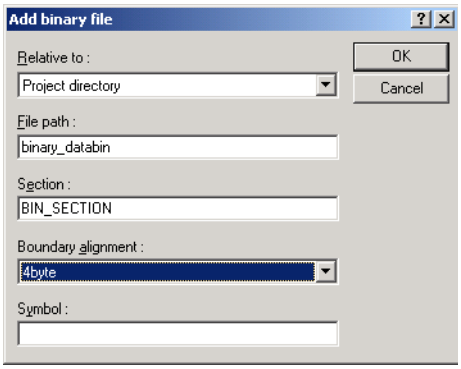


图 11.11“添加二进制文件”(Add binary file) 对话框

说明

输入二进制文件时的边界对齐号指定受连接编辑程序 9 或以上版本的支持。

有关详情，请参考第 9.1.1(4) 节“二进制文件”。

11.3 标准程序库

11.3.1 可重入的函数和标准程序库

问题

使函数可重入时我需要特别注意的警惕有哪些？

解答

使用全局变量的函数不可重入。

此外，即使函数创建为可重入，如果使用标准程序库 时采用以下标准包含文件，将会使用全局变量，因此该函数将不再是可重入。

以下标示可重入程序库函数的的列表。在表中，_errno 变量设定为以三角形表示的函数；若 _errno 未在程序中参考，可重入将可用。

您也可以使标准程序库可重入。要获得有关如何使标准程序库可重入的详情，请参考第 11.3.2 节“我要在标准程序库文件中使用可重入程序库函数”。

表 11.14 可重入程序库函数列表 (1)

可重入的列　O： 可重入; X： 不可重入; Δ: _errno 变量集

编号	标准包含文件		函数名称	可重入	编号	标准包含文件		函数名称	可重入
1	stddef.h	1	offsetof	O	4	math.h	18	acos	Δ
2	assert.h	2	assert	x			17	asin	Δ
3	ctype.h	3	isalnum	O			18	atan	Δ
		4	isalpha	O			19	atan2	Δ
		5	iscntrl	O			20	cos	Δ
		6	isdigit	O			21	sin	Δ
		7	isgraph	O			22	tan	Δ
		8	islower	O			23	cosh	Δ
		9	isprint	O			24	sinh	Δ
		10	ispunct	O			25	tanh	Δ
		11	isspace	O			26	exp	Δ
		12	isupper	O			27	frexp	Δ
		13	isxdigit	O			28	ldexp	Δ
		14	tolower	O			29	log	Δ
		15	toupper	O			30	log10	Δ

表 11.14 可重入程序库函数列表 (2)

编号	标准包含文件		函数名称	可重入	编号	标准包含文件		函数名称	可重入
4	math.h	31	modf	Δ	7	stdio.h	61	fputs	x
		32	pow	Δ			62	getc	x
		33	sqrt	Δ			63	getchar	x
		34	ceil	Δ			64	gets	x
		35	fabs	Δ			65	putc	x
		36	floor	Δ			66	putchar	x
		37	fmod	Δ			67	puts	x
5	setjmp.h	38	setjmp	O			68	ungetc	x
		39	longjmp	O			69	fread	x
6	stdarg.h	40	va_start	O			70	fwrite	x
		41	va_arg	O			71	fseek	x
		42	va_end	O			72	ftell	x
7	stdio.h	43	fclose	x			73	rewind	x
		44	fflush	x			74	clearerr	x
		45	fopen	x			75	feof	x
		46	freopen	x			76	ferror	x
		47	setbuf	x			77	perror	x
		48	setvbuf	x	8	stdlib.h	78	atof	Δ
		49	fprintf	x			79	atoi	Δ
		50	fscanf	x			80	atoll	Δ
		51	printf	x			81	strtod	Δ
		52	scanf	x			82	strtol	Δ
		53	sprintf	Δ			83	rand	x
		54	sscanf	Δ			84	srand	x
		55	vfprintf	x			85	calloc	x
		56	vprintf	x			86	free	x
		57	vsprintf	Δ			87	malloc	x
		58	fgetc	x			88	realloc	x
		59	fgets	x			89	bsearch	O
		60	fputc	x			90	qsort	O

表 11.14 可重入程序库函数列表 (3)

编号	标准包含文件		函数名称	可重入	编号	标准包含文件		函数名称	可重入
8	stdlib.h	91	abs	○	9	string.h	103	memchr	○
		92	div	△			104	strchr	○
		93	labs	○			105	strcspn	○
		94	ldiv	△			106	strpbrk	○
9	string.h	95	memcpy	○			107	strrchr	○
		96	strcpy	○			108	strspn	○
		97	strncpy	○			109	strstr	○
		98	strcat	○			110	strtok	x
		99	strncat	○			111	memset	○
		100	memcmp	○			112	strerror	○
		101	strcmp	○			113	strlen	○
		102	strncmp	○			114	memmove	○

11.3.2 我要在标准程序库文件中使用可重入程序库函数。

问题

我要在标准程序库文件中使用可重入程序库函数。

解答

可重入函数列表在 [11.3.1 可重入程序库] 上提供。 可重入函数可以通过在 SHC 7.0 或以上版本中设定程序库生成程序来生成。

- 在命令行上，使用 lbgsh -reent 选项。
- HEW 中的这项设定在图 11.12 中显示。

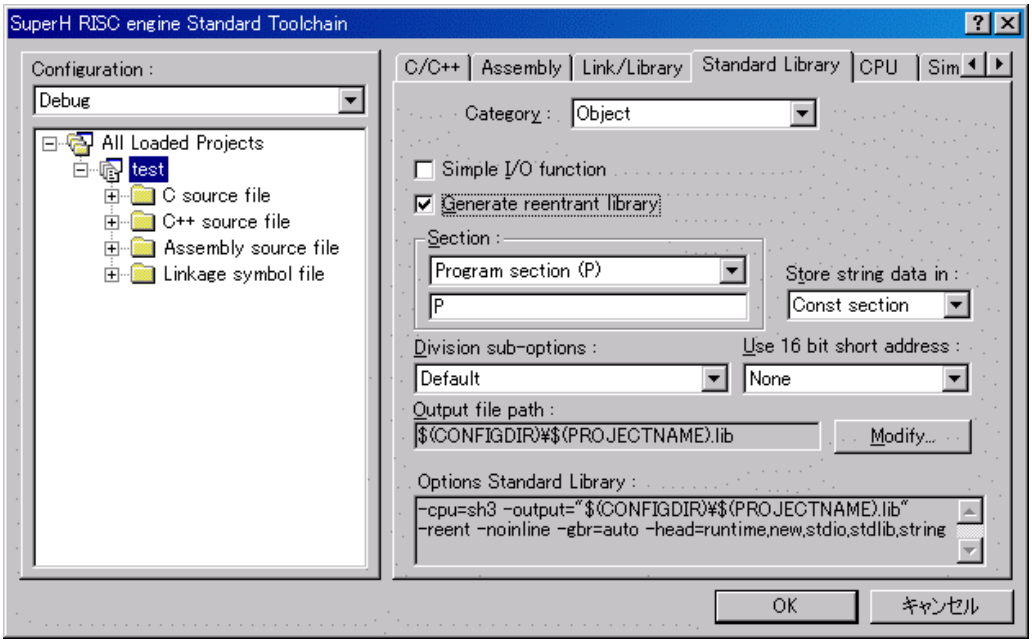


图 11.12“标准程序库”(Standard Library) 对话框

11.3.3 没有标准程序库文件。（SHC V6、7、8、H8C V4、5）

问题

没有标准程序库文件。（SHC V6、7、8、H8C V4、5）

解答

请使用程序库生成程序来生成标准程序库文件，因为标准程序库文件并没有在 SHC V6、H8C V4 或以上版本产品中连接。

11.3.4 创建标准程序库时的警告消息

问题

生成标准程序库文件时，可能会输出 [L1200(W) 将文件“a.lib”备份到“b.lbk”中 (L1200(W) Backed up file "a.lib" into "b.lbk")]。

解答

这只是一则警告消息，说明 HEW 将会在它生成新的程序库文件时备份文件。

如果您在 HEW/[选项(OPTIONS)]/[SuperH RISC engine 标准工具链(SuperH RISC engine Standard Toolchain)] 中的 [标准程序库(Standard Library)] 模式：选择“创建程序库文件（选项更改）”(Build a library file (option changed))，将不会发出警告。 当您在 HEW 中选择“全部创建”(BUILD ALL) 时，连接编辑程序将会先生成一个标准程序库。 对于您创建的第一个工程，创建标准程序库是必要的，因此您必须在 HEW/[选项(OPTIONS)]/[SuperH RISC engine 标准工具链(SuperH RISC engine Standard Toolchain)] 中的 [标准程序库(Standard Library)] 模式，选择“创建程序库文件”(Build a library file)。

然而，一旦在文件中指定“全部创建”(BUILD ALL)，标准程序库就会在该文件中创建，因此不需要对该文件进行自动生成标准程序库。 在此情形下，由于每一个“全部创建”(BUILD ALL) 指定都会自动生成标准程序库，现有程序库将会进行备份。

如果您选择“创建程序库文件（选项更改）”(Build a library file (option changed))，此警告消息将可以避免。 另外，此操作也可以节省“全部创建”(BUILD ALL) 时自动生成标准程序库所需的时间。

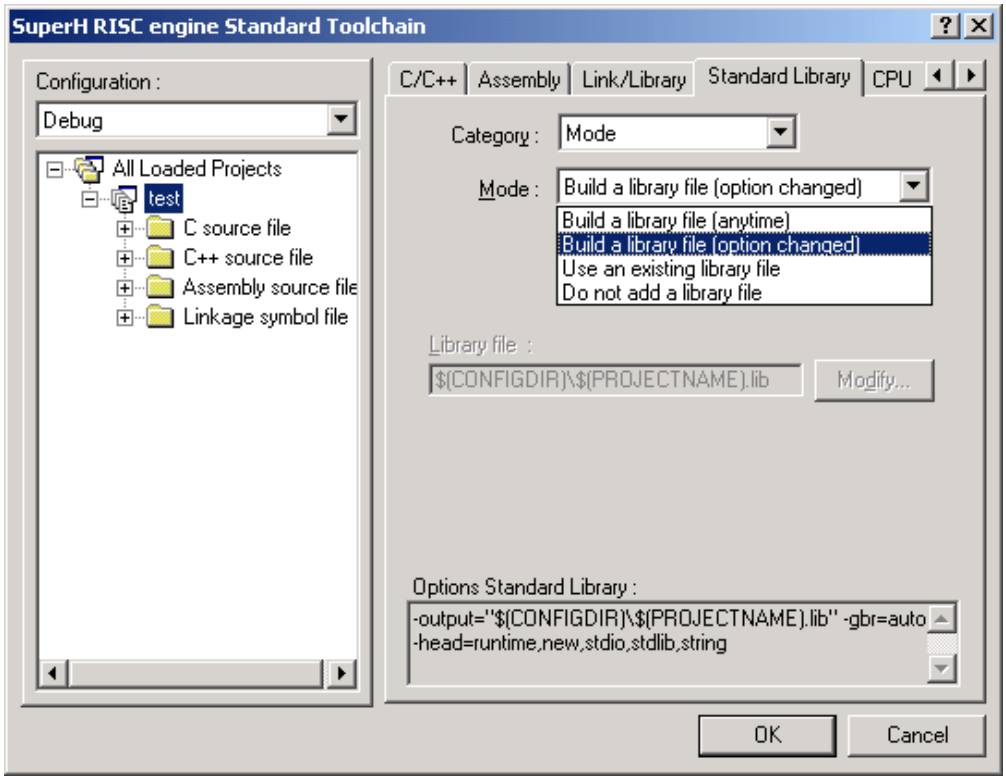


图 11.13“标准程序库”(Standard Library) 对话框

11.3.5 用作堆的存储器大小

问题

请告诉我如何计算用作堆的存储器大小。

解答

用作堆的存储器大小是 C/C++ 程序中的存储器管理程序库函数（calloc、malloc、ralloc、new）分配的总存储器区域。但是，这些函数在每次被调用时将四个字节用作管理区域。堆的大小可以通过将此大小加实际分配区域的大小计算出来。

编译程序以 1024 字节单位管理堆。作为堆 (HEAPSIZE) 分配的区域的大小计算如下：

HEAPSIZE = 1024 x n (n≥1)

（由存储器管理程序库分配的区域大小）+（管理区域大小≤HEAPSIZE）

I/O 程序库函数在内部处理中使用存储器管理程序库函数。I/O 期间分配的区域大小是 516 字节 x 同时打开的文件的最大数量。

注意： 由存储器管理程序库函数释放或删除后所释放的区域，由用于分配的存储器管理程序库函数重新使用。即使释放区域的总大小足够，重复分配将会导致释放区域划分为更小的区域，使分配大区域的新请求无法实现。要防止此情形的发生，请按照以下建议使用堆区域。

a. 大的区域应该在程序开始运行后立即分配。

b. 要释放和重新使用的数据区域的大小必须是常数。

11.3.6 编辑程序库文件

问题

我应该如何编辑现有的程序库文件，以便让我能够重新使用它？

解答

现有程序库文件可以使用优化连接编辑程序的选项编辑。 以下说明每个编辑功能。

H 系列库管理程序界面专为从 GUI 启动优化连接编辑程序而提供。

启动 H 系列库管理程序界面

要启动 H 系列库管理程序界面，从 HEW，选择 [工具 (Tools) -> H 系列库管理程序界面(H Series Librarian Interface)]。

（A） 更改程序库中模块的段名称。

您可以更改段名称，然后将该段放置在程序库中特定模块的特定地址。

(1) 打开适当的程序库，然后选择您要将它分配到特定地址的模块。

(2) 选择 [操作 (Action) -> 重命名段 (Rename Section) …] 以显示下列对话框，然后单击 [之后 (After)] 按钮以更改段名称。

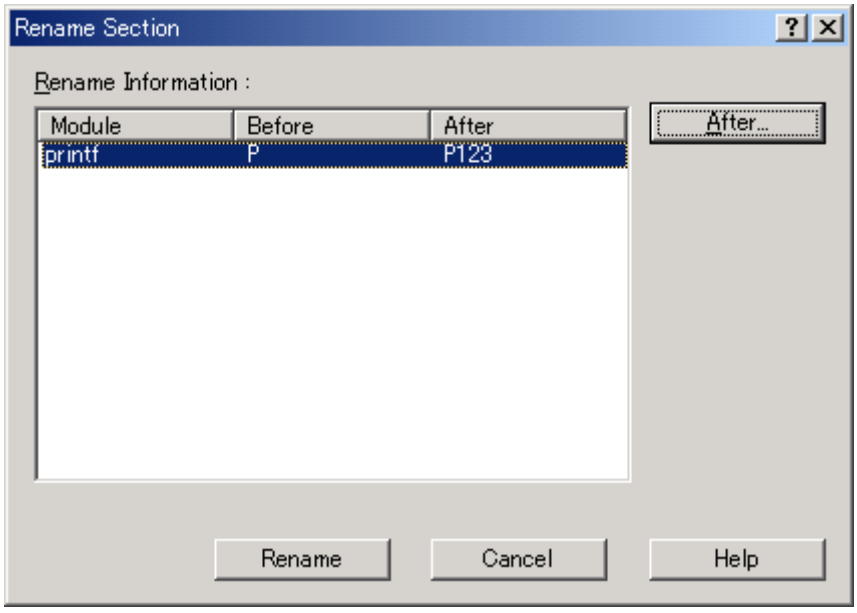


图 11.14 “重命名段” (Rename Section) 对话框

[对于命令行]

optlnk –form=lib –lib=*程序库文件名* -rename=*程序库中模块的名称* (P=P123)

(B) 交换程序库中的模块并将新的模块添加到程序库中

您可以交换程序库模块，也可以添加新的。

(1) 打开适当的程序库，然后选择 [操作(Action) -> 添加/替换(Add/Replace)...]。

(2) 打开要交换的具备相同名称的模块。 如果打开具备不同名称的模块，该模块将会添加。

[对于命令行]

optlnk –form=lib –lib=*程序库文件名* -replace=*程序库中模块的名称*

(C) 删除程序库中的模块

您可以删除程序库模块。

(1) 打开适当的程序库，然后选择您要删除的一个或多个模块。

(2) 选择 [操作 (Action) -> 删除 (Delete) …] 以显示“删除” (Delete) 对话框，然后单击 [删除 (Delete)] 按钮。

[对于命令行]

optlnk –form=lib –lib=*程序库文件名* -delete=*程序库中模块的名称*

(D) 从程序库提取模块

您可以提取程序库模块。

(1) 打开适当的程序库，然后选择您要提取的一个或多个模块。

(2) 选择 [操作 (Action) -> 提取 (Extract) …] 以显示下列对话框，设定输出目的地，然后单击 [确定 (OK)] 按钮。

(3) 模块将会输出到设定的输出目的地（在下例中是 C:\ ）。

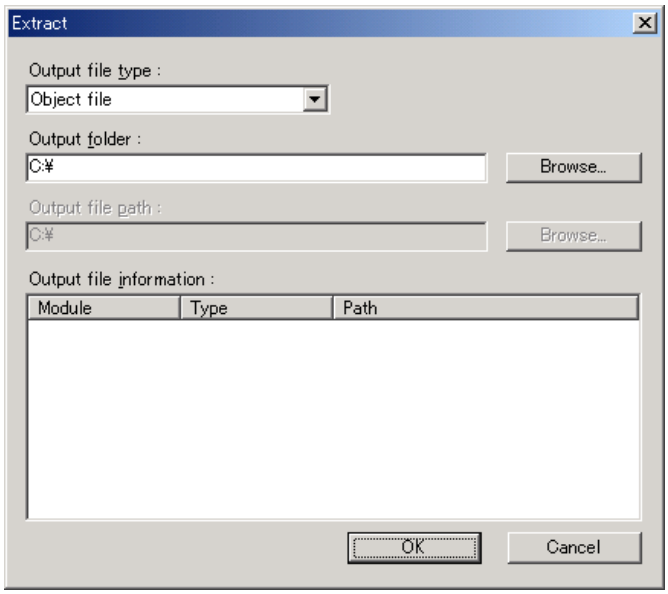


图 11.15 “提取” (Extract) 对话框

[对于命令行]

optlnk –lib=*程序库文件名* -extract=*程序库中模块的名称* -form=*输出文件格式*

请注意，此例的输出格式是目标。

11.4 HEW

11.4.1 显示对话框菜单失败

问题

HIM 和 HEW 的“工具选项”（Tools Option）对话框未正确显示。

解答

如果使用了旧版本（如 400.950a）的 Windows®95，在打开 C/C++ 编译程序、汇编程序或 IM OptLinker 的选项时，将会产生应用程序错误，HEW 可能会异常终止操作或选项对话框不正确显示。 此问题是因为位于 Windows 目录的系统目录内的 COMCTL32.DLL 文件版本太旧所造成。 在此情形下，必须升级 Windows®95。

11.4.2 目标文件的连接顺序

问题

我要指定 HEW 上的目标文件连接的顺序。

解答

请按下 [添加(Add)] 以添加目标文件，然后 从“ SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 的“连接/程序库”(Link/Library) 标签中的类别 [输入(Input)]， 选择“显示有关项目： ”(Show entry for:) [可再定位文件和目标文件 (Relocatable files and object files)]。 这次，目标将以指定的顺序连接。

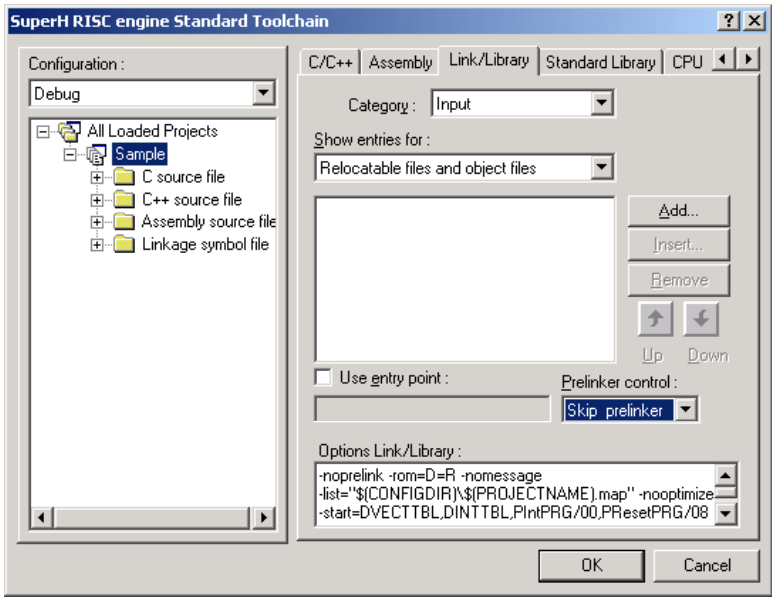


图 11.16“连接/程序库”(Link/Library) 对话框

SHC 8.00 版本 02 或以上版本简化了连接顺序的指定。

要显示用于自定义连接顺序的对话框，请选择 [创建 (Build)]，然后选择 [指定连接顺序 (Specify link order)] 。

在此处指定连接顺序。 列表中越高位置的项目将会先连接。

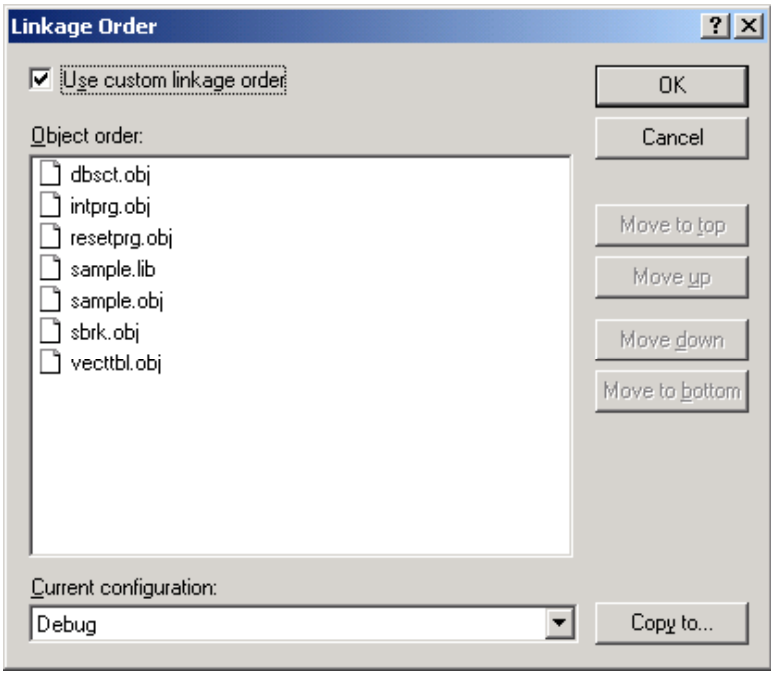


图 11.16“连接顺序”(Linkage Order) 对话框

11.4.3 指定 MAP 优化

问题

指定 MAP 优化将导致警告消息显示。

解答

如果您在 SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain) 的 C/C++ 标签类别： [优化(Optimize)] 中核选“包含映像文件”(Include map file)，警告消息将会显示，如图 11.18 所示。这将自动允许“连接/程序库”(Link/Library) 标签类别：[输出(Output)] 中的“生成映像文件” (Generate map file)。

要指定 MAP 优化，请从 C/C++ 标签类别：[优化(Optimize)] 选择“包含映像文件”(Include map file)，以及从 SuperH RISC engine 标准工具链 (SuperH RISC engine Standard Toolchain) 的“标准程序库”(Standard Library) 标签类别： [优化 (Optimize)]) 选择“包含映像文件”(Include map file)。

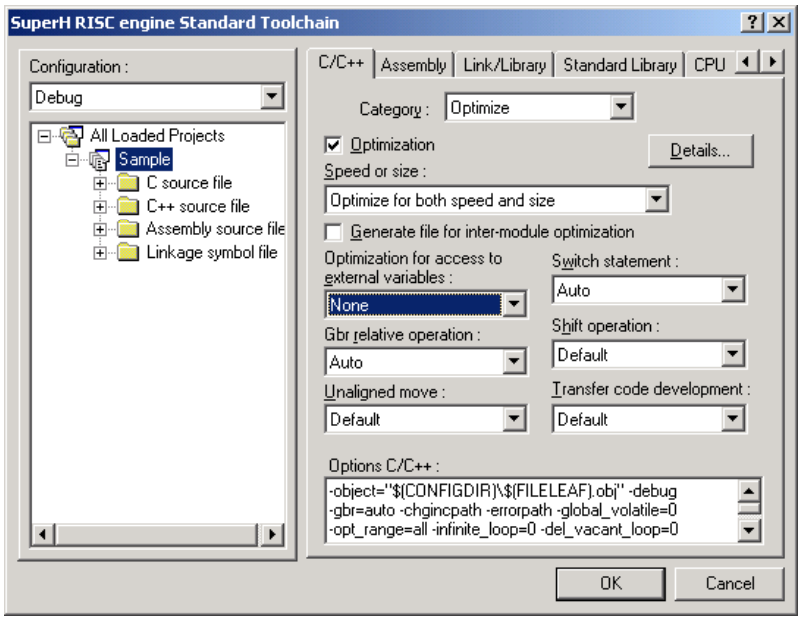


图 11.18 C/C++ 对话框

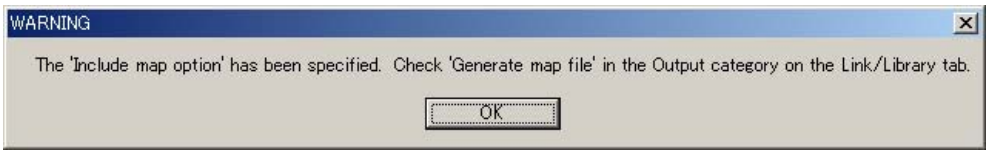


图 11.19 警告消息

11.4.4 排除工程文件

问题

我要暂时从“创建”(Build) 删除工程文件。

解答

如果在工作空间窗口中的“工程”(Projects) 标签的文件右击鼠标来选择 [排除创建 <文件>(Exclude Build <file>)]， 该文件将会从“创建”(Build) 删除。 如果要再次将文件传送回“创建”(Build)，请在工作空间窗口中的“工程”(Projects) 标签的文件上右击鼠标来选择 [纳入创建 <文件>(Include Build <file>)]。

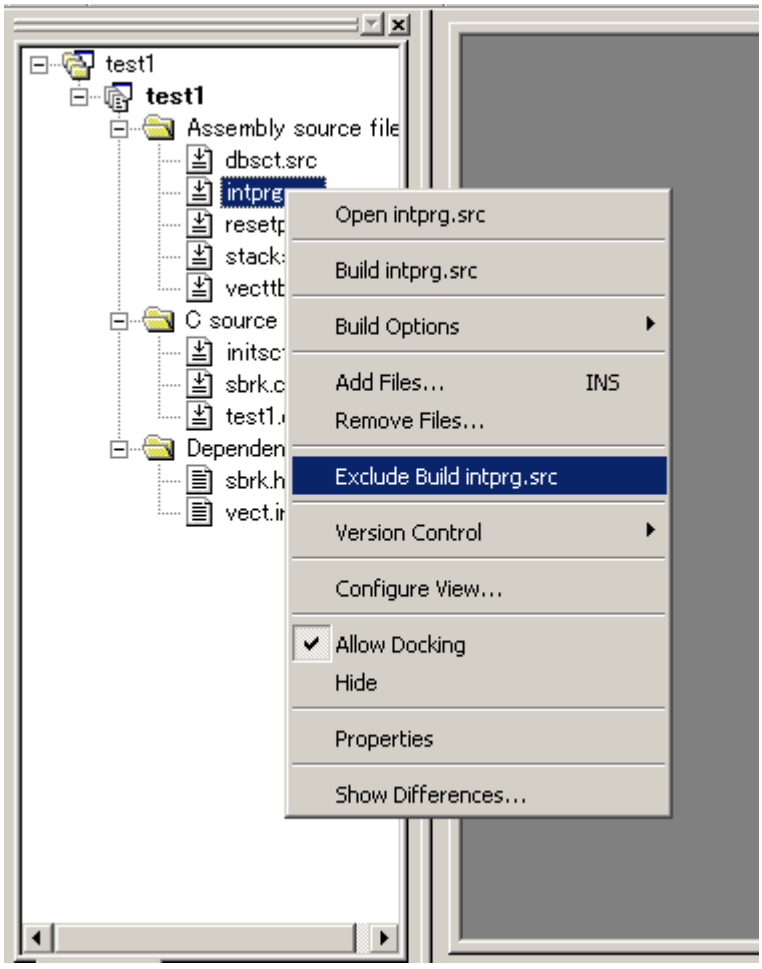


图 11.20 排除创建菜单

11.4.5 为工程文件指定默认选项

问题

我希望在添加工程到文件时自动指定默认选项到文件中。

解答

文件列表在“SuperH RISC engine 标准工具链”(SuperH RISC engine Standard Toolchain) 的左侧显示（请参考下图）。 请在文件组中打开要使用文件列表指定默认选项的文件夹。“Default Option” 图标将显示在文件夹中。 请选择该图标，在“选项”(option) 对话框的右侧指定选项，然后单击“确定”（OK）。 此选项可以在文件组中的文件首次添加到工程时应用。

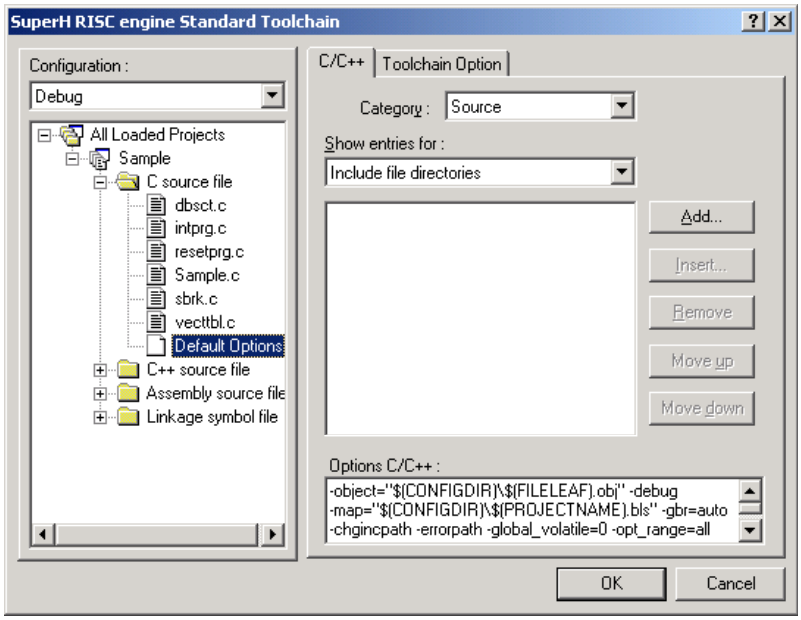


图 11.21 默认选项

11.4.6 更改存储器映像

问题

无法更改存储器映像。

解答

当存储器窗口的存储器源已经映像时，存储器映像将不能在系统配置窗口中更改。请在存储器资源的映像释放时更改存储器映像。

11.4.7 如何在网络上使用 HEW

问题

(1) HEW 可以安装在网络上吗？
(2) 工程和程序可以安装在网络上吗？

解答

(1) HEW 系统本身不可在网络上安装。
(2) 没问题。 但要确保单一文件不会同时被两个以上的用户所存取。

11.4.8 使用 HEW 创建文件和目录名称的限制

问题

HEW 系统启动时，显示“在保存文件 <文件名> 时出错” (“Error has occurred whilst saving file <filename>”) 的消息。 为什么？

解答

在 HEW 系统中创建的文件和目录存有限制。

对于以下项目的指定，仅可使用半角字母数字字符以及半角下划线：

- 要安装的目录名称
- 将要创建工程的目录名称
- 工程名称

11.4.9 使用 HEW 编辑程序或 HDI 显示日文字体失败

问题

(1) HEW 编辑程序不显示日文字体。
(2) HEW 编辑程序的日文字符会旋转 90 度。
(3) 模块间优化器生成“语法错误 (SYNTAX ERROR)”消息。

解答

使用 HEW 编辑程序编码日文时，指定日文字体如下：

<HEW 2.0 或以上版本>

使用工具 (Tools)-> 格式视图 (Format Views) 中的字体 (Font) 标签上的字体 (Font)：

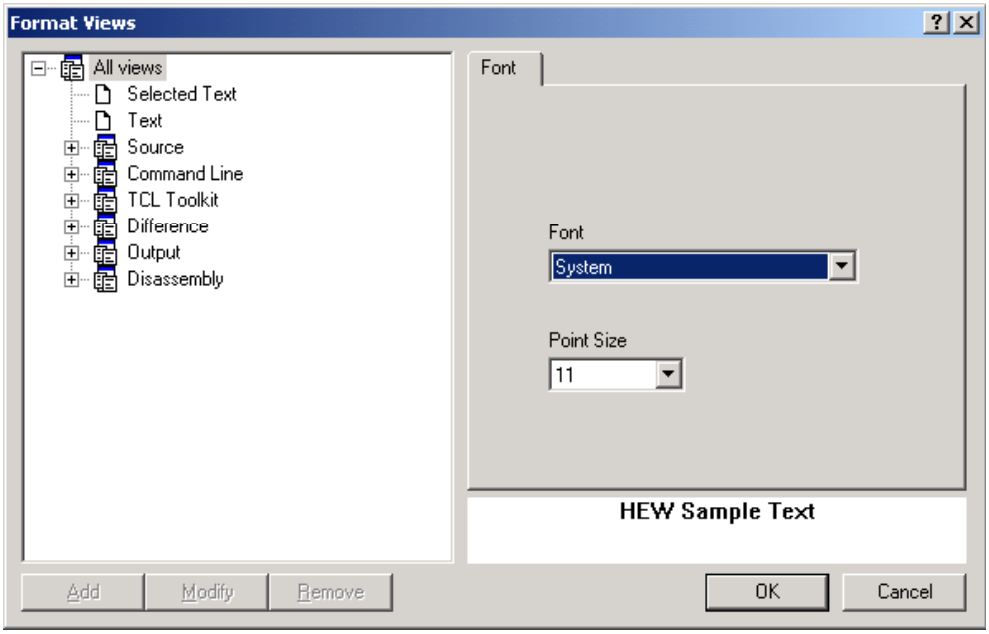


图 11.22 “字体”(Font) 对话框

若日文字体不在 HDI 中正确显示，请如下修改：

[设定 (Setup)->定制 (Customize)->字体 (Font)…]

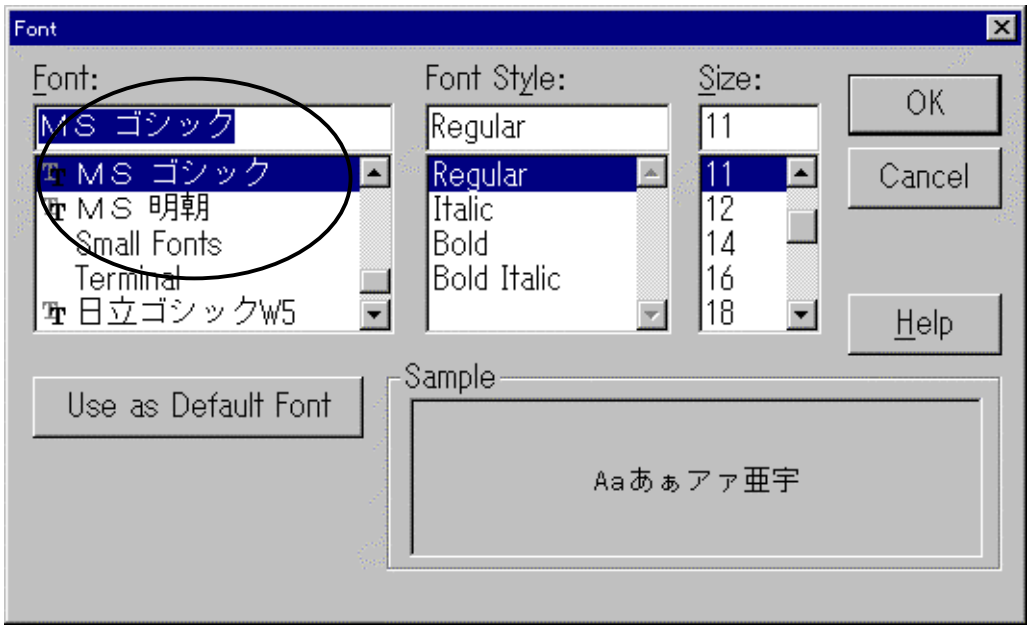


图 11.23 “字体”(Font) 对话框

11.4.10 如何将程序从 HIM 转换到 HEW

问题

我该如何在 HEW 上使用 HIM（Hitachi 集成管理器）所创建的工程？

解答

使用 HEW 系统所提供，称为“HIM 到 HEW 的工程转换器 (HIM To HEW Project Converter)”的工具，可将工程从 HIM 转换到 HEW。

要获取此工具的详细资料，请参考“瑞萨高性能嵌入式工作区发行说明”(Renesas High-performance Embedded Workshop Release Notes) 第 3 节“将工程从 HIM 转换到 HEW” (Converting a Project from HIM to HEW)。

11.4.11 设置 HEW 工程时相应的设备不可用

问题

尝试使用 HEW 设置工程时，我要选择的设备没有供选择。 我该怎么做？

解答

从瑞萨网站下载“设备更新程序”(Device Updater)。

“设备更新程序”(Device Updater) 是由 HEW 生成的工具，用于更新工程文件。 新 CPU 的工程支持按顺序执行。

如果在使用“设备更新程序”(Device Updater) 更新工程文件后相应的设备仍然无法选取，HEW 生成的文件将需要人工重写。 例如，以设定 SH-2 核心的 SH7018 工程为例。 首先，在 HEW 中创建新工程，在 CPU 系列中选择 **SH-2**，以及在 CPU 类型中选择“**其他 (Other)**”。 从瑞萨网站下载 I/O 寄存器定义文件，然后将 SH7018.H 添加到工程。 如果在设置工程时没有选择“**其他 (Other)**”请将 SH7018.H 重新命名为 iodefine.h ，在 HEW 生成文件时，覆盖具备相同名称的文件。

另外，使用函数中断时，必须更改 HEW 生成的文件。 有关详情，请参考第 *11.1.41 节“我要编写中断处理”*。

11.4.12 我要在最新的 HEW 中使用旧编译程序（工具链）。

问题

我拥有旧编译程序封装。 购买仿真程序时，新的 HEW 随产品配套提供。

为了使用新的 HEW 创建和调试，我要在新的 HEW 中使用旧工具链。

我可以这样做吗？

解答

这将取决于您正在使用的编译程序封装版本。 请参考下文。

[SHC 4 或更早版本]

< 创建 >

工具链不能在最新的 HEW 中注册。 因此，不可使用新的 HEW 创建。

< 调试 >

不能使用绝对文件 (*.abs)。 您只可以使用 S 类型格式文件。

此外，也无法进行 C 源代码级调试。 只可在汇编程序级进行。

[SHC V.5.0]

< 创建 >

工具链不能在最新的 HEW 中注册。 因此，不可使用新的 HEW 创建。

（注意）

“HIM 到 HEW 的工程转换器 (HIM To HEW Project Converter)”只有在您拥有 SHC 5.1 版本的编译程序封装时可使用。

通过使用此工具，您可以将 HIM 工程转换为 HEW 工程。您可以在转换后将 SHC 5.1 版本与新的 HEW 配合使用。

< 调试 >

不能使用绝对文件 (*.abs)。 您只可以使用 S 类型格式文件。

此外，也无法进行 C 源代码级调试程序。 只可在汇编程序级进行。

[SHC V.5.1]

< 创建 >

工具链可以在最新的 HEW 中注册。 因此，可以使用新的 HEW 创建。

但是您不能使用最新的 HEW 创建新工程。

在创建新工程的情形下，您必须使用与其他较旧版本的编译程序封装配套提供的 HEW 1 版本。

使用 HEW 1 版本创建工程后，您可以在新的 HEW 中将它打开。

< 调试 >

不能使用绝对文件 (*.abs)。 您只可以使用 S 类型格式文件。

此外，也无法进行 C 源代码级调试程序。 只可在汇编程序级进行。

[SHC V.6]

< 创建 >

工具链可以在最新的 HEW 中注册。 因此，可以使用新的 HEW 创建。

但是您不能使用最新的 HEW 创建新工程。

在创建新工程的情形下，您必须使用与其他较旧版本的编译程序封装配套提供的 HEW 1 版本。

使用 HEW 1 版本创建工程后，您可以在新的 HEW 中将它打开。

< 调试 >

可以使用绝对文件 (*.abs)。

通过注册绝对文件，可以在 C 源代码级执行调试。

[SHC 7 或以上版本]

< 创建与调试 >

没有限制。 您可以使用新的 HEW 的所有功能。

SuperH RISC Engine C/C++编译程序应用笔记

运行时例程的命名规则

附录 A　　运行时例程的命名规则

为运行时例程的函数名称命名的规则如下：

(1) 为整数运算、浮点运算、符号转换和位字段函数命名的规则

```
[operation name]  [size]  [sign]  [r]  [p]  [nm]
[size]      : b   …1 字节
            : w   …2 字节
            : l   …4 字节
            : s   …4 字节（单精度浮点）
            : d   …8 字节（双精度浮点）
[sign]      : s   …带符号
            : u   …无符号
[r]         : 仅限 _subdr、_divdr；仅在参数的进栈顺序分别不同于 _subd、_divd 时
[p]         : 仅在外围处理中添加
[nm]        : 无屏蔽；仅在无中断屏蔽时在外围处理中添加
例外： _muli
```

注意： [sign] 标识符仅为整数运算添加。

(2) 转换函数的命名规则

```
 _[size]to[size]

[size]      : I …带符号，四字节
            : u …无符号，四字节
            : s …单精度浮点
            : d …双精度浮点
```

(3) 移位函数的命名规则

```
 _[sta_]  sft  [direction]  [sign]  [number of bits]

[sta_]      : 仅在添加了位数时添加
[direction] : l …左移
            : r …右移
[sign]*1    : l …逻辑移位
            : a   …算术移位
[number of bits]*2:  0 至 31
```

注意： 1. [sign] 仅在 [direction] 为 r 时添加。
2. [number of bits] 仅在添加了 [sta_] 时添加。

(4) 其他函数的命名规则

存储器区域移动、字符串比较及字符串复制函数为特例。

附录 B 附加功能

B.1 1.0 版本与 2.0 版本的附加功能

表 B.1 概述 SHC 编译程序 2.0 版本的附加功能。

表 B.1 SHC 编译程序 2.0 版本的附加功能概述

编号	功能	描述
1	支持 SH7600 系列	除了 SH7000 系列，也可以创建使用 SH7600 系列指令的目标。
2	位置无关代码	可通过将程序段分配到任意地址来创建 SH7600 系列目标。
3	指定字符串的输出区域	可使用选项来选择要将字符串数据放置在常数段 (ROM) 或数据段 (RAM) 中。
4	注解嵌套	支持指定是否嵌套注解的选项。
5	速度或大小的优化	提供指定将在目标创建时进行速度或大小优化的选项。
6	支持段名称切换	通过在程序中途使用 #pragma 指令，可切换目标的输出段名称。
7	mac 嵌入式函数	支持使用 MAC 指令在两个数组上执行乘法累加运算的嵌入式函数。
8	系统调用的嵌入式函数	支持对 ITRON 规格操作系统 HI-SH7 进行直接系统调用的嵌入式函数。
9	单精度初等函数程序库	支持单精度初等函数程序库。
10	char 类型位字段	支持 char 类型位字段。

B.2 2.0 版本与 3.0 版本的附加功能

表 B.2 概述 SHC 编译程序 3.0 版本的附加功能。

表 B.2 SHC 编译程序 3.0 版本的附加功能概述

编号	功能	描述
1	加强的优化	优化性能获得大大增强。 同时，还制定有选择性使用速度或大小优化选项的规定。
2	支持 SH-3	提供生成 SH-3 目标的选项，并支持 SH-3 的 little-endian 格式特性。另外，SH-3 数据的预取指令作为嵌入式函数被支持。
3	编译程序限制的扩展	可一次编译的文件数量、包含文件的最大嵌套级和其他编译程序限制都获得扩展。
4	支持字符串中的日文字码	添加了包含移位 JIS 和 EUC 日文字码的字符串数据的相关规定。
5	使用文件的选项指定	文件可用来指定命令行选项。
6	SH-2 除法器的使用	通过使用 SH-2 除法器生成除法运算代码。
7	内联扩展	可为以 C 及汇编语言编写的用户例程内联扩展添加指定。
8	短地址指定的使用	可为短寻址指定变量，包括二字节地址及 GBR 相对数据。
9	寄存器保存/恢复操作的控制	可添加语句来制止寄存器保存/恢复操作、增进函数速度和大小。

(1) 加强的优化

3.0 版本中的优化提供强调速度（-SPEED 选项）和大小（-SIZE 选项）的选项，而这两种优化都获得了加强。

若要增强速度，获得增进的循环优化与内联扩展的使用使执行速度提高大约 10%，达到 1 MIPs/MHz 的执行速度。

为了缩减程序大小，削减代码大小的指令被生成，同时重叠的处理也被结合，以获取明显的增进，使目标大小减小了大约 20%。同时，通过使用在 3.0 版本中引进的扩展功能（8. 短地址指定的使用，及 9. 寄存器保存/恢复操作的控制），可使目标大小进一步缩减。

(2) 支持 SH-3

除了 SH-1 和 SH-2，如今也可以创建 SH-3 的目标（使用 -CPU=SH3 选项）。另外，也支持 SH-3 的下列功能。

(a) 支持与设定存储器中位顺序的功能对应的 -ENDIAN 选项 (-ENDIAN=BIG, -ENDIAN=LITTLE)。
(b) 支持生成高速缓存预取指令 (PREF) 的预取扩展嵌入式函数。

(3) 编译程序限制的扩展

编译程序的限制获得如下表所述的扩展。

表 B.3 扩展的编译程序限制

编号	描述	2.0 版本	3.0 版本
1	可一次编译的源程序数量	16 个文件	无限*
2	每个文件的源代码行数	32,767 行	65535 行
3	完全编译单位内的源代码行数	32,767 行	无限
4	#include 嵌套级的最大数量	8 级	30 级

(4) 支持字符串中的日文字码

移位 JIS 和 EUC 日文字码也可作为字符串数据被包含在程序中。

当输入代码是移位 JIS（-SJIS 选项）时，输出代码也是移位 JIS，当输入代码是 EUC（-EUC 选项）时，输出代码也是 EUC。

然而，图形用户界面目前不支持日文字码数据的显示。

(5) 使用文件的选项指定

通过使用 -SUBCOMMAND 选项来指定文件名，与其包含在命令行上，选项可被包含到特定的文件中。因此，不需要每次在命令行输入多个复杂的选项。

(6) SH-2 除法器的使用

支持下列选项以使 SH-2 除法器可被使用。

(a) 不使用除法器的目标可通过 -DIVISION=CPU 选项来生成。
(b) 使用除法器的目标可通过使用 -DIVISION=PERIPHERAL 选项来生成。在除法器使用期间，中断将被禁用。
(c) 使用除法器的目标可通过 -DIVISION=NOMASK 选项生成。此选项假设除法器不会在中断处理期间被使用。

(7) 内联扩展

(a) C 函数的内联扩展
当使用了 -SPEED 选项时，编译程序将自动对小型函数进行内联扩展。同时，通过使用 -INLINE 选项，可修改用于内联扩展的函数的最大大小。内联扩展也可通过使用 #pragma 语句来明确指定。“#pragma inline”语句指定以 C 编写的用户函数的内联扩展。

实例 （C 函数的内联扩展）：

Example (inline expansion of C function):

```
#pragma inline(func)
int func(int a,int b)
{
      return(a+b)/2;
}

main()
{
```

```
i=func(10,20); /* expanded to i=(10+20)/2 */
}
```

(b) 汇编程序函数的内联扩展

“#pragma inline_asm”选项可被用来指定以汇编语言编写的用户函数的内联扩展。然而，当使用“#pragma inline_asm”进行内联扩展时，编译程序的输出是汇编语言源代码。在这种情况下将无法进行 C 语言级的调试。

实例（汇编程序函数的内联扩展）：

```
#pragma inline_asm(rotl)
int rotl(int a)
{
      ROTL  R4
      MOV   R4,R0
}

main()
{
      i=rotl(i); /* 将变量 i 设定在寄存器 R4 中，然后扩展函数 rotl 的代码 */
}
```

(8) 短地址指定的使用

(a) 指定二字节地址变量

通过使用“#pragma abs16(<变量名称>)”语句，可对可使用二字节（-32768 至 32767）来定址的地址范围分配指定变量。通过这种方法，将可缩减对有关变量进行参考的目标大小。

(b) GBR 基址变量的指定

通过使用“#pragma gbr(<变量名称>)”语句，可对 GBR 相对寻址模式的参考指定变量。通过这种方法，参考此变量的目标将可缩减大小，并可使用 GBR 相对寻址模式基于存储器的特定位操作指令。

(9) 寄存器保存/恢复操作的控制

“#pragma noregsave(<函数名称>)”语句可被用来制止函数入口及出口点的寄存器保存/恢复操作。这将可产生不具有寄存器保存/恢复内务操作的快速、简练的函数。指定了“#pragma noregsave”的函数将无法被普通函数调用，但可被为调用指定了“#pragma noregsave”的函数而明确（使用“#pragma regsave”）指定的 C 语言函数所调用。

通过将“#pragma noregsave”和被频繁执行的函数一起使用，程序大小将可被缩减，而执行速度获得提高。

B.3 3.0 版本与 4.1 版本的附加功能

SuperH RISC engine C/C++ 编译程序 4.1 版本的附加功能概述如下。

(1) 外部变量的寄存器分配

“#pragma global_register(<变量名称>=<寄存器数量>)”语句可被用来将外部变量分配到寄存器。

(2) 善用高速缓存的优化

支持可分配具有 16 字节对齐的标签的“-align16”选项，以有效使用高速缓存存储器和取指令。

(3) 加强的内联扩展功能

添加了一项功能，使因为内联扩展的结果而不再被使用的函数被删除。内联扩展后变得不必要的函数应使用“static”来声明。同样的，未被地址调用或参考的 static 函数也将被删除。

实例：

```
#pragma inline(func)          #pragma inline(func)
int a;                           int a;
static int func(){               /* func() 函数本身被删除 */
   a++;
}
main(){                       main(){
   func();                          a++; /* 内联扩展
}                                }
```

(4) 递归内联扩展

函数的递归内联扩展添加了一项功能。可使用“-nestinline”选项来指定递归的深度。

(5) 循环扩展优化的选项

“-loop”和“-noloop”选项可独立于“-speed”和“-size”选项（这些选项将在指定了省略优化的选项时无效），用来指定是否在优化中扩展循环处理。

(6) 用于二字节地址变量的选项

之前，具有二字节地址的变量必须使用“#pragma abs16”语句来个别指定，但现在“-abs16”选项允许所有变量的一次指定。选项“-abs16=run”仅为运行时例程指定二字节地址，“-abs16=all”为所有变量和函数指定二字节地址，包括运行时例程。

(7) 函数返回值的高位字节被保证

之前，由 (unsigned) char 和 short 类型的函数所返回的值不保证高位字节。通过指定“-rtnext”选项，返回值的高位字节现在获得保证（R0 的高位字节被符号扩展或零扩展）。

(8) 更完整的列表文件

与之前的版本相比，包含在目标列表和汇编列表中的信息如今更为完整及易于读取。

列表文件中 C 源代码与汇编语言源代码语句单位的同时输出，使它们之间的对应更容易掌握（使用“-show=source,object”选项）。

（此外，“-show”选项的默认值从 source 更改为 nosource。）

添加了一个在函数中使用的运行时例程名称列表，作为计算函数所使用的堆栈空间数量的信息。

对于从常数库加载的数据，由指令加载的数据被添加为注解。

实例：

```
1:  float x;
2:  func(){
3:      x/=1000;
4:  }

Listing file

func.c   1      float x;
func.c   2      func(){*(a) C 源代码和汇编语言代码的同时输出
                       code
000000          _func:                    ; 函数：func
                                        ; 帧大小=4

            ; 所使用的运行时程序库名称：

            ; divs *(b) 运行时例程名称
000000     4F22           STS.L   PR,@-R15
func.c     3 x/=1000;
000002     D404           MOV.L   L216+2,R4 ; x
000004     D004           MOV.L   L216+6,R0 ; H'447A0000 *(c) 加载数据
000006     D305           MOV.L   L216+10,R3 ; __divs
000008     430B           JSR     @R3
00000A     6142           MOV.L   @R4,R1
func.c     4        }
00000C     4F26          LDS.L    @R15+,PR
00000E     000B          RTS
000010     2402          MOV.L    R0,@R4
000012       L216:
000012     00000002       .RES.W     1
000014     <00000000> .DATA.L  _x
000018     447A0000       .DATA.L  H'447A0000
00001C     <00000000> .DATA.L  __divs
000000       _ x:                        ; static:      x
000000 00000004       .RES.L     1
```

(9) 更完整的错误消息

通过指定“-message”选项来输出信息消息，将使对编程错误的检查变得更轻松。

实例：

```
1:  void func(){
2:        nt a;
3:        a++;
4: sub(a);
5:    }
```

信息消息

```
line 3: 0011 (I) Used before set symbol: "a"    （未定义自动变量的参考）
line 4: 0200 (I) No prototype function            （没有原型声明）
```

此外，造成错误的标识符、记号或编号被添加到消息，以便容易查找发生错误的位置。

实例：

```
: 2118 (E) Prototype mismatch "identifer"
: 2119 (E) Not a parameter name "identifer"
: 2201 (E) Cannot covert parameter "number"
: 2225 (E) Undeclared name "identifer"
: 2500 (E) Illegal token "token"
```

(10) 日文字码的自动转换

当包含 EUC 或移位 JIS 日文字码的字符串被输出到目标文件时，日文字码将被自动转换到由编码选项所指定的编码。

(a) “-outcode=euc”选项促使自动转换为 EUC 代码。
(b) “-outcode=sjis”选项促使自动转换为移位 JIS 代码。

(11) 通过环境变量对 CPU 类型进行指定

现在可以使用环境变量来代替命令行选项指定 CPU 类型。

环境变量指定

```
SHCPU=SH1      （等同于“-cpu=sh1”选项）
SHCPU=SH2      （等同于“-cpu=sh2”选项）
SHCPU=SH3      （等同于“-cpu=sh3”选项）
```

(12) 将 double 数据类型处理为 float 类型的选项

通过使用“-double=float”选项，被声明为 double 类型的数据可被读取为 float 类型。在不要求 double 类型的精度的程序中，执行速度可在不需修改源代码的情况下获得增进。

B.4 4.1 版本与 5.0 版本的附加功能

SuperH RISC engine C/C++ 编译程序 5.0 版本的附加功能概述如下。

(1) 一行中所包含字符数量的扩展

单个逻辑行中的字符数量限制从 4,096 个字符扩展到 32,768 个字符。

(2) 移除编译程序的源行限制

单一文件中 65,535 行的编译限制被移除。然而，超过 65,535 行的文件部分将无法被调试。

(3) 与 SH-4 的指令兼容

此编译程序版本也与 SH-4 兼容，以保持对 SH 系列 CPU 的符合。通过使用“-cpu=sh4”选项，将可生成 SH-4 目标。

(4) 正常化模式的添加

通过使用“-denormalize=on|off”选项，将可选择是否处理非正常化数字或将它们设定为零。这仅在使用了“-cpu=sh4”时有效。

然而，当“-denormalize=on”时，若非正常化数字被输入到 FPU，将会发生异常。因此必须编写一项异常处理程序来处理非正常化数字的处理。

(5) 舍入模式的添加

通过使用“-round=nearest|zero”选项，将可选择要舍入到零或最近的数字。这仅在使用了“-cpu=sh4”时有效。

(6) 编译程序选项环境变量与 SH-4 兼容

与其使用命令行选项来指定 CPU，通过设定“SHCPU=SH4”，环境变量“SHCPU”可被用来指定 SH-4。

(7) 与 SH-2E 兼容

通过使用“-cpu-sh2e”选项，将可生成 SH-2E 的目标。

(8) 编译程序选项环境变量与 SH-2E 兼容

与其使用命令行选项来指定 CPU，通过设定“SHCPU=SH2E”，环境变量“SHCPU”可被用来指定 SH-2E。

(9) 使用扩展名来区分 C 和 C++ 文件

通过选择性的使用 shc 及 shcpp 命令，编译程序使所使用的语法可被决定。如今，即使在使用 shc 命令时，C++ 文件也可以根据文件扩展名或选项来进行编译。有关详情，请参考下表。

表 B.4 决定编译语法的条件

命令	选项	所编译文件的扩展名	用于编译的语法
shcpp	任意	任意	编译为 C++
shc	-lang=c	任意	编译为 C
	-lang=cpp		编译为 C++
	无 -lang 选项	*.c	编译为 C
		.cpp、.cc、*.cp、*.CC	编译为 C++

B.5 5.0 版本与 5.1 版本的附加功能

SuperH RISC engine C/C++ 编译程序 5.1 版本的附加功能概述如下。

(1) SH3-DSP 程序库的支持

除了版本较旧的 SH-DSP，现在也为可被应用到 SH3-DSP 的程序库提供支持。

(2) 嵌入式 C++ 语言的支持

现在可获得为嵌入式 C++ 语言规格提供的支持，即与嵌入式系统兼容的 C++ 规格。

- bool 类型的支持
- 多个继承的警告
- 嵌入式 C++ 语言类程序库的支持

(3) 模块间优化函数的支持

施行下列优化，然后生成具有最佳大小/速度的目标。

通过这项优化，大小可获得大约 10% 的缩减，而执行速度则可增进 7 到 8%。

- 减少冗余寄存器的保存/恢复代码
- 删除未被参考的变量/函数
- 公用代码的例程化
- 函数调用代码的优化

(4) 被增进的编译速度

通过增进的优化处理加快了编译速度。

最大可达到倍速，而平均速度增加达到 130%。

(5) 限制的扩展

- 命令行长度的限制从 256 扩展至 4,096。
- 文件名长度的限制从 128 扩展至 251。
- 字符串字面长度的限制从 512 扩展至 32,767。

(6) 加强的优化

可增进目标性能的各种优化均获得加强。

(7) C++ 注解的支持

在 C 语言中，可使用“//”的注解。

(8) 集成环境的更改（PC 版本）

较旧的 PC 集成环境 HIM (Hitachi Integration Manager) 已被新的集成环境 HEW (High-performance Embedded Workshop) 所取代。

与 HIM 相比，下列为新添加的功能。

- 工程生成程序
  自动生成为每个 CPU 定义外围 I/O 的标头文件。

- 具有版本管理工具的组合界面
  支持具有由第三方提供的版本管理工具的界面。
- 分层工程支持
  可以定义一个工程内的多个子工程并进行分层管理。
- 网络支持
  提供 WindowsNT CSS 下的开发环境。

B.6 5.1 版本与 6.0 版本的附加功能

SuperH RISC engine C/C++ 编译程序 6.0 版本的附加功能概述如下。

(1) 限制的放宽

源程序和命令行的限制被大大的放宽。

- 文件名称长度： 251 字节 → 无限
- 符号长度： 251 字节 → 无限
- 符号的数量： 32,767 个符号 → 无限
- 源程序行数： C/C++： 32,767 行， ASM： 65,535 行 → 无限
- C 程序字符串长度： 512 个字符 → 32,766 个字符
- 汇编程序的行长度： 255 个字符 → 8,192 个字符
- 子命令文件的行长度： ASM： 300 字节，optlnk： 512 字节 → 无限
- 优化连接编辑程序 rom 选项的参数数量： 64 个参数 → 无限

(2) 目录名称及文件名称中的破折号 (-)

现在目录名称及文件名称中将可以指定破折号 (-)。

(3) 版权通知的删除

现在通过指定 logo/nologo 选项，将可以指定是否显示版权通知。

(4) 错误消息的前缀

除了支持集成开发环境中的错误帮助功能，编译程序及优化连接编辑程序中的错误消息起始部分也有了前缀。

(5) fpscr 选项的添加

现在当指定了 cpu=sh4 选项，而未指定 fpu 选项时，可以指定是否在调用函数前后保证 FPSCR 的寄存器精度模式。

(6) #pragma 扩展

#pragma 扩展现在可以不需使用 ( ) 来编写。

(7) 嵌入式函数的添加

添加了 trace 函数。

(8) 隐含声明的添加

_ _HITACHI_ _ 和 _ _HITACHI_VERSION_ _ 以 #define 进行隐含声明。

(9) 静态标签名称

为使文件中的静态标签可被 #pragma inline_asm 参考，标签名称已被更改为 _ _$ (名称)。然而，它在连接列表中显示为 _(名称)。

(10) 语言规格的扩展及更改

- union 初始化时的错误已被消除。

实例：

```
union{
```

```
char c[4];
} uu={{'a','b','c'}};
```

- 现在已能在替换 structure 的同时做出声明。

实例：

```
struct{
int a, int b;
} s1
void test()
{
struct S s2=s1;
}
```

- bool 类型数据的边界对齐现在是 4 字节。
- 异常处理和模板函数现在可被支持为 C++ 语言规格。
- C 预处理程序现在符合 ANSI/ISO。

B.7 6.0 版本与 7.0 版本的附加功能

- SuperH RISC engine C/C++ 编译程序 7.0 版本的附加功能概述如下。

(1) 外部存取优化功能（支持映像选项）

这项功能根据变量及函数在连接时分配的地址来执行外部变量存取和函数位移指令的优化。优化是通过重新编译优化连接编辑程序在第一次连接时输出（被指定到映像选项）的外部符号分配信息文件来施行。

(2) GBR 相对存取代码的自动生成（支持 gbr 选项）

若指定了 gbr=auto，编译程序将自动生成 GBR 设定及 GBR 相对存取代码。在函数调用前后，GBR 值可被保证。然而，GBR 的相对嵌入式函数将无法使用。

(3) 加强的 speed/size 选择选项

添加了 speed/size 选择选项（shift、blockcopy、division、approxdiv 选项），现在可以进行更细致的大小/速度调整。

(4) 嵌入式系统的加强函数

- 嵌入式函数的添加
  双精度乘法、SWAP 指令、LDTLB 指令、NOP 指令
- #pragma 扩展的添加及更改
  支持 #pragma entry 入口函数指定及 SP 设定
  支持 #pragma stacksize 堆栈大小指定
  支持 #pragma interrupt sp=<变量>+<常数> 及 sp=&<变量>+<常数>
- 支持段运算符
  支持以 C 语言为大小参考进行编码的函数。
- 地址转型错误的放宽
  在初始化外部变量时有关地址初始化的转型表达式错误被放宽。

(5) 增进的程序库

- 支持可重入的程序库
  若使用程序库创建工具指定了 reent 选项，将生成可重入的程序库。
- malloc 预留大小的单位及输入和输出文件的数量被制定为变量。
  现在可以在 C/C++ 程序库函数的初始设定中使用 _sbrk_size 来指定 malloc 预留大小及使用 _nfiles 来指定输入和输出文件的数量。这会大大缩减 RAM 的容量。
  若省略这项指定，malloc 预留大小是 520，输入和输出文件的数量是 20。
- 支持简易 I/O
  若使用程序库创建工具指定了 nofloat 选项，浮点转换将不被支持，并会生成小型的 I/O 例程。

(6) 优化选项的添加（7.0.06 版本）

- 添加的选项

下面显示被添加到 7.0.06 版本的选项。大写字母表示缩写，具有下划线的字符表示默认值。

表 B.5 添加的选项

	项目	命令行格式	指定
1	全局变量的处理	GLOBAL_Volatile = { 0 |	将全局变量视为符合非易失性标准来处理，除了符合易失性标准的变量
		1 }	将全局变量视为符合易失性标准来处理
2	优化全局变量的范围	OPT_Range = {All |	优化整个函数中的所有全局变量
		NOLoop |	制止全局变量在循环外的动作或循环控制变量的优化
		NOBlock }	制止转移或循环内的全局变量优化
3	空循环的删除	DEL_vacant_loop ={ 0 |	制止空循环的删除
		1}	删除空循环
4	指定最大的解开因数	MAX_unroll = <数值> <数值>:1-32	指定最大的循环解开因数 默认值： 1 （当指定了速度或循环选项时，默认值是 2）
5	在无穷循环之前删除赋值	INFinite_loop = {0 |	制止在无穷循环之前删除全局变量的赋值
		1 }	在无穷循环之前删除全局变量的赋值
6	全局变量的分配	GLOBAL_Alloc = {0 |	制止全局变量的寄存器分配
		1 }	分配全局变量的寄存器
7	struct/union 成员的分配	STRUCT_Alloc = {0 |	制止 struct 或 union 成员的寄存器分配
		1 }	分配 struct 或 union 成员的寄存器
8	符合 const 标准的变量的传播	CONST_Var_propagate = {0 |	制止符合 const 标准的变量的传播
		1 }	传播符合 const 标准的变量
9	常数加载的内联扩展	CONST_Load = {Inline |	执行常数加载的内联扩展
		Literal }	从文字库加载常数数据
10	预设指令	SChedule = {0 |	制止预设指令
		1 }	预设指令

GLOBAL_Volatile

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)][将全局变量视为符合易失性标准来处理 (Treat global variables as volatile qualified)]

命令行格式

```
GLOBAL_Volatile = { 0 | 1 }
```

描述

当指定了 **global_volatile=0** 时，编译程序将优化符合非易失性标准的全局变量的存取，因此对全局变量的存取计数或顺序可能与 C/C++ 程序不同。

当指定了 **global_volatile=1** 时，所有全局变量会被视为符合易失性标准来处理。因此对全局变量的存取计数或顺序可能与 C/C++ 程序相同。

此选项的默认值是 **global_volatile=**0。

说明

当指定了 **global_volatile=1** 时，**schedule=0** 将变成默认值。

OPT_Range

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)][指定优化范围 (Specify optimizing range)：]

命令行格式

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

描述

当指定了 **opt_range=all** 时，编译程序会在一个函数中优化对所有全局变量的存取。

当指定了 **opt_range=noloop** 时，编译程序将不会优化对循环或循环条件表达式中所使用的全局变量的存取。

当指定了 **opt_range=noblock** 时，编译程序将不会优化转移或循环内对全局变量的存取。

此选项的默认值为 **opt_range=all**。

实例

(1) 转移内的优化实例（指定了 opt_range=all 或 noloop）

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

〈优化后的源图像〉

```
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1; /* 删除变量 A 的参考及传播 A=1 */
}
```

(2) 对循环的优化实例（指定了 opt_range=all）

```
int A,B,C[100];
void f() {
    int i;
    for (i=0;i<A;i++) {
```

```
            C[i] = B;
        }
    }
```

〈优化后的源图像〉

```
void f() {
        int i;
        int temp_A, temp_B;
        temp_A = A; /* 移除在循环条件表达式中对变量 A 的参考 */
        temp_B = B; /* 移除在循环中对变量 B 的参考 */
        for (i=0;i<temp_A;i++) { /* 删除对变量 A 的参考 */
            C[i] = temp_B; /* 删除对变量 B 的参考 */
        }
}
```

说明

当指定了 **opt_range=noloop** 时，默认值将变为 **max_unroll=1**。
当指定了 **opt_range=noloblock** 时，**max_unroll**=1、**const_var_propagate**=0 和 **global_alloc=0** 将变成默认值。

DEL_vacant_loop

优化 (Optimize) [详细资料 (Details)][杂项 (Miscellaneous)][删除空循环 (Delete vacant loop)]

命令行格式

```
DEL_vacant_loop = { 0 | 1 }
```

描述

当指定了 **del_vacant_loop=0** 时，编译程序将不会删除空循环。

当指定了 **del_vacant_loop=1** 时，编译程序将会删除空循环。

此选项的默认值是 **del_vacant_loop**=0。

说明

注意　7.0.04 版本与 7.0.06 版本之间的默认值不同。
到 7.0.04 版本为止： 删除空循环
7.0.06 版本或以上： 不删除空循环

MAX_unroll

优化 (Optimize) [详细资料 (Details)][杂项 (Miscellaneous)][指定最大的解开因数 (Specify maximum unroll factor)：]

命令行格式

```
MAX_unroll = <数值>
```

描述

指定当循环被扩展时的最大解开因数。

<数值>接受从 1 到 32 的十进制数字。 若<数值>的指定超出范围，将会发生错误。

当指定了 **speed** 或 **loop** 选项时，此选项的默认值是 **max_unroll**=2。

否则，此选项的默认值将是 **max_unroll**=1。

说明

当指定了 **opt_range=noloop** 或 **opt_range=noblock** 时，此选项的默认值是 **max_unroll**=1。

INFinite_loop

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)]

[在无穷循环之前删除全局变量的赋值 (Delete assignment to global variables before an infinite loop)]

命令行格式

```
INFinite_loop = { 0 | 1 }
```

描述

当指定了 **infinite_loop=0** 时，编译程序不会在无穷循环前删除全局变量的赋值。

当指定了 **infinite_loop=1** 时，编译程序会在无穷循环前删除在无穷循环中不被参考的全局变量的赋值。

此选项的默认值为 **infinite_loop** =0。

实例

```
int A;

void f()
{
    A = 1; /* 对变量 A 的赋值 */
    while(1) {}  /* 变量 A 在循环中不被参考 */
}
```

〈优化后的源图像〉

```
void f()
{
  /* 删除对变量 A 的赋值 */
    while(1) {}
}
```

说明

注意 7.0.04 版本与 7.0.06 版本之间的默认值不同。

到 7.0.04 版本为止： 在无穷循环前删除在无穷循环中不被参考的全局变量的赋值

7.0.06 版本或以上： 不会在无穷循环前删除全局变量的赋值

GLOBAL_Alloc

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)][将寄存器分配给全局变量 (Allocate registers to global variables)：]

命令行格式

```
GLOBAL_Alloc = { 0 | 1 }
```

描述

当指定了 **global_alloc=0** 时，编译程序不会将寄存器分配给全局变量。

当指定了 **global_alloc=1** 时，编译程序将寄存器分配给全局变量。

此选项的默认值是 **global_alloc=**1。

说明

当指定了 **opt_range=noblock** 时，默认值将变为 **global_alloc=0**。

当指定了 **optimize=0** 时，注意 7.0.04 版本与 7.0.06 版本之间的默认值不同。

到 7.0.04 版本为止： 将寄存器分配给全局变量
7.0.06 版本或以上： 不会将寄存器分配给全局变量

STRUCT_Alloc

优化 (Optimize) [详细资料 (Details)][杂项 (Miscellaneous)][将寄存器分配给 struct/union 成员 (Allocate registers to struct/union members)]

命令行格式

```
STRUCT_Alloc = { 0 | 1 }
```

描述

当指定了 **struct_alloc=0** 时，编译程序不会将寄存器分配给 struct 或 union 成员。

当指定了 **struct_alloc=1** 时，编译程序将寄存器分配给 struct 或 union 成员。

此选项的默认值为 **struct_alloc=**1。

说明

当指定了 **opt_range=noblock** 或 **global_alloc=**0，及 **struct_alloc=1** 时，编译程序仅将寄存器分配给局部的 struct 或 union 成员。

当指定了 **optimize=0** 时，注意 7.0.04 版本与 7.0.06 版本之间的默认值不同。

到 7.0.04 版本为止： 将寄存器分配给 struct 或 union 成员
7.0.06 版本或以上： 不会将寄存器分配给 struct 或 union 成员

CONST_Var_propagate

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)][传播符合 const 标准的变量 (Propagate variables which are const qualified)：]

命令行格式

```
CONST_Var_propagate = { 0 | 1 }
```

描述

当指定了 **const_var_propagate=0** 时，编译程序不会传播符合 const 标准的全局变量。

当指定了 **const_var_propagate=1** 时，编译程序会传播符合 const 标准的全局变量。

此选项的默认值是 **const_var_propagate=**1。

实例

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

〈优化后的源图像〉

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

说明

当指定了 **opt_range=noblock** 时，此选项的默认值是 **const_var_propagate=**0。

C++ 程序中符合 const 标准的变量，即使在指定了 **const_var_propagate=0** 时也始终会被传播。

CONST_Load

优化 (Optimize) [详细资料 (Details)][杂项 (Miscellaneous)][常数值加载为 (Load constant value as)：]

命令行格式

```
CONST_Load = { Inline | Literal }
```

描述

当指定了 **const_load=inline** 时，所有 2 字节常数数据或一些 4 字节常数数据的加载将被扩展。

当指定了 **const_load=literal** 时，所有 2 字节或 4 字节常数数据将从文字库加载。

此选项的默认值如下。

当指定了 **speed** 选项时：

默认值是 **const_load=inline**。

当指定了 **size** 或 **nospeed** 选项时：

若 2 字节或 4 字节的常数数据可被分别扩展入 2 或 3 个指令，

const_load=inline 将被应用。

否则，默认值为 **const_load=literal**。

实例

```
int f() {
    return (257);
}
```

(1) 当指定了 const_load=inline 或 speed 选项时：

```
MOV #1,R0 ; R0 <- 1
SHLL8  R0 ; R0 <- 256 (1<<8)
RTS
ADD #1,R0 ; R0 <- 257 (256+1)
```

(2) 当指定了 const_load=literal、size 或 nospeed 时：

```
MOV.W  L11,R0
RTS
NOP
L11:
.DATA.W   H'0101
```

SChedule

优化 (Optimize) [详细资料 (Details)][全局变量 (Global variables)][预设指令 (Schedule instructions)：]

命令行格式

```
SChedule = { 0 | 1 }
```

描述

当指定了 **schedule=0** 时，编译程序不会预设指令，它们会按以 C/C++ 程序编写的顺序来执行。

当指定了 **schedule=1** 时，编译程序将参照流水线或超标量（仅限于 SH-4）机制来预设指令。

此选项的默认值是 **schedule=1**。

说明

当指定了 **opt_range=noblock** 时，**schedule=0** 将变成默认值。

- 默认值是 optimize=0

当指定了 **optimize=0** 时，所添加选项的默认值如下。

```
global_volatile=0
opt_range=noblock
del_vacant_loop=0
max_unroll=1
infinite_loop=0
global_alloc=0
struct_alloc=0
const_var_propagate=0
const_load=literal
schedule=0
```

下列选项的默认值不同于 **optimize**=1。

	optimize=0	**optimize=1**
opt_range	noblock	all
global_alloc	0	1
struct_alloc	0	1
const_var_propagate	0	1
const_load	literal	视 speed/size/nospeed 而定
schedule	0	1

-

- **7 版本的兼容（到 7.0.04 版本为止）**

下列选项的默认值在 7.0.04 及 7.0.06 版本之间不同。

(i) 空循环的删除 (del_vacant_loop)

到 7.0.04 版本为止 : 删除空循环

7.0.06 版本或以上 : 不删除空循环

(ii) 在无穷循环前删除赋值 (infinite_loop)

到 7.0.04 版本为止 : 在无穷循环前删除无穷循环中未被参考的全局变量的赋值

7.0.06 版本或以上 : 不会在无穷循环前删除全局变量的赋值

以下对 **optimize=0** 的指定在 7.0.04 及 7.0.06 版本之间不同。

(i) 全局变量的分配 (global_alloc)

到 7.0.04 版本为止 : 将全局变量分配给寄存器

7.0.06 版本或以上 : 不会将全局变量分配给寄存器

(ii) struct 或 union 成员的分配 (struct_alloc)

到 7.0.04 版本为止 : 将 struct 或 union 成员分配给寄存器

7.0.06 版本或以上 : 不会将 struct 或 union 成员分配给寄存器

- **优化系统**

全局变量的优化等级显示如下。当在 Hew 中选择了其中一个等级时，将可联合控制与全局变量的优化相关的选项。

等级的设定在优化 (Optimize) [详细资料 (Details)][等级 (Level)：] 进行。

(i) 等级 1

全局变量的所有优化会被制止。

```
global_volatile=1
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=0
```

(ii) 等级 2

非符合易失性标准的全局变量的优化在基本块中进行

（指令的顺序没有标签或转移，除了在起始或结束部分）。

```
global_volatile=0
opt_range=noblock
infinite_loop=0
global_alloc=0
```

```
const_var_propagate=0
schedule=1
```

(iii) 等级 3

会进行所有符合非易失性标准的全局变量的优化。

```
global_volatile=0
opt_range=all
infinite_loop=0
global_alloc=1
const_var_propagate=1
schedule=1
```

(iv) 自定义

用户根据程序指定这些选项。

当指定了等级 1、等级 2 或等级 3 时，上述选项将无法个别更改。

- SuperH RISC engine C/C++ 优化连接编辑程序 7.0 版本的附加功能概述如下。

(7) 支持通配符

可为输入文件及启动选项段名称指定通配符。

(8) 搜索路径

可使用环境变量 (HLNK_DIR) 为多个输入文件及程序库文件指定搜索路径。

(9) 加载模块的个别输出

可执行绝对加载模块文件的个别输出。

(10) 更改的错误等级

可分别更改信息、警告及错误等级消息的错误等级，及指定是否将它们输出。

(11) 支持二进制与 HEX

现在可输入和输出二进制文件。

此外，现在也可以选择以 Intel HEX 格式输出。

(12) 堆栈容量使用信息的输出

使用 stack 选项，将可以输出堆栈分析工具的数据文件。

(13) 调试信息删除工具

使用 strip 选项，将可以仅删除加载模块文件及程序库文件内的调试信息。

SuperH RISC engine C/C++ 优化连接编辑程序 7.1 版本的附加功能概述如下。

(14) 输出外部符号分配信息文件（支持映像选项）

若指定了映像选项，编译程序将生成用于外部变量存取优化的外部符号分配信息文件。

B.8 7.0 版本与 7.1 版本的附加功能

- SuperH RISC engine C/C++ 编译程序 7.1 版本的附加功能概述如下。

(1) 加强的优化

(a) 在 MOVT 后立即删除 EXTU

在 MOVT 之后立即删除不必要的 EXTU。
（由于不可设定 1 或 0 以外的其他值，EXTU 变得不必要）

优化之前		优化之后	
f:		_f:	
MOV.L	L12+2,R6 ; _a1	MOV.L	L12+2,R6 ; _a1
MOV.B	@R6,R0	MOV.B	@R6,R0
TST	#128,R0	TST	#128,R0
MOVT	R0	MOVT	R0
EXTU.B	R0,R0		

由于不可为 R0 设定 1 或 0 以外的其他值，EXTU 变得不必要。

(b) 在零扩展寄存器的右移后删除 EXTU

即使零扩展的寄存器在右移后被零扩展，但值保持不变，因此它将被删除。

优化之前			优化之后		
_f:			_f:		
	MOV.L	L13+2,R2; _a2		MOV.L	L13+2,R2; _a2
	MOV	#1,R5		MOV	#1,R5
	MOV.W	@R2,R6		MOV.W	@R2,R6
	EXTU.W	R6,R6		EXTU.W	R6,R6
	MOV	R6,R2		MOV	R6,R2
	SHLR2	R2		SHLR2	R2
	SHLR	R2		SHLR	R2
	EXTU.W	R2,R2		CMP/GE	R5,R2
	CMP/GE	R5,R2		:	
	:				

由于高位 2 字节使用 EXTU 进行零清除，即使再次执行 EXTU 也不会更改值。

(c) 统一连续的 AND

若对相同变量连续进行 AND，它们可被组合成 1 个 AND。

优化之前	优化之后
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 AND #3,R0 RTS AND #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS AND #1,R0
组合成 1 个 AND。	

(d) 位字段的比较与结合

统一对多个位字段的判断 (TST#n, R0)。

优化之前	优化之后

_f:
:
MOV R4,R0
TST #64,R0
BF L12
TST #32,R0
BF L12
MOV R6,R0
:
:
	_f:
	:
	MOV R4,R0
	TST #96,R0
	BF L12
	MOV R6,R0
	:
	:

统一位字段的条件，并以 1 个判断来替换它们。	

(e) 删除连续 EXTS+EXTU 的 EXTS

在 EXTS 后，若相同大小的 EXTU 被执行，EXTS 将变得不必要，因此会被删除。

优化之前	优化之后
_f: : EXTS.B R6,R2 EXTU.B R2,R0 : :	_f: : EXTU.B R6,R0 : :
EXTU 在取自 EXTS 的值上执行，因此 EXTS 是不必要的。	

(f) MOVT(+XOR)+EXTU+CMP/EQ 的删除

在 TST 后删除不必要的 MOVT(+XOR)+EXTU+CMP/EQ，并进行转换以使用直接的转移指令参考 T 位。

优化之前	优化之后
_f: : TST #4,R0 MOVT R0 MOV.L L23+6,R6 ; _st2 XOR #1,R0 EXTU.B R0,R0 CMP/EQ #1,R0 MOV.B @R6,R0 BF L16 :	_f: : TST #4,R0 MOV.L L23+6,R6; _st2 MOV.B @R6,R0 BT L16 :
直接参考 T 位。	

(g) AND #imm, R0+CMP/EQ #imm, R0 → TST #imm, R0

以 TST #imm, R0 来替换 AND #imm, R0+CMP/EQ #imm, R0。

优化之前	优化之后
L17: MOV.B @R6,R0 AND #1,R0 CMP/EQ #1,R0 BF L19 MOV.B @R5,R0 AND #1,R0	L17: MOV.B @R6,R0 TST #1,R0 BT L19 MOV.B @R5,R0 AND #1,R0

(h) 当比较 (==) unsigned char 及 constant 时删除 EXTU

在紧贴着加载比较 unsigned char 及 constant 时删除不必要的 EXTU。

优化之前	优化之后
_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 EXTU.B R6,R6 EXTU.B R2,R2 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6	_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6
删除不必要的扩展。	

(i) 在位字段加载后/存储前删除扩展

在加载后及存储前删除不必要的位字段扩展。

优化之前	优化之后
_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 EXTU.B R2,R0 OR #128,R0 :	_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 OR #128,R0 :
删除不必要的扩展。	

(j) 当判断 switch-case 时删除复制

在执行各个 switch 语句的 case 判断时删除值的复制。

优化之前

```
_f:
                :
        MOV     R0,R2
        MOV     R2,R0
        CMP/EQ  #1,R0
        BT      L24
        CMP/EQ  #2,R0
        BT      L26
        MOV     R2,R0
        CMP/EQ  #3,R0
        BT      L28
        MOV     R2,R0
        CMP/EQ  #4,R0
        BT      L30
        MOV     R2,R0
                :
```

优化之后

```
_f:
                :
        MOV     R0,R2
        MOV     R2,R0
        CMP/EQ  #1,R0
        BT      L24
        CMP/EQ  #2,R0
        BT      L26
        CMP/EQ  #3,R0
        BT      L28
        CMP/EQ  #4,R0
        BT      L30
                :
```

删除不必要的复制。

(k) 统一连续的 OR

若对相同变量连续进行 OR，它们可被组合成 1 个 OR。

优化之前

```
_f:
    MOV.L   L11+2,R6    ; _a5
    MOV.B   @R6,R0
    OR      #3,R0
    RTS
    OR      #1,R0
```

优化之后

```
_f:
    MOV.L   L11+2,R6    ; _a5
    MOV.B   @R6,R0
    RTS
    OR      #3,,R0
```

组合成 1 个 OR。

(l) 在紧贴着 AND #imm,R0 或 TST #imm,R0 之前删除 EXTS

在紧贴着下列项目前删除不必要的扩展：

(i) AND #imm,R0
(ii) TST #imm,R0

优化之前	优化之后
_f: : EXTS.B R6,R0 AND #32,R0 :	_f: : AND #32,R0 :
_f: : EXTS.B R6,R0 TST #32,R0 :	_f: : TST #32,R0 :
删除不必要的扩展。	

(m) 删除连续 EXTU+EXTS 的 EXTU

在 EXTU 后，若相同大小的 EXTS 被执行，EXTU 将变得不必要，因此被删除。

优化之前	优化之后
_f: : EXTU.B R6,R2 EXTS.B R2,R0 :	_f: : EXTS.B R6,R0 :
EXTS 在取自 EXTU 的值上执行，因此 EXTU 是不必要的。	

(n) 在 MOVT 之后删除紧贴着 XOR #imm,R0(OR,AND) 的 EXTU

紧贴着下列项目删除不必要的 EXTU；

(i) XOR #imm,R0
(ii) OR #imm,R0
(iii) AND #imm,R0

在 MOVT 之后

优化之前	优化之后
: MOVT R0 XOR #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS XOR #1,R0
MOVT R0 OR #2,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS OR #2,R0
: MOVT R0 AND #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS AND #1,R0
删除不必要的扩展。	

(o) 在进行比较时删除不必要的 EXTS

在符号扩展后比较寄存器时删除冗余的 EXTS 再输出。

优化之前	优化之后
_f: : EXTS.B R6,R6 CMP/GT R6,R2 BF L13 :	_f: : CMP/GT R6,R2 BF L13 :
若 R6 之前已被扩展，EXTS 将是不必要的。	

(p) 禁用（立即）对寄存器分配常数值

禁用对寄存器分配功能性参数常数（-128 到 127）。

优化之前		优化之后	
_f:		_f:	
PUSH	R14		
:		:	
MOV.B	#127,R14		
:		:	
MOV.B	R14,R4	MOV.B	#127,R4
BSR	sub	BSR	sub
:		:	
POP	R14		

在未分配到寄存器的情况下，直接将常数值加载到参数寄存器。

(q) 加强的 DT 指令

为分配给寄存器的变量执行 DT 指令。

优化之前		优化之后	
_f:		_f:	
MOV.L	L16+2,R6; _x	MOV.L	L16+2,R6; _x
MOV.L	@R6,R2	MOV.L	@R6,R2
ADD	#-1,R2	DT	R2
TST	R2,R2	BT/S	L12
BT/S	L12	:	
:			

执行 DT 指令。

(r) 增进的文字输出位置

决定文字数据输出位置时的指令大小计算精确度获得提高，并可在稍后输出文字数据输出位置。

(s) 1byte&=1byte 冗余 EXTU 的删除

当 1byte&=1byte 时删除不必要的 EXTU。

优化之前		优化之后	
_f:		_f:	
:		:	
MOV.B	@(R0,R7),R6	MOV.B	@(R0,R7),R6
MOV.B	@R5,R2	MOV.B	@R5,R2
EXTU.B	R6,R6	AND	R6,R2
AND	R6,R2	MOV.B	R2,@R5
MOV.B	R2,@R5	MOV.B	@R14,R2
MOV.B	@R14,R2	:	
:			

删除不必要的扩展。

(t) 2 字节文字扩展

防止相同的代码被扩展两次。

优化之前		优化之后	
_f:		_f:	
MOV.L	L13+4,R4 ; _b	MOV.L	L13+4,R4 ; _b
SHLL8	R0	SHLL8	R0
ADD	#-48,R0	ADD	#-48,R0
MOV.W	@(R0,R4),R2	MOV.W	@(R0,R4),R2
MOV	#8,R0	MOV	#8,R0
SHLL8	R0	SHLL8	R0
ADD	#-46,R0	ADD	#-46,R0
EXTU.W	R2,R6	EXTU.W	R2,R6
MOV.W	@(R0,R4),R2	MOV.W	@(R0,R4),R2
MOV	#8,R0	ADD	#2,R0
SHLL8	R0	EXTU.W	R2,R5
ADD	#-44,R0	MOV.W	@(R0,R4),R2
EXTU.W	R2,R5		
MOV.W	@(R0,R4),R2		

防止相同的代码被扩展两次。

(u) 增进循环条件决定的扩展

若大小具有优先级，在执行循环条件决定时不会执行循环决定的复制。

优化之前	优化之后（7 版本）	优化之后（7.1 版本）

while (cond) {
  :
}
	if (cond) {
	  do {
	  :
	  } while (cond);
	}
		goto L1;
		do {
		  :
		L1:;
		} while (cond);

cond 出现在一个地点，而非两个地点。		

(v) 冗余 if 语句条件决定的删除

当首个 if 语句的结果使到之后的 if 语句变得不必要时，之后的 if 语句将被删除。

优化之前	优化之后

if (cond)
  t=65;
else
  t=67;
if (t == 65)
  fx();
else
  fy();
	if (cond) {
	  t=65;
	  fx();
	} else {
	  t=67;
	  fy();
	}

当首个 if 语句的结果使到之后的 if 语句变得不必要时，之后的 if 语句将被删除。	

(w) 临时变量的直接运算

禁用冗余 temp 变量的替换，并更改方程式的运算顺序。

优化之前	优化之后
k = i + prime;	p = i + prime + flags;
p = flags + k;	
k 没有在之后使用，因此对 temp 的冗余替换未被执行。	

(x) 增量后寻址

为加载 4 字节变量使用 MOV.L @Rm+,Rn。

优化之前	优化之后

:
L11:
MOV.L @R5,R2
ADD #4,R5
DT R6
ADD R2,R4
BF L11
:
	:
	L11:
	MOV.L @R5+,R2
	DT R6
	ADD R2,R4
	BF L11
	:

使用一个指令执行 MOV.L @Rm+,Rn。	

(y) 增进循环终止条件

放宽循环终止的优化执行条件，并使优化易于应用。

优化之前	优化之后

int a, b;
func() {
unsigned short sx;
 	
for (sx=0; sx<1; sx++) {
a++;
b++;
f();
}
}
	int a, b;
	func() {
	
	a++;
	b++;
	f();
	
	}

执行循环终止。	

(z) 1 位判断的优化

将参考多个 1 位宽度位字段的条件表达式组合为 1，并生成使用位 AND 同时执行取数据及值的比较的代码。

优化之前	优化之后

struct S {
  char bit0:1;
  char bit1:1;
  char bit2:1;
  char bit3:1;
}ss1;
if((ss1.bit0|ss1.bit1|ss1.bit2)!= 0){
  :
  :
}
	struct S {
	  char bit0:1;
	  char bit1:1;
	  char bit2:1;
	  char bit3:1;
	}ss1;
	if ((*(char *)&ss1 & 0xe0)!= 0){
	  :
	  :
	}

使用 AND 同时执行取数据及比较。	

B.9 7.1 版本与 8.0 版本的附加功能

SuperH RISC engine C/C++ 编译程序 8.0 版本的附加功能概述如下。

(1) 支持新的 CPU

现在支持 SH-4A 及 SH4AL-DSP。

(2) 扩展及更改语言规格

- 现在支持 SP-C。
- 现在支持 long long 和 unsigned long long 类型。

(3) 增进内建函数

- 添加 DSP 的内建函数
  绝对值、MSB 侦测、算术移位、舍入运算、位样式复制、取模寻址设置、取模寻址取消及 CS 位设定
- 添加 SH-4A 及 SH4AL-DSP 的内建函数
  正弦及余弦计算、平方根的倒数、指令高速缓存块失效、指令高速缓存块预取以及数据运算的同步化
- 添加及更改 #pragma 扩展
  #pragma ifunc 制止浮点寄存器的保存或恢复
  #pragma bit_order 指定位字段的顺序
  #pragma pack 指定 structure、union 或 class 的对齐数量

(4) 自动选择枚举类型的大小（支持 auto_enum 选项）

枚举类型被处理为可包含枚举类型的最小类型。

(5) 指定 structure、union 或 class 成员的对齐数量（支持 pack 选项）

可指定 structure、union 或 class 成员的对齐数量。

(6) 指定位字段的顺序（支持 bit_order 选项）

可指定位字段成员的顺序。

(7) 更改错误等级（支持 change_message 选项）

可更改每则信息及警告消息的错误等级。

(8) 限制的撤消

可允许的最大 switch 语句数量现在增加至 2048。

(9) 支持 DSP 程序库的定点

现在支持 DSP 程序库的定点。

B.10 8.0 版本与 9.0 版本的附加功能

- SuperH RISC engine C/C++ 编译程序 9.0 版本的附加功能概述如下。

(1) 支持新的 CPU

支持 SH-2A 及 SH2A-FPU。

添加了一个选项及 #pragma 扩展，以在 SH-2A 及 SH2A-FPU 中使用 TBR。

(2) 语言规格的扩展与更改

- 下列项目符合 ANSI 标准。
  - 数组索引

```
int iarray[10], i=3;
i[iarray] = 0; /* 与 iarray[i] = 0 相同;*/
```

  - 启用了 union 位字段指定

```
union u {
  int a:3;
};
```

  - 常数运算

```
static int i=1||2/0; /*被零除由错误消息更改为警告消息*/
```

  - 程序库和宏的添加

```
strtoul, FOPEN_MAX
```

- 下列项目在指定了 strict_ansi 选项时符合 ANSI 标准，可能在 9 版本与旧版本之间产生不同的结果。
  - unsigned int 和 long 运算
  - 浮点运算的联合
- 有指定寄存器存储类的变量将在指定了 enable_register 选项时被优先分配给寄存器。

(3) 固有函数的增强

- 添加了 SH-2A 及 SH2A-FPU 的固有函数。

饱和运算及 TBR 设定与参考

- 为不可以 C 编写的指令添加了固有函数。

T 位的参考及设定、相联的存储器的中部抽出、具有进位的加法、具有借位的减法、正负转换、1 位除法、旋转和移位。

(4) 放宽值的限制

放宽了下列限制。

- 重复语句（while、do 和 for）及选择语句（**if** 和 **switch**）的组合中的嵌套级： 32 级 -> 4096 级
- 允许在一个函数中使用的转至 (goto) 标签数： 511 -> 2,147,483,646
- switch 语句的嵌套级： 16 级 -> 2048 级
- 允许在一个 switch 语句中使用的 case 标签数： 511 -> 2,147,483,646
- 允许在一个函数定义或函数调用中使用的参数数量： 63 ->2,147,483,646
- 段名称的长度： 31 字节 -> 8192 字节
- 允许在一个文件的 #pragma 段中使用的段数： 64 -> 2045

(5) 存储器空间分配的扩展

可为存储器空间分配进行更详细的设定。

- abs16/abs20/abs28/abs32 选项
- #pragma abs16/abs20/abs28/abs32

(6) 变量的绝对地址的指定（支持 #pragma 地址）

可为外部变量指定绝对地址。

(7) 外部变量存取的优化扩展（支持 smap 选项）

优化被应用到对在所要编译的文件中定义的外部变量的存取。不需进行映像选项所需的重新编译。

(8) 算术程序库的精确度获得增进

使用算术程序库的运算精确度获得增进，可能造成 9 版本与旧版本之间的结果不同。

附录 C 版本升级的注意事项

本节描述当为旧版本（SuperH RISC engine C/C++ 编译程序封装 6.x 或以下版本）进行升级时的注意事项。

C.1 受保证的程序操作

在版本升级后开发程序时，程序的运作可能更改。在创建程序时，请留意下列事项并充分测试您的程序。

(1) 依赖执行时间或时序的程序

C/C++ 语言规格不指定程序执行时间。因此，编译程序的版本差异，可能造成由程序执行时间和外围，如 I/O 的时序落后，或异步处理中，如在中断中的处理时间差异，所引起的操作更改。

(2) 包含具有两项或以上副作用的表达式的程序

当两项或以上的副作用包含在一项表达式内时，操作可能根据版本更改。

实例

```
a[i++]=b[i++];          /* i 增量顺序未定义。                    */
f(i++,i++) ;            /*参数值根据增量顺序而更改。          */
                        /*这在 i 的值为 3 时产生 f(3, 4)或 f(4, 3)。*/
```

(3) 具溢出结果或非法操作的程序

当发生溢出或执行了非法操作时，结果的值将不被保证。操作可能根据版本更改。

实例

```
int a, b;
x=(a*b)/10;             /* 根据 a 和 b 的值范围，这可能造成溢出。*/
```

(4) 无变量初始化或类型不均等

当变量未被初始化，或调用与被调用的函数间的参数或返回值类型不匹配时，不正确的值被存取。操作可能根据版本更改。

文件 1：

```
int f(double d)
{
      :
}
```

文件 2：

```
int g(void)
{
      f(1);
}
```

由于调用函数的参数为 int 类型，而被调用函数的参数为 double 类型，因此，值不能被正确参考。

此处所提供信息不包含所有可能发生的情形。请谨慎使用此编译程序，并牢记版本间的差异来充分测试您的程序。

C.2 与旧版本的兼容性

下列备注包含有关编译程序（5.x 或以下版本）所生成的文件，要和旧版本所生成的文件，或由汇编程序（4.x 或以下版本）或连接编辑程序（6.x 或以下版本）输出的目标文件或程序库文件进行连接的情形。备注也涵盖了有关使用由旧版本编译程序所提供的现有调试程序的说明。

(1) 目标格式

标准的目标文件格式从 SYSROF 更改为 ELF。调试信息的标准格式也被更改为 DWARF2。

当由旧版本的编译程序（5.x 或以下版本）或汇编程序（4.x 或以下版本）所输出的目标文件 (SYSROF) 要被输入到优化连接编辑程序时，先使用文件转换程序来将它转换成 ELF 格式。然而，由连接编辑程序所输出的可再定位文件（扩展名：rel），及包含一个或以上可再定位文件的程序库文件不能被转换。

(2) 包含文件的原点

在旧版本中，当以相对目录格式指定的包含文件被搜索时，搜索将从编译程序的目录开始。在新版本中，搜索将从包含源文件的目录开始。

(3) C++ 程序

因为编码规则和执行方法被更改了，由旧版本编译程序所建立的 C++ 目标文件不能被连接。确保重新编译这些文件。

用以设定执行环境的全局类目标初始/后处理程序库函数，其名称也被更改。请参考 9.2.2 节，执行环境设定，并修改名称。

(4) 公用段的废除（汇编程序）

由于目标格式的更改，对公用段的支持因此废除。

(5) 通过 .END 的项目指定（汇编程序）

只有被外部定义的符号可被 .END 指定。

(6) 模块间优化

由旧版本编译程序（5.x 或以下版本）或汇编程序（4.x 或以下版本）输出的目标文件不是模块间优化的目标。请确保重新编译及重新汇编这类文件，以使它们成为模块间优化的目标。

附录 D ASCII 码表

表 D.1 ASCII 码表

低四位 \ 高四位	0	1	2	3	4	5	6	7
0	NULL	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