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.

前言

本应用笔记说明如何使用 C/C++ 编译程序包有效的建立应用程序，以在下列微型计算机家族上运行：H8SX、H8S/2600、H8S/2000、H8/300H、H8/300，以及 H8/300L。

本应用笔记所涵盖主题的进一步详细资料，可在下列相关手册中找到：

高性能嵌入式工作区 3.0 用户手册 (High-performance Embedded Workshop 3.0 User's Manual)

高性能嵌入式工作区 3.0 HEW 创建程序用户手册 (High-performance Embedded Workshop 3.0 HEW Builder User's Manual)

高性能嵌入式工作区 3.0 HEW 调试程序用户手册 (High-performance Embedded Workshop 3.0 HEW Debugger User's Manual)

H8S 及 H8/300 系列高性能嵌入式工作区教程 (H8S and H8/300 Series High-performance Embedded Workshop Tutorial)

H8S 及 H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S and H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)

H8S 及 H8/300 系列模拟程序调试程序用户手册 (H8S and H8/300 Series Simulator Debugger User's Manual)

每一种产品的硬件和编程手册

本应用笔记编排如下：

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。

第 3 节 说明使用于用户程序开发的扩展函数。

第 4 节 说明 HEW 选项。

第 5 节 说明如何使用模块间优化器的优化功能及优化函数。

第 6 节 阐释有效的编程技术。

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 提供用户常见问题的解答。

附录所涵盖的主题如下：

A：浮点操作功能列表

B：限制列表

C：ASCII 代码表

D：索引

本应用笔记主要涵盖 HEW3.0 以及 H8 编译程序版本 6.0。若 HEW1.2 的操作不同于 H8 编译程序版本 3.0，其不同点将分开论述。

本应用笔记所使用的符号以及惯例如下。

[] ：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。

… ：表示紧随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。

[**菜单－>菜单选项**]：加粗的字母和字符 －> 表示一个菜单选项。

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。

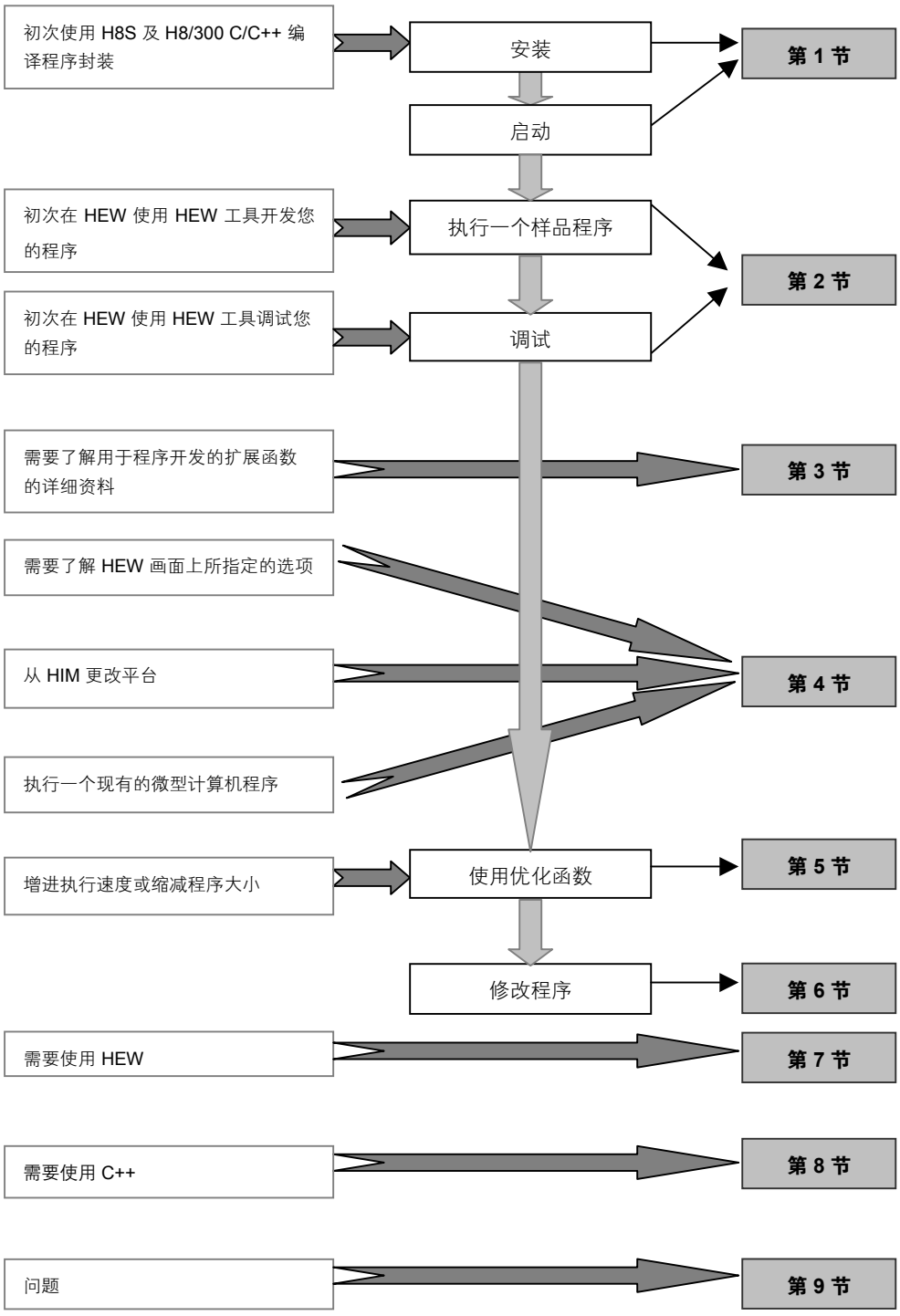
MS-DOS® 是 Microsoft Corporation 在美国及其他国家（地区）的注册商标。

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。

使用应用笔记

Renesas 建议依照以下顺序阅读应用笔记：



目录

H8S, H8/300 系列 C/C++编译程序应用笔记

概述

第 1 节　概述

1.1　摘要

H8S 及 H8/300 C/C++ 编译程序汲取 Renesas Technology H8S 及 H8/300 系列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 或 C++ 程序语言的创建更具效率。

本编译程序支持以下的 CPU：

- H8SX 系列 (H8SX)
- H8S/2600 系列 (H8S/2600)
- H8S/2000 系列 (H8S/2000)
- H8/300H 系列 (H8/300H)
- H8/300 系列 (H8/300)
- H8/300L 系列 (H8/300L)
- AE5 系列 (AE5)

本文档将说明使用此 C/C++ 编译程序创建应用程序的步骤。

本文档主要说明编译程序版本 6.0（HEW2.0 或以上的版本），同时也在需要时解说旧版本 3.0（HEW1.2）。

1.2 功能

H8S 及 H8/300 C/C++ 编译程序提供下列重要的功能。

Windows® 版本

H8S 及 H8/300 C/C++ 编译程序的 Windows® 版本支持集成环境。

允许用户在 Windows® 显示屏上彻底开发程序的 HEW（高性能嵌入式工作区，High-performance Embedded Workshop）。

HEW 提供以下的功能：

- 工程生成程序
  自动生成每个 CPU 的模板软件工程。
- 具有版本管理工具的组合界面。
  支持由第三方所提供，具有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。

UNIX 版本

H8S 及 H8/300 C/C++ 编译程序的 UNIX 版本支持集成开发管理员 (integrated development manager, IDM)，以允许用户执行从编辑到调试的程序开发。

IDM 提供以下的功能：

- 编辑程序可在编译或汇编出现错误时启动。
  （光标显示在出现错误的源代码行。）
- 从汇编/编译、目标模块连接，到装入到调试程序的程序开发都能自动执行。
- 支持使用图形用户界面在源代码级进行调试。

1.3 安装方法

1.3.1 PC 版本

本节叙述与 Windows® 98、Windows® Me、WindowsNT® 4.0、Windows® 2000 或 Windows® XP 兼容的 H8S 及 H8/300 C/C++ 编译程序封装的操作环境，以及将它安装到 Windows® 98、Windows® Me、WindowsNT® 4.0、Windows® 2000 或 Windows® XP 系统的步骤。

(1) 操作环境

主机计算机：IBM-PC 兼容型机器

（CPU：CPU 可运行 Windows® 98、Windows® Me、WindowsNT® 4.0、Windows® 2000 或 Windows® XP）

OS：Windows® 98、Windows® Me、WindowsNT® 4.0、Windows® 2000 或 Windows® XP

存储器大小：建议 128 MB 或以上

集成开发环境的硬盘容量：需要 100 MB 或以上的磁盘空间（供完全安装）

Acrobat® Reader：需要 10 MB 或以上的磁盘空间

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

执行下列步骤以在您的 PC 上安装编译程序：
在开始安装步骤前，确保已关闭所有应用程序。

(a) 安装 H8S 及 H8/300 C/C++ 编译程序封装：

(i) 将编译程序封装的 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 (Start) 菜单，单击 [运行 (Run) …]。
(iii) 在 [运行…] 对话框内，指定在 CD-ROM 根目录内的 Setup.EXE（例如：D:\Setup.EXE），然后单击 [确定 (OK)]。
(iv) 按照画面上的安装说明执行。

注意在集成开发环境 (Integrated Development Environment) 的安装上：

将集成开发环境安装到仅由半角字母数字字符以及半角下划线所组成的目录路径内。使用不包含全角字符或空格的目录路径。

(i) 注意勿将 HEW （高性能嵌入式工作区，High-Performance Embedded Workshop）和 HIM（Hitachi 集成管理员，Hitachi Integration Manager）安装到相同的目录内。
(ii) 即使在网络上使用时，请将高性能嵌入式工作区安装到每一架 PC 的驱动器上。工具链、库管理程序界面、Hitachi 调试界面，以及在线手册可安装到网络驱动器上。要获取从您的 PC 定义安装在其他 PC 上的工具链或库管理程序界面的详细资料，请参阅“高性能嵌入式工作区 V.4.00 用户手册 (High-performance Embedded Workshop V.4.00 User's Manual)”第 5 节，工具管理 (Tools Administration)。
(iii) 如果 [高性能嵌入式工作区 (High-performance Embedded Workshop)] 在 HEW 被安装后无法在 Windows® 开始 (Start) 菜单的 [程序 (Programs)] 内显示，请重新启动 Windows®。
(iv) 如果安装程序在 Windows® 98 之下进行安装时无故中断，请重新启动计算机然后重新安装。

(b) 安装 Acrobat® Reader:

(i) 将编译程序封装的 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 (Start) 菜单，单击 [运行 (Run) …]。

(iii) 在 [运行…] 对话框内指定 CD-ROM 上 [PDF_READ\Japanese] 目录内的 Ar505jpn.exe（日文）或 [PDF_read\English] 目录内的 Ar505eng.exe（英文）。（例如：D:\PDF_Read\Japanese\Ar505jpn.exe），然后单击 [确定 (OK)]。

(iv) 按照画面上的安装说明执行。

(c) 参考在线手册（Online Manual）以及其他文档

- 如果安装了在线手册：

  单击 Windows® 开始 (Start) 菜单、[程序 (Programs)]、[高性能嵌入式工作区 (High-performance Embedded Workshop)] 菜单内的 Online Manual [H8S,H8/300]－English(xx xx)（英文）PDF 文件或 Online Manual [H8S,H8/300]-Japanese(xx.xx)（日文）PDF 文件，(xx xx) 表示年份和月份。

  （例如：Online Manual [H8S,H8/300]-Japanese(01 10)）

- 如果在线手册未被安装：

(i) 将编译程序封装的 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 (Start) 菜单，单击 [运行 (Run) …]。

(iii) 在 [运行…] 对话框内指定 CD-ROM 上 [手册 (Manuals)] 目录内的 jH8_xxxx.PDF（日文）或 eH8_xxxx.PDF（英文）（其中 xxxx 表示年份和月份）（例如：D:\Manuals\jH8_0110.PDF），然后单击 [确定 (OK)]。

1.3.2 UNIX 版本

在 UNIX 系统上安装 H8S 及 H8/300 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空格。

(1) 记录媒介

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) 安装编译程序封装

安装编译程序封装的步骤如下：

(i) 为编译程序封装创建一个路径

使用任何任意名称，为编译程序文件的存储创建一个路径。

（下文中，安装目录将被假设为 /usr/cross_soft。）

% mkdirΔ/usr/cross_soft (RET)

(ii) 安装 CD-ROM

如下所示安装 CD-ROM。如果安装自动执行，则不需要执行以下命令。

[在 Solaris]

% mountΔ–rΔ–FΔhsfsΔ/dev/dsk/c0t6d0s2/h8s_sparcΔ/cdrom/h8s_sparc (RET)

[在 HP-UX]

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

(iii) 复制编译程序封装

移到新建的路径，然后从 CD-ROM 将 SuperH RISC engine C/C++ 编译程序封装的文件解压缩至前述步骤（i）所创建的路径。

[在 Solaris]

% cdΔ/usr/cross_soft (RET)

% tarΔxvfΔ/cdrom/h8s_sparc/Program.tar (RET)

[在 HP-UX]

% cdΔ/usr/cross_soft (RET)

% tarΔxvfΔ/cdrom/"PROGRAM.TAR;1" (RET)

(iv) 更换环境设定

环境变量与路径名称设定如下：（双星号 ** 表示应该指定一个适当的值。）要获取有关环境变量的详细资料，请参考“H8S 及 H8/300 C/C++ 编译程序用户手册 (H8S and H8/300 C/C++ Compiler User’s Manual)”。

下面示范为 C 外壳设定环境变量以及路径名称的例子。

% setenvΔCH38Δ/usr/cross_soft (RET)

设定系统包含文件的存储区域。

% setenvΔCH38TMPΔ/usr/tmp (RET)

指定存储由编译程序或模块间优化程序所创建的中间文件的目录。（这里我们假设目录为 /usr/tmp。）

如果未指定目录，一个当前目录将作为默认目录使用。

% setenvΔH38CPUΔ**:** (RET)**

从 2000n、2000a、2600n、2600a、300hn、300ha、300，以及 3001 之间选择 CPU 的操作模式。如果将 CPU 选择为 2000a、2600a，或 300ha，地址空间的大小也可被指定。

（例如：% setenv H38CPU 2600a:24(RET)）

% setenvΔHLNK_TMPΔ/usr/tmp (RET)

指定存储由连接编辑程序或模块间优化程序所创建的中间文件的目录。

（这里我们假设目录为 /usr/tmp）

如果未指定目录，一个当前目录将作为默认目录使用。

% setenvΔHLNK_LIBRARY1Δ/usr/cross_soft/****.lib (RET)**

% setenvΔHLNK_LIBRARY2Δ/usr/cross_soft/****.lib (RET)**

在连接状态下，可将一个程序库暗中输入而不使用程序库 (LIBRARY) 选项或子命令选项。

要获取详细资料，请参考 H8S，H8S/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S,H8S/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)。

(v) 卸装 CD-ROM

[在 Solaris]

% umountΔ/cdrom/h8s_sparc (RET)

[在 HP-UX]

% umountΔ/cdrom (RET)

(b) 安装集成开发管理员以及集成开发管理员定义文件

(i) 从 CD-ROM 上的 tarfile 装入安装程序。（这里假设 CD-ROM 驱动程序的设备名称为 /cdrom。）

% tarΔxvfΔ/cdrom/idm.tarΔidm_install (RET) [在 Solaris]

(ii) 启动安装程序。

% idm_install (RET)

按照画面上的安装说明执行。要获取详细资料，请参考 H8S 及 H8/300 定义文件安装程序。

(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、Windows®Me、WindowsNT®4.0、Windows®2000，或 Windows®XP 的计算机上。

Acrobat® Reader 版权所有© 2002 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。

下列步骤被用于执行安装。任何运行中的应用程序必须在继续安装之前被停止。

(i) 将集成开发环境 (Integrated Development Environment) 的 CD-ROM 插入 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 (Start) 菜单，单击 [运行 (Run) …]。

(iii) 在 [运行…] 对话框内指定 CD-ROM 上 [PDF_READ\Japanese] 目录中的 Ar40jpn.exe（日文）或 [PDF_read\English] 中的 Ar40eng.exe（英文）（例如：D:\PDF_Read\Japanese\Ar40jpn.exe），然后单击 [确定 (OK)]。

(iv) 按照画面上的安装说明执行。

(d) 参考在线手册 (Online Manual) 以及其他文档

- 如果安装了在线手册：

  单击 Windows® 开始 (Start) 菜单、[程序 (Programs)]、[高性能嵌入式工作区 (High-performance Embedded Workshop)] 菜单上的 Online Manual [H8S,H8/300]-English(xx xx)（英文）PDF 文件或 Online Manual [H8S,H8/300]-Japanese(xx xx)（日文）PDF 文件，(xx xx) 表示年份和月份。（例如：Online Manual [H8S,H8/300]-Japanese(01 10)）

- 如果在线手册未被安装：

(i) 将集成开发环境 (Integrated Development Environment) 的 CD-ROM 插入 CD-ROM 驱动器。（这里将 CD-ROM 驱动器假设为驱动器 D。）

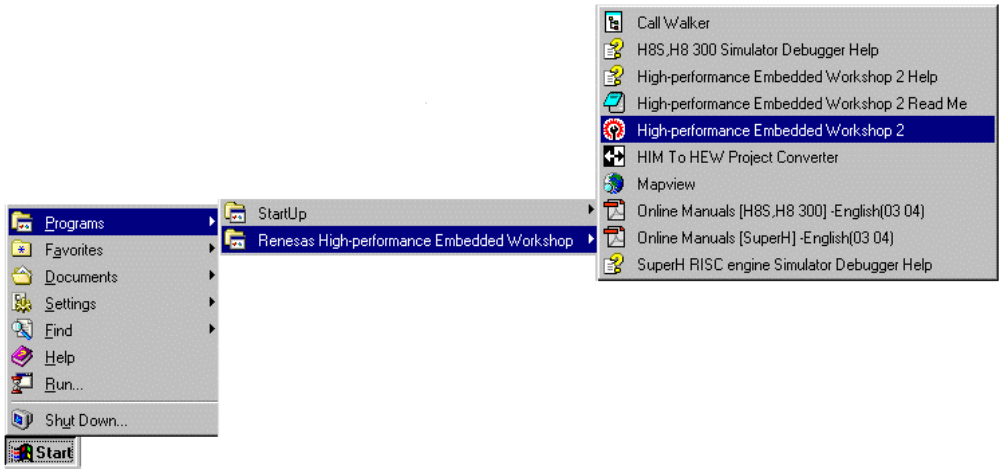
(ii) 从 Windows® 的开始 (Start) 菜单，单击 [运行 (Run) …]。

(iii) 在 [运行…] 对话框内指定 CD-ROM 上 [手册 (Manuals)] 目录内的 jH8_xxxx.PDF（日文）或 eH8_xxxx.PDF（英文）（其中 xxxx 表示年份和月份）（例如：D:\Manuals\jH8_0110.PDF），然后单击 [确定 (OK)]。

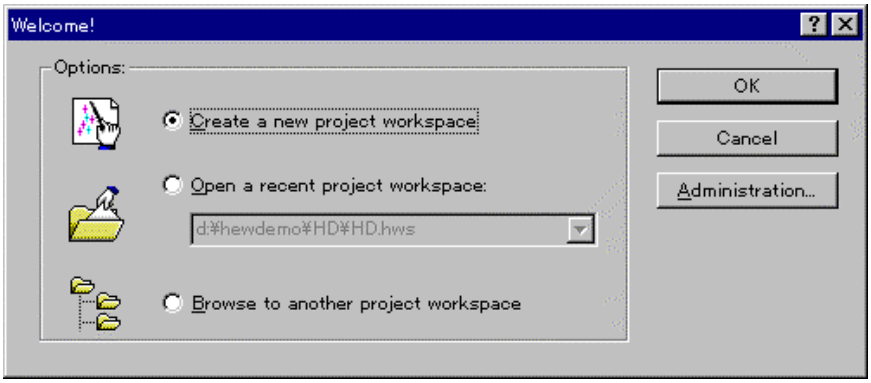
1.4 启动方法

1.4.1 启动 HEW

在 H8S 及 H8/300 C/C++ 编译程序封装安装完成后，高性能嵌入式工作区 (High-performance Embedded Workshop, HEW) 的安装程序将在 Windows® 开始 (Start) 菜单的程序 (Programs) 文件夹中创建一个命名为高性能嵌入式工作区 (High-performance Embedded Workshop) 的文件夹，在这个文件夹内可以启动高性能嵌入式工作区的程序。



将显示下列“欢迎使用 (Welcome)！”对话框：



从以上画面选择所需的工程工作空间：

创建新的工程工作空间（Create a new project workspace）	创建新的工程工作空间。
打开最近的工程工作空间（Open a recent project workspace）	打开一个最近使用的现有工作空间。
浏览至另一个工程工作空间（Browse to another project workspace）	打开另一个工作空间。

通过选择 [管理 (Administration)]，您可以注册或删除所要使用的系统工具。

1.4.2 使用命令启动编译程序

在本子节内，将举例说明执行 H8S 及 H8/300 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“H8S 及 H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S and H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”第 2 节，C/C++ 编译程序操作方法 (C/C++ Compiler Operating Method)。

下面说明使用编译程序的基本步骤。

(1) 启动编译程序

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。

ch38 (RET)

(2) 编译程序

C 源程序测试 1.c 已被编译。

ch38Δtest1.c (RET)

C++ 源程序测试 2.cpp 已被编译。

ch38Δtest2.cpp(RET)

多个 C/C++ 程序可同时被编译。

ch38Δtest1.cΔtest2.cpp(RET)

(3) 指定选项

选项（优化 (goptimize)、调试、显示＝目标、分配，等等）以一个破折号（－）为前缀，多个选项由空格（Δ）区隔。

当指定多个子选项时，必须以逗号（，）区隔。

ch38Δ-goptimizeΔ-debugΔ-show=object,allocationΔtest1.c (RET)

下面的短格式也可用于选项指定。

ch38Δ-gΔ-debΔ-sh=o,aΔtest1.c (RET)

当编译多个程序时，程序上的选项是否有效将因选项被指定的位置而异。

<实例 1：为所有的源程序指定选项>
先于第一个源程序被指定的选项对所有的源程序有效。

ch38Δ-gΔ-debΔ-sh=o,aΔtest1.cΔtest2.cpp (RET)

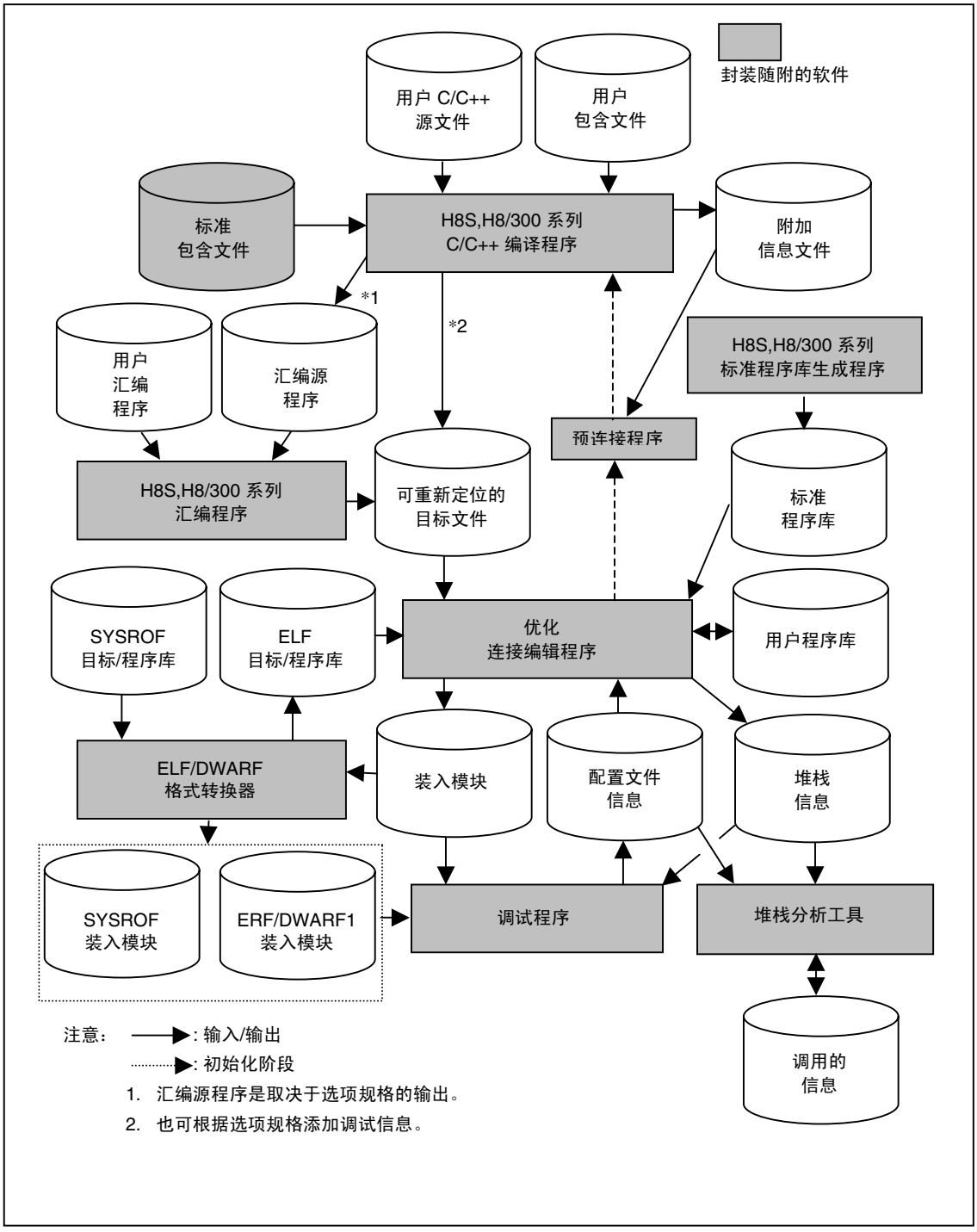
<实例 2：每个程序的选项分开指定>
在源程序测试 2.cpp 之后指定的选项只对测试 2.cpp 有效。

ch38Δtest1.cΔtest2.cppΔ-debΔ-sh=o,a (RET)

注意： (1) 编译程序根据文件扩展名，以及 －lang 和 lang 选项来区别 C 和 C++ 文件。要获取有关文件扩展名的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S,H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”第 8 节，文件规格 (File Specifications)。

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。



H8S, H8/300 系列 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

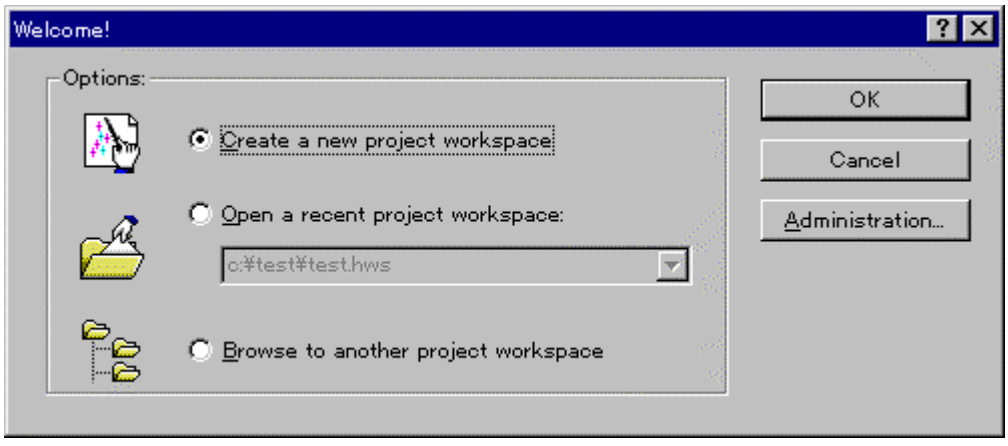
装入模块的创建步骤因其创建时所处的特殊工作环境以及 HEW 的版本而异。从以下列表选取您的环境，以适当的创建一个装入模块。

使用 HEW1 (HEW1.2) 创建新的工程。	→ 2.1.1 节
使用 HEW2 (HEW2.0) 创建新的工程。	→ 2.1.2 节
绕过 HEW 使用命令行。	→ 2.1.3 节
将 HIM 工程转换为 HEW 工程。	→ 7.1.8 节
使用现有文件在 HEW 内创建工程。	→ 4.3 节

在 2.3 节有关使用 HDI 进行调试的叙述中，将假设使用以 HEW 创建的新工程。

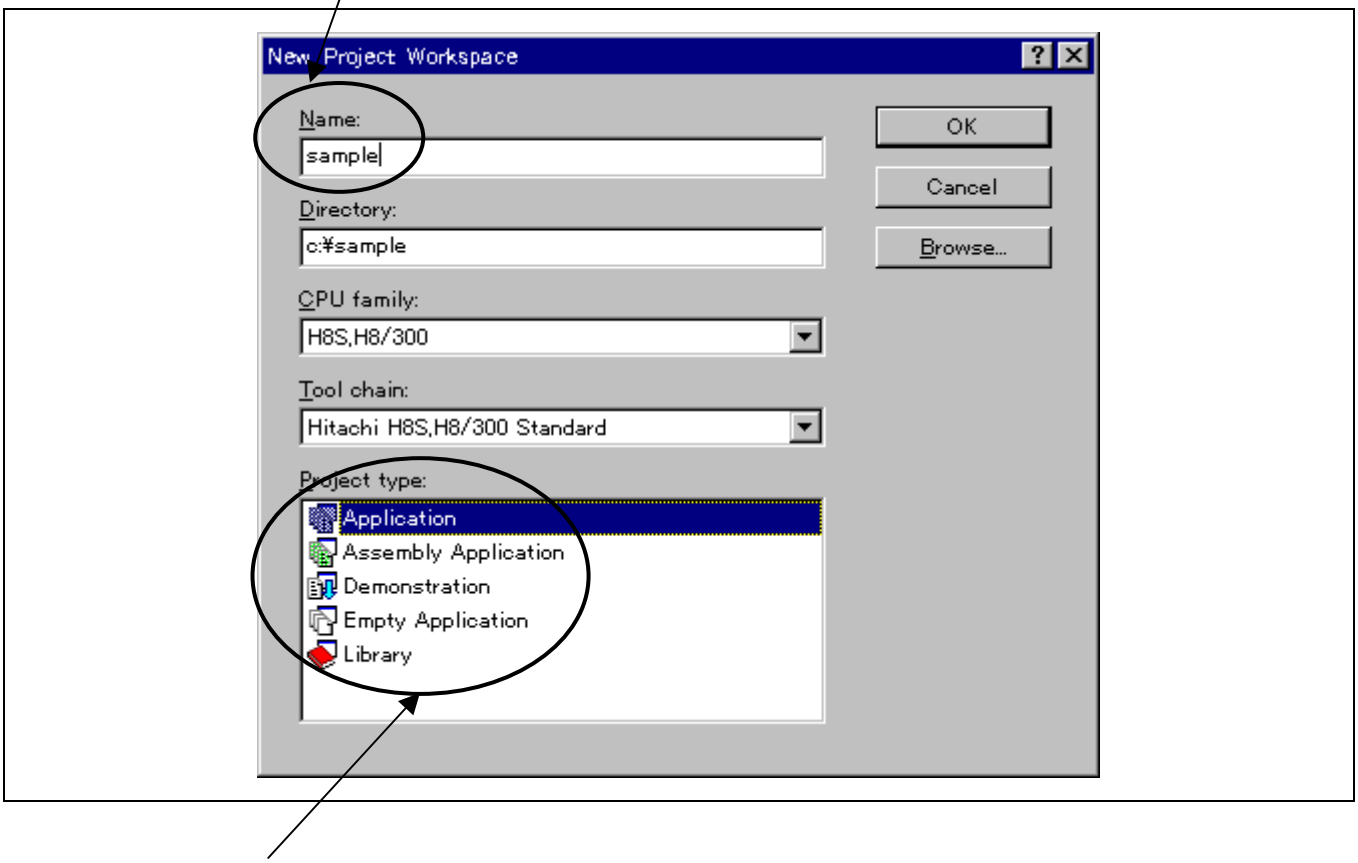
2.1.1　创建新的工作空间 1 (HEW1.2)

要创建新的工程工作空间，从“欢迎使用 (Welcome)！”对话框内选取创建新的工程工作空间 (Create a new project workspace)。



(1) 设定工程类型

当出现以下画面时，在**名称 (Name)** 字段内输入所需的工程名称。



然后，选取**工程类型 (Project type)**：列。

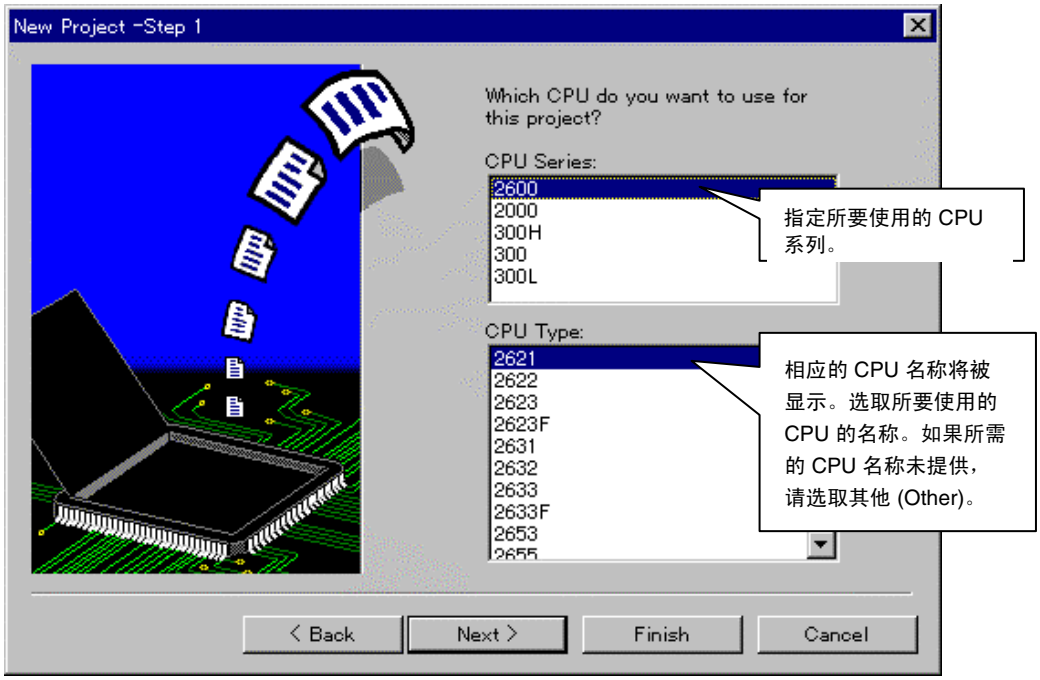
工程类型	描述
应用程序 (Application)	该工程类型创建包含 C/C++ 程序文件的应用程序
汇编应用程序 (Assembly Application)	该工程类型创建仅包含汇编语言程序的应用程序
演示 (Demonstration)	样品工程类型
空的应用程序 (Empty Application)	空的工程创建
程序库 (Library)	程序库创建工程类型

在选取所需的工程类型后单击 [确定 (OK)] 按钮，然后您可以前进到初始化新工程的步骤。

以下的阐释假设您已选定应用程序 (Application) 为工程类型。

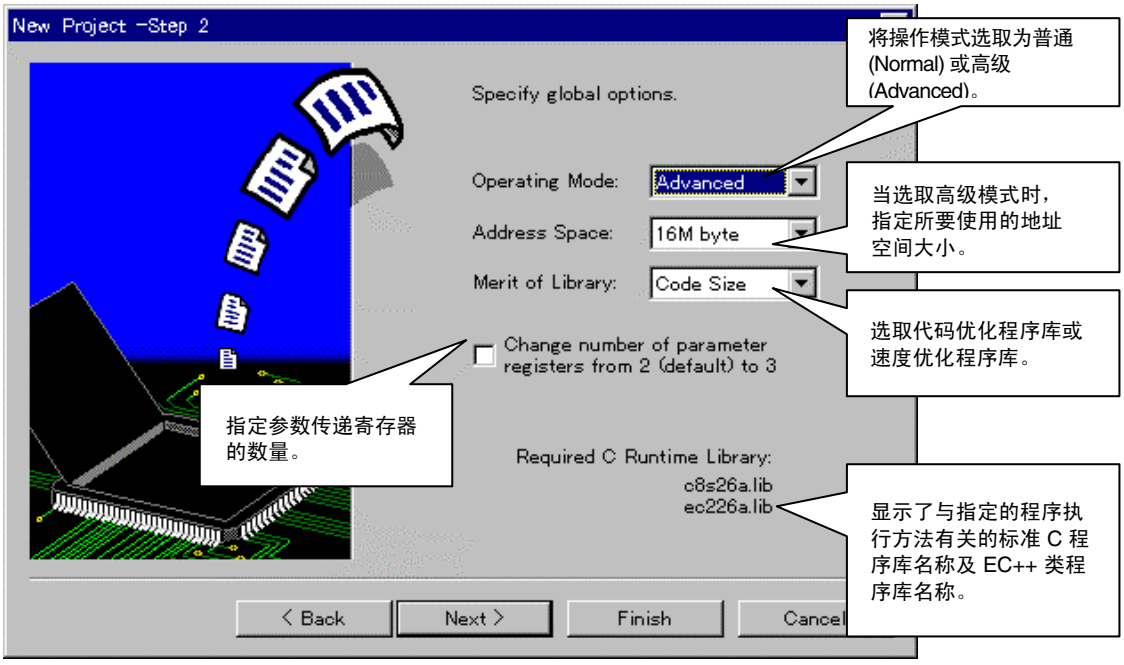
(2) 新的工程 — 步骤 1

指定将使用的 CPU 然后按下 下一步 (NEXT)> 。



(3) 新的工程 — 步骤 2

指定所需的全局选项，然后按下 下一步 (NEXT)> 。



同一套全局选项应被使用到所有工程文件上。

可用的全局选项类别如下：

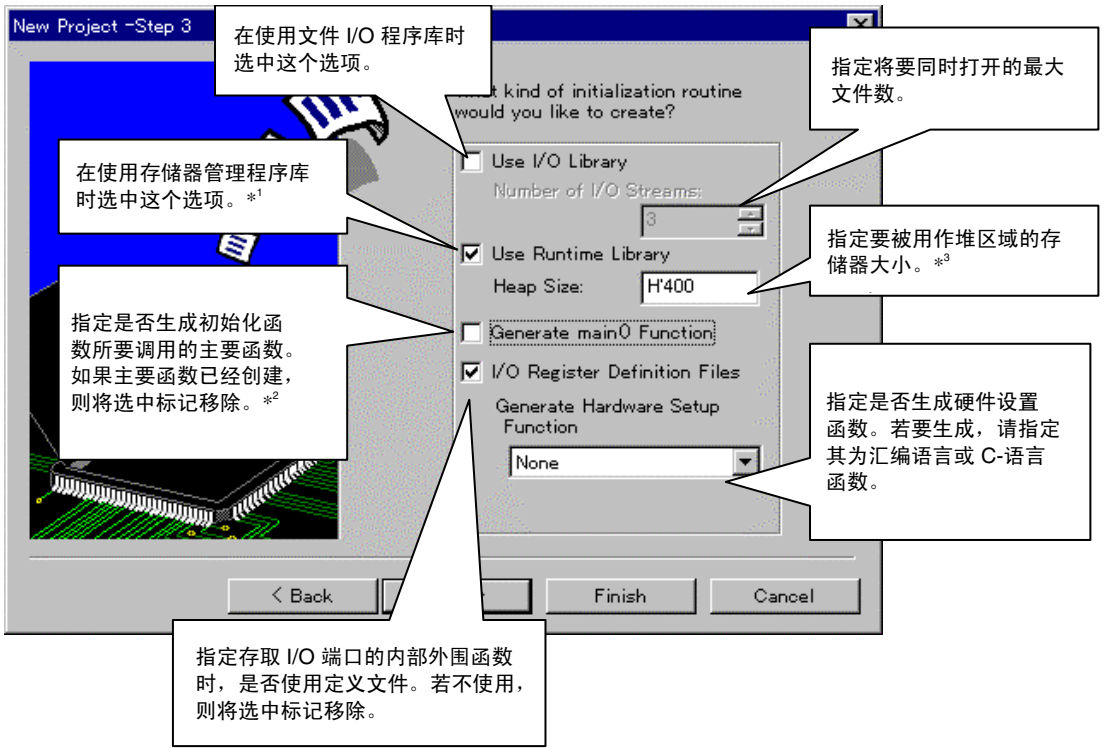
- CPU 类型
- 参数传递寄存器的数量

要在新的工程被初始化后更改这个对话框中所指定的全局选项，则必须修改所需连接的标准程序库的规格。

要获取如何更改全局选项和标准程序库的详细资料，请参考 11.2.1 节，“未定义外部符号”的输出。

(4) 新的工程 — 步骤 3

在这个画面上，指定初始化程序的内容然后按下 下一步 (NEXT)> 。



注意： 1. 可用的存储器程序库函数为 malloc、realloc、calloc，及 new。

2. 在这个对话框上，不应该创建主要函数。 在步骤（9）当中，添加了一个包含主要函数的样品程序以作为对 2.3 节，使用 HDI 进行调试的准备。

3. 堆区域所需的大小可通过下列方法计算：

（堆区域大小）≥
（由存储器管理程序库分配的区域大小）+（管理区域大小）

管理区域的大小如下：

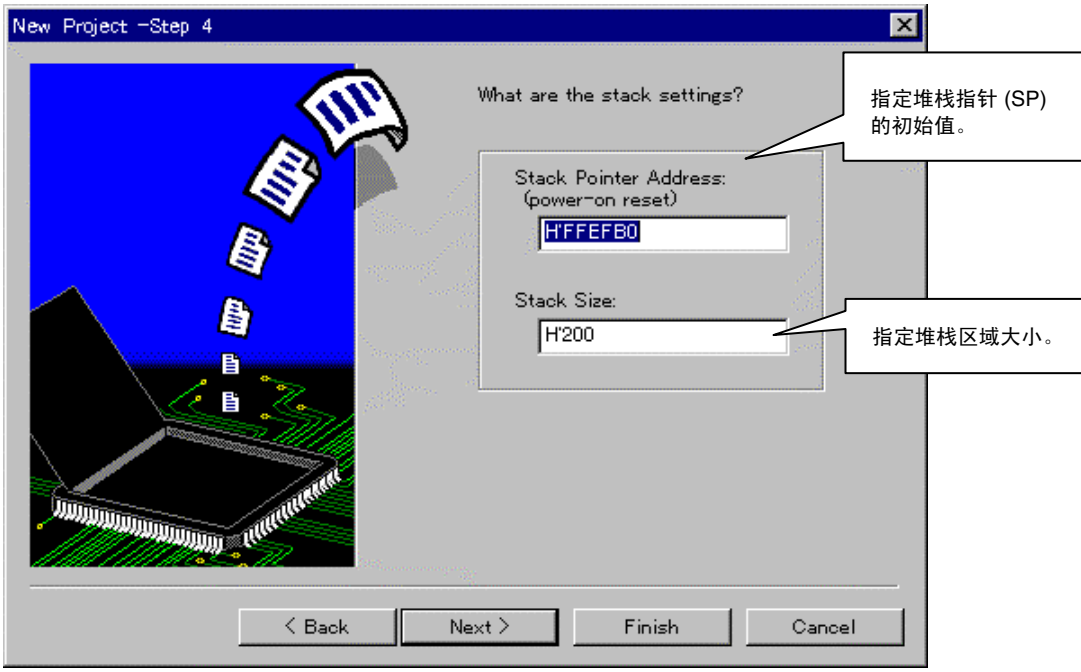
CPU 类型	管理区域大小
H8S/2600 ADV、H8S/2000 ADV、H8/300H ADV	16 字节
H8S/2600 NRM、H8S/2000 NRM、H8/300H NRM、H8/300	8 字节

ADV： 高级模式，NRM： 普通模式

要在新工程被初始化后修改本节中所指定的堆区域大小，请参考 2.2.1（2）节，分配堆区域。

(5) 新的工程 — 步骤 4

设定将使用的堆栈然后按下 下一步 (NEXT)> 。



所要使用的堆栈大小可通过下列方法决定：

为函数间调用关系内最深的调用嵌套计算堆栈区域大小。通过这种方式获得的最大值既是堆栈的区域大小。

例如，如果最深的函数调用嵌套如下，相加所有的堆栈大小：

主要函数（堆栈大小： 10 字节） → func 函数（20 字节） → sub 函数（30 字节）

堆栈大小在这里将为 60 字节。

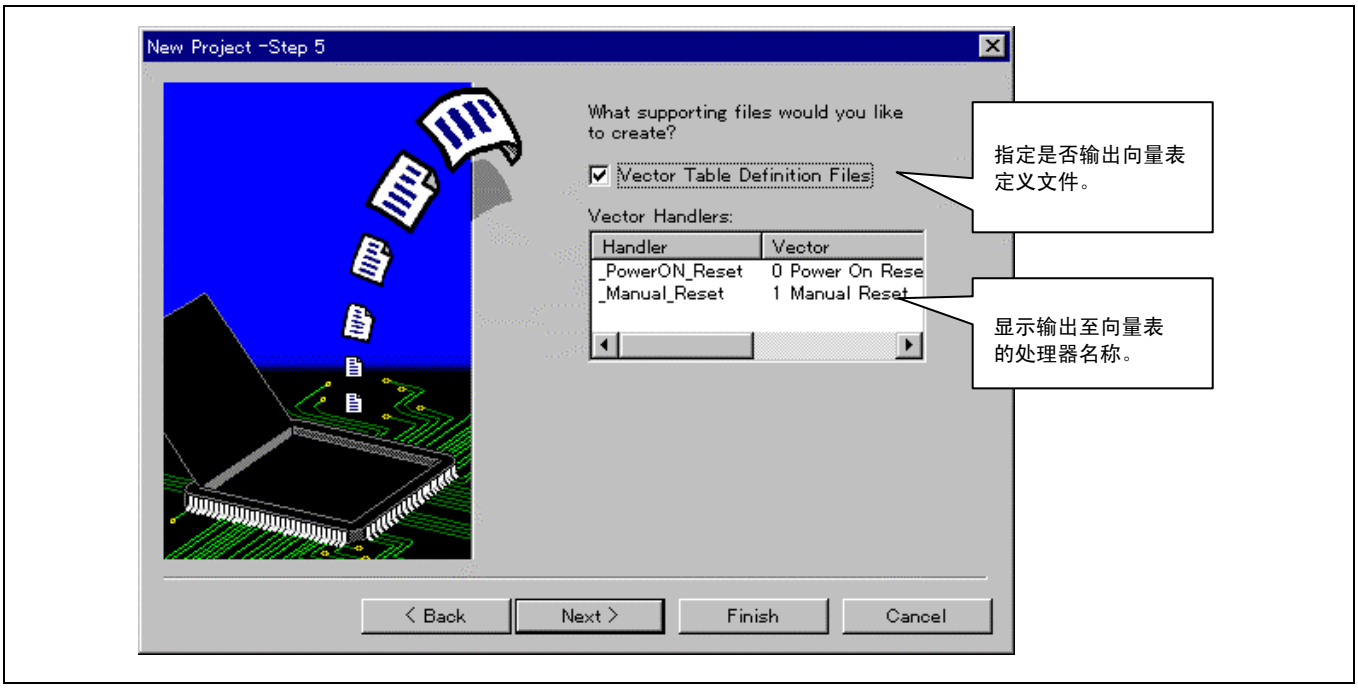
当符号分配信息输出被指定为目标列表文件输出规格的一部分时，函数的堆栈大小将被输出。

至于运行时例常程序，请参考 H8S 及 H8/300 系列 C/C++ 编译程序随附手册中的“标准程序库所使用的堆栈大小列表 (List of Stack Sizes Used by the Standard Library)”。

当要在新工程被初始化后修改本节中所指定的堆栈大小时，请参考 2.2.1（8）节，设定堆栈大小。

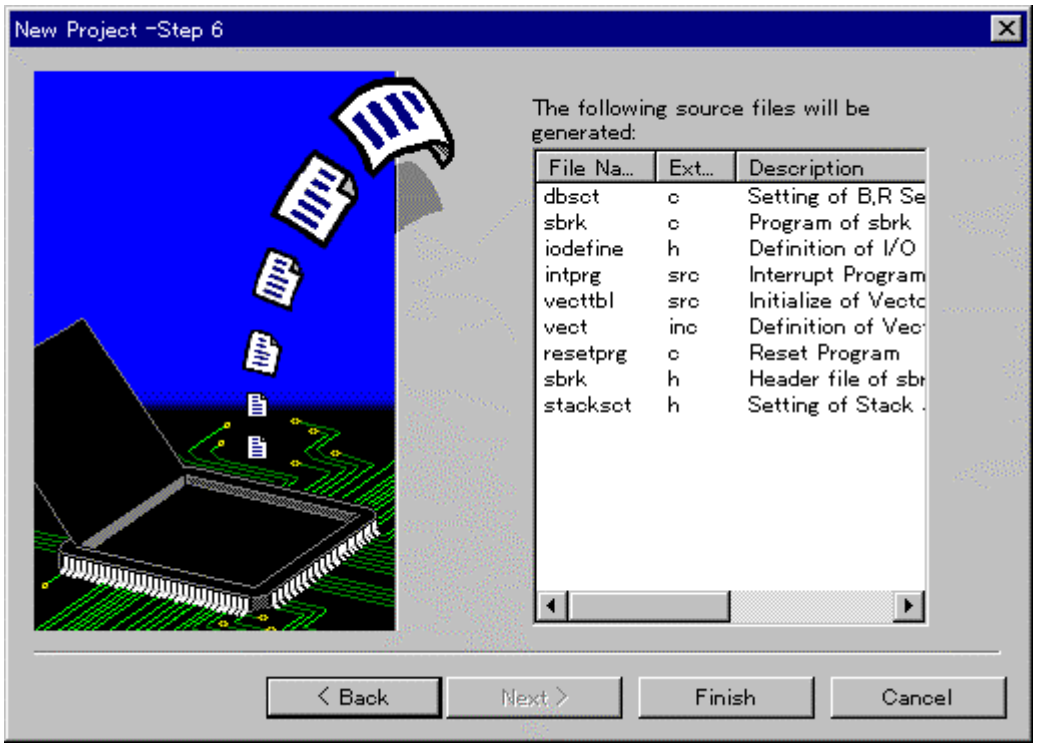
(6) 新的工程 － 步骤 5

指定向量表的设定然后按下 下一步 (NEXT)> 。



(7) 新的工程 — 步骤 6

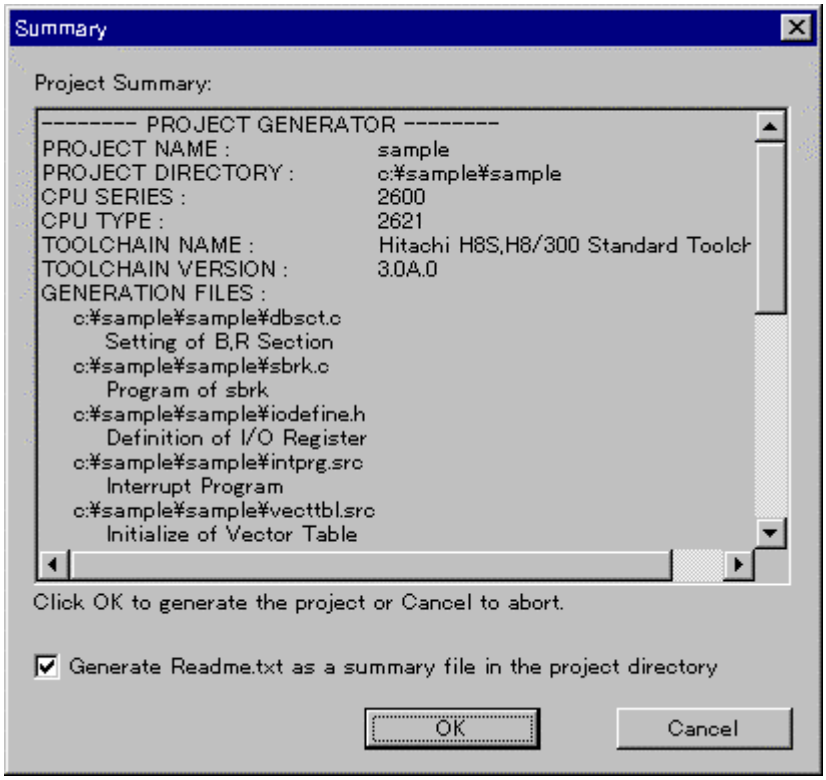
显示由工程生成程序所创建的文件。 按下完成 完成 (FINISH) 以前进到步骤 7。



有关本节中所创建的文件的详细资料，请参考 2.2 节，样品程序简介。

(8) 新的工程 — 步骤 7

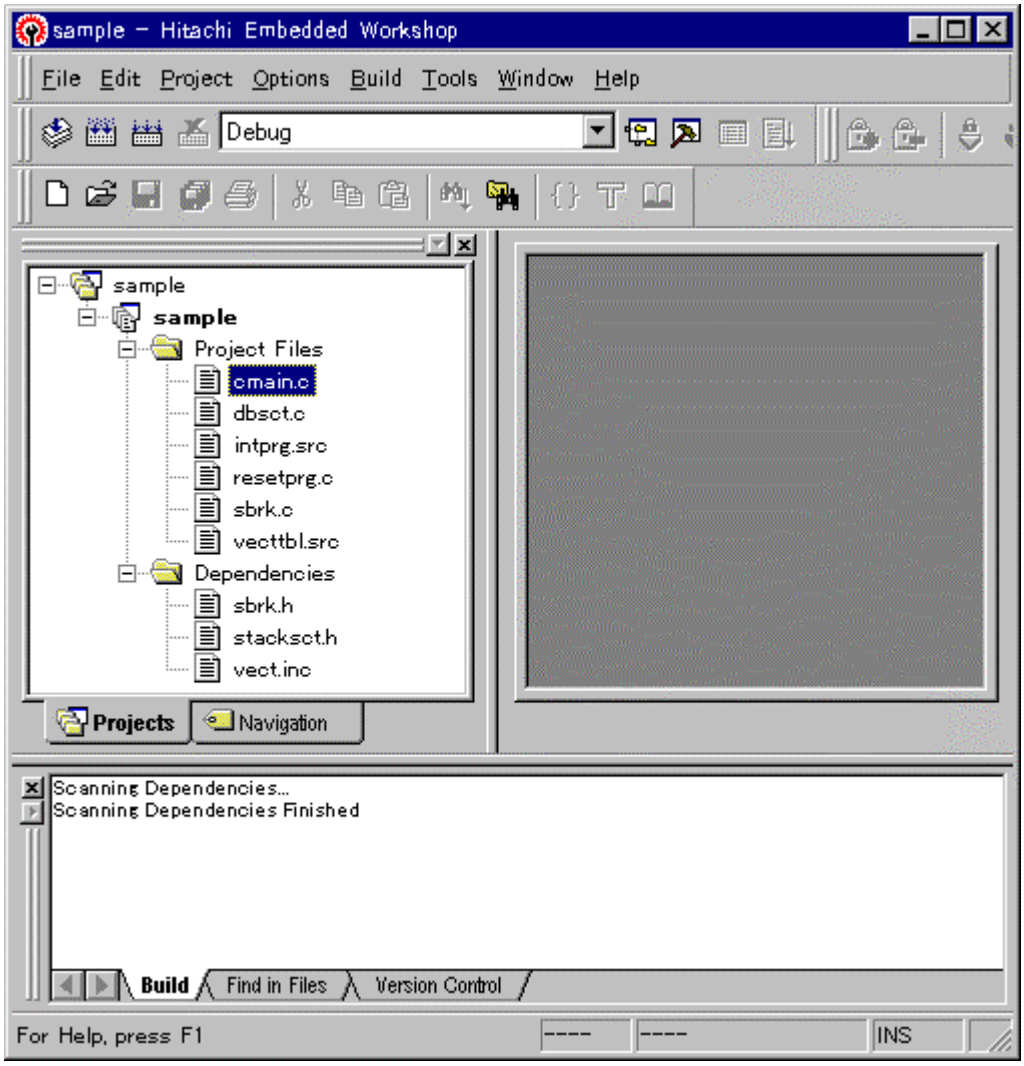
指定“完成 (Finish)”将显示下列画面：



(9) 添加主要文件

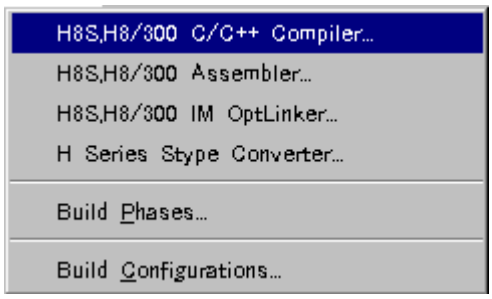
添加 cmain.c 文件到已完成的工程以进行主要处理。

在 [工程 (Project) → 添加文件 (Add Files) …] 内，指定 HEW 目录 \Tools\HITACHI\H8\3_0a_0\sample\cmain.c。



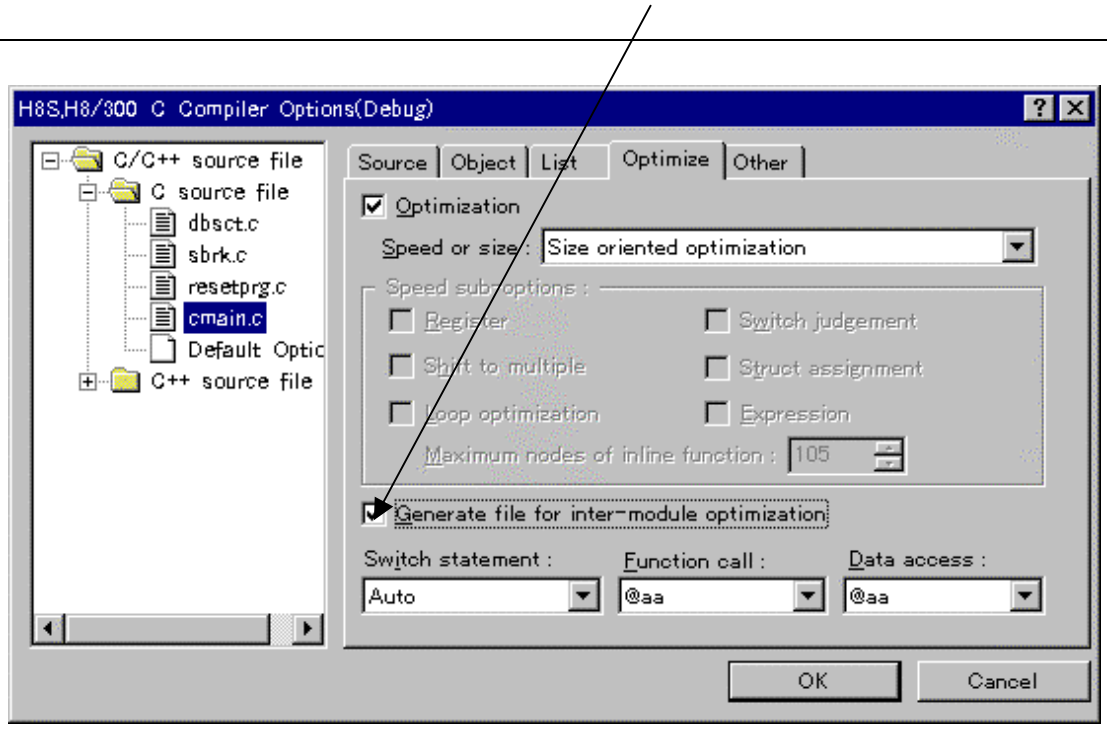
(10) 设定选项

从选项 (options) 菜单选取 H8S，H8/300 C/C++ 编译程序 (H8S,H8/300 C/C++ Compiler) …。



指定 cmain.c 的编译程序选项。

在这个对话框上，通过选中以下所示项目以指定模块间优化器加载信息文件的输出：

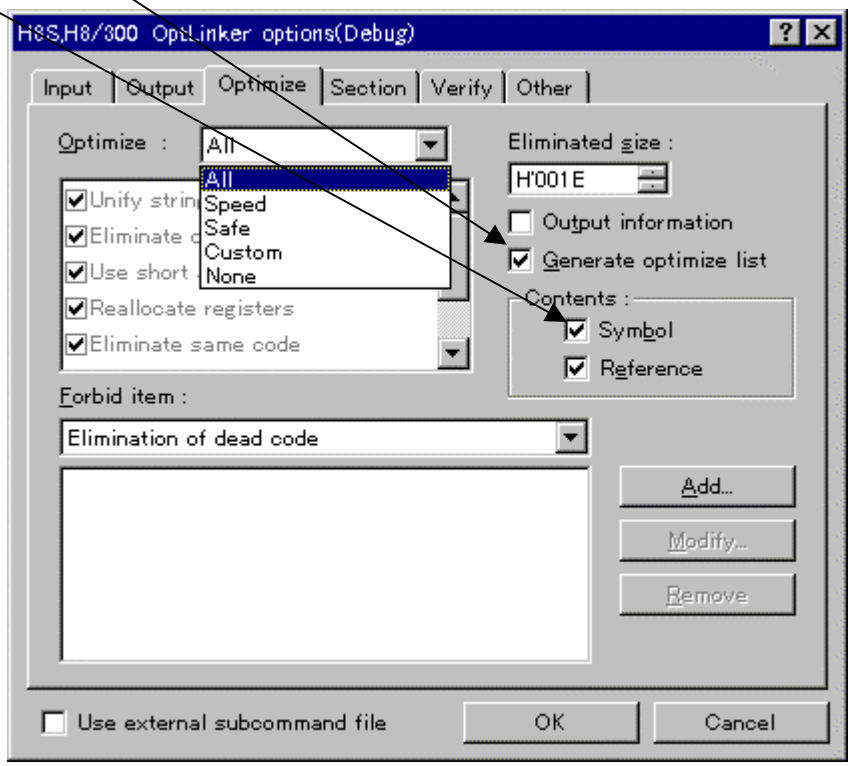


在下一个步骤中，选取 [选项 (Options) → H8S，H8/300 IM Optlinker...] 以指定模块间优化器的选项。

首先，在优化 (Optimize) 标签内，指定全部 (All) 以启用所有模块间优化功能。

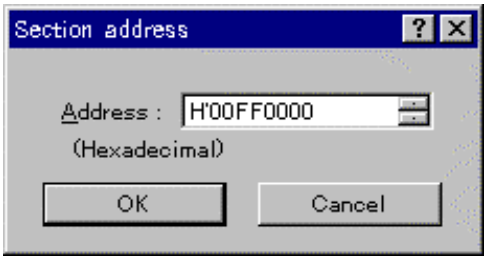
在这个标签上，在此指定优化信息列表的输出。

同时也在这里指定符号优化信息的输出以及这个列表的符号参考的数量。

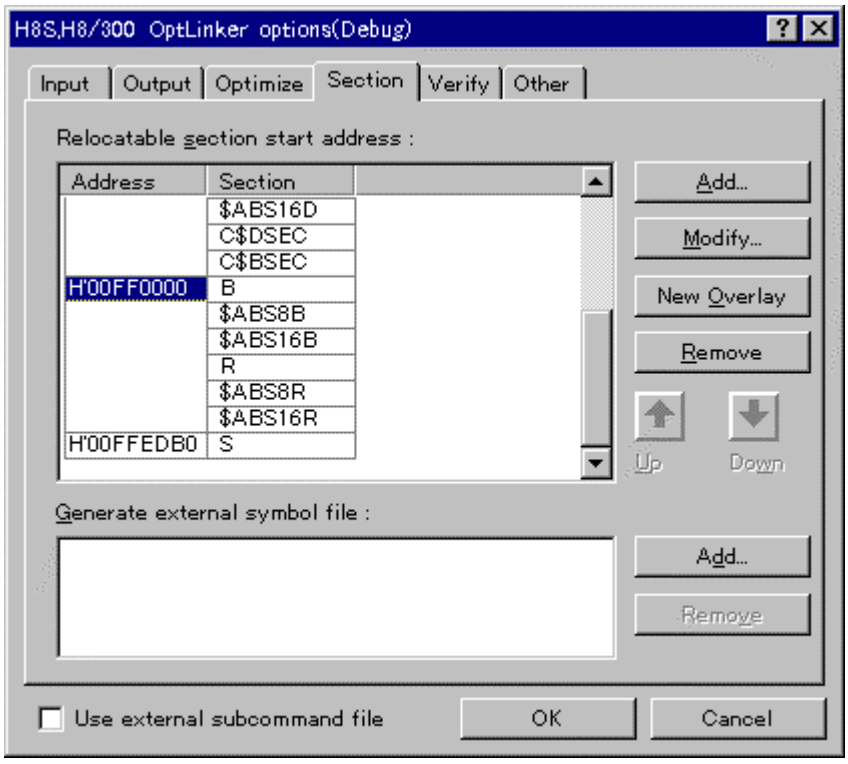


在下一个步骤中，在段 (section) 标签内指定文件将在连接上被分配的方式。

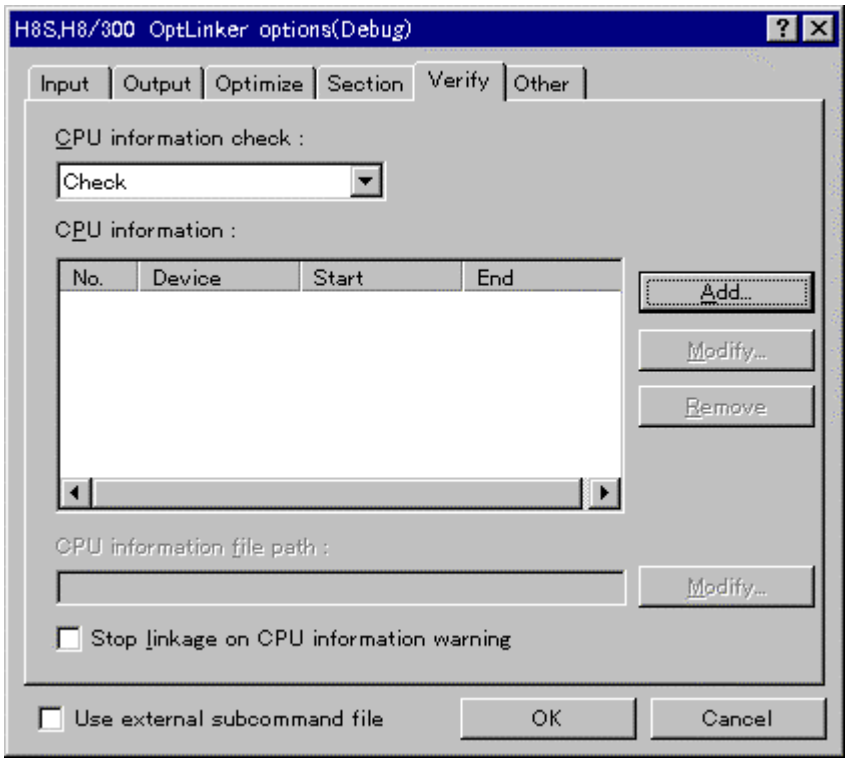
在这里，更改段的地址以使段 B 被分配给 H'00FF00。 首先单击地址 (Address) 字段，然后按下修改 (Modify) 按钮以指定地址。



地址的修改如下所示。

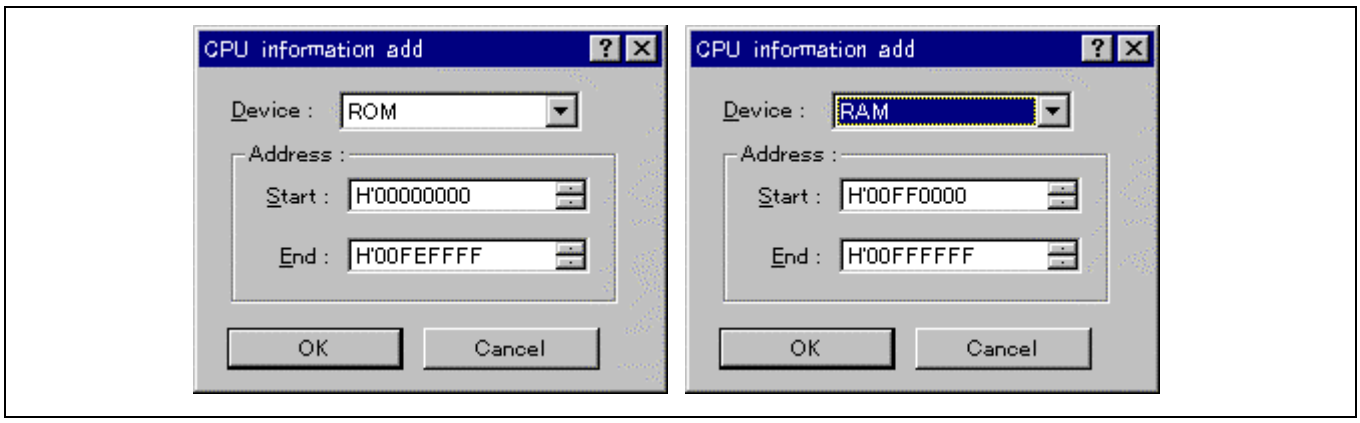


在下一个步骤中，在验证 (Verify) 标签内，创建 CPU 信息以检查 CPU 赋值。



在 CPU 信息检查 (CPU information check) 字段内选取检查 (Check) 可允许用户检查 CPU 信息。

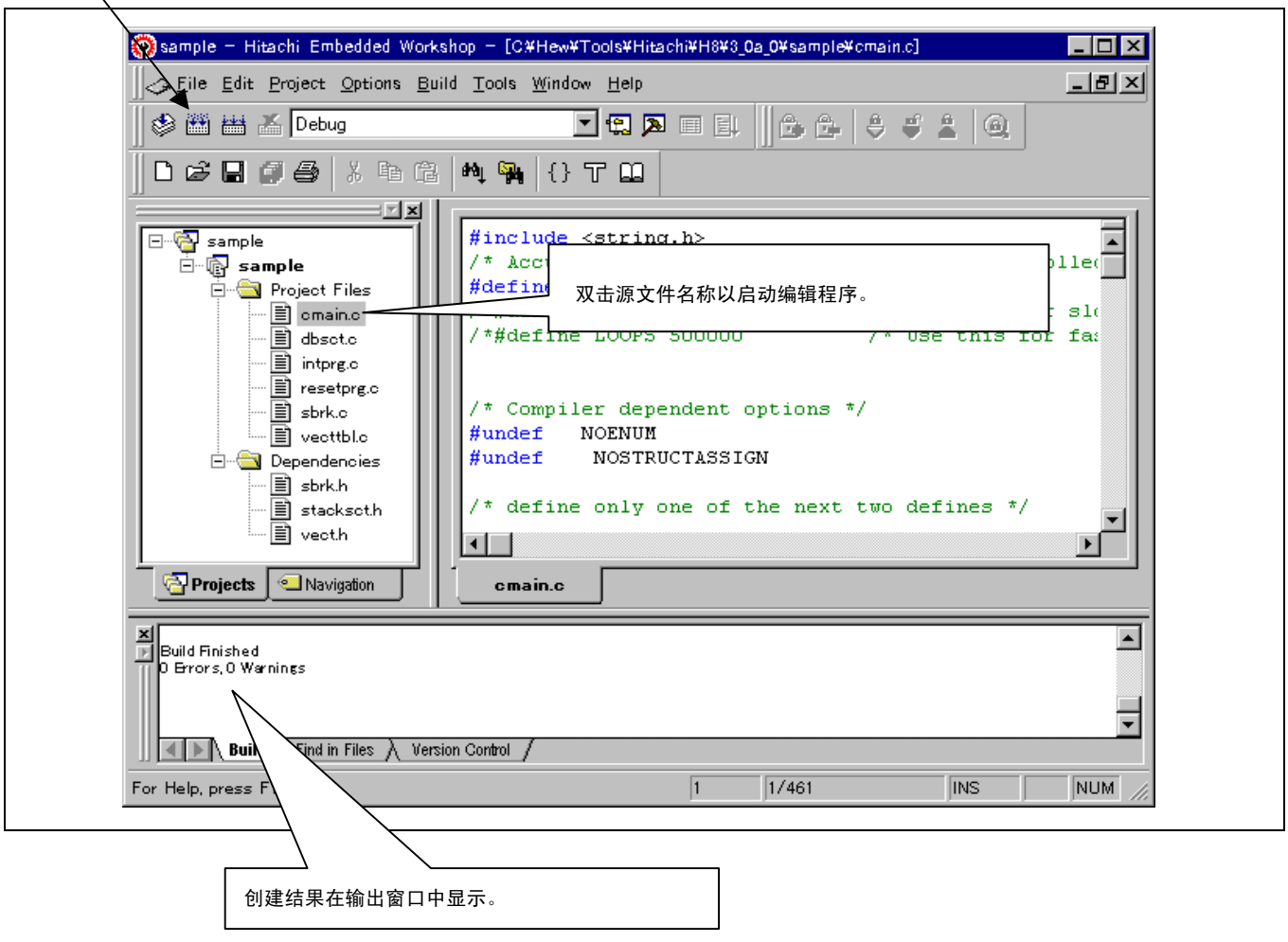
按下 [添加 (Add) …] 按钮将显示一个对话框，可在对话框上如下所示指定 ROM 和 RAM 区域。



(11) 执行创建过程

执行创建过程以生成装入模块。

在这里按下命令按钮可执行创建。

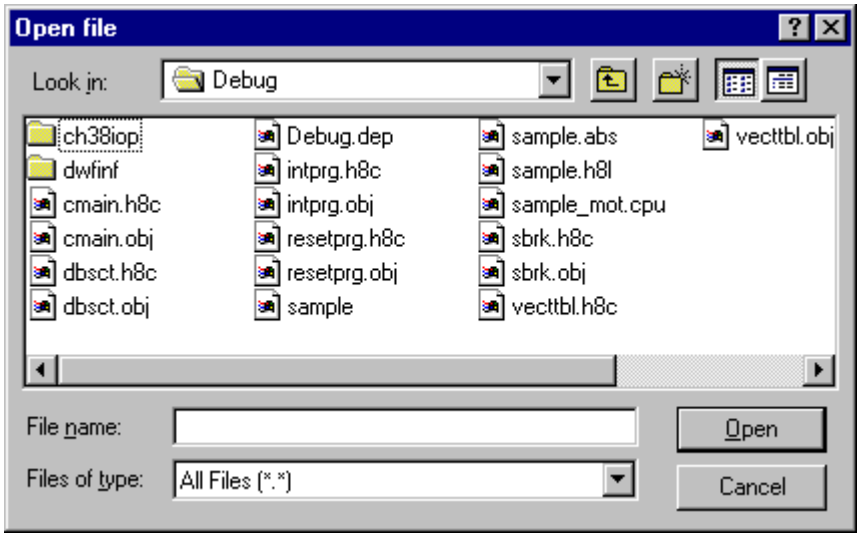


(12) 验证生成的文件

下列文件在创建过程完成后即被生成。

一个名称和工程名称相同的目录被创建于工程目录下。 绝对装入模块以 sample.abs 的名称格式生成于新目录的调试目录内。

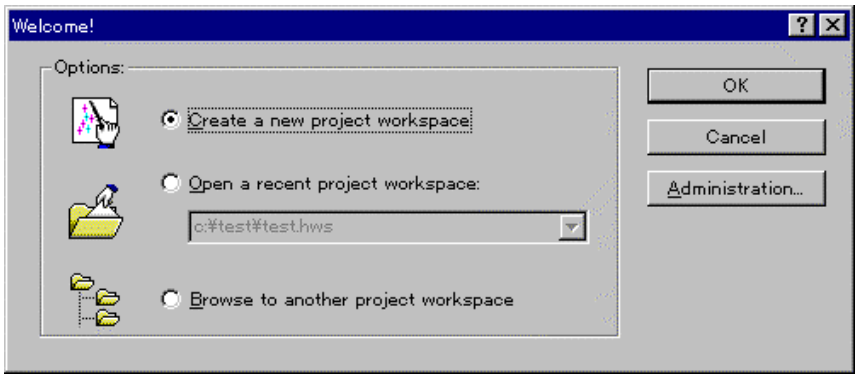
创建过程期间所生成的映像文件以及优化信息列表文件被存储在相同的目录内，可通过单击 [文件 (File) → 打开 (Open)] 来打开和检查这些文件。



映像文件以 sample.map 的名称生成；优化信息列表文件以 sample.lop 的名称生成。

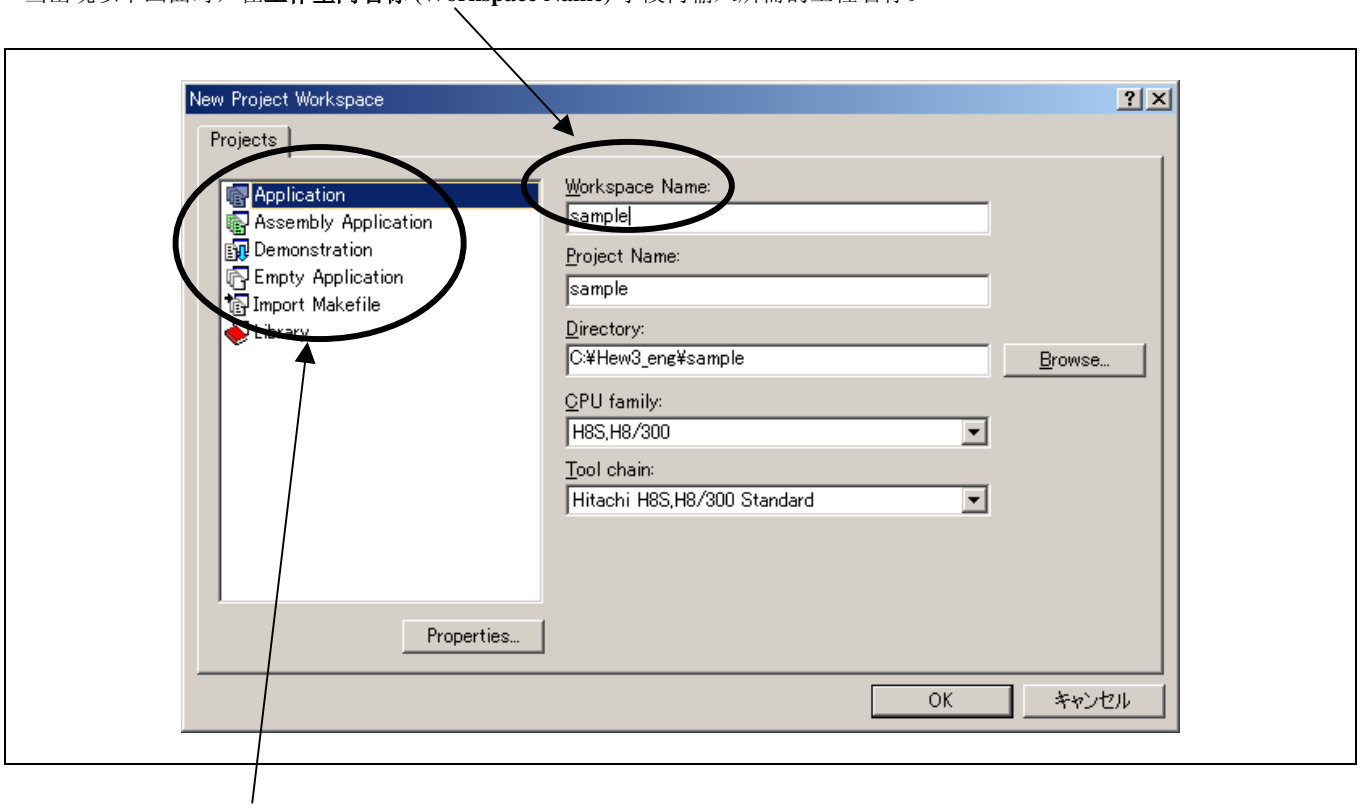
2.1.2 创建新的工作空间 2 (HEW2.0)

要创建新的工程工作空间，从“欢迎使用 (Welcome)！”对话框内选取创建新的工程工作空间 (Create a new project workspace)。



(1) 设定工程类型

当出现以下画面时，在**工作空间名称 (Workspace Name)** 字段内输入所需的工程名称。



然后，选取**工程 (Projects)** 列。

工程类型	描述
应用程序 (Application)	该工程类型创建包含 C/C++ 程序文件的应用程序
汇编应用程序 (Assembly Application)	该工程类型创建仅包含汇编语言程序的应用程序
演示 (Demonstration)	样品工程类型
空的应用程序 (Empty Application)	空的工程创建
程序库 (Library)	程序库创建工程类型

在这个对话框内，设定工作空间名称（当创建新的工作空间工程时，工程名称在默认情况下是相同的）、CPU 类型和工程类型。

若您在 [工作空间名称 (Workspace Name)] 字段内输入 “sample” 为工作空间名称，则 [工程名称 (Project Name)] 将为 “sample” 且 [目录 (Directory)] 也将是 “c:\hew2\sample”。要更改工程名称，请在 [工程名称 (Project Name)] 字段内直接键入一个名称。 要更改用作工作空间的目录，可通过单击 [浏览 (Browse) …] 或直接在 [目录 (Directory)] 字段内输入目录路径来选取目录。

在选取所需的工程类型后单击 [确定 (OK)] 按钮，然后您可以前进到初始化新工程的步骤。

以下的阐释假设您已选定应用程序 (Application)为工程类型。

(2) 新的工程 － 1/9

指定将使用的 CPU 然后按下 下一步 (NEXT)>。

在新的工程工作空间 (New Project Workspace) 对话框内单击 [确定 (OK)] 将启动工程生成程序。 首先，选取所要使用的 CPU。所要使用的 CPU 类型（[CPU 类型 (CPU Type)]）已根据每个 CPU 系列（[CPU 系列： (CPU Series:)]）进行分类。 为所要开发的程序选取 CPU 类型，因为生成的文件将根据所选的 [CPU 系列：(CPU Series:)] 和 [CPU 类型：(CPU Type:)] 而有所不同。若所需的 CPU 类型未提供，则请选取拥有类似硬件规格的或“其他 (Other)”CPU 类型。

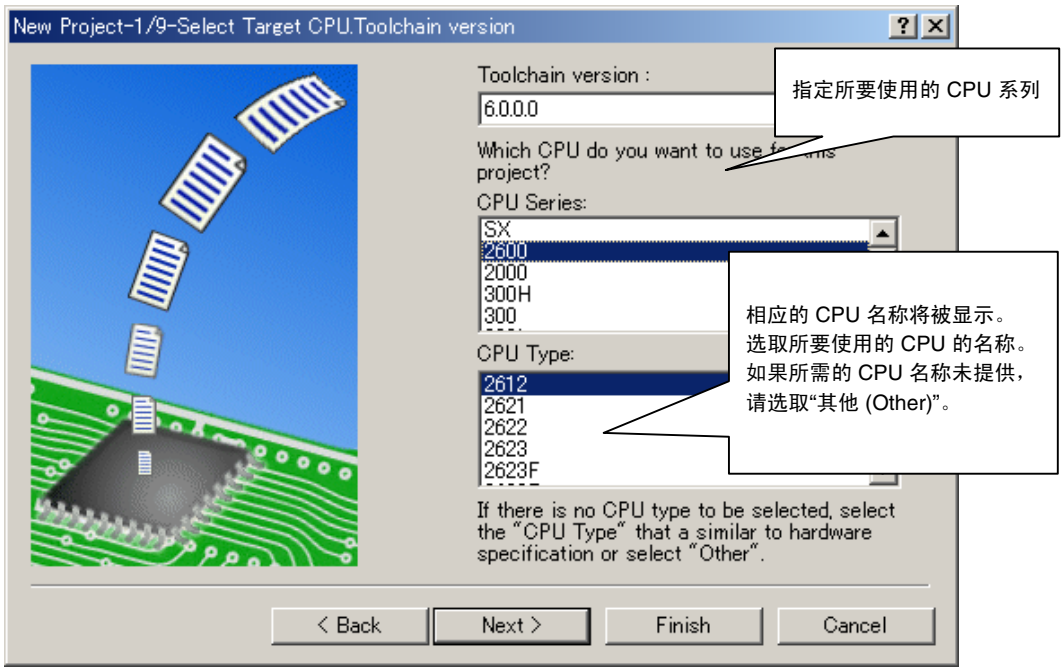
单击 下一步 (NEXT)> 以显示下列画面。

单击 <返回 (Back) 以在此画面之后显示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) 以检索新的工作空间 (New Workspace) 对话框。

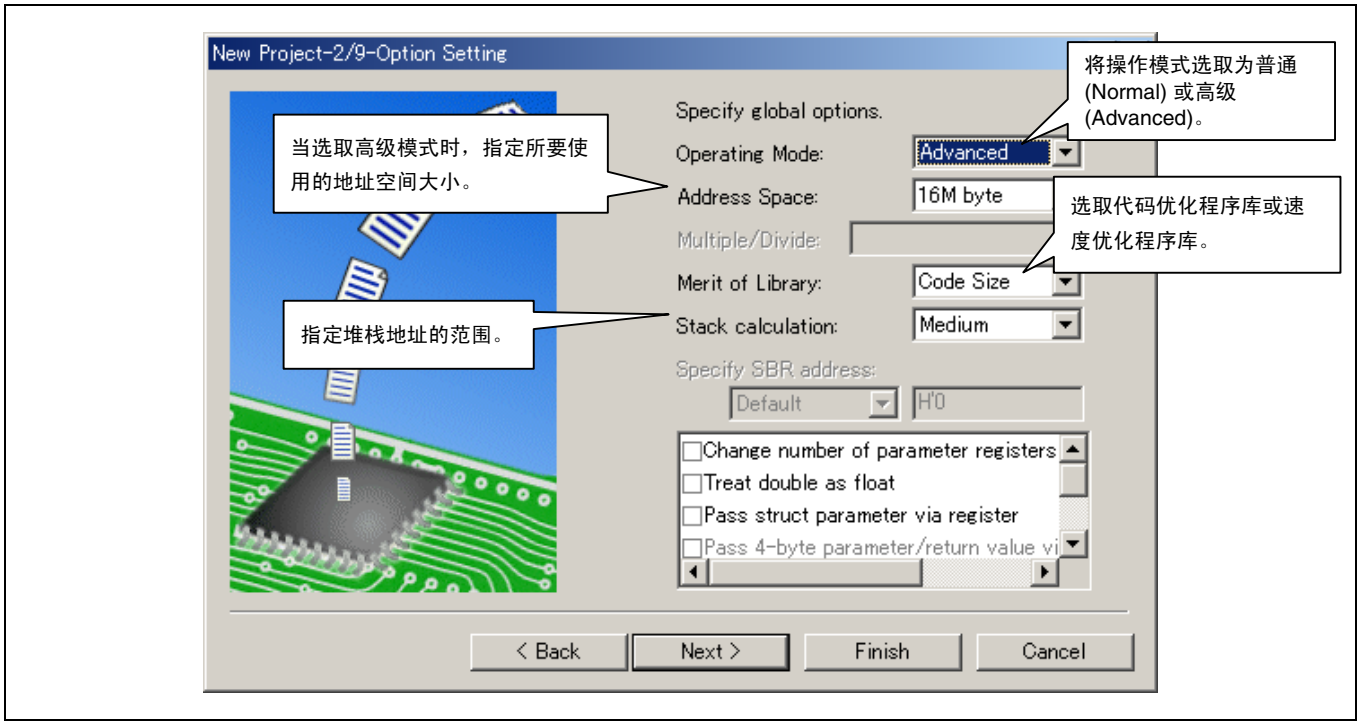
<返回 (Back)、下一步 (NEXT)>、完成 (Finish)、和取消 (Cancel) 等函数在本向导对话框内被广泛使用。



(3) 新的工程 － 2/9

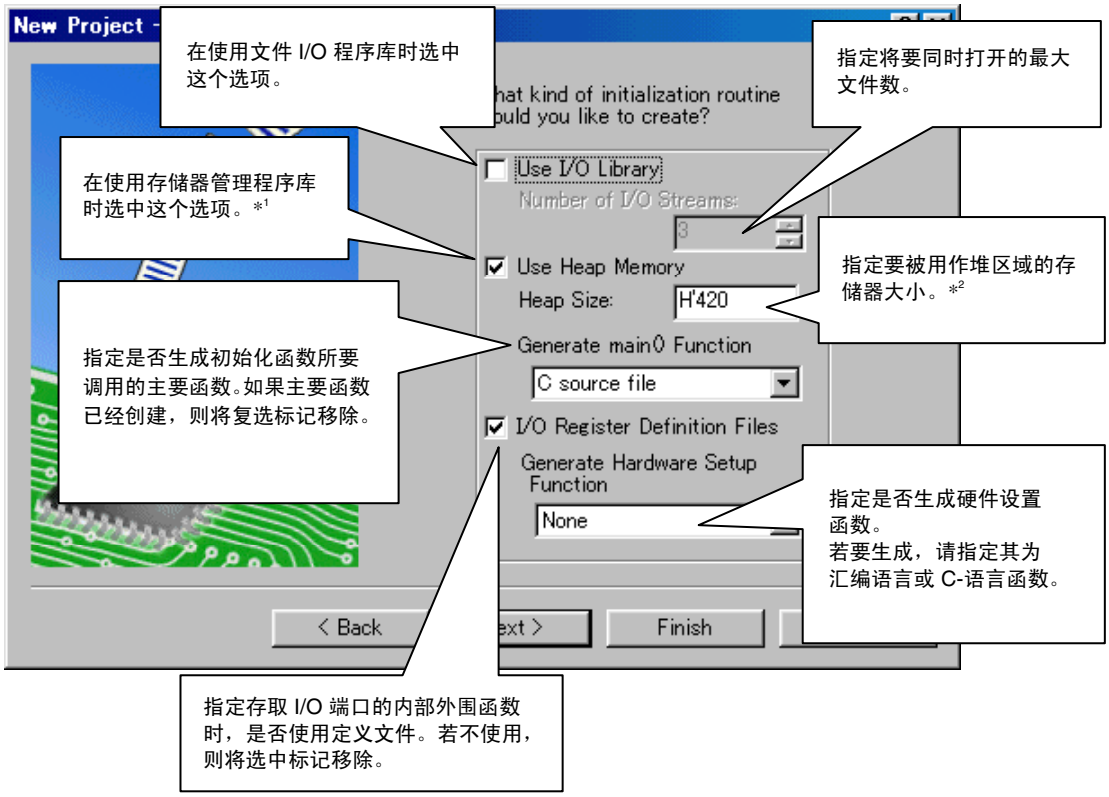
指定所需的全局选项，然后按下 下一步 (NEXT)>。

在这个画面上，设定所有工程文件的普遍选项。 设定选项的项目可能根据步骤 1 画面中所选取的 CPU 系列而更改。 如果您要在创建工程后更改选项，您可以在 HEW 内 [CPU 标签 (CPU Tab)] 上的 [选项 (Options)-> H8S，H8/300 标准工具链 (H8S,H8/300 Standard Toolchain)] 进行。



(4) 新的工程 — 3/9

在本画面上，指定初始化程序的内容然后按下 下一步 (NEXT)>。



注意： 1. 可用的存储器程序库函数为 malloc、realloc、calloc，及 new。

2. 堆区域所需的大小可通过下列方法计算：

（堆区域大小）≥
（由存储器管理程序库分配的区域大小）+（管理区域大小）

管理区域的大小如下：

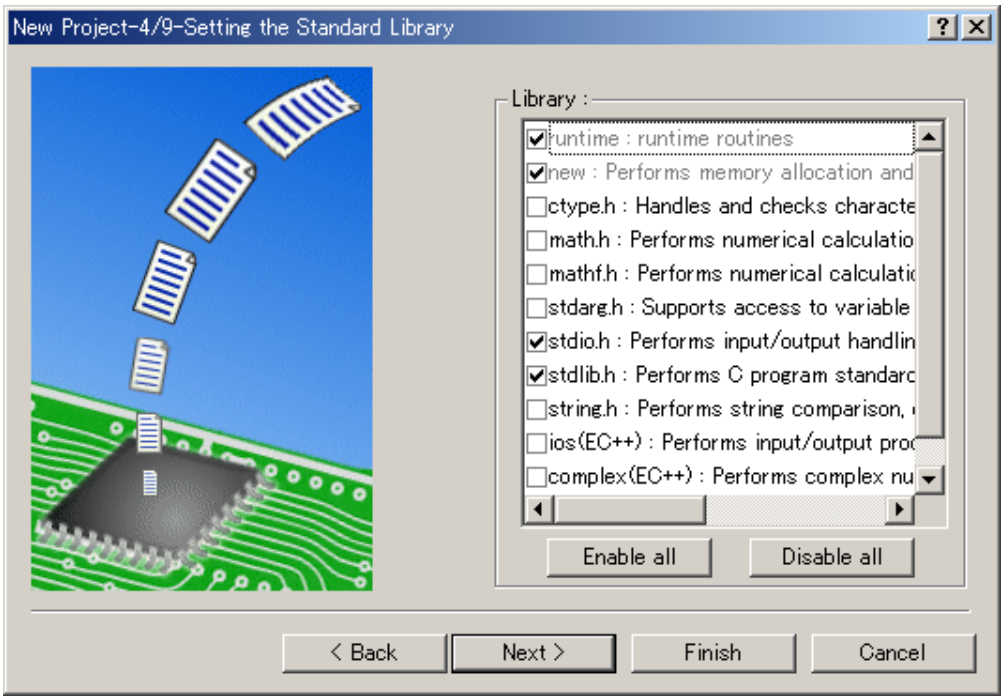
CPU 类型	管理区域大小
H8S/2600 ADV、H8S/2000 ADV、H8/300H ADV	16 字节
H8S/2600 NRM、H8S/2000 NRM、H8/300H NRM、H8/300	8 字节

ADV： 高级模式，NRM： 普通模式

要在新工程被初始化后修改本节中所指定的堆区域大小，请参考 2.2.1（2）节，分配堆区域。

(5) 新的工程 － 4/9

在这个画面上，决定 C/C++ 编译程序所要使用的标准程序库组织。如果您要在创建工程后更改标准程序库组织，您可以在 HEW 内 [标准程序库标签 (Standard Library Tab)] 上的 [选项 (Options)-> H8S，H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) …] 进行。

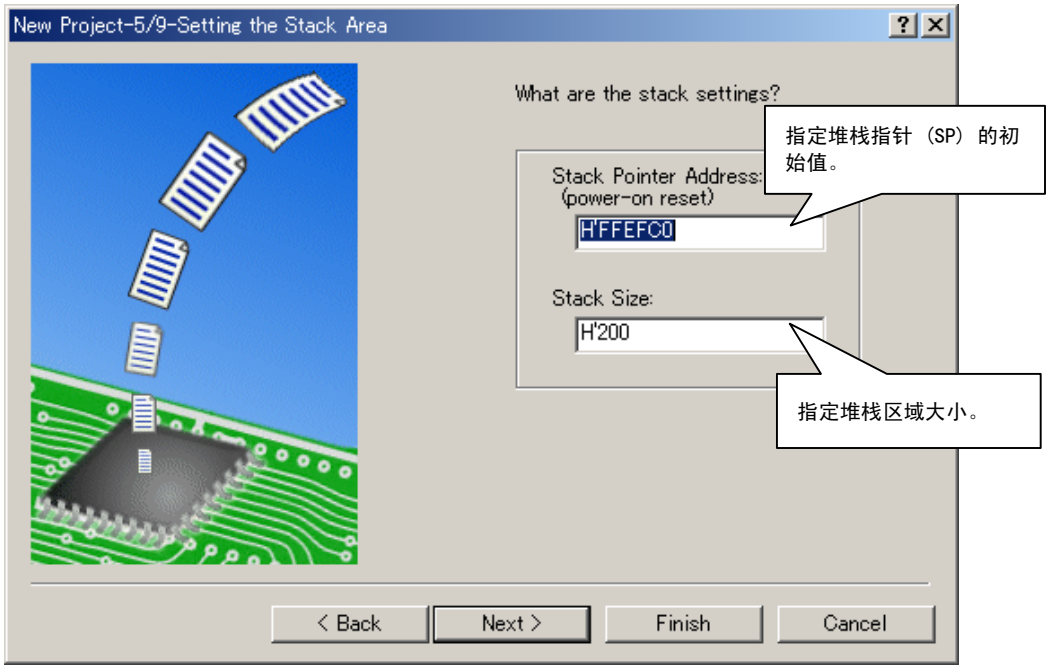


(6) 新的工程 — 5/9

设定将使用的堆栈然后按下 下一步 (NEXT)>。

在这个画面上，设定堆栈区域。要设定为堆栈区域的初始化数值将根据步骤 1 画面上的 [CPU 类型：(CPU Type:)] 而有所变化。

如果您要在创建工程后更改堆栈大小，您可以在 HEW 内的 [工程 (Project)-> 编辑工程配置 (Edit ProjectConfiguration)] 上进行。



所要使用的堆栈大小可通过下列方法决定：

为函数间调用关系内最深的调用嵌套计算堆栈区域大小。 通过这种方式获得的最大值既是堆栈的区域大小。

例如，如果最深的函数调用嵌套如下，相加所有的堆栈大小：

主要函数（堆栈大小： 10 字节） → func 函数（20 字节） → sub 函数（30 字节）

堆栈大小在这里将为 60 字节。

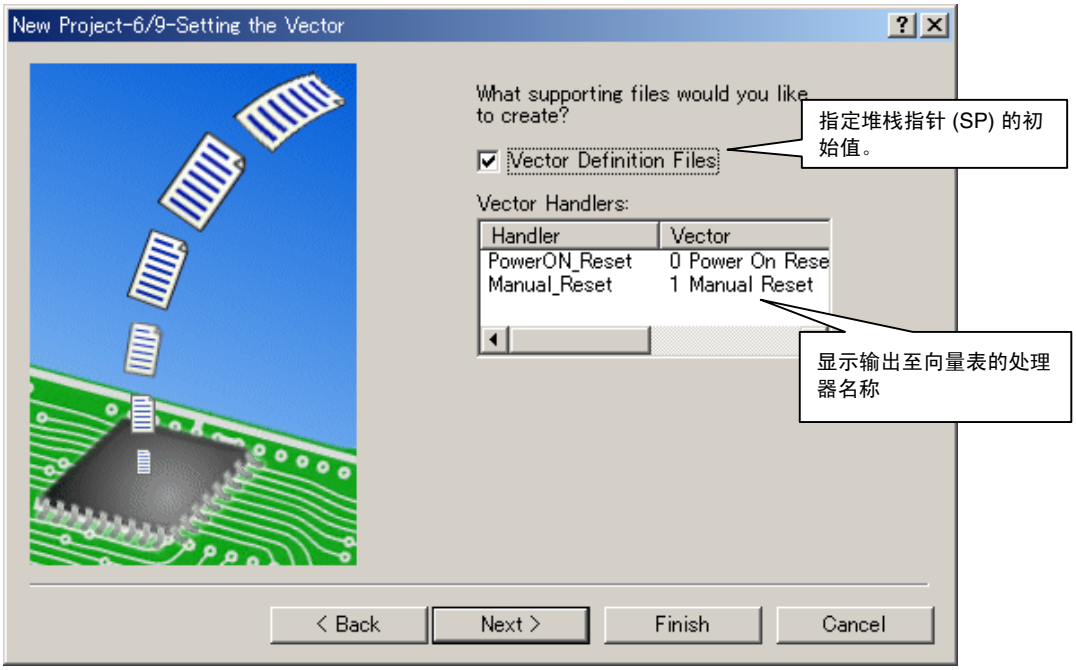
当符号分配信息输出被指定为目标列表文件输出规格的一部分时，函数的堆栈大小将被输出。

C/C++ 程序和标准程序库所使用的堆栈区域最大空间，可在指定优化连接编辑程序 (Optimizing Linkage Editor) 的堆栈选项并输出堆栈信息文件时，以堆栈分析工具 (stack analysis tools) 算出。 要获取有关使用堆栈分析工具的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S,H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”第 6 节，操作堆栈分析工具 (Operating Stack Analysis Tool)。

(7) 新的工程 － 6/9

指定向量表的设定然后按下 下一步 (NEXT)>。

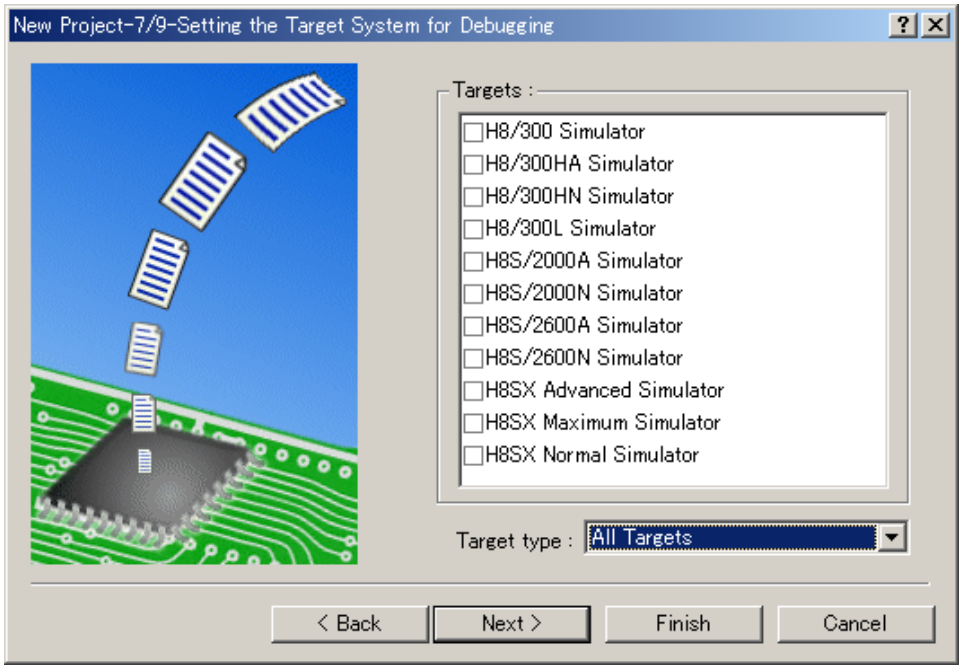
要修改处理器程序 (Handler Program)，请选择处理器程序名称，点按该名称，然后输入。 请注意在处理器程序被修改后将不生成复位程序 (reset program, reserprg.c)。



(8) 新的工程 － 7/9

指定调试程序目标，然后按下 下一步 (NEXT)>。

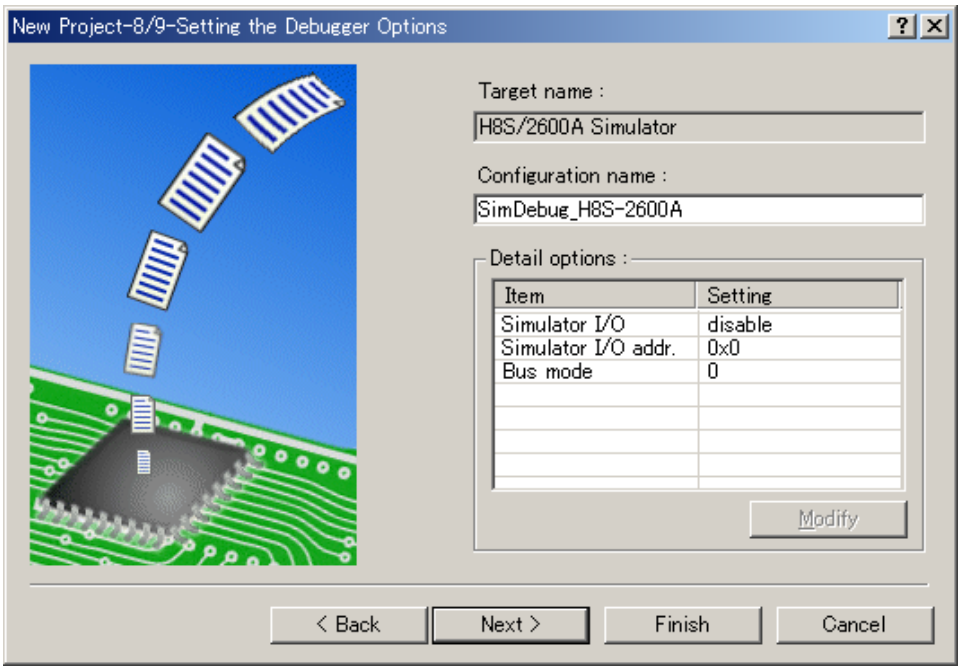
从 [目标： (Target:)] 选取（选中）所要使用的调试程序目标。 您可选择无调试程序目标或多个调试程序目标。



(9) 新的工程 － 8/9

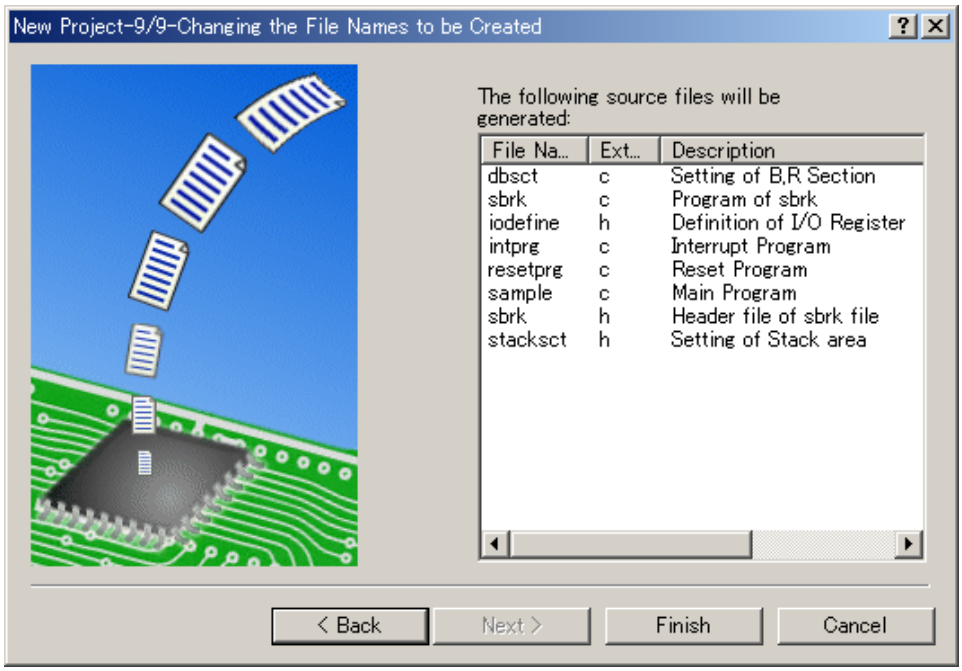
为选取的调试程序目标设定选项，然后按下 下一步 (NEXT)>。

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）。可在 [配置名称：(Configuration name:)] 内修改配置名称。调试目标的选项将显示在 [资料选项：(Detail options:)] 之下。若要更改设定，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。



(10) 新的工程 － 9/9

显示由工程生成程序所创建的文件。 然后按下 完成 (Finish)。

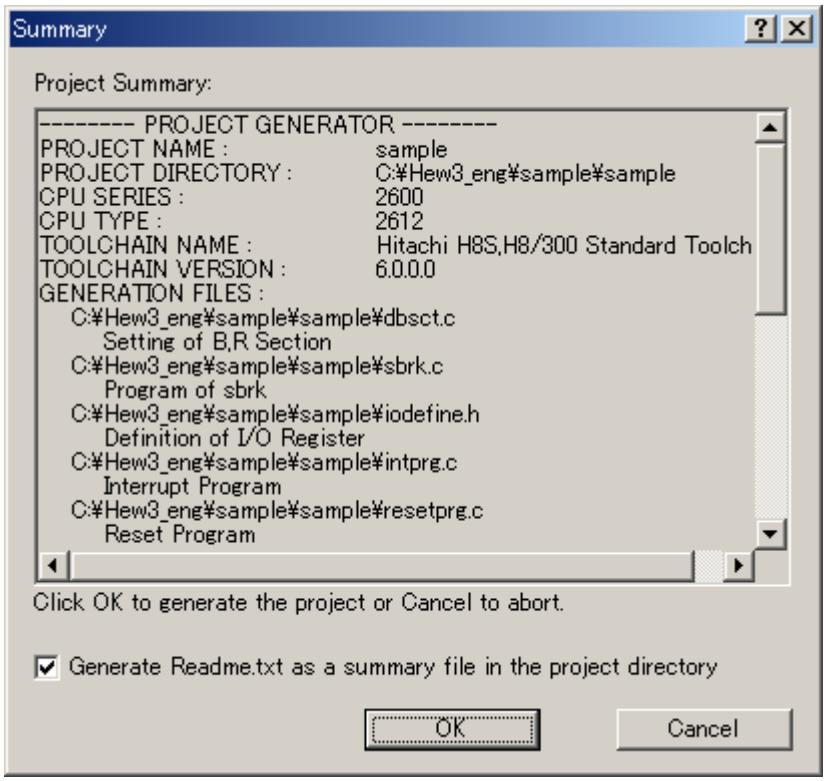


有关本节中所创建的文件的详细资料，请参考 2.2 节，样品程序简介。

(11) 新的工程 – 摘要

在步骤 9 的画面上单击 [完成 (Finish) >] 以使工程生成程序显示将被生成在摘要 (Summary) 对话框内的工程信息。检查这些信息，然后单击 [确定 (OK)]。

通过选中 [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)]，可将摘要对话框内所显示的工程信息以命名为 “Readme.txt” 的文本文件保存在工程目录 (Project Directory) 内。



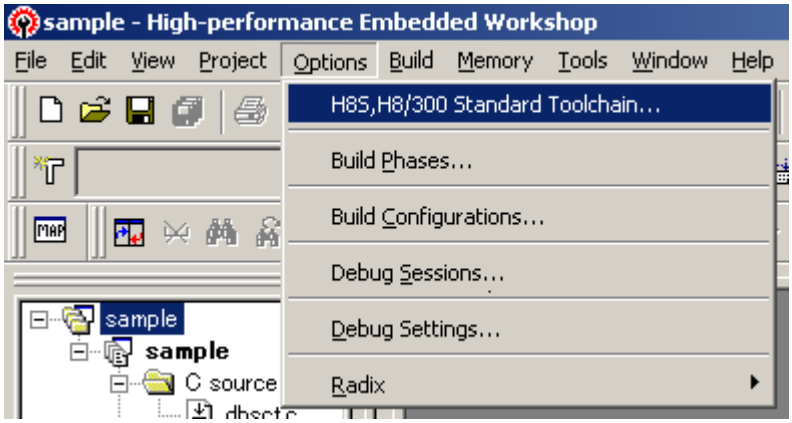
(12) 其他

若从工程类型 (Project Type) 选取了演示 (demonstration)，则将包含在模拟程序调试时可被使用的低层程序库样品。所要添加的文件如下：

- 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）

(13) 设定选项

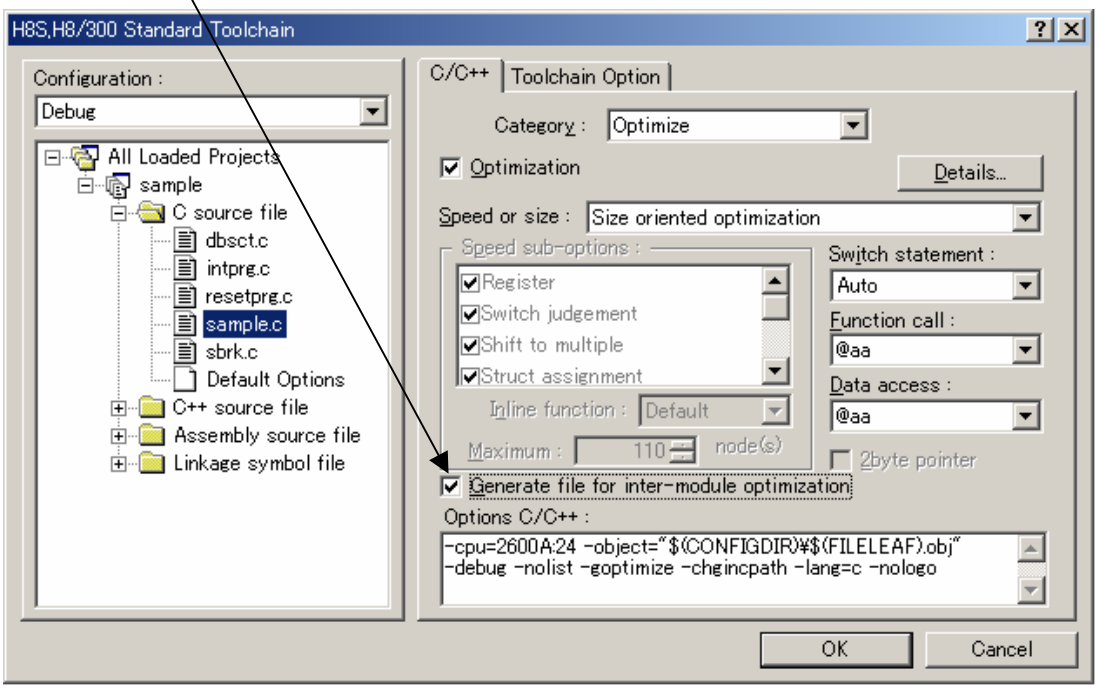
从选项 (options) 菜单选取 H8S，H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) …。



指定 sample.c 的编译程序选项。

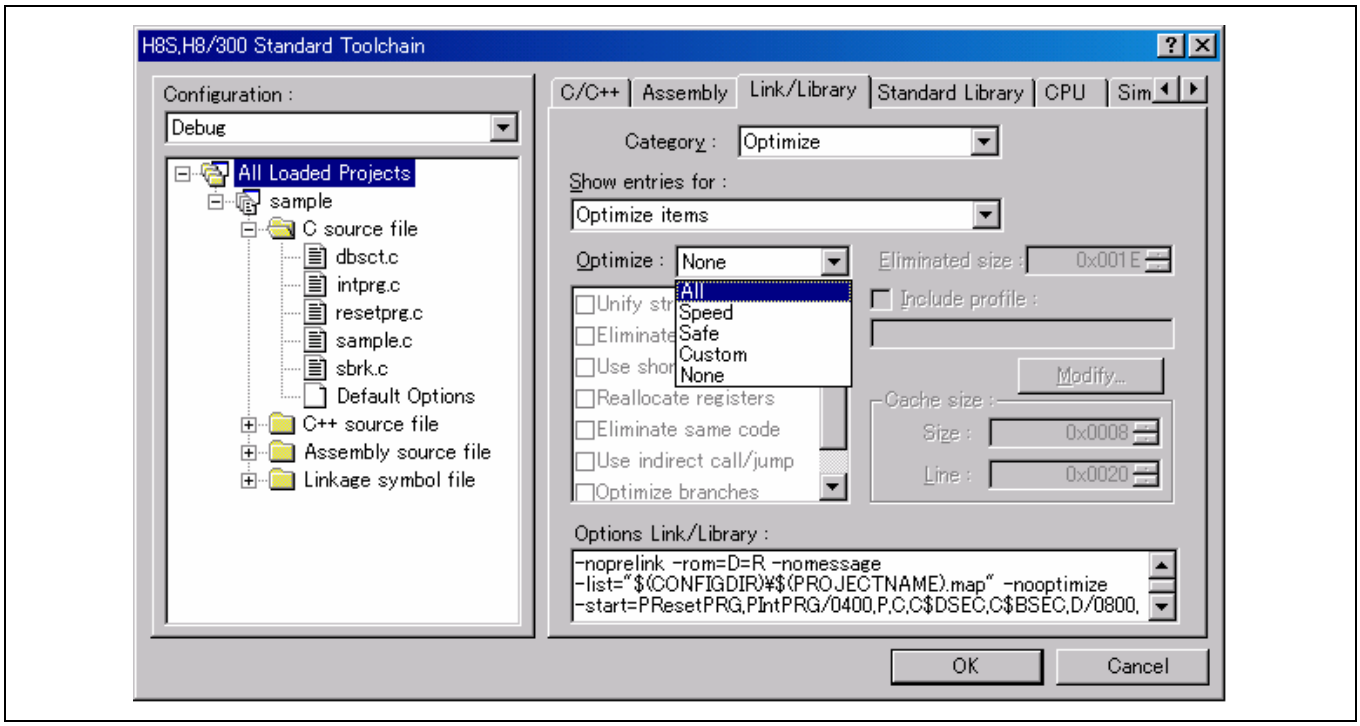
在 HEW 内选取 [选项 (Options)-> H8S，H8/300 标准工具链 (H8S,H8/300 Standard Toolchain)] 的 [C/C++ 标签 (C/C++ Tab)] [类别/优化 (Category/Optimize)]。

在这个对话框上，通过选中以下所示项目以指定模块间优化器加载信息的输出：

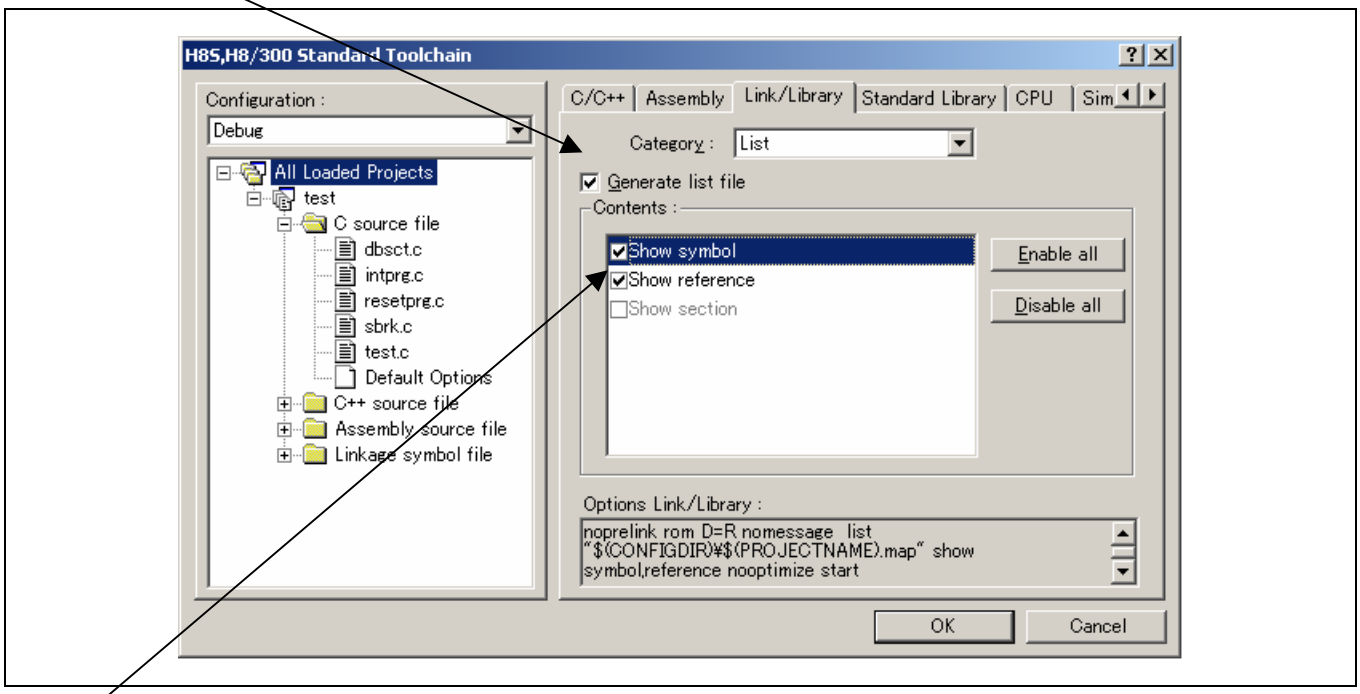


在下一个步骤中，在工程文件列表内指定全部装入的工程 (All Loaded Project) 以选取 [连接/程序库标签 (Link/Library Tab)] [类别/优化 (Category/Optimize)]，并指定模块间优化器的选项。

首先，在优化 (Optimize) 标签内，指定全部 (All) 以启用所有模块间优化功能。



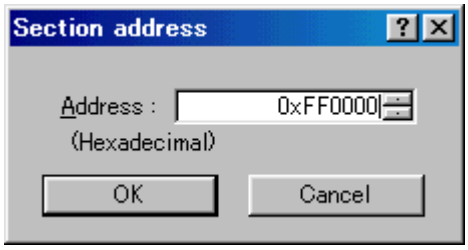
同时指定这里以在 [连接/程序库标签 (Link/Library Tab)] [类别/列表 (Category/List)] 上输出优化信息列表。



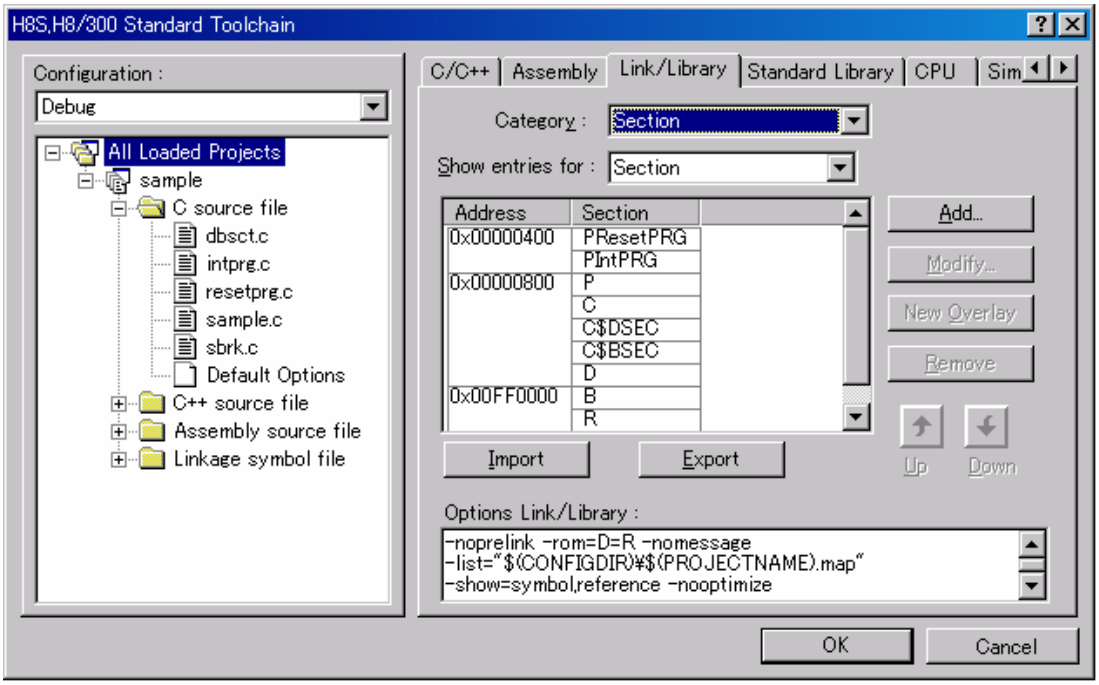
也在这里指定符号优化信息的输出以及这个列表的符号参考的数量。

在下一个步骤中，在 [连接/程序库标签 (Link/Library Tab)] [类别/段 (Category/Section)] 内指定文件将在连接上被分配的方式。

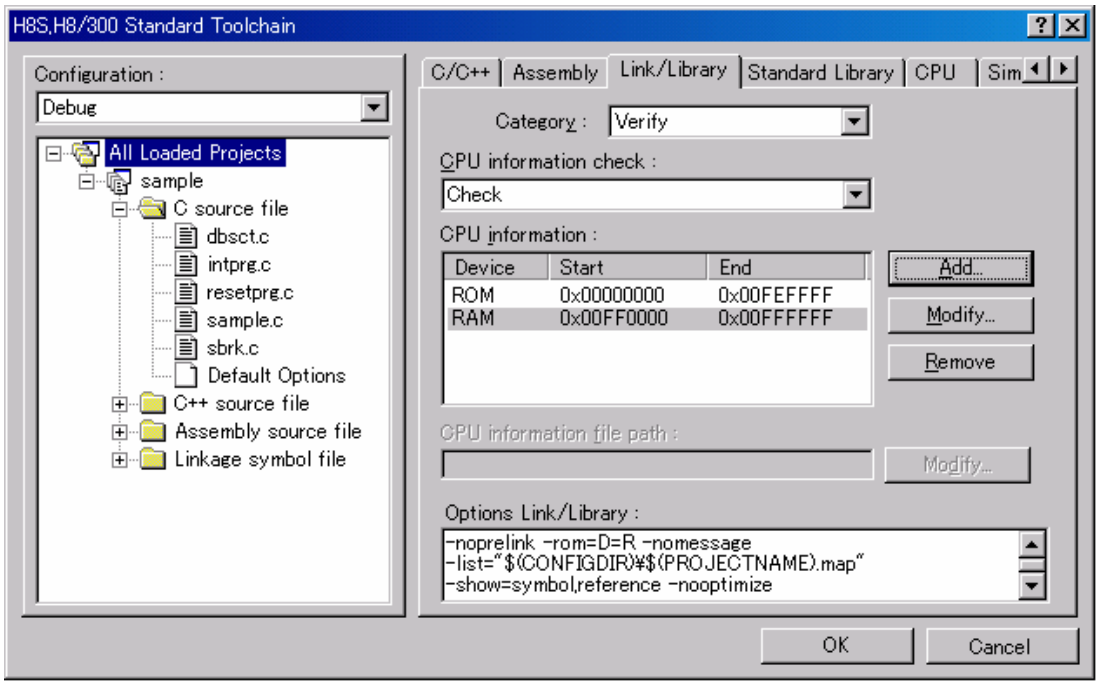
在这里，更改段的地址以使段 B 被分配给 H’00FF0000。 首先单击地址 (Address) 字段，然后按下修改 (Modify) 按钮以指定地址。



地址的修改如下所示。

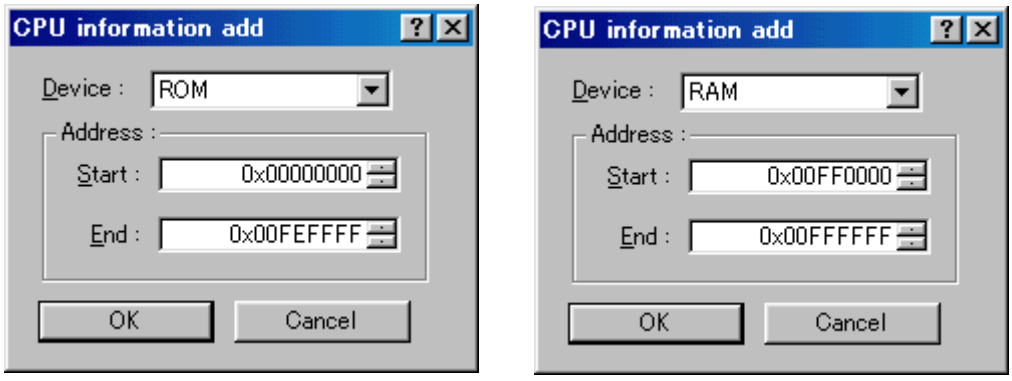


在下一个步骤中，在 [连接/程序库标签 (Link/Library Tab)] [类别/验证 (Category/Verify)] 内，创建 CPU 信息以检查 CPU 赋值。



在 CPU 信息检查 (CPU information check) 字段内选取检查 (Check) 可允许用户检查 CPU 信息。

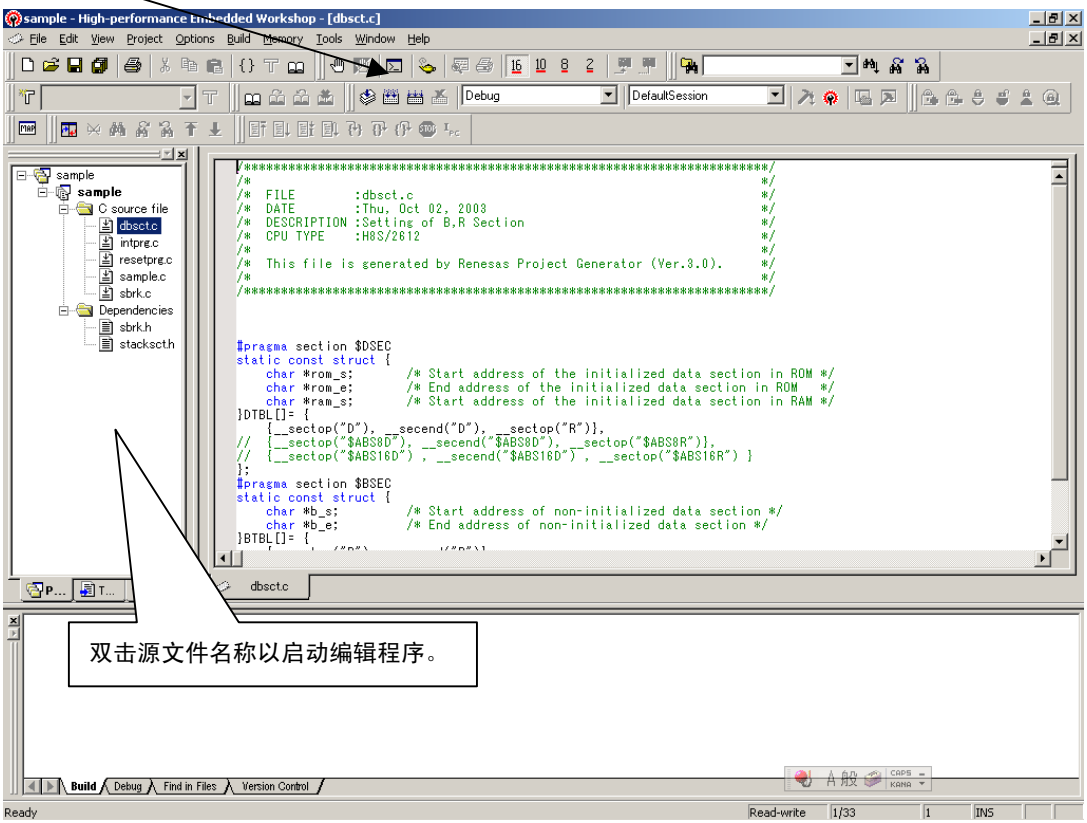
按下 [添加 (Add) …] 按钮将显示一个对话框，可在对话框上如下所示指定 ROM 和 RAM 区域。



(14) 执行创建过程

执行创建过程以生成装入模块。

在这里按下命令按钮可执行创建。

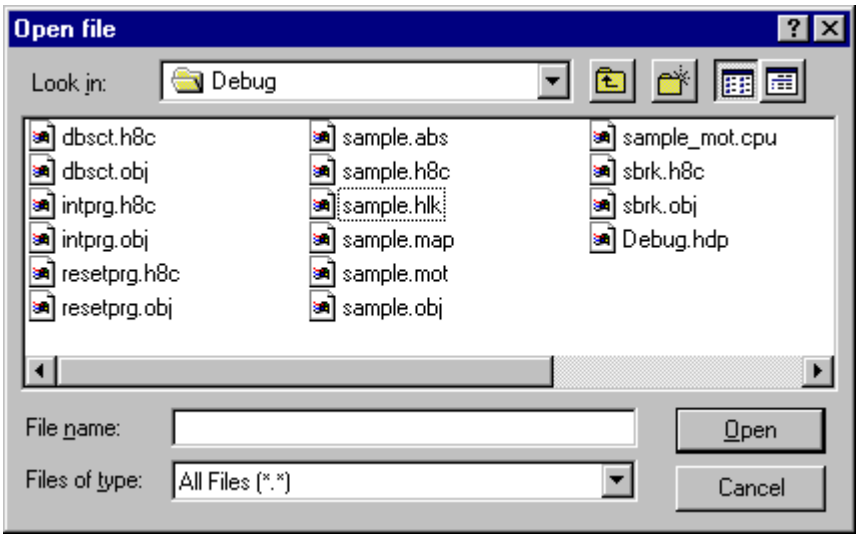


(15) 验证生成的文件

下列文件在创建过程完成后即被生成。

一个名称和工程名称相同的目录被创建于工程目录下。 绝对装入模块以 sample.abs 的名称格式生成于新目录的调试目录内。

创建过程期间所生成的映像文件被存储在相同的目录内，可通过单击 [文件 (File)→ 打开 (Open)] 来打开和检查这个文件。



映像文件以 sample.map 的名称生成。

2.1.3 从命令行启动工具

工具可从命令行启动，如下所示：

这个实例使用了 H8S/2600 高级模式的 CPU。

在 HEW1.2 内，样品程序在 HEW 目录
\Tools\HITACHI\H8\3_0a_0\sample 中提供。

编号	HEW1.2 文件	描述
1	init.c	初始化例程
2	vectbl.c	向量表设定
3	scttbl.c	段初始化例程
4	cmain.c	主要函数文件
5	2600a.cpu	CPU 信息文件
6	c2600a.sub	模块间优化器的子命令文件

HEW2.0 或以上的版本未提供样品程序。因此必须准备用户自制的样品程序或将创建样品工程时所要生成的下列文件用作样品程序。

根据 2.1.2 节，创建新的工作空间 2 (HEW 2.0)，选取演示 (Demonstration) 为工程类型设定，以创建样品工程。

编号	HEW2.0 或以上版本文件	描述
1	resetprg.c	初始化例程
2	intprg.c	向量表设定
3	dbsct.c	段初始化例程
4	main.c	主要函数文件
5	2600a. sub（用户自制）	子命令文件

(1) 设定所需的环境

- PC 版本
  设定路径 (path)=<HEW 安装目录 (HEW install directory)>\tools\hitachi\h8\v3_0a_0\bin;%path%
  设定 CH38=<HEW 安装目录 (HEW install directory)>\tools\hitachi\h8\v3_0a_0\include
  设定 hlnk_linrary1=<HEW 安装目录 (HEW install directory)>\tools\hitachi\h8\v3_0a_0\lib\c8s26a.lib
- unix 版本
  请参考 1.3 节，安装方法。

(2) 编译

编译 C 程序文件。

ch38Δ–cpu=2600aΔ–debugΔinit.cΔvectbl.cΔscttbl.c (RET)

ch38Δ–cpu=2600aΔ–debugΔ-show=allocation,objectΔ-goptimizeΔcmain.c (RET)

(3) 创建 CPU 信息文件。（地址范围只能为 HEW1.2 指定，而无法为 HEW2.0 或以上的版本指定。）

在 unix 版本内，启动 cia38 以指定所要使用的 ROM/RAM 地址范围。

要获取有关使用 cia38 的描述，请参考 H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 的附录 J，创建 CPU 信息文件 (Creating a CPU Information File)。

在 PC 版本内，您可以使用 HEW。 请参考 2.1.1 (10) 节，设定选项。

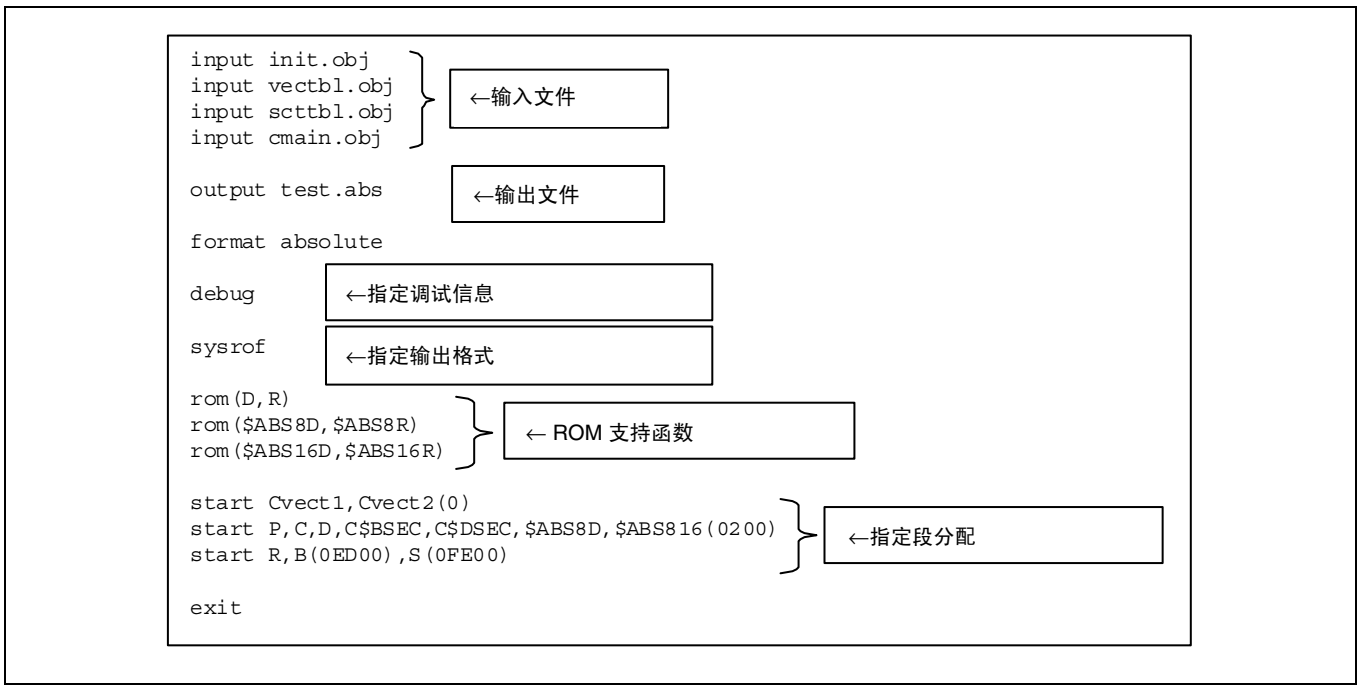
这个实例使用位于样品目录内的 CPU 信息文件 2600a.cpu。

创建模块间优化程序的子命令文件。

创建模块间优化程序所要指定的子命令文件。

这个实例使用位于样品目录内的 c2600a.sub 文件。（文件只能用于 HEW1.2。用户自制的文件应该根据下列子命令文件为 HEW2.0 创制。）

<c2600a.sub>（修改子命令文件）



使用下列子命令文件以执行模块间优化程序：

optlnk38Δ–sub=test.sub (RET)　(HEW1.2 命令行)

optlnkΔ–sub=test.sub (RET)　(HEW2.0 命令行)

优化程序输出装入模块文件 sample.abs；在 HEW1.2 中，它也输出存储器分配信息至连接列表 sample.map 及符号优化信息至优化信息列表 sample.lop。在 HEW2.0 中，它输出存储器分配信息和符号优化信息至连接列表 sample.map。

(4) 转换目标文件。

为了创建 ROM 程序，使用下列方法将目标装入模块（在这里为 SYSROF 类型）转换为 S 类型格式：

cnvsΔtest.abs (RET)　(HEW1.2 命令行)

由于优化连接编辑程序具有用于 HEW2.0 或以上版本的转换功能，因此 S 类型格式可在不需使用转换器的情况下进行转换。

在子命令文件中描述 form=stype 以输出 S 类型格式。

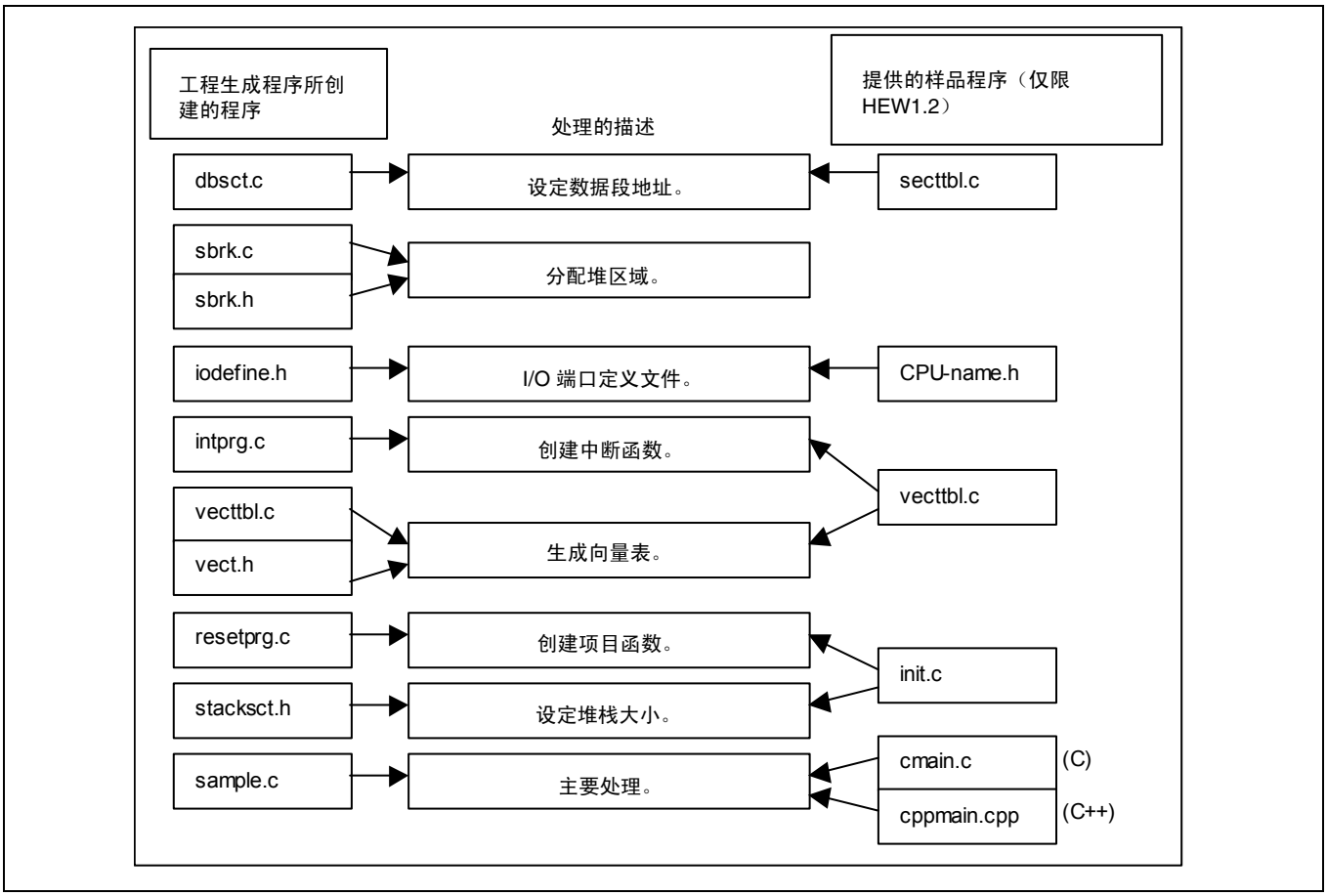
2.2 样品程序简介

2.2.1 ROM 程序所需的初始化

下列描述主要以由工程生成程序所创建的程序为例。

下列图示显示由工程生成程序所创建程序的文件组织以及由本产品所提供的样品程序。

（样品程序仅随 HEW1.2 提供，HEW2.0 或以上的版本未提供。由于 intprg.c 扩展函数的一个描述，样品程序 vecttbl.c 及 vect.h 不再由 HEW2.0 或以上版本生成。）



(1) 设定数据段地址

（HEW 工程文件名称： dbsct.c，样品程序名称： scttbl.c）

```
/************************************************************************/
/*                                                                      */
/*  文件         :dbsct.c                                               */
/*  日期         :1999 年 11 月 4 日，星期四                             */
/*  描述         :B, R 段的设定                                         */
/*  CPU 类型     :H8S/2621                                              */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.1）生成。                     */
/*                                                                      */
/************************************************************************/


#pragma section $DSEC
static const struct {
   char *rom_s;        /* ROM 内已初始化数据段的起始地址 */
   char *rom_e;        /* ROM 内已初始化数据段的终止地址 */
   char *ram_s;        /* RAM 内已初始化数据段的起始地址 */
}DTBL[]= {
   {__sectop("D"), __secend("D"), __sectop("R")},
   {__sectop("$ABS8D"), __secend("$ABS8D"), __sectop("$ABS8R")},
   {__sectop("$ABS16D") , __secend("$ABS16D") , __sectop("$ABS16R") }
                                                                        
};

#pragma section $BSEC
static const struct {
   char *b_s;          /* 未初始化数据段的起始地址 */
   char *b_e;          /* 未初始化数据段的终止地址 */
}BTBL[]= {
   {__sectop("B"), __secend("B")},
   {__sectop("$ABS8B"), __secend("$ABS8B")},
   {__sectop("$ABS16B"), __secend("$ABS16B")}
                                                                        
};
```

设定已初始化数据区域的段地址。

设定未初始化数据区域的段地址。

设定由将它们初始化的例程所使用的已初始化和未初始化数据段的地址。

要添加已初始化数据区段的名称，如前一行般将段名称添加到这里。

要添加未初始化数据区段的名称，如前一行般将段名称添加到这里。

_ _sectop 及 _ _secend 是用来决定段地址的增强函数。

这些函数将在 3.3 节，段地址运算符中加以说明。

(2) 分配堆区域

（HEW 工程文件名称： sbrk.c、sbrk.h，样品程序名称： sbrk.c、lowsrc.c、otherlb.c）
这些程序生成一项对存储器管理程序库所使用的堆区域的分配函数。

```
/*********************************************************************/
/*                                                                   */
/*  文件            :sbrk.c                                          */
/*  日期            :1999 年 11 月 4 日，星期四                      */
/*  描述            :sbrk 程序                                       */
/*  CPU 类型        :H8S/2621                                        */
/*                                                                   */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                  */
/*                                                                   */
/*********************************************************************/
#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=              /* 指定定义的堆区域            */
                                        /* 的最小单位                  */
extern char *_s1ptr;
extern void srand(unsigned int);

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（通过）                           */
/*                    -1                （失败）                     */
/*********************************************************************/
char  *sbrk(unsigned long size)         /* 分配的区域大小              */
{
      char  *p;

      if(brk+size>heap_area.heap+HEAPSIZE)   /* 空的区域大小          */
      return (char *)-1 ;

      p=brk ;                           /* 区域赋值                    */
      brk += size ;                     /* 终止地址更新                */
      return p ;
}

/*********************************************************************/
/* _INIT_OTHERLIB                                                    */
/* 需要时，初始化 C 程序库函数。                                     */
/* 在汇编程序选项 (Assembler Option) 上定义 OTHERLIB。               */
/*********************************************************************/

void _INIT_OTHERLIB(void)
{
   srand(1);
   _s1ptr=NULL;
}
```

有关创建低层界面例程的方法描述，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户指南 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.2.2 节，执行环境设定 (Execution Environment Settings)。

（HEW 工程文件名称： sbrk.h）

```
/************************************************************************/
/*                                                                      */
/*  文件            :sbrk.h                                             */
/*  日期            :1999 年 11 月 4 日，星期四                         */
/*  描述            :sbrk 文件的标题文件                                */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/
/*由 sbrk 管理的区域大小*/
#define HEAPSIZE 0x400
```

这是用于低层例程定义的包含文件，sbrk。 包含文件显示堆区域的大小。 要在工程被指定后更改堆区域的大小，请修改这项值。

实例：将堆区域的大小更改为 514 (0x202) 字节：

```
#define HEAPSIZE 0x202
```

(3) 定义 I/O 端口文件

（HEW 工程文件名称： iodefine.h，样品程序名称： <CPU 名称>.h)
定义了 I/O 端口文件以便 I/O 端口可使用变量名称存取。

```
/************************************************************************/
/*                                                                      */
/*  文件            :iodefine.h                                         */
/*  日期            :1999 年 11 月 4 日，星期四                         */
/*  描述            :I/O 寄存器的定义                                   */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/


/************************************************************************/
/*      H8S/2623 系列包含文件                    版本 1.1               */
/************************************************************************/
struct st_hcan {                                    /*    struct HACN  */
               union {                              /*    MCR          */
                     unsigned char BYTE;            /*    字节存取     */
                     struct {                       /*    位存取       */
                            unsigned char SLPME:1;  /*    SLPME        */
                            unsigned char      :1;  /*                 */
                            unsigned char SLPM :1;  /*    SLPM         */
                            unsigned char      :2;  /*                 */
                            unsigned char MSM  :1;  /*    MSM          */
                            unsigned char HALT :1;  /*    HALT         */
                            unsigned char RST  :1;  /*    RST          */
                     }             BIT;             /*                 */
               }                MCR;                /*                 */
               union {                              /*    GSR          */
                     unsigned char BYTE;            /*    字节存取     */
                     struct {                       /*    位存取       */
                            unsigned char wk  :4;   /*                 */
                            unsigned char RSF :1;   /*    RSF          */
                            unsigned char MSEF:1;   /*    MSEF         */
                            unsigned char SRWF:1;   /*    SRWF         */
                            unsigned char BOF :1;   /*    BOF          */
                     }             BIT;             /*                 */
               }                GSR;                /*                 */
                                (omitted)

#define HCAN    (*(volatile struct st_hcan  *)0xFFF800) /* HCAN  地址*/
#define SCRX    (*(volatile union  un_scrx  *)0xFFFDB4) /* SCRX  地址*/
#define SBYCR   (*(volatile union  un_sbycr *)0xFFFDE4) /* SBYCR 地址*/
#define SYSCR   (*(volatile union  un_syscr *)0xFFFDE5) /* SYSCR 地址*/
#define SCKCR   (*(volatile union  un_sckcr *)0xFFFDE6) /* SCKCR 地址*/

                                (cont)
```

当不使用工程生成程序时，在随产品提供的样品中定位命名与 CPU 相同的包含文件和 C 源文件。 在小心检查其为正确文件后使用这些文件。

(4) 创建中断 (Interrupt) 函数

（HEW 工程文件名称： intprg.c，样品程序名称： vecttbl.c）
这些程序将定义产生中断调用的函数。

<HEW1.2>

```
/************************************************************************/
/*                                                                      */
/*  文件            :intprg.c                                           */
/*  日期            :1999 年 11 月 4 日，星期四                         */
/*  描述            :中断程序                                           */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/


#include    <machine.h>
#include    "vect.h"
#pragma section IntPRG
//  向量 2 已预留

//  向量 3 已预留

//  向量 4 已预留

//  向量 5 跟踪
void    INT_Treace(void) {/* sleep(); */}
//  向量 6 已预留

//  向量 7 NMI
void    INT_NMI(void) {/* sleep(); */}
//  向量 8 用户断点陷阱
void    INT_TRAP1(void) {/* sleep(); */}
//  向量 9 用户断点陷阱
void    INT_TRAP2(void) {/* sleep(); */}
//  向量 10 用户断点陷阱
void    INT_TRAP3(void) {/* sleep(); */}
//  向量 11 用户断点陷阱
void    INT_TRAP4(void) {/* sleep(); */}
//  向量 12 已预留

//  向量 13 已预留

//  向量 14 已预留
```

←定义段名称

↓定义中断功能

注意： 如果指定了 #pragma section IntPRG，函数将被分配给命名为 PIntPRG 的段。在更改由模块间优化器所创建的段名称前必须警惕。

<HEW2.0 或以上版本>

```
/************************************************************************/
/*                                                                      */
/*  文件            :intprg.c                                           */
/*  日期            :2002 年 8 月 20 日，星期二                         */
/*  描述            :中断程序                                           */
/*  CPU 类型        :H8S/2612                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/


#include    <machine.h>
#pragma section IntPRG
//  向量 2 已预留

//  向量 3 已预留

//  向量 4 已预留

//  向量 5 跟踪
__interrupt(vect=5) void INT_Trace(void) {/* sleep(); */}
//  向量 6 已预留

//  向量 7 NMI
__interrupt(vect=7) void INT_NMI(void)    {/* sleep(); */}
//  向量 8 用户断点陷阱
__interrupt(vect=8) void INT_TRAP0(void)  {/* sleep(); */}
//  向量 9 用户断点陷阱
__interrupt(vect=9) void INT_TRAP1(void)  {/* sleep(); */}
//  向量 10 用户断点陷阱
__interrupt(vect=10) void INT_TRAP2(void) {/* sleep(); */}
//  向量 11 用户断点陷阱
__interrupt(vect=11) void INT_TRAP3(void) {/* sleep(); */}
//  向量 12 已预留

//  向量 13 已预留

//  向量 14 已预留

//  向量 15 已预留
```

_interrupt(vect=5) 的描述自动生成向量表。

要获取中断函数的详细资料，请参考 3.1 节，指定中断函数。

(5) 创建向量表

（HEW 工程文件名称： vecttbl.c，样品程序名称： vecttbl.c）

这些程序将设定向量表内中断函数的地址。（仅在 HEW1.2 生成）

```
/************************************************************************/
/*                                                                      */
/*  文件            :vecttbl.c                                          */
/*  日期            :1999 年 11 月 4 日，星期四                         */
/*  描述            :向量表的初始化                                     */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/


#include    "vect.h"

#pragma section VECTTBL
void *RESET_Vectors[] = {
//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 加电复位
   PowerON_Reset,
//;<<VECTOR DATA END (POWER ON RESET)>>
//;<<VECTOR DATA START (MANUAL RESET)>>
//;1 手动复位
   Manual_Reset
//;<<VECTOR DATA END (MANUAL RESET)>>
};
#pragma section INTTBL
void *INT_Vectors[] = {
   // 2 已预留
   (void *) Dummy,
   // 3 已预留
   (void *) Dummy,
   // 4 已预留
   (void *) Dummy,
   // 5 跟踪
   (void *) INT_Treace,
   // 6 已预留
   (void *) Dummy,
   // 7 NMI
   (void *) INT_NMI,
   // 8 用户断点陷阱
   (void *) INT_TRAP1,
   // 9 用户断点陷阱
   (void *) INT_TRAP2,
   // 10 用户断点陷阱
   (void *) INT_TRAP3,
   // 11 用户断点陷阱
   (void *) INT_TRAP4,
   // 12 已预留
   (void *) Dummy,
   // 13 已预留
   (void *) Dummy,
```

在 CVECTTBL 段内创建命名为 RESET_Vectors 的向量表。

在 INTTBL 段内创建命名为 INT_Vectors 的向量表。

（待续）

注意： 在 #pragma section 内指定段名称将使名称被附加到默认段名称。 因此，当使用模块间优化器分配地址时，您需要更改段名称。

(6) vect.h

这项程序声明当设定了向量表时，所参考的内建函数的原型。

（仅在 HEW1.2 生成）

```
; /************************************************************************/
/*                                                                      */
/*  文件            :vect.h                                             */
/*  日期            :1999 年 11 月 4 日，星期四                          */
/*  描述            :向量的定义                                         */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/




//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 加电复位
extern void PowerON_Reset(void);
//;<<VECTOR DATA END (POWER ON RESET)>>
//;<<VECTOR DATA START (MANUAL RESET)>>
//;1 手动复位
extern void Manual_Reset(void);
//;<<VECTOR DATA END (MANUAL RESET)>>
// 2 已预留

// 3 已预留

// 4 已预留

// 5 跟踪
#pragma interrupt INT_Treace
extern void INT_Treace(void);
// 6 已预留

// 7 NMI
#pragma interrupt INT_NMI
extern void INT_NMI(void);
// 8 用户断点陷阱
#pragma interrupt INT_TRAP1
extern void INT_TRAP1(void);
// 9 用户断点陷阱
#pragma interrupt INT_TRAP2
extern void INT_TRAP2(void);
// 10 用户断点陷阱
#pragma interrupt INT_TRAP3
extern void INT_TRAP3(void);
// 11 用户断点陷阱
#pragma interrupt INT_TRAP4
extern void INT_TRAP4(void);
// 12 已预留

// 13 已预留
```

<-通过指定 #pragma interrupt 为中断函数，RTE 指令在返回函数值时被生成。
要获取中断函数的详细资料，请参考 3.1 节，指定中断函数。

（待续）

(7) 创建项目函数

（HEW 工程文件名称： resetprg.c，样品程序名称： init.c）

```
/************************************************************************/
/*                                                                      */
/*  文件            :resetprg.c                                         */
/*  日期            :1999 年 11 月 4 日，星期四                         */
/*  描述            :复位程序                                           */
/*  CPU 类型        :H8S/2621                                           */
/*                                                                      */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                     */
/*                                                                      */
/************************************************************************/


#include <machine.h>
#include "stacksct.h"

#pragma entry PowerON_Reset

extern void main(void);

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

//#ifdef __cplusplus          // 在您使用 SIM I/O 时移除注解
//extern "C" {
//#endif
//extern void _INIT_IOLIB(void);
//extern void _CLOSEALL(void);
//#ifdef __cplusplus
//}
//#endif

//extern void srand(unsigned int);  // 在您使用 rand() 时移除注解
//extern char *_s1ptr;              // 在您使用 strtok() 时移除注解
```

包括嵌入式函数包含文件

指定 PowerON_Reset 为项目函数。
编译程序输出用以初始化 SP 至项目函数的代码。

（待续）

```
                                （继前一页）

//#ifdef __cplusplus        // 在您使用硬件设置 (Hardware Setup) 时移除注解
//extern "C" {
//#endif
//extern void HardwareSetup(void);
//#ifdef __cplusplus
//}
//#endif


#pragma section ResetPRG

void PowerON_Reset(void);                    将 CCR 中断标志设定为被允许
void PowerON_Reset(void)
{
    set_imask_ccr(1);                        调用段初始化例程

   _INITSCT();

// _INIT_IOLIB();           // 在您使用 SIM I/O 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();         // 在您使用硬件设置 (Hardware Setup) 时移除注解

   main();                                   调用主要函数

// _CLOSEALL();             // 在您使用 SIM I/O 时移除注解

   sleep();                                  进入低功耗模式
}

void Manual_Reset(void);
void Manual_Reset(void)
{
}
```

(8) 设定堆栈大小

（HEW 工程文件名称： stacksct.h）

```
/*********************************************************************/
/*                                                                   */
/*  文件            :stacksct.h                                      */
/*  日期            :1999 年 11 月 4 日，星期四                      */
/*  描述            :设定堆栈区域                                    */
/*  CPU 类型        :H8S/2621                                        */
/*                                                                   */
/*  本文件由 Renesas 工程生成程序（版本.3.0）生成。                  */
/*                                                                   */
/*********************************************************************/
#pragma stacksize 0x200
```

指定所需的堆栈大小。这项规格将创建 512 字节的堆栈段，并以 S 为固定名称。

堆栈段的大小相等于函数调用关系内最深的嵌套级的堆栈大小。

通过参考目标列表分配信息中所输出的总帧大小 (Total Frame Size) 来计算堆栈大小。

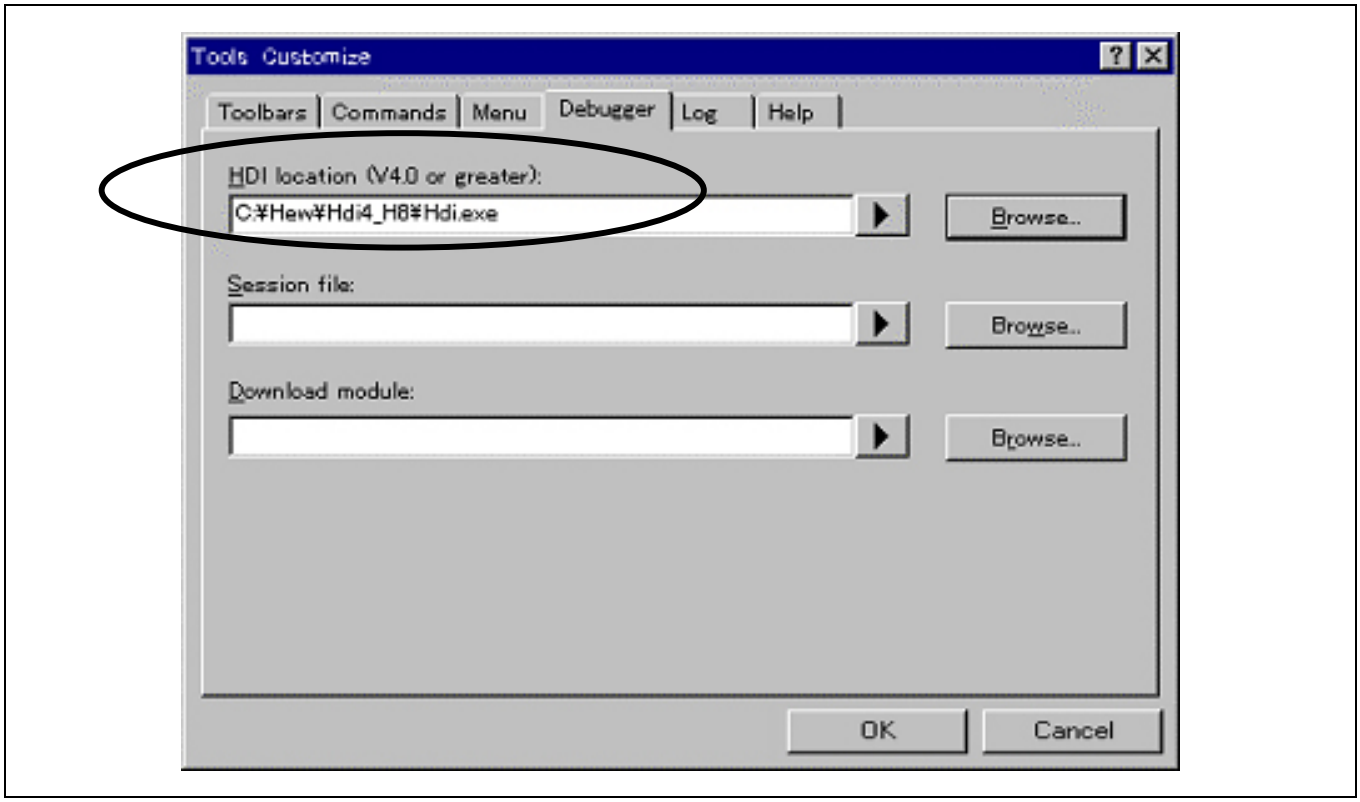
若要更改堆栈大小的规格，请修改本程序中的值。

2.3 使用 HDI 进行调试

让我们用新建的 HEW 工作空间来执行使用 HDI 进行的调试。（HDI 可在 HEW1.2 及 HEW2.0 或以上的版本中操作。）

2.3.1 使用 HEW 运行（1）

在 HEW 菜单内的工具 (Tools) 上选择自定义 (Customize) …，以打开工具自定义 (Tools Customize) 对话框，并在 HDI 位置 (HDI location) 字段中指定 HDI.exe 的位置。 然后，即可在 HEW 菜单上按下启动调试程序 (Launch Debugger) 按钮来启动 HDI。



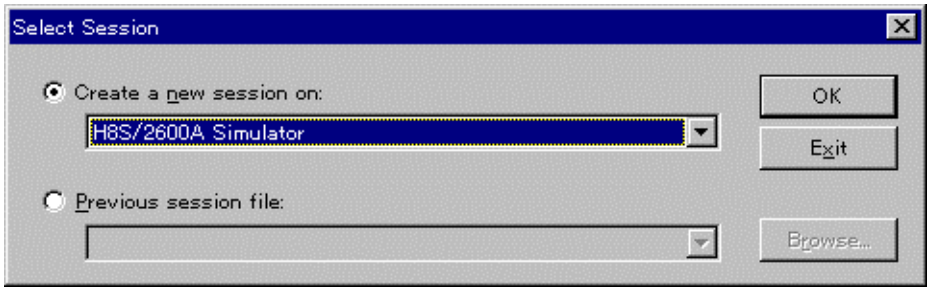
启动调试程序

2.3.2 选取目标

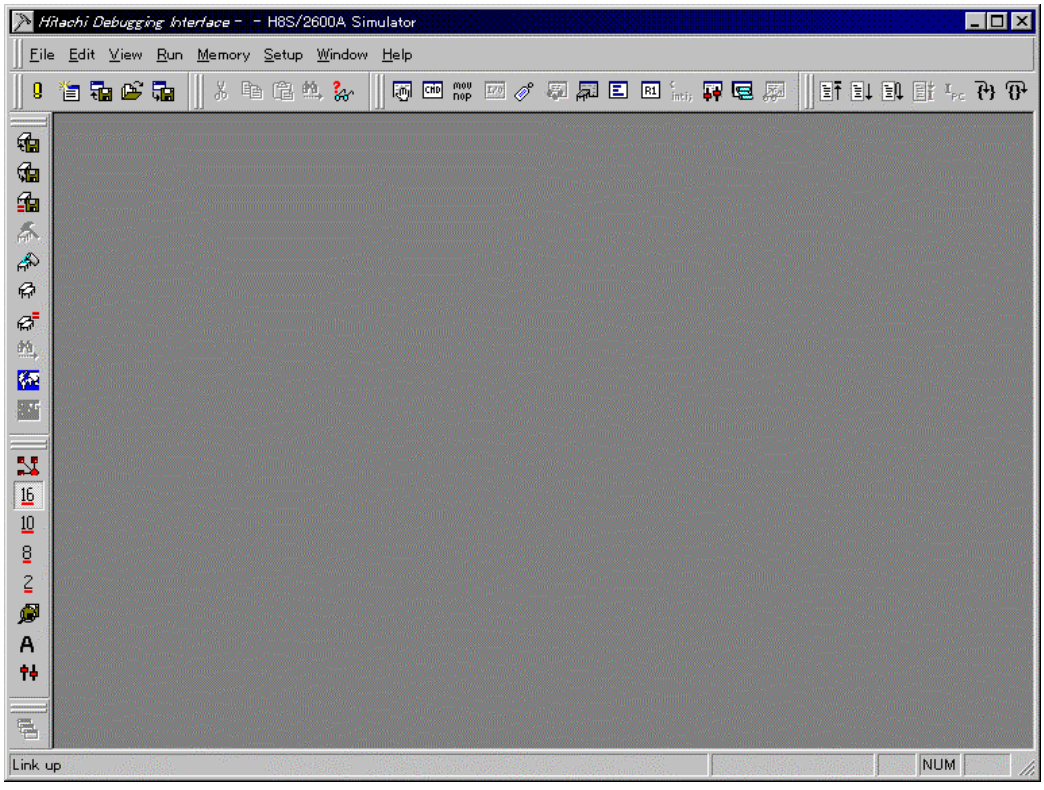
在以下的画面上，选取所需的 CPU 类型和调试程序类型。

这个实例，在之前选取的 H8S/2600 高级模式下，选取了 H8S/2600A 模拟程序。

选定了目标之后，单击 [确定 (OK)]。



在显示了启动窗口之后，将打开 Hitachi 调试界面 (Hitachi Debugging Interface) 窗口：

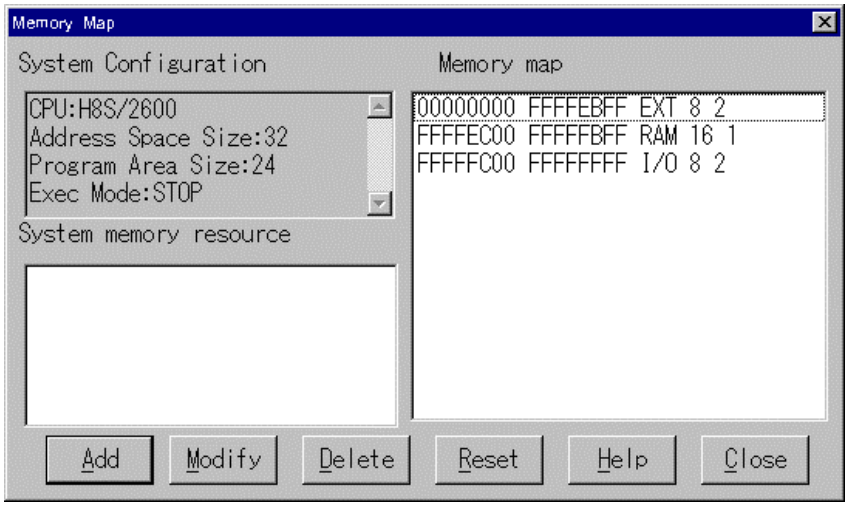


2.3.3 分配存储器资源

在下一个步骤中，分配操作装入模块所需的存储器资源。
可从 [视图 (View)] 菜单选取 [存储器映像窗口 (Memory Mapping Window)] 或在工具栏 (Toolbar) 上单击存储器映像 (Memory Mapping) 按钮：


存储器映像

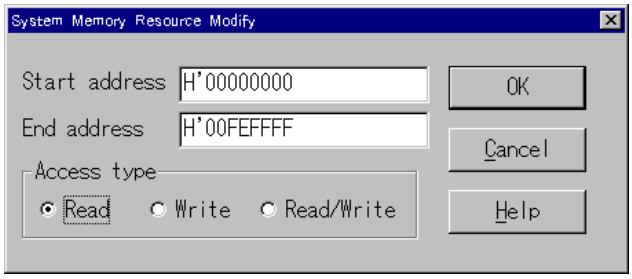
这将显示存储器映像 (Memory Map) 对话框：

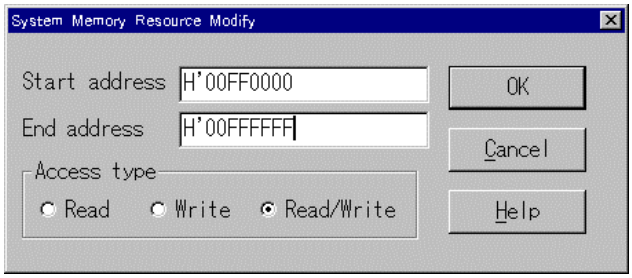


按下 [添加 (Add)] 按钮以在系统存储器资源修改 (System Memory Resource Modify) 画面上分配存储器资源。

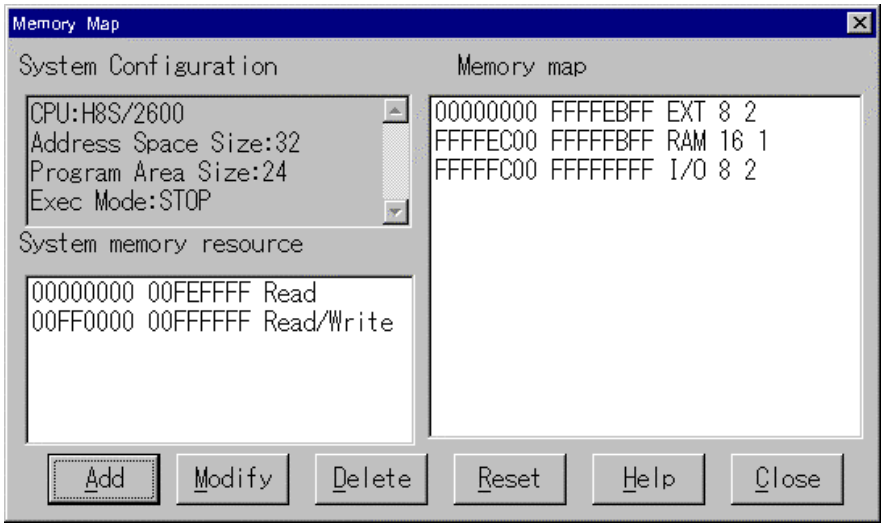
在这里，指定所有区域。对于 ROM 区域，指定地址 H’ 0 至 H’ 00FEFFFF 的存储器区域，对于 RAM 区域，指定 H’00FF0000 至 H’ 00FFFFFF。 将 ROM 区域和 RAM 区域的存取类型，分别指定为读取 (Read) 和读/写 (Read/Write)。

单击 [确定 (OK)] 按钮。


System Memory Resource Modify
Start address
H'00000000
End address
H'00FEFFFF
Access type
Read
Write
Read/Write
OK
Cancel
Help


System Memory Resource Modify
Start address
H'00FF0000
End address
H'00FFFFFF
Access type
Read
Write
Read/Write
OK
Cancel
Help

指定的存储器资源如上所示。



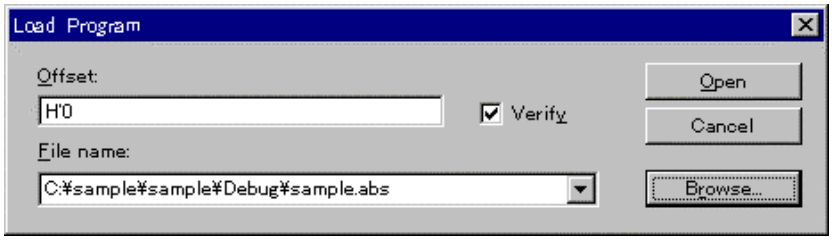
按下 [关闭 (Close)] 按钮以关闭这个窗口。

2.3.4 下载装入模块

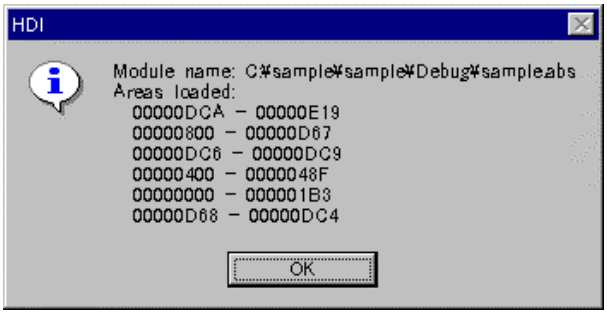
从 [文件 (File)] 菜单选取 [装入程序 (Load Program)]。选取所要调试的绝对装入模块 (absolute load module)。若使用按钮，单击工具栏 (Toolbar) 上的装入程序 (Load Program) 按钮。


装入程序

选取 sample.abs 文件，然后单击 [打开 (Open)]。

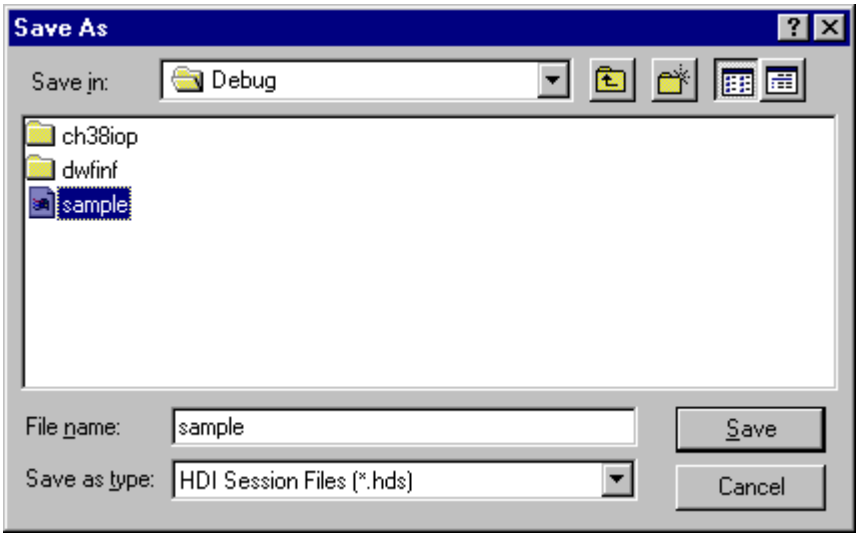


将显示下列画面：
文件已被装入。 画面显示写入了程序代码的存储器区域的信息。

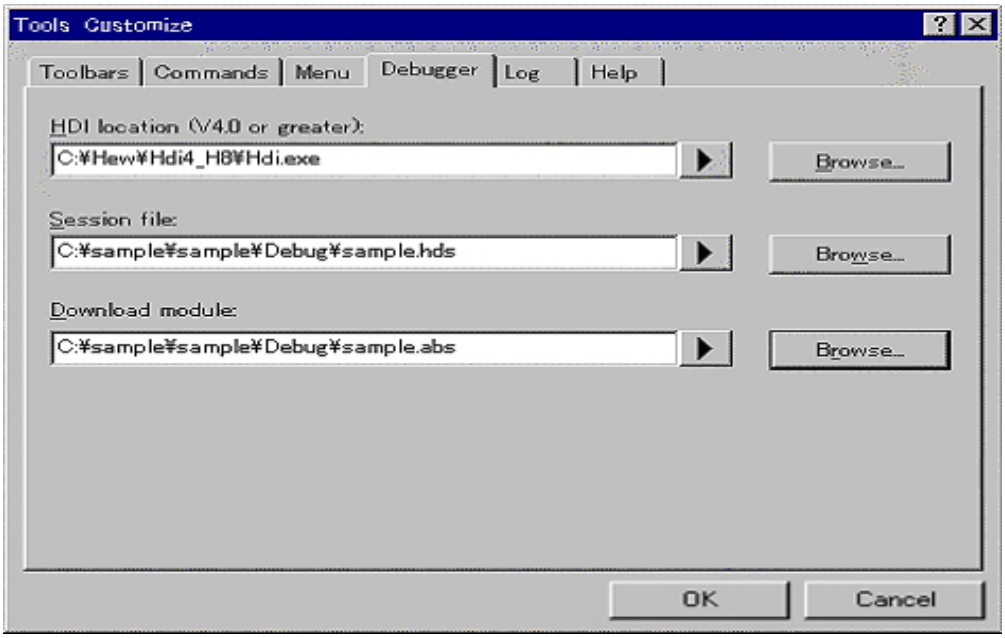


2.3.5 使用 HEW 操作 HDI（2）

从文件 (File) 菜单选取会话另存为 (Save Session As) …。

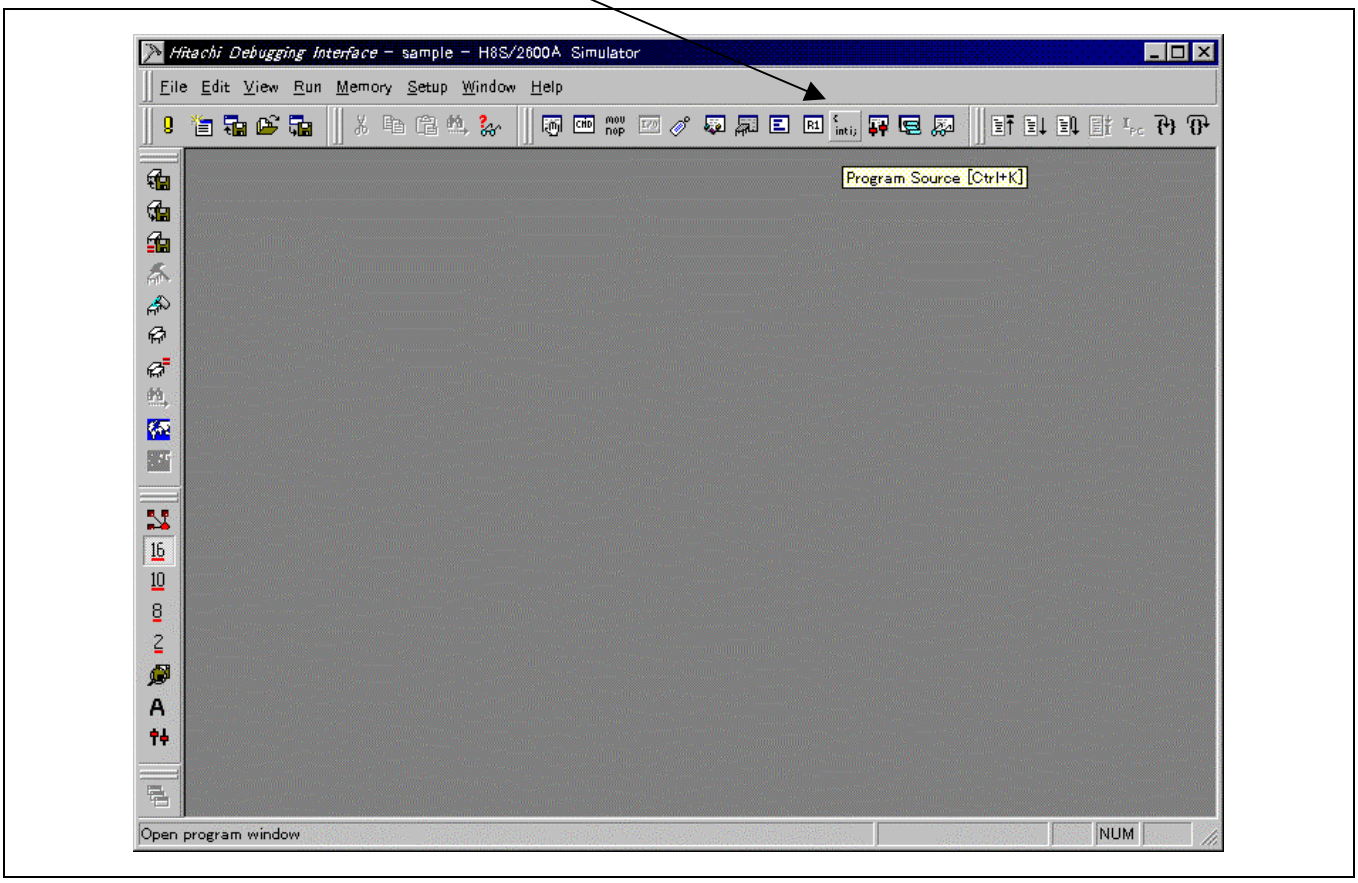


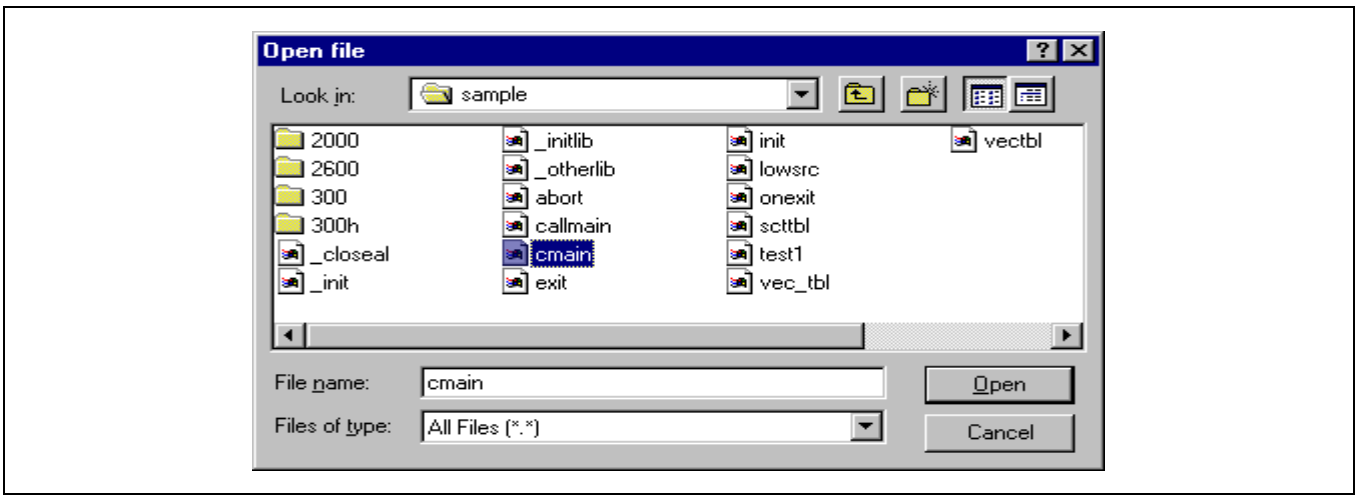
从 HEW 菜单上的工具 (Tools) 选择自定义 (Customize) … 将打开工具自定义 (Tools Customize) 对话框。在这个对话框内，在会话文件 (Session file) 字段内指定会话文件名称及在下载模块 (Download module) 字段内指定装入模块名称。然后，即可在 HEW 菜单上按下启动调试程序 (Launch Debugger) 按钮以在启动 HDI 后装入会话。


Tools Customize
Toolbars
Commands
Menu
Debugger
Log
Help
HDI location (V4.0 or greater):
C:¥Hew¥Hdi4_H8¥Hdi.exe
Browse...
Session file:
C:¥sample¥sample¥Debug¥sample.hds
Browse...
Download module:
C:¥sample¥sample¥Debug¥sample.abs
Browse...
OK
Cancel

2.3.6 显示源程序

单击程序源（Program Source）按钮。



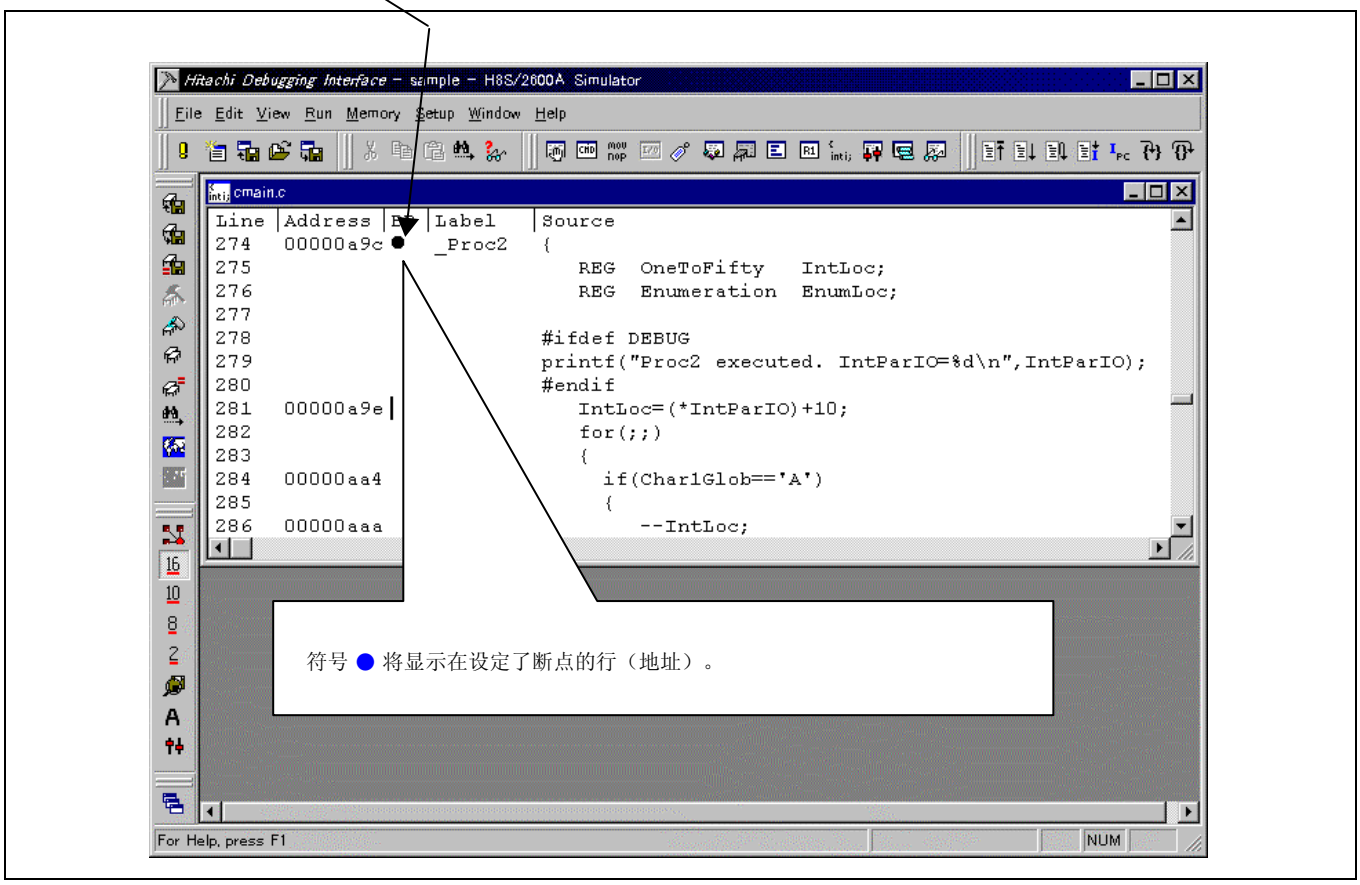


选取 cmain.c 文件。

2.3.7 设定断点

在程序窗口内的断点 (BP) 列上，在要设定断点的源行上双击。

例如，双击这里以在启动了主要函数时设定断点。

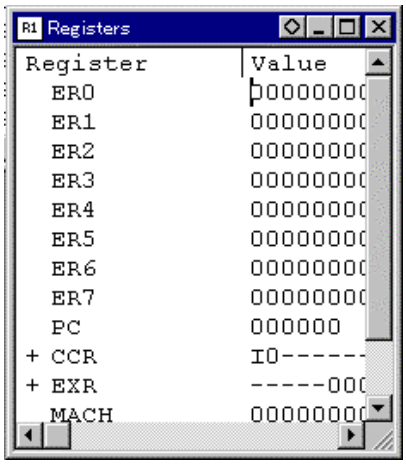


2.3.8 显示寄存器状态

可从 [视图 (View)] 菜单选取 [寄存器 (Registers)] 或在工具栏 (Toolbar) 上单击 CPU 寄存器 (CPU Registers) 按钮。

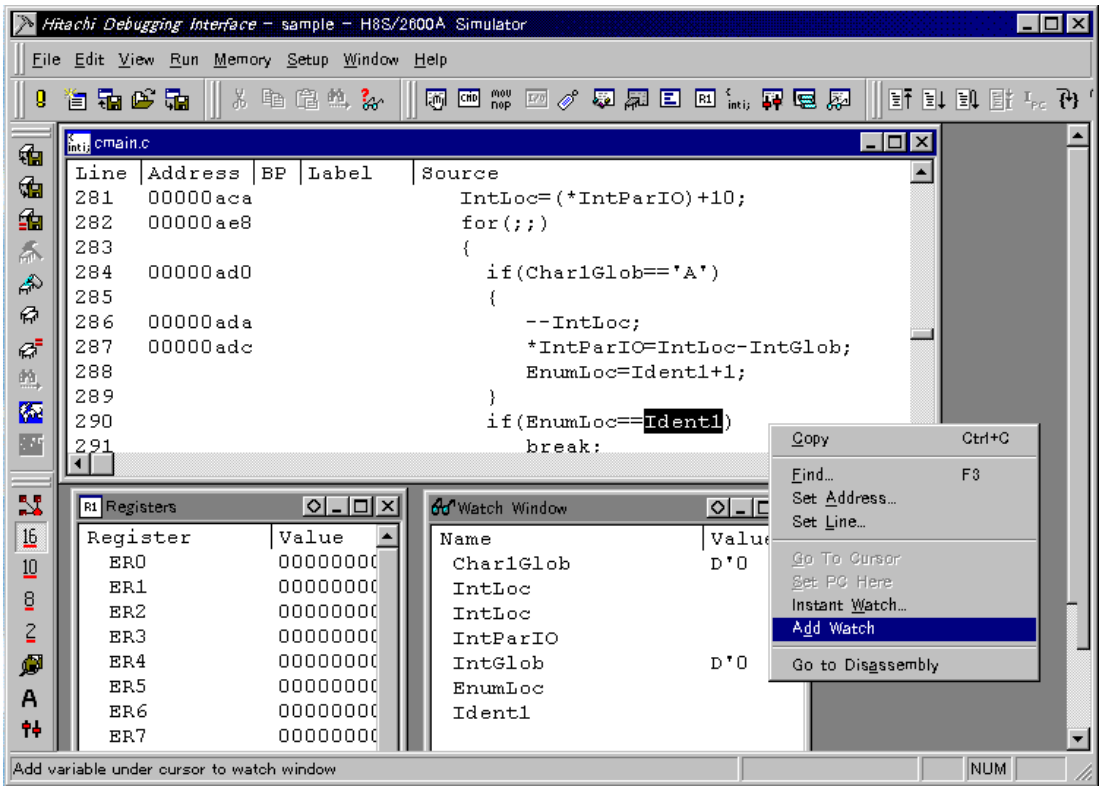
通过从 [视图 (View)] 菜单打开寄存器窗口 (Register Window)，您可以查阅寄存器的状态。

CPU 寄存器



2.3.9 参考外部变量

选取所要参考的变量名称。 单击鼠标右键以从快捷菜单选取添加监视 (Add Watch)。 在监视窗口 (Watch Window) 上，您可以参考变量值。或者，您可以将鼠标光标置于变量上方以显示变量值。



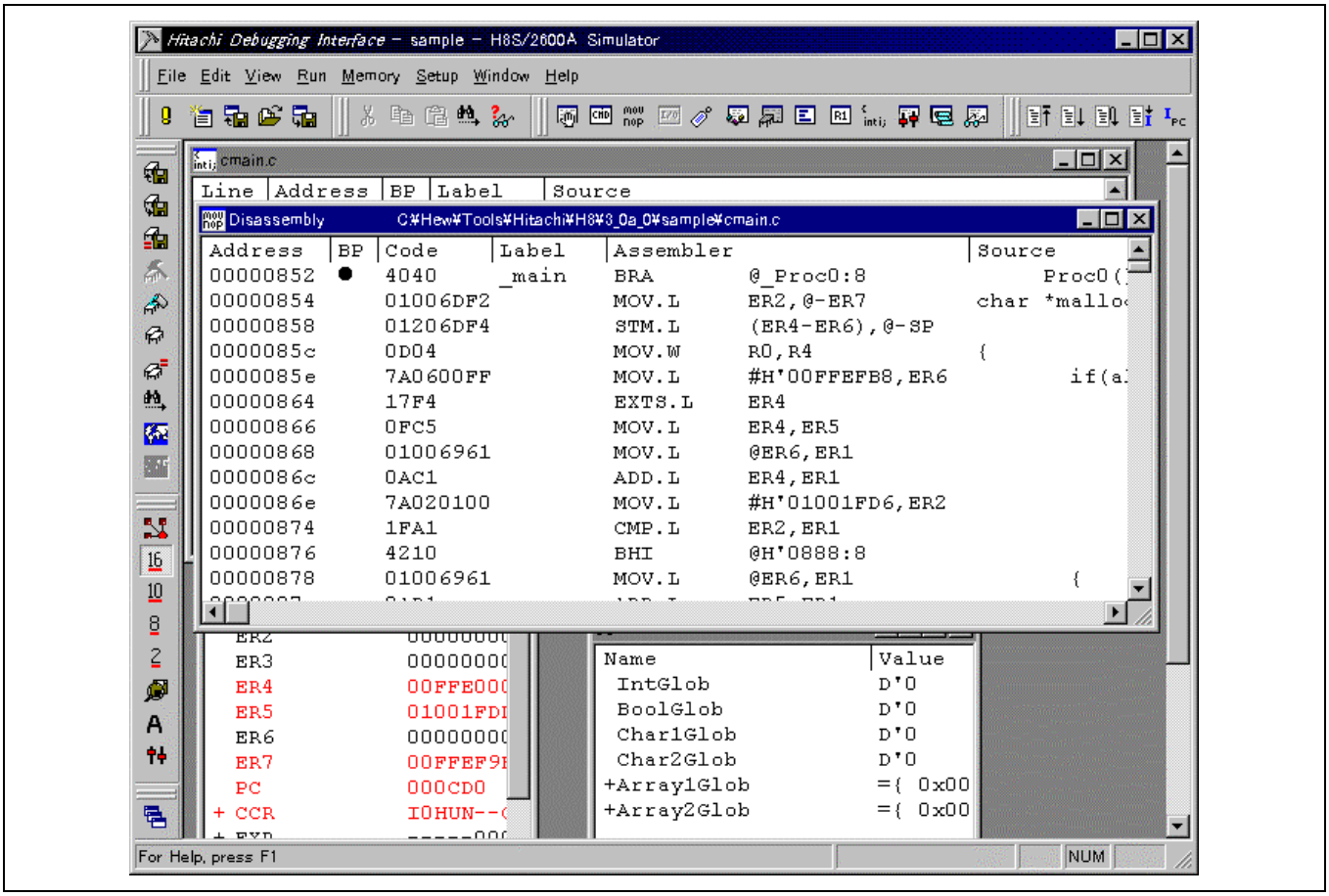
做好这些准备后，尝试执行程序。

2.3.10 ResetGo 命令

从 [运行 (Run)] 菜单选取 ResetGo 将使系统执行程序直到 PC 达至断点。

ResetGo

在 C 源程序上，单击右键以显示快捷菜单并选取前进到反汇编 (Go to Disassembly) 以显示反汇编 (Disassembly) 窗口。 反汇编窗口最右边的列为源 (Source) 列，和已反汇编代码有关的 C 源程序在此显示。



2.3.11 参考局部变量

从 [视图 (View)] 菜单选取 [局部 (Locals)] 将使系统显示局部 (Locals) 窗口，可从当前 PC 位置参考的局部变量及它们的值在此显示。

按下逐步 (Step) 按钮将允许用户进入该函数。 下一节将描述程序的逐步执行。

2.3.12 程序的逐步执行

现在让我们使用 [运行 (Run)] 菜单的跳入 (Step In)、跳过 (Step Over)，及跳出 (Step Out) 来逐步执行程序。

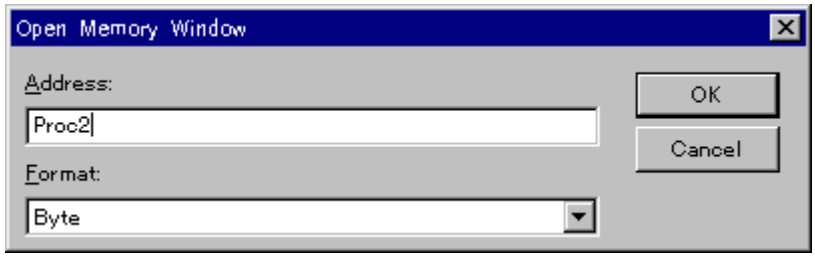
在子例程调用的情形下，跳入 (Step In) 将把 PC 移入子例程。
跳过 (Step Over) 将把 PC 从一个子例程调用行移到另一个。
跳出 (Step Out) 将把 PC 从一个子例程调用行移到下一行。

从 [运行 (Run)] 菜单选取 [运行 (Run) …] 将使系统打开运行 (Run) 对话框，用户可在对话框内更改逐步单元。

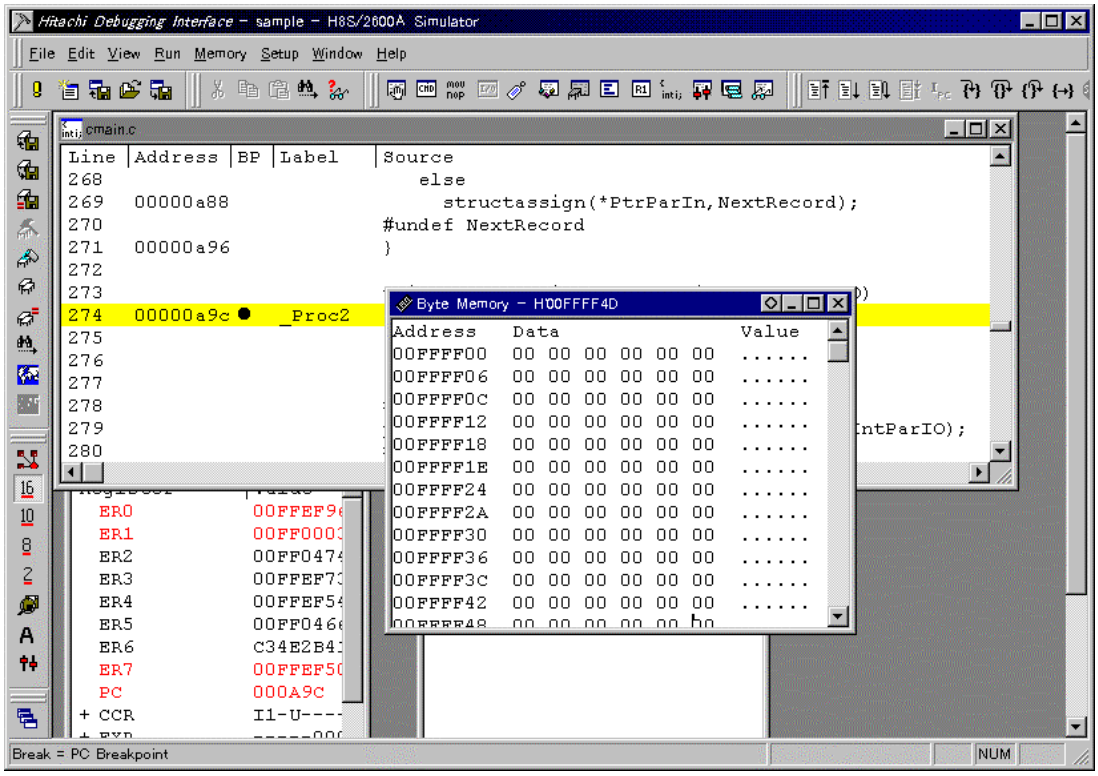
在这个实例中，一步对应于 C 源程序的一行。

2.3.13 显示存储器内容

指定 [视图 (View) → 存储器 (Memory) …] 将显示打开存储器窗口 (Open Memory Window) 对话框。 在地址 (Address) 字段内输入符号名称。

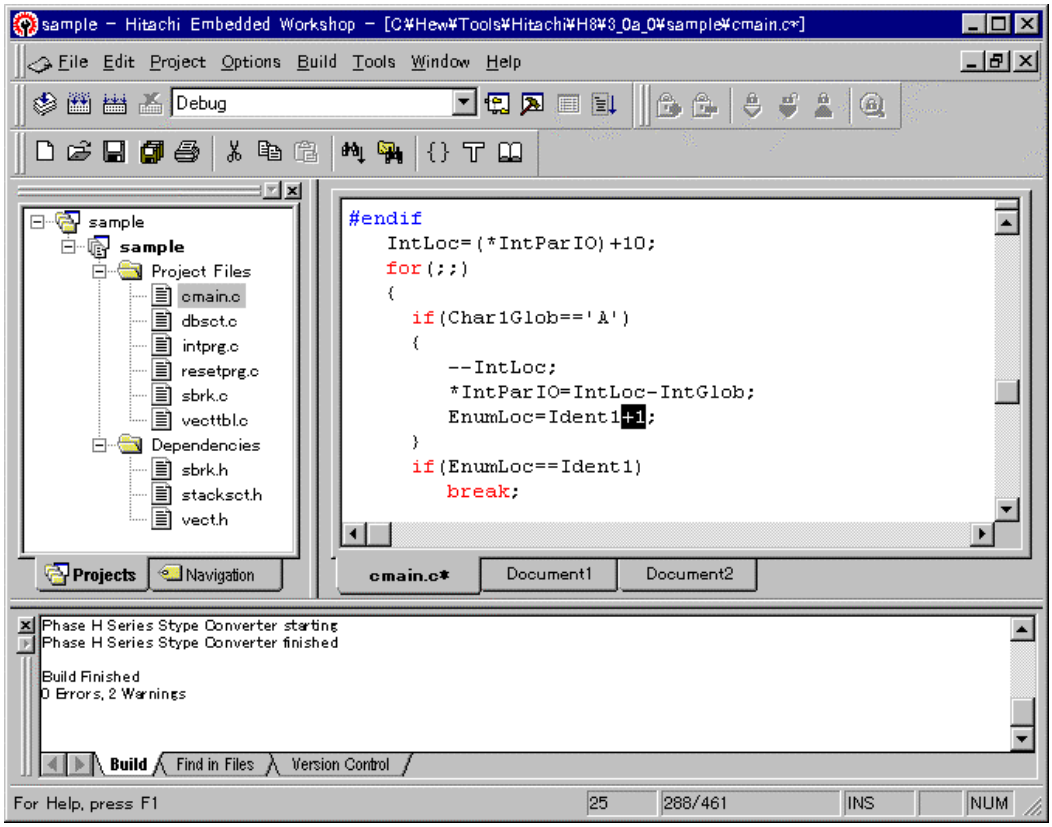


存储器的内容可在下列字节存储器 (Byte Memory) 的画面中显示：



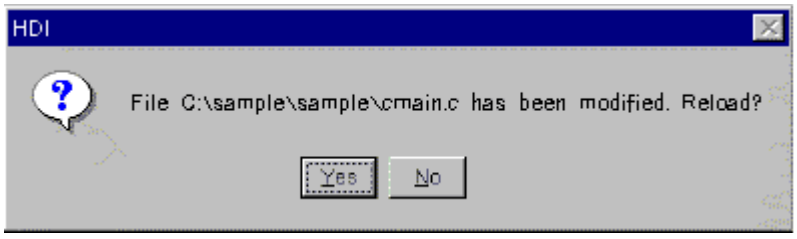
2.3.14 使用 HEW 操作 HDI（3）

要从 HEW 启动 HDI，双击 HDI 源 (HDI source) 窗口以在 HEW 编辑程序上打开所需的文件。



编辑并保存这个文件以将它重新编译。（注意样品程序将不能被修改，因为它们是只读文件。）

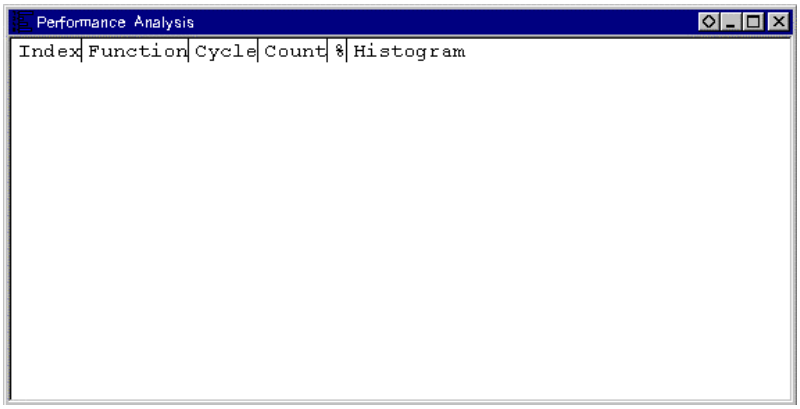
启动后，HDI 将显示信息对话框并询问是否重新装入程序。



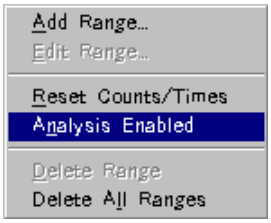
选取是 (Yes) 将使 HDI 重新装入程序。

调试程序可以这种形式执行。

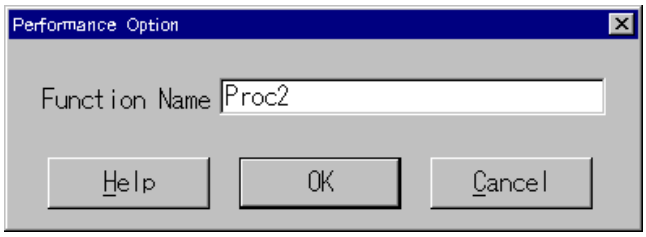
HDI 也提供性能分析 (performance analysis) 的函数。 若要衡量程序的性能，从 [视图 (View)] 菜单选取性能分析 (Performance Analysis)，将打开性能分析 (Performance Analysis) 窗口。

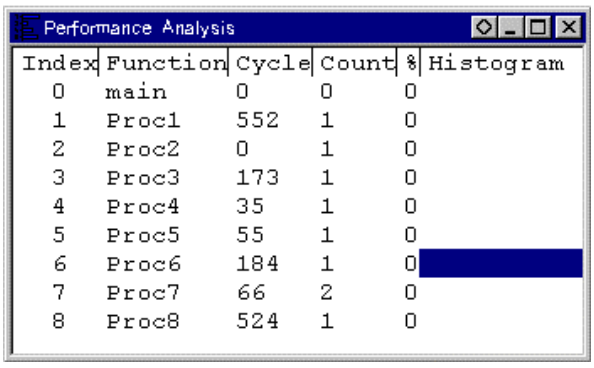


若要衡量性能，在弹出窗口上选取允许分析 (Analysis Enabled)：



在快捷菜单的添加范围 (Add Range) 选项上，指定要衡量性能的标签。





在执行程序后，每个标签的性能将显示。

要获取 HDI 功能的详细资料，请参考“Hitachi 调试界面用户手册 (Hitachi Debugging Interface User’s Manual)”。

2.4 使用模拟程序调试程序进行调试

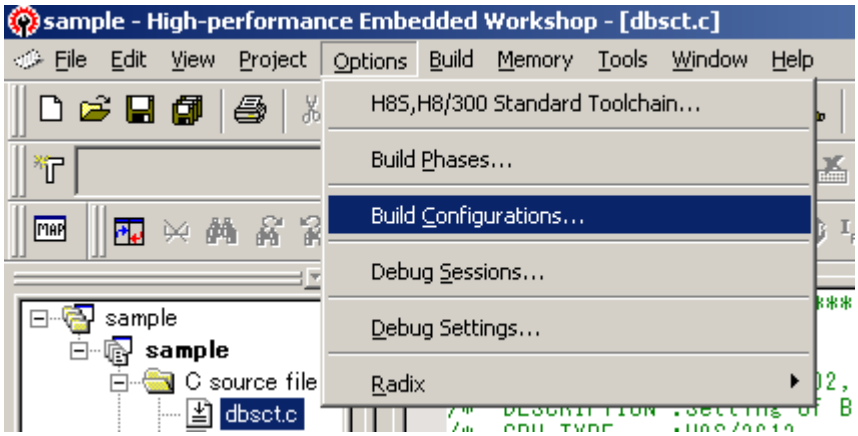
HEW 从 HEW2.0 开始允许调试。（注意这在 HEW1.2 并不可用。）

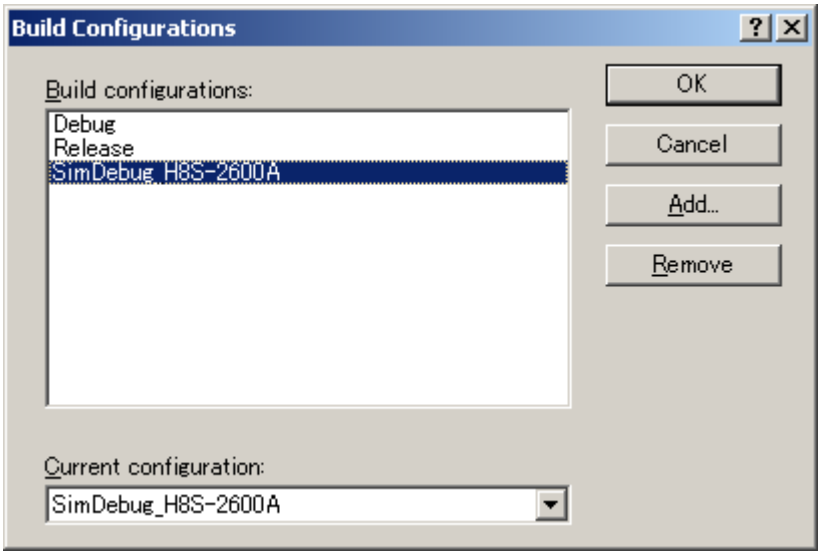
使用由选取演示 (Demostration) 为工程类型设定所创建的样品工程来执行模拟程序调试程序。

2.4.1 设定配置

- 从 [选项 (Options)] 菜单选取 [创建配置 (Build Configurations) …] 以调用创建配置 (Build configurations) 画面并选取所要使用的环境。 在这里，选取 [SimDebug_H8-2600A]。

若您修改配置，请执行创建过程。

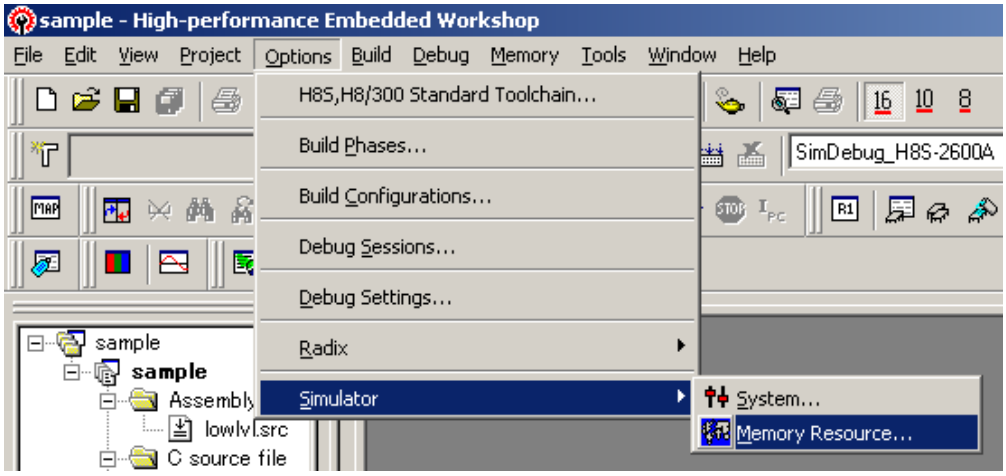


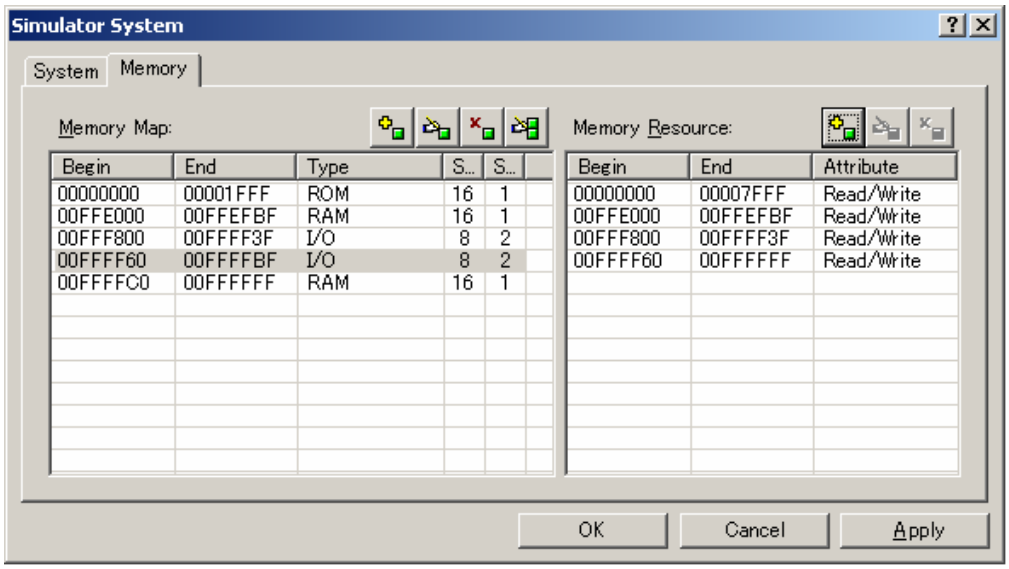


2.4.2 分配存储器资源

应分配存储器资源以运行已开发的应用程序。 请检查设定因为演示工程 (Demonstration Project) 中的存储器资源将自动分配。

- 从 [选项 (Options)] 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。





从 H’00000000 至 H’00007FFF 的可读/写区域被分配为程序区域，从 H’00FFEC00 至 H’00FFFFFF 的区域则被分配为堆栈区域。

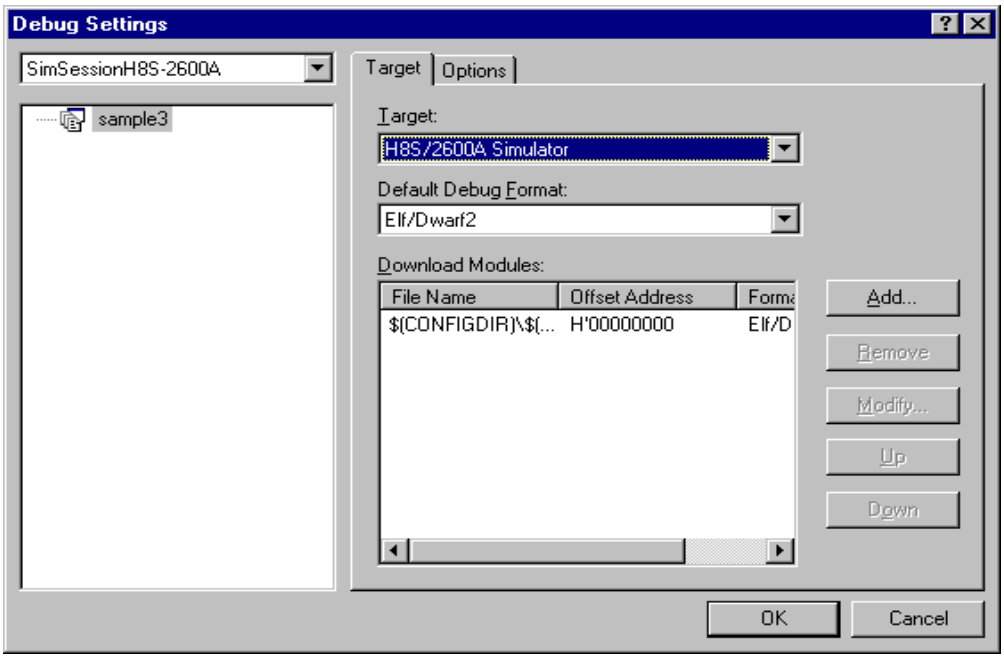
- 单击 [关闭 (Close)] 按钮以关闭对话框。

存储器资源也可在 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) 对话框的模拟程序 (Simulator) 标签上被参考或修改。相互的修改将被反映。

2.4.3 下载样品程序

请检查设定因为所要下载的样品程序在演示程序 (Demonstration Program) 中将自动设定。

- 从 [选项 (Options)] 菜单选取 [调试设定 (Debug Settings) …] 以打开调试设定 (Debug Settings) 对话框。



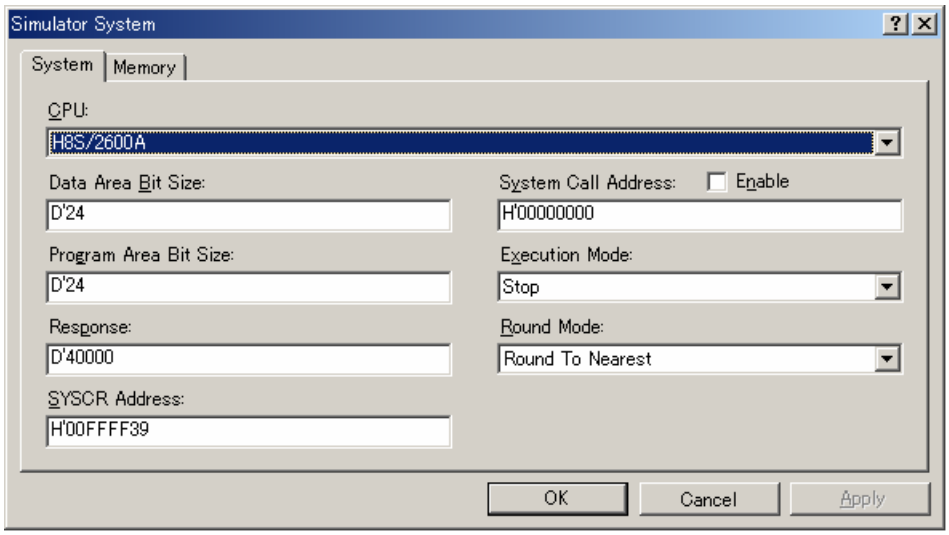
在 [下载模块 (Download Modules)] 中设定的文件将被下载。

- 单击 [确定 (OK)] 按钮以关闭调试设定 (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

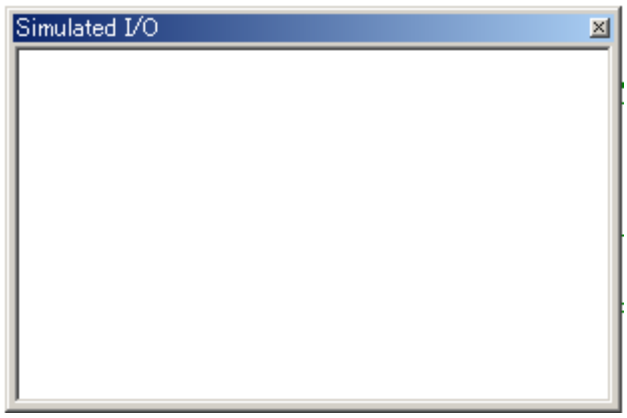
检查设定因为模拟的 I/O 将在演示工程 (Demonstration Project) 中自动设置。

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



- 检查 [系统调用地址 (System Call Address)] 中已选取 [允许 (Enable)]。
- 单击 [确定 (OK)] 按钮以允许模拟的 I/O (Simulated I/O)。
- 从 [视图 (View)] 菜单选取 [模拟的 I/O (Simulated I/O)] 以打开模拟的 I/O (Simulated I/O) 窗口。

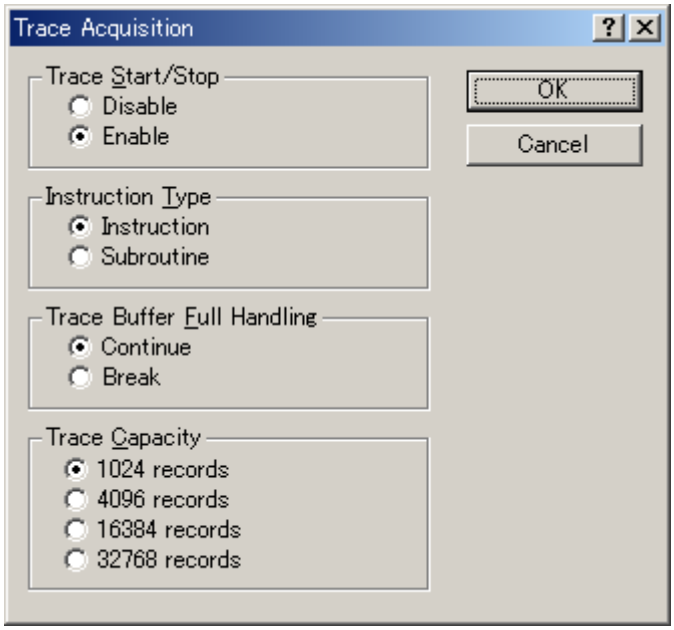
若没有打开模拟的 I/O 窗口，则模拟的 I/O 未被允许。



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。 在跟踪窗口上单击右键以显示快捷菜单并选取 [采集 (Acquisition) …]。

将显示跟踪采集 (Trace Acquisition) 对话框如下。

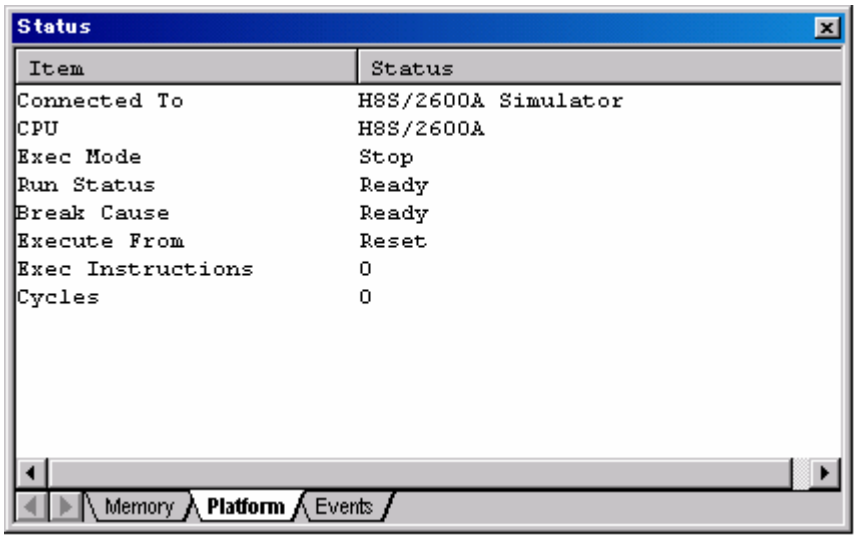


- 在跟踪采集对话框中将 [跟踪开始/停止 (Trace Start/Stop)] 设定为 [允许 (Enable)]，然后单击 [确定 (OK)] 按钮以允许跟踪信息采集 (Trace Information Acquisition)。

2.4.6 状态窗口

终止的原因将可在状态 (Status) 窗口上确认。

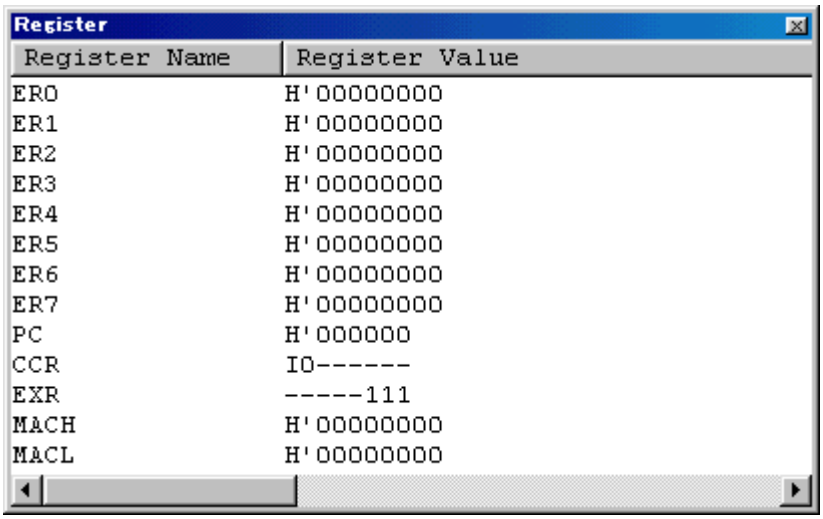
- 从 [视图 (View)] 菜单选取 [CPU->状态 (Status)] 以打开状态 (Status) 窗口。 从状态窗口内显示 [平台 (Platform)] 表。



2.4.7 寄存器 (Registers) 窗口

寄存器的值可在寄存器 (Registers) 窗口上确认。

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

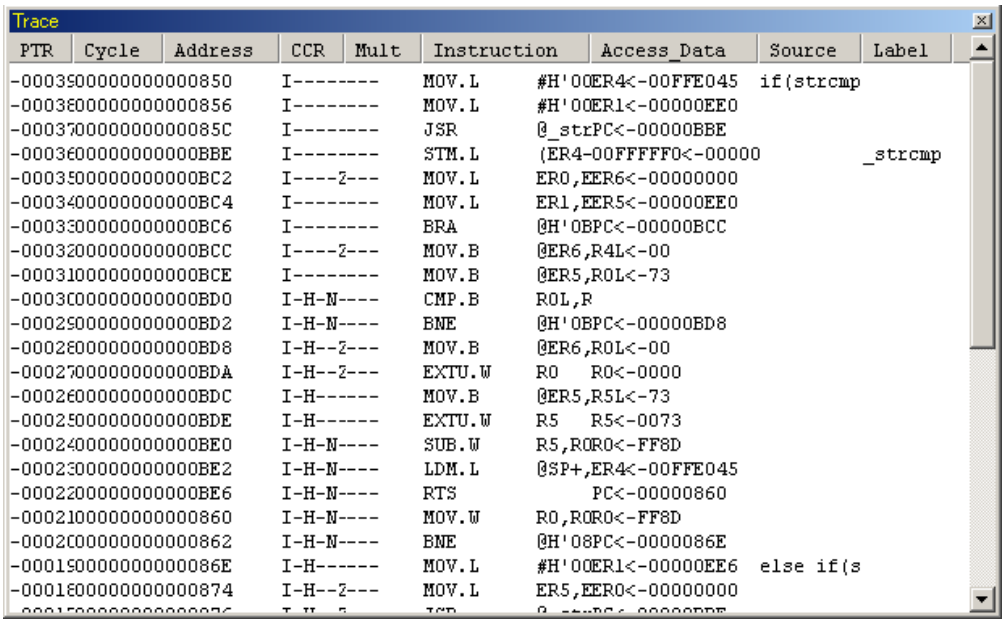


2.4.8 使用跟踪

(1) 跟踪缓冲器

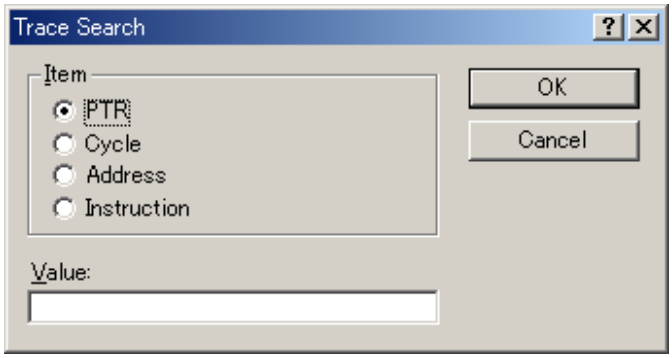
使用跟踪缓冲器 (trace buffer)，您可以查看指令的执行履历。

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

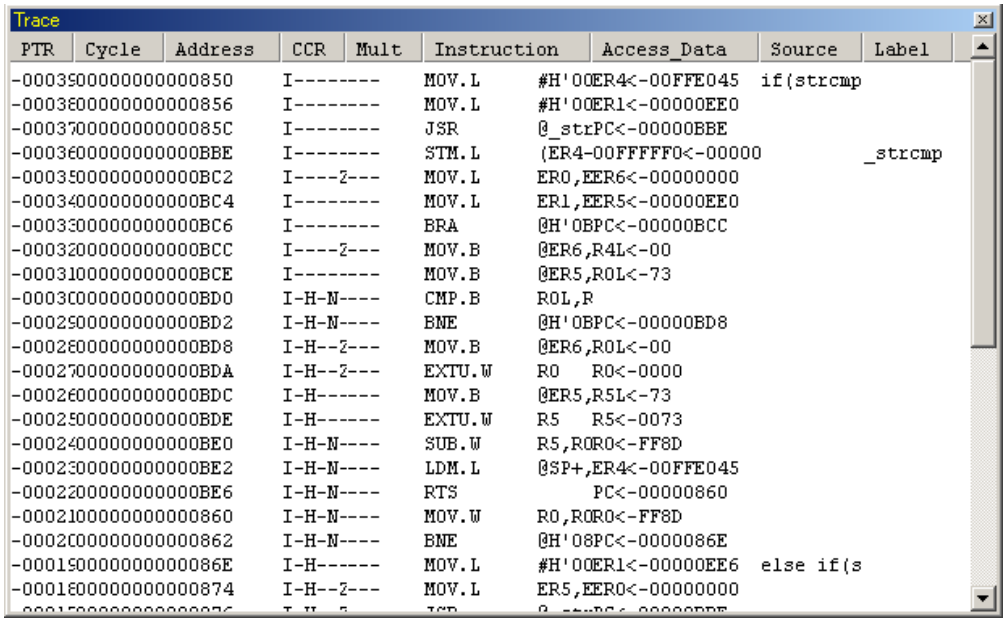


(2) 跟踪搜索

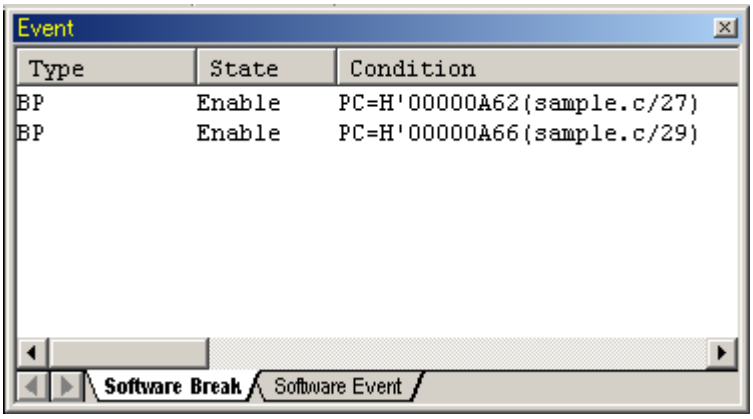
首先，在跟踪 (Trace) 窗口上单击右键以显示快捷菜单并选取 [查找 (Find) …] 以打开跟踪搜索 (Trace Search) 对话框。



设定搜索项目 [项目 (Item)] 和搜索内容 [值 (Value)]，单击 [确定 (OK)]，然后执行跟踪搜索 (Trace Search)。 若您查找到相应的跟踪信息，突出显示第一行。 若您要使用相同的搜索内容 [值 (Value)] 继续跟踪搜索，在跟踪窗口上单击右键以显示快捷菜单并选取 [查找下一个 (Find Next)]。 在下一个步骤中，突出显示下一行。



2.4.9 显示断点



所有在程序中设定的断点列表可在事件点 (Eventpoints) 窗口上显示。

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

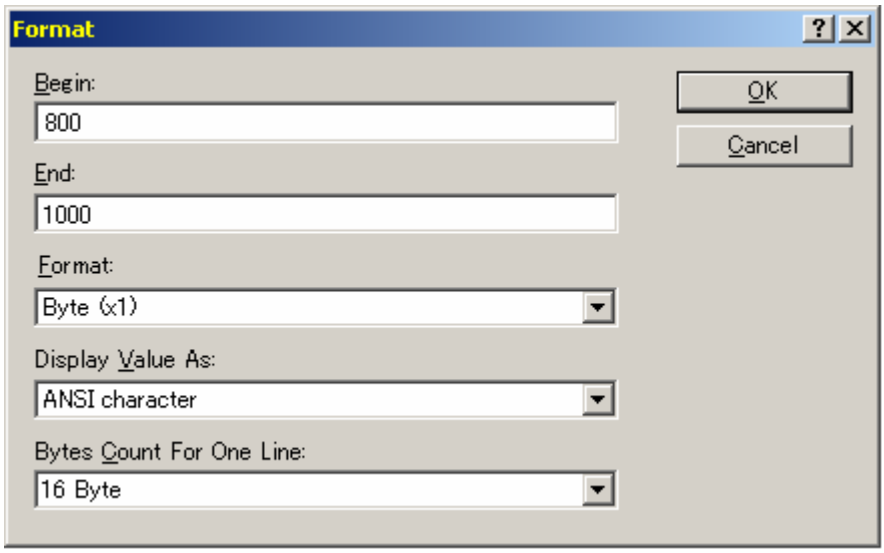
事件点窗口允许用户设定断点、定义新的断点及显示断点。

关闭断点窗口。

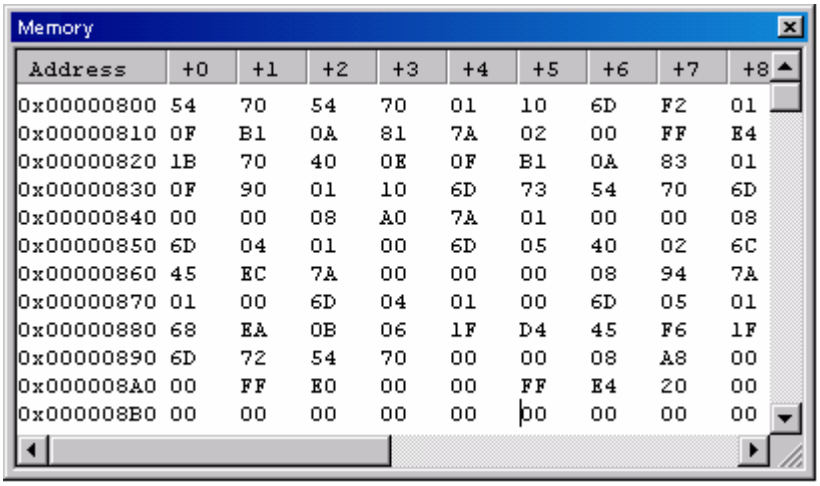
2.4.10 显示存储器内容

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。



- 单击 [确定 (OK)] 按钮以打开存储器 (Memory) 窗口，指定的存储器区域在此显示。



H8S, H8/300 系列 C/C++编译程序应用笔记

编译程序

第 3 节　编译程序

本节将描述在开发 C/C++ 程序时所使用的有效函数。

下列所描述的函数允许您执行不被大多数 C/C++ 程序所支持的中断处理以及其他类型的处理。

3.1　指定中断函数

描述

#pragma interrupt <函数名称 (function name)> 声明一项中断函数。声明的中断函数，在 RTE 指令上返回，函数内所使用的所有寄存器被保证（被保存及恢复）。这允许中断函数从异常处理中返回。

[格式]

#pragma interrupt（<函数名称 (function name)>[（<中断规格 (interrupt specs)>）][,< 函数名称 (function name)>[（<中断规格 (interrupt specs)>）]…]）

实例

要声明中断函数 f1。这项函数在完成其处理后于 RTE 指令上返回。

C/C++ 程序

```
extern unsigned char a;
#pragma interrupt (f1)
void f1(void)
{
    a=0;
}
```

← 函数 f1 被定义为中断函数。

（编译的汇编语言扩张代码）

```
_f1:
            PUSH.W      R0
            SUB.B       R0L,R0L
            MOV.B       R0L,@_a:32
            POP.W       R0
            RTE
            .END
```

← 中断函数在 RTE 指令上返回。

说明与备注

中断函数的声明支持下列函数：堆栈切换规格、陷阱指令返回规格、中断完整函数规格，及向量表规格。

编号	项目	格式	选项	描述
1	堆栈切换规格	sp=	<变量 (variable)>| &<变量 (variable)>| <常量 (constant)>| <变量 (variable)>+ <常量 (constant)>| &<变量 (variable)>+ <常量 (constant)>	指定具有变量或常量的新堆栈地址。 <可输入值的 (valuable)>：可输入值的（指针 (pointer)） &<可输入值的 (valuable)>：可输入值的（目标类型 (object type)）地址 <常量 (constant)>：常量值
2	陷阱指令返回规格	tn=	<常量 (constant)>	指定 TRAPA 指令的结束。 <常量 (constant)>：常量值（陷阱向量号）
3	中断完整函数规格	sy=	<函数名称 (function name)>| <常量 (constant)>| $<函数名称 （function name）>	指定跳转至中断函数的结束。 <函数名称 (function name)>：中断函数名称 <常量 (constant)>：绝对地址 $<函数名称 (function name)>：不具有下划线的中断函数名称。

3.1.1 堆栈切换规格

描述

这项函数指定一个单独的中断函数堆栈区域。

当出现外部中断时，堆栈切换规格 (sp=) 切换堆栈指针至指定的地址，以便中断函数可以使用该堆栈操作。返回后，这项函数将指针复位到出现中断前所存在的条件。

实例

要指定具有变量或常量的新堆栈地址。在下列实例中，数组 STK[100] 被设定为中断函数 f 所要使用的堆栈：

（C/C++ 程序）

```
extern int STK [100];
extern unsigned char a;
#pragma interrupt (f(sp=STK+100))

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

← 指定一项中断函数并切换堆栈指针。

（编译的汇编语言扩展代码）

```
_f:
            MOV.L       SP,@_STK+96:32
            MOV.L       #_STK+96:32,SP
            PUSH.W      R0
            SUB.B       R0L,R0L
            MOV.B       R0L,@_a:32
            POP.W       R0
            MOV.L       @SP,SP
            RTE
            .END
```

← 更改堆栈指针。

← 中断函数在 RTE 指令上返回。

说明与备注

(i) 这项规格可以和陷阱指令返回规格 (trap instruction return specification) 或中断函数完整规格 (interrupt function complete specification) 一起设定。

(ii) 堆栈切换规格 “sp=” 必须始终使用小写字符指定。

3.1.2 陷阱指令返回规格

描述

#pragma interrupt 所声明的函数通常由执行 RTE 指令返回。然而，当陷阱指令返回规格 (tn=) 被允许时，它们将由执行 TRAPA 指令返回。

实例

要在中断函数完成时执行 TRAPA #2 指令以启动陷阱异常处理：

（C/C++ 程序）

```
extern unsigned char a;
#pragma interrupt (f(tn=2))

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

←中断函数 f 在 TRAPA 指令的执行上返回。

（编译的汇编语言扩展代码）

```
_f1:
            PUSH.W      R0
            SUB.B       R0L,R0L
            MOV.B       R0L,@_a:32
            POP.W       R0
            TRAPA       #2
            .END
```

← 在 TRAPA 指令的执行上返回。

说明与备注

(i) 这项规格，可与堆栈切换规格一同设定，但不能与中断函数完整规格一同设定。

(ii) 陷阱指令返回规格 “tn=” 必须始终使用小写字符指定。

(iii)这项规格在 CPU 操作模式指定为 300 时无法使用。

3.1.3 中断函数完整规格

描述

#pragma interrupt 所声明的函数通常由执行 RTE 指令返回。然而，当中断函数完整规格 (sy=) 被允许时，它们将在 JMP 指令跳转至指定的地址。

实例

在 JMP 指令跳转至函数 f2 的地址：

（C/C++ 程序）

```
extern int f2();
extern unsigned char a;
#pragma interrupt (f1(sy=$ f2))

void f1(void)
{
    a=0;
}
```

← 在中断函数 f1 的结束部分，通过执行 JMP 指令跳转到函数 f2 的地址。

（编译的汇编语言扩展代码）

```
_f1:
            PUSH.W      R0
            SUB.B       R0L,R0L
            MOV.B       R0L,@_a:32
            POP.W       R0
            JMP         @f2:24
            .END
```

← 在 JMP 指令上返回。

说明与备注

(i) 这项规格，可与堆栈切换规格一同设定，但不能与陷阱指令返回规格一同设定。

(ii) 若被指定为 $<函数名称 (function name)>，则函数名称将被汇编语言程序参考为不具下划线的名称。

(iii)中断函数完整规格 “sp=” 必须始终使用小写字符指定。

3.1.4 向量表自动生成函数

描述

通过指定 #pragma interrupt 的向量号，函数的向量表将自动生成。

[格式]

#pragma interrupt（<函数名称>[（vect=<向量号>）]）

实例

要指定一个向量号以创建向量表。

（C/C++ 程序）

(CPU=2600a)

```
#pragma entry f1(vect=0)
void f1(){
}
#pragma interrupt (f2(vect=4))
void f2(void){
}
#pragma indirect (f3(vect=5))
unsigned char f3(void){
}
```

← 分配项目函数 f1 至向量号 0。

← 分配中断函数 f2 至向量号 4。

← 分配间接存储器存取函数 f3 至向量号 5。

（存储器映像内容）

```
$VECT0   00000000   00000003
$VECT4   00000010   00000013
$VECT5   00000014   00000017
```

说明与备注

(i) 向量表规格“vect=”必须始终使用小写字符指定。
(ii) 必须谨慎所分配的向量号不与其他向量表重复。
(iii) C/C++ 编译程序 4.0 或以上版本支持向量表自动生成函数。

3.2 内建函数

C/C++ 语言规格内所不支持的 CPU 指令，如条件码寄存器的设定，被支持为扩展内建函数。

当使用内建函数时，确保声明系统包含文件 machine.h。

编号	项目	函数	参考的节
1	条件码寄存器 (CCR)	设定中断屏蔽	3.2.1
2		参考中断屏蔽	
3		设定 CCR	
4		参考 CCR	
5		按逻辑 AND CCR	
6		按逻辑 OR CCR	
7		按逻辑 XOR CCR	
8	扩展寄存器 (EXR)	设定中断屏蔽	3.2.2
9		参考中断屏蔽	
10		设定 EXR	
11		参考 EXR	
12		按逻辑 AND EXR	
13		按逻辑 OR EXR	
14		按逻辑 XOR EXR	
15	向量基址寄存器 (VBR) *	设定 VBR	3.2.3
16	具溢出测试的操作	执行 1 字节加法并根据结果设定 CCR	3.2.4
17		执行 2 字节加法并根据结果设定 CCR	
18		执行 4 字节加法并根据结果设定 CCR	
19		执行 1 字节减法并根据结果设定 CCR	
20		执行 2 字节减法并根据结果设定 CCR	
21		执行 4 字节减法并根据结果设定 CCR	
22		左移 1 字节数据并根据结果设定 CCR	
23		左移 2 字节数据并根据结果设定 CCR	
24		左移 4 字节数据并根据结果设定 CCR	
25		执行 1 字节数据的符号转换并根据结果设定 CCR	
26		执行 2 字节数据的符号转换并根据结果设定 CCR	
27		执行 4 字节数据的符号转换并根据结果设定 CCR	

编号	项目	函数	参考的节
28	转移指令	MOVFPE 指令	3.2.5
29		MOVTPE 指令	
30	算术指令	十进制加法	3.2.6
31		十进制减法	
32		TAS 指令	
33		MAC 指令	
34		64 位乘法*	
35	移位指令	向左循环 1 字节数据	3.2.7
36		向左循环 2 字节数据	
37		向左循环 4 字节数据	
38		向右循环 1 字节数据	
39		向右循环 2 字节数据	
40		向右循环 4 字节数据	
41	系统控制指令	TRAPA 指令	3.2.8
42		SLEEP 指令	
43	块转移指令	EEPMOV 指令	3.2.9
		EEPMOV 指令（中断请求）	
44	块转移指令 (H8SX)	MOVMD 指令	3.2.10
		MOVSD 指令	
45	NOP 指令	NOP 指令	

注意： * 只能用于 H8SX。

3.2.1 设定和参考条件码寄存器 (CCR)

描述

对于设定和参考条件码寄存器，编译程序提供了下表所列的函数：

编号	项目	格式	描述
1	设定中断屏蔽	void set_imask_ccr(unsigned char mask)	设定屏蔽值（0 或 1）至 CCR 的中断屏蔽位。
2	参考中断屏蔽	unsigned char get_imask_ccr(void)	参考 CCR 中断屏蔽位 (I) 的值（0 或 1）。
3	设定 CCR	void set_ccr(unsigned char ccr)	设定 ccr 的值（8 位）至 CCR。
4	参考 CCR	unsigned char get_ccr(void)	参考 CCR 的值。
5	AND CCR	void and_ccr(unsigned char ccr)	按逻辑 AND CCR 及 ccr；并将结果存储在 CCR 内。
6	OR CCR	void or_ccr(unsigned char ccr)	按逻辑 OR CCR 及 ccr；并将结果存储在 CCR 内。
7	XOR CCR	void xor_ccr(unsigned char ccr)	按逻辑 XOR CCR 及 ccr；并将结果存储在 CCR 内。

实例

要操作条件码寄存器，然后将它复位到操作 CCR 前所存在的条件：

（C/C++ 程序）

```
#include <machine.h>          ←内建功能的包含文件
void main(void)
{
    unsigned char mask;

    if (mask=get_imask_ccr()){          /* 保存中断屏蔽的值 */
        set_imask_ccr(1);               /* 指定异常的起始 */
        and_ccr((unsigned char)0xFC) ; /* 在 CCR 内保存 CCR 和 0xFC. 的“求与 (AND)” */
    }
    set_imask_ccr(mask);                /* 返回中断屏蔽的值 */
}
```

（编译的汇编语言扩展代码）

```
_main:   STC.B       CCR,R0L
          AND.B       #-128:8,R0L
          ROTL.B      R0L
          BEQ         L48:8
          ORC.B       #-128:8,CCR
          ANDC.B      #-4:8,CCR
L48:     STC.B       CCR,R0H
          BLD.B       #0,R0L
          BST.B       #7,R0H
          LDC.B       R0H,CCR
          RTS
          .END
```

说明与备注

CCR 是指明 CPU 内部状态的 8 位寄存器。

<条件码寄存器 (Condition code register)>

I	UI	H	U	N	Z	V	C

I:　中断屏蔽位
UI: 用户位/中断主位
H:　半进位标志
U:　用户位
N:　负标志
Z:　零标志
V:　溢出标志
C:　进位标志

3.2.2 设定和参考扩展寄存器

描述

对于设定与参考扩展寄存器，编译程序提供下列函数：

编号	项目	格式	描述
1	设定中断屏蔽	void set_imask_exr(unsigned char mask)	设定屏蔽值（0 至 7）至 EXR 的中断屏蔽位（I2 至 I0）。
2	参考中断屏蔽	unsigned char get_imask_exr(void)	参考 EXR 中断屏蔽位（I2 至 I0）的值（0 至 7）。
3	设定 EXR	voir set_exr(unsigned char exr)	在 EXR 内设定 exr 的值（8 位）。
4	参考 EXR	unsigned char get_exr(void)	参考 EXR 的值。
5	获得 EXR 的 AND	void and_exr(unsigned char exr)	按逻辑 AND EXR 及 exr；并将结果存储在 EXR 内。
6	获得 EXR 的 OR	void or_exr(unsigned char exr)	按逻辑 OR EXR 及 exr；并将结果存储在 EXR 内。
7	获得 EXR 的 XOR	void xor_exr(unsigned char exr)	按逻辑 XOR EXR 及 exr；并将结果存储在 EXR 内。

实例

在不更改 EXR 中断屏蔽位值的情况下更改 EXR 状态：

（C/C++ 程序）

```
#include <machine.h>
extern unsigned char e;

void main()
{
    unsigned char mask;

    if (mask=get_imask_exr()){
        set_exr((unsigned char)0x05);
        xor_exr((unsigned char)0xff);
        e=get_exr();
    }
    set_imask_exr(mask);
}
```

←保存中断屏蔽位。

←在 EXR 内设定一个值，按逻辑 XOR，并将结果设定在外部变量 e 内。

←恢复中断屏蔽位。

（编译的汇编语言扩展代码）

```
_main:
            STC.B       EXR,R1L
            AND.B       #7:8,R1L
            BEQ         L49:8
            MOV.B       #5:8,R0L
            LDC.B       R0L,EXR
            XORC.B      #-1:8,EXR
            STC.B       EXR,R0L
            MOV.B       R0L,@_e:32
L49:
            AND.B       #7:8,R1L
            STC.B       EXR,R1H
            AND.B       #-8:8,R1H
            OR.B        R1L,R1H
            LDC.B       R1H,EXR
            RTS
            .END
```

说明与备注

设定及参考扩展寄存器的内建函数只在 CPU/操作模式为 2600n、2600a、2000n，或 2000a 时有效。

<扩展寄存器 (Extended register)>

T	–	–	–	–	I2	I1	I0

（T）跟踪位

（I2 至 I0）中断屏蔽位

3.2.3 设定向量基址寄存器

描述

H8SX 具有可在任何地址为异常处理分配向量区域的函数。

在 H8/300、H8/300H、H8S 系列中，异常处理的向量区域固定为从零起。

当 CPU 为 H8SX 时，用户可通过指定向量基址寄存器 (VBR) 来修改用于异常处理的向量区域分配地址。

编译程序为向量基址寄存器的设定提供了下列函数：

编号	项目	格式	描述
1	设定 VBR	void set_vbr(void* vbr)	设定 vbr 的值（32 位）至 VBR。

实例

设定向量基址寄存器（VBR）的值：

（C/C++ 程序）

```
#include <machine.h>                    ←内建函数的包含文件
void main(void)
{
    set_imask_ccr(1);             /* 设定中断屏蔽位                */
    set_vbr((void*)0x20000);      /* 将 0x20000 设定到 VBR         */
    set_imask_ccr(0);             /* 清除中断屏蔽位                */
}
```

（编译的汇编语言扩展代码）

```
_main:
            ORC.B       #H'80:8,CCR
            SUB.L       ER0,ER0
            MOV.W       #2:3,E0
            LDC.L       ER0,VBR
            ANDC.B      #H'7F:8,CCR
            RTS
            .END
```

说明与备注

(1) 设定向量基址寄存器的内建函数仅在 CPU/操作模式为 H8SXN、H8SXM、H8SXA 或 H8SXX 时有效。
(2) 当 CPU/操作模式为 H8SXN 时，向量基址寄存器指定值的低 16 位有效。
(3) 有关切换向量表地址的详细资料，请参考 3.8.3 节，切换向量表地址。
(4) 切换向量基址寄存器 (VBR) 应在中断屏蔽 (interrupt mask) 状态中完成。若不在中断屏蔽状态中进行，当切换向量基址寄存器 (VBR) 期间发生中断处理时，将无法保证异常处理的正确运行。

3.2.4 具溢出（V 标志）测试的操作

描述

下列内建函数只在执行具溢出（V 标志）测试的操作时可用：

（CC:条件码）

编号	项目	格式	描述
1	1 字节加法及 CCR 设定	int ovfaddc(char dst,char src,char *rst)	相加 dst 和 src，各长 1 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
2	2 字节加法及 CCR 设定	int ovfaddw(int dst,int src,int *rst)	相加 dst 和 src，各长 2 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
3	4 字节加法及 CCR 设定	int ovfaddl(long dst,long src,long *rst)	相加 dst 和 src，各长 4 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
4	1 字节减法及 CCR 设定	int ovfsubc(char dst,char src,char *rst)	从 dst 减去 src，各长 1 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
5	2 字节减法及 CCR 设定	int ovfsubw(int dst,int src,int *rst)	从 dst 减去 src，各长 2 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
6	4 字节减法及 CCR 设定	int ovfsubl(long dst,long src,long *rst)	从 dst 减去 src，各长 4 字节；若 rst≠0 则将结果存储在以 rst 表示的区域。
7	1 字节左移及 CCR 设定	int ovfshalc(char dst, char *rst)	将 1 字节数据 dst 算术左移 1 位；若 rst≠0 则将结果存储在以 rst 表示的区域。
8	2 字节左移及 CCR 设定	int ovfshalw(int dst, int *rst)	将 2 字节数据 dst 算术左移 1 位；若 rst≠0 则将结果存储在以 rst 表示的区域。
9	4 字节左移及 CCR 设定	int ovfshall(long dst, long *rst)	将 4 字节数据 dst 算术左移 1 位；若 rst≠0 则将结果存储在以 rst 表示的区域。
10	1 字节符号转换及 CCR 设定	int ovfnegc(char dst, char *rst)	获取 1 字节数据 dst 的 2 的补数；若 rst≠0 则将结果存储在以 rst 表示的区域。
11	2 字节符号转换及 CCR 设定	int ovfnegw(int dst, int *rst)	获取 2 字节数据 dst 的 2 的补数；若 rst≠0 则将结果存储在以 rst 表示的区域。
12	4 字节符号转换及 CCR 设定	int ovfnegl(long dst, long *rst)	获取 4 字节数据 dst 的 2 的补数；若 rst≠0 则将结果存储在以 rst 表示的区域。

实例

要测试以查看加法的结果是否已溢出；执行适当的处理。

（C/C++ 程序）

```
#include <machine.h>
extern int dst, src;
void f()
{
    if (ovfaddw(dst,src,0))
        dst++;
    else
        dst--;
}
```

←检查 dst 和 src 的相加是否生成溢出(overflow)。

（编译的汇编语言扩展代码）

```
_f:
            PUSH.L      ER6
            MOV.L       #_dst:32,ER6
            MOV.W       @ER6,R0
            MOV.W       @_src:32,R1
            ADD.W       R1,R0
            BVC         L48:8
            MOV.W       @ER6,R0
            INC.W       #1,R0
            BRA         L50:8
L48:
            MOV.W       @ER6,R0
            DEC.W       #1,R0
L50:
            MOV.W       R0,@ER6
            POP.L       ER6
            RTS
            .END
```

说明与备注

条件码操作函数只能在以 if、do、while，或 for 语句来测试条件的表达式中指定。

3.2.5 转移指令

描述

下列函数可用以增强系统控制转移指令：

编号	项目	格式	描述
1	MOVFPE 指令	void movfpe(char *addr,char data) char _movfpe(char *addr) *1	扩展入与 E 时钟同步转移数据的 MOVFPE 指令。
2	MOVTPE 指令	void movtpe(char data ,char *addr)	扩展入与 E 时钟同步转移数据的 MOVTPE 指令。

注意： 1. 仅限于 H8SX

实例

(a) MOVFPE 指令

要和 E 时钟同步从由 16 位绝对地址所指定的存储器地址装入数据：
_movfpe 与 movfpe 函数相同，除了它返回目标 (Destination) 数据为其函数值。

（C/C++ 程序）

```
#include <machine.h>
#define P1DR (*(unsigned char *)0x00FFFF60)
extern unsigned char data;
void f()
{
    movfpe((char*)&P1DR,data);
}
```

←执行 MOVFPE 指令。

（编译的汇编语言扩展代码）

```
_f:
            MOVFPE.B    @16777056:16,R0L
            MOV.B       R0L,@_data:32
            RTS
            .END
```

（C/C++ 程序）

```
#include <machine.h>
#define P1DR (*(unsigned char *)0x00FFFF60)
extern unsigned char data;
void f()
{
    data =  movfpe((char*)&P1DR);
}
```

← 执行 MOVFPE 指令。

（编译的汇编语言扩展代码）

```
_f:
            MOVFPE.B    @16777056:16,R0L
            MOV.B       R0L,@_data:32
            RTS
            .END
```

(b) MOVTPE 指令

要和 E 时钟同步存储数据到由 16 位绝对地址所指定的存储器区域：

（C/C++ 程序）

```
#include <machine.h>
extern unsigned char data;
#define P1DR (*(unsigned char*)0x00FFFF60)
void f()
{
    movtpe(data,(char*)&P1DR);
}
```

执行 MOVTPE 指令。

（编译的汇编语言扩展代码）

```
_f:
            MOV.B       @_data:32,R0L
            MOVTPE.B    R0L,@16777056:16
            RTS
            .END
```

3.2.6 算术操作指令

描述

下列函数可用以增强算术操作指令：

编号	项目	格式	描述
1	十进制加法	void dadd(unsiged char size, char*ptr1, char*ptr2, char*rst)	在起始自 ptr1 的大小字节数据和起始自 prt2 的大小字节数据之间执行十进制加法；并将结果存储到起始自 rst 的大小字节区域。
2	十进制减法	void dsub(unsiged char size, char*ptr1, char*ptr2, char*rst)	在起始自 ptr1 的大小字节数据和起始自 prt2 的大小字节数据之间执行十进制减法；并将结果存储到起始自 rst 的大小字节区域。
3	TAS 指令	void tas(char*addr)	扩展入测试与设定 (test-and-set) 指令 TAS。
4	MAC 指令	long mac(long val, int*ptr1,int *ptr2, unsigned long count) long macl(long val, int*ptr1, int*ptr2, unsigned long count, unsigned long mask)	扩展入乘法累积 (multiply-accumulate) 指令 MAC。
5	64 位乘法*1	long mulsu(long val1,long val2) unsigned long muluu(unsigned long val1,unsigned long val2)	扩展入 MULS/U，MULU/U

注意： 1. 仅限于 H8SX

实例

(1) 十进制操作

要以十进制相加起始自 ptr1 指定地址的 6 位数 4 位 BCD 数据（3 字节）和起始自 ptr2 指定地址的 4 位 BCD 数据，并将结果存储到起始自 rst 指定地址的 3 字节区域内：

（C/C++ 程序）

```
#include <machine.h>
char ptr1[3]={0,1,2};
char ptr2[3]={2,1,0};
char rst[3];
void f()
{
    dadd((char)3,ptr1,ptr2,rst);
}
```

输出 DAA 指令。

（编译的汇编语言扩展代码）

```
_f:       STM.L       (ER4-ER6),@-SP
          MOV.L       #_ptr1+2:32,ER0
          MOV.L       #_ptr2+2:32,ER1
          MOV.L       #_rst+3:32,ER5
          MOV.B       #3:8,R6L
          ANDC.B      #-34:8,CCR
L49:    MOV.B       @ER0,R4L
          MOV.B       @ER1,R4H
          ADDX.B      R4H,R4L
          DAA.B       R4L
          MOV.B       R4L,@-ER5
          DEC.L       #1,ER0
          DEC.L       #1,ER1
          DEC.B       R6L
          BNE         L49:8
          LDM.L       @SP+,(ER4-ER6)
          RTS
```

说明与备注

函数 dadd 和 dsub 的第一个参数为常量 1 至 255。

(2) TAS 指令

要在测试存储器内容后（通过和 0 比较）将存储器内容的 MSB（位 7）设定为“1”：

（C/C++ 程序）

```
extern unsigned char data;
#define ADR (*(volatile unsigned char *)0x00fff000)
#include <machine.h>
void main()
{
    tas((char*)&ADR);

    if (data=get_ccr())
        and_ccr(data);
    else
        or_ccr(data);
}
```

←将存储器内容与 0 比较；在 CCR 内设定结果。

←根据存储器内容存储 AND 或 OR 至 CCR。

（编译的汇编语言扩展代码）

```
_main:  MOV.L       #16773120:32,ER0
          TAS         @ER0
          MOV.L       #_data:32,ER1
          STC.B       CCR,R0L
          MOV.B       R0L,@ER1
          BEQ         L47:8
          MOV.B       @ER1,R1L
          STC.B       CCR,R1H
          AND.B       R1L,R1H
          LDC.B       R1H,CCR
          RTS
L47:      MOV.B       @ER1,R1L
          STC.B       CCR,R1H
          OR.B        R1L,R1H
          LDC.B       R1H,CCR
          RTS
          .END
```

说明与备注

函数 tas 只在 CPU 操作模式为 2600a、2600n、2000a，或 2000n 时有效。

(3) MAC 指令

H8S/2600 微型计算机包含乘加寄存器 (multiply-accumulate register, MAC)，它是存储乘法累积运算结果的 64 位寄存器。下列图示显示这个寄存器如何组织。

MAC 指令在存储器数据项目之间执行乘法并把结果添加到 MAC 寄存器。 使用这个寄存器，16 x 16 位 + 32 位 = 32 位乘法累积运算可被执行。

下列解释从下面提供的实例中得出：

<函数 mac (Function mac)>

将值 100 作为初始值分配给 MAC 寄存器。在带符号的基准上相乘以 ptr1 及 ptr2 表示的 2 字节数据项目，将所得 4 字节数据添加到 MAC 寄存器，并以 2 为增量增加 ptr1 及 ptr2。重复这个步骤四次，并在最后返回 MAC 寄存器的内容。

<函数 mac1 (Function macl)>

以 ~4 执行乘法累积运算，因为函数使用 ptr2 的数据进行环形缓冲。

由于函数将 ptr2&mask 用作地址，ptr2 必须被分配给一个是 8 的整数倍数的地址。

（C/C++ 程序）

```
#include <machine.h>
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
int ptr3[2]={9,8};
long l1,l2;
void func()
{
    l1=mac(100,ptr1,ptr2,4);
                                    /* l1=100+0*9+1*8+2*7+3*6 */
    l2=macl(100,ptr1,ptr3,4,~4);
                                    /* l2=100+0*9+1*8+2*9+3*8 */
}
```

← 乘法累积运算

（编译的汇编语言扩展代码）

```
_func:   PUSH.L      ER2
         MOV.L       #100:32,ER0
         CLRMAC
         LDMAC.L     ER0,MACL
         MOV.L       #_ptr2:32,ER0
         MOV.L       #_ptr1:32,ER1
         MAC         @ER1+,@ER0+
         MAC         @ER1+,@ER0+
         MAC         @ER1+,@ER0+
         MAC         @ER1+,@ER0+
         STMAC.L     MACL,ER0
         MOV.L       ER0,@_l1:32
         MOV.L       #100:32,ER0
         CLRMAC
         LDMAC.L     ER0,MACL
         MOV.L       #_ptr3:32,ER0
         MOV.L       #_ptr1:32,ER1
         MOV.L       #-5:32,ER2
         MAC         @ER1+,@ER0+
         AND.L       ER2,ER0
         MAC         @ER1+,@ER0+
         AND.L       ER2,ER0
         MAC         @ER1+,@ER0+
         AND.L       ER2,ER0
         MAC         @ER1+,@ER0+
         AND.L       ER2,ER0
         STMAC.L     MACL,ER0
         MOV.L       ER0,@_l2:32
         POP.L       ER2
         RTS
```

说明与备注

函数 mac 和 mac1 只能在 CPU 操作模式指定为 2600a 或 2600n 时使用。

(4) MULS/U，MULU/U 指令

mulsu/muluu 被扩展至 MULS/U 或 MULU/U 指令，这些指令执行 32 位 x 32 位 = 64 位乘法。

这项原有函数的 32 位参数（val1 及 val2）被倍乘且上端 32 位被返回为操作结果。

（C/C++ 程序）

```
#include <machine.h>
long sval1, sval2, sans;
unsigned long uval1, uval2, uans;
void f(void)
{
    sans = mulsu(sval1, sval2);     ←带符号 32 位乘法的上端 32 位

    uans = muluu(uval1, uval2);     ←无符号 32 位乘法的上端 32 位
}
```

（编译的汇编语言扩展代码）

```
_f:
            PUSH.L      ER2
            MOV.L       @sval1:32,ER1
            MOV.L       @sval2:32,ER2
            MULS/U.L    ER2,ER1
            MOV.L       ER1,@sans:32
            MOV.L       @uval1:32,ER1
            MOV.L       @uval2:32,ER2
            MULU/U.L    ER2,ER1
            MOV.L       ER1,@uans:32
            RTS/L       ER2
```

说明与备注

这项函数 mulsu/muluu 仅在 CPU 为具 H8SX*:{M | MD} 的 H8SX 时有效。

3.2.7 移位指令

描述

下列内建函数可用以增强循环指令：

编号	项目	格式	描述
1	向左循环 1 字节数据	char rotlc(int count,char data)	以计数位向左循环 1 字节数据；返回结果。
2	向左循环 2 字节数据	int rotlw(int count,int data)	以计数位向左循环 2 字节数据；返回结果。
3	向左循环 4 字节数据	long rotll(int count,long data)	以计数位向左循环 4 字节数据；返回结果。
4	向右循环 1 字节数据	char rotrc(int count,char data)	以计数位向右循环 1 字节数据；返回结果。
5	向右循环 2 字节数据	int rotrw(int count,int data)	以计数位向右循环 2 字节数据；返回结果。
6	向右循环 4 字节数据	long rotrl(int count,long data)	以计数位向右循环 4 字节数据；返回结果。

实例

要循环数据位。

（C/C++ 程序）

```
#include <machine.h>
extern unsigned char data;
char i;
void func()
{
    i=rotlc(2,data);
}
```

←向左循环 2 位。

（编译的汇编语言扩展代码）

```
_func:
          MOV.B       @_data:32,R0L
          ROTL.B      #2,R0L
          MOV.B       R0L,@_i:32
          RTS
          .SECTION    B,DATA,ALIGN=2
_i:
          .RES.B      1
```

3.2.8 系统控制指令

描述

下列函数可用以增强系统控制指令：

编号	项目	格式	描述
1	TRAPA 指令	void trapa(unsigned int trap_no)	扩展入无条件陷阱 TRAPA #trap_no。
2	SLEEP 指令	void sleep(void)	扩展入低功耗 (low-power-consumption) 模式指令 SLEEP。

实例

(1) TRAPA 指令

要转移至向量地址的内容所表示的地址，该地址与指定的向量表号 0 有关：

（C/C++ 程序）

```
#include <machine.h>
#define dummy (void*)0
extern void f1(void);
extern void f2(void);
extern void f3(void);
void (*const vect_table[])(void)={
    f1,dummy,f2,f3
};
void func()
{
    trapa(0);
}
```

←到函数 f1 的陷阱指令

注意：在这种情形下，向量表应被分配到一个中断向量地址。

（编译的汇编语言扩展代码）

```
_func:
          TRAPA       #0
          RTS
          .SECTION    C,DATA,ALIGN=2
_vect_table:
          .DATA.L     _f1
          .DATA.L     H'00000000
          .DATA.L     _f2,_f3
          .END
```

说明与备注

(i) 只有常量 0 至 3 可被分配给函数 trapa 的参数。
(ii) 这项函数只在 300 以外的 CPU 操作模式被指定时有效。

(2) SLEEP 指令

发出 SLEEP 指令以把 CPU 置于低功耗模式。

低功耗模式维护当前 CPU 状态，延缓任何在 SLEEP 指令之后的指令执行，并等待直到中断请求被生成。在中断请求之后，CPU 退出低功耗。

（C/C++ 程序）

```
#include <machine.h>
extern int a;
void func()
{
    while(a);
    sleep();
}
```

←发出 SLEEP 指令。

（编译的汇编语言扩展代码）

```
_func:
          MOV.W       @_a:32,R0
L49:
          BNE         L49:8
          SLEEP
          RTS
```

3.2.9 块转移指令

描述

下列函数可用以增强系统控制块转移指令：

编号	项目	格式	描述
1	EEPMOV 指令	void eepmov(void*dst, const void*src, unsigned char size) void eepmov(void*dst, const void*src, unsigned int size)	扩展入块转移指令 EEPMOV。
		void eepmovb(void*dst, const void*src, unsigned char size) *1	始终将其扩展至 EEPMOV.B。 大小可以是变量。
		void eepmovw(void*dst, const void*src, unsigned int size) *1	始终将其扩展至 EEPMOV.W。 大小可以是变量。
		void eepmovi(void*dst, const void*src, unsigned int size) *1	将其扩展至 EEPMOV。 可在中断之后恢复转移。 大小可以是变量。
2	EEPMOV 指令（具有 ECR 设定）	void eepromb(void*dst, const void*src, unsigned char size, volatile unsigned char*ecr, unsigned char ecrval) void eepromw(void*dst, const void*src, unsigned int size, volatile unsigned char*ecr, unsigned char ecrval)	将值设定至 ECR。 将其扩展至 EEPMOV.B, EEPMOV/P.W。 大小可以是变量。
	EEPMOV 指令（具有 EPR 和 ECR 设定）	void eepromb_epr(void*dst, const void*src, unsigned char size, volatile unsigned char*ecr, unsigned char ecrval, volatile unsigned char*epr, unsigned char eprval) void eepromw_epr(void*dst, const void*src, unsigned int size, volatile unsigned char *ecr,unsigned char ecrval, volatile unsigned char*epr, unsigned char eprval)	将值设定至 ECR，EPR。 将其扩展至 EEPMOV.B, EEPMOV/P.W。 大小可以是变量。

注意： 1. 仅限于 H8SX

实例

(1) eepmov, eepmovb, eepmovw

要以第三个参数所表示的字节从第二个参数所表示的地址执行块转移至第一个参数所表示的地址。

（C/C++ 程序）

```
#include <machine.h>
struct STR{
    char a[300];
}ST1;
struct STR ST2={0};
void f()
{
    eepmov((char*)&ST1,(char*)&ST2,255);
}
```

←执行 EEPMOV 指令。

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       #_ST2:32,ER5
            MOV.B       #-1:8,R4L
            MOV.L       #_ST1:32,ER6
            EEPMOV.B
            LDM.L       @SP+,(ER4-ER6)
            RTS
```

说明与备注

(i) 当 CPU 操作模式为 300 时，可被块转移的最大数据大小为 255 字节。

(ii) 当 CPU 操作模式为 300 以外时，可被块转移的最大数据大小为 65535 字节。当数据大小为 256 至 65535 字节时，指令被扩展入 EEPMOV.W，并可能受 NMI 中断影响。

若要获取这项中断的详细资料，请参考相应的产品编程手册。

(2) eepmovi

要以第三个参数所表示的字节从第二个参数所表示的地址执行块转移至第一个参数所表示的地址。

这项函数被扩展以使 EEPMOV 指令在从中断返回后可以恢复转移。

（C/C++ 程序）

```
#include <machine.h>
struct STR{
    char a[300];
}ST1;
struct STR ST2={0};
void f()
{
    eepmovi((char*)&ST1,(char*)&ST2,256);
}
```

←执行 EEPMOV 指令。

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       #_ST1,ER6
            MOV.L       #_ST2,ER5
            MOV.W       #256:16,R4
L28:
            EEPMOV.W
            MOV.W       R4,R4
            BNE         L28:8           ←执行，直到所剩转移大小为零。
            RTS/L       (ER4-ER6)
```

说明与备注

这项函数仅在 CPU 为 H8SX 时有效。

(3) eepromb,eepromw

要以第三个参数所表示的字节从第二个参数所表示的地址执行块转移至第一个参数所表示的地址。

eepromb 固有函数使用 EEPMOV.B 指令转移存储块，**eepromw** 则使用 EEPMOV/P.W 指令。

这些固有函数设定存储器的第一、第二和第三个参数，将 **ecrval** 设定到 **ecr** 所指示的地址，然后转移存储块。

若转移成功完成，将返回 0。 若转移失败，将返回存储块的剩余大小。

eepromb 的**大小**可以是 0 至 255，而 **eepromw** 的**大小**可以是 0 至 65535。但若**大小**是 0，将不发生转移。

（C/C++ 程序）

```
#include <machine.h>
#define ecr_ptr ((volatile unsigned char *)(0x123456))
char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromw(b, a, 10, ecr_ptr, 1);      ←执行 EEPMOV/P.W 指令。
}
```

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       #_b,ER6
            MOV.L       #_a,ER5
            MOV.W       #H'000A:16,R4
            MOV.B       #1:4,@H'00123456:32
            EEPMOV/P.W
            MOV.B       R4L,@_x:32
            RTS/L       (ER4-ER6)
```

说明与备注

(i) 此固有函数在 CPU 类型为 AE5，或在指定了 H8SX 及 -eeprom 选项时有效。
(ii) 有关 ECR、EPR 的详细资料及其他相关事项，请参考硬件手册。

(4) eepromb_epr,eepromw_epr

要以第三个参数所表示的字节从第二个参数所表示的地址执行块转移至第一个参数所表示的地址。

eepromb_epr 固有函数使用 EEPMOV.B 指令转移存储块，**eepromw_epr** 则使用 EEPMOV/P.W 指令。

这些固有函数设定存储器的第一、第二和第三个参数，将 **eprval** 设定到 **epr** 所指示的地址，将 **ecrval** 设定到 **ecr** 所指示的地址，然后转移存储块。

若转移成功完成，将返回 0。 若转移失败，将返回存储块的剩余大小。

eepromb_epr 的**大小**可以是 0 至 255，而 **eepromw_epr** 的**大小**可以是 0 至 65535。但若**大小**是 0，将不发生转移。

（C/C++ 程序）

```
#include <machine.h>
#define ecr_ptr ((volatile unsigned char *)(0x123456))
#define epr_ptr ((volatile unsigned char *)(0x123457))
char a[10], b[10];
unsigned char x;
void f(void)
{
    x = eepromw_epr(b, a, 10, ecr_ptr, 1, epr_ptr, 1);
}
```

←执行 EEPMOV/P.W 指令

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       #_b,ER6
            MOV.L       #_a,ER5
            MOV.W       #H'000A:16,R4
            MOV.B       #1:4,@H'00123457:32
            MOV.B       #1:4,@H'00123456:32
            EEPMOV/P.W
            MOV.B       R4L,@_x:32
            RTS/L       (ER4-ER6)
```

说明与备注

(i) 此固有函数在 CPU 类型为 AE5，或在指定了 H8SX 及 -eeprom 选项时有效。
(ii) 有关 ECR、EPR 的详细资料及其他相关事项，请参考硬件手册。

3.2.10 H8SX 的块转移指令

描述

下列函数可用以增强 H8SX 的块转移指令。

编号	项目	格式	描述
1	MOVMD 指令	void movmdb(void*dst, const void*src, unsigned int count) void movmdw(int*dst, const int*src, unsigned int count) void movmdl(long*dst, const long*src, unsigned int count)	扩展入 MOVMD 指令。
2	MOVSD 指令	unsigned int movsd(char*dst, const char*src, unsigned int size)	扩展入 MOVSD 指令。

实例

(1) movmdb, movmdw, movmdl

MOVMD.B、MOVMD.W 或 MOVMD.L 指令分别转移 1、2 或 4 字节的存储块，转移的次数以从 **src** 指定地址至 **dst** 指定地址的**计数 (count)** 来指定。

在以下实例中，100 字节的转移，movmdb 每次转移 1 字节共 100 次，movmdw 每次转移 2 字节共 50 次，movmdl 每次转移 4 字节共 25 次。

（C/C++ 程序）

```
#include <machine.h>
char s1[100], d1[100];
int s2[50], d2[50];
long s4[25], d4[25];
void f(void)
{
        movmdb(d1, s1, 100);
        movmdw(d2, s2, 50);
        movmdl(d4, s4, 25);
 }
```

←执行 MOVMD 指令。

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       # d1,ER6
            MOV.L       # s1,ER5
            MOV.W       #100:16,R4
            MOVMD.B
            MOV.L       # d2,ER6
            MOV.L       # s2,ER5
            MOV.W       #50:16,R4
            MOVMD.W
            MOV.L       # d4,ER6
            MOV.L       # s4,ER5
            MOV.W       #25:16,R4
            MOVMD.L
            RTS/L       (ER4-ER6)
```

说明与备注

(i) 这项函数仅在 CPU 为 H8SX 时有效。

(ii) **计数 (count)** 采用从零至 65535 的值。然而，若**计数**为零，它将被解释为 65536。

(2) movsd

使用块转移指令 MOVSD 从 **src** 指定地址转移存储块至 **dst** 指定地址，直到数值为零 (H'00) 的字节已被转移或转移的大小已经达到**大小 (size)** 的限制。返回的值是从所限的**大小 (size)** 减去实际转移的字节大小后的所得。

（C/C++ 程序）

```
#include <machine.h>
const char *s = "1234";
cahr d[100];
unsigned int remain;
void f(void)
{
        remain = movsd(d, s, 100);

}
```

←在 100 字节的限制内执行 MOVSD 指令。

（编译的汇编语言扩展代码）

```
_f:
            STM.L       (ER4-ER6),@-SP
            MOV.L       # d,ER6
            MOV.L       @ s:32,ER5
            MOV.W       #100:16,R4
            MOVSD.B     ($+4)
            MOV.W       R4,@ remain:32
            RTS/L       (ER4-ER6)
```

←设定从所提供**大小 (size)** 减去实际转移大小后所得的值。

说明与备注

(i) 这项函数仅在 CPU 为 H8SX 时有效。

(ii) **大小 (size)** 采用从零至 65535 的值。然而，若**大小**为零，它将被解释为 65536。

3.3 段地址运算符

描述

段地址可以使用编译程序所提供的 _ _sectop 及 _ _secend 运算符指定。

在编译程序的目标输出中，段地址通常无法被指定，因为段赋值目标未被定义。然而，有了 _ _sectop 和 _ _secend 运算符，您可以指定将要使用模块间优化工具 (Inter-Module Optimization Tool) 来设定在所连接程序内的段的最终地址。

这两个运算符可被指定如下：

[格式]

_ _sectop（“<段名称 (section name)>”）

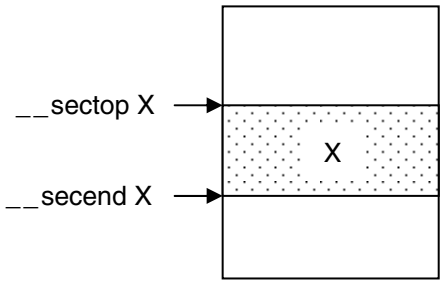
_ _secend（“<段名称 (section name)>”）

当段命名为 X 时，包含运算符的语句将被扩展如下：

__sectop(“X”) → STARTOF X

__secend(“X”) → STARTOF X+SIZEOF X

段状态的实例



STARTOF 和 SIZEOF 为汇编程序运算符。

STARTOF 决定段在被连接后设定的起始地址。

SIZEOF 决定段在被连接后设定的大小。

实例

要把段 X 的内容复制到段 Y：

（C/C++C/C++ 程序）

```
char *X_BGN;
char *X_END;
char *Y_BGN;
void func(void)
{
    char *p, *q;

    X_BGN=(char *)__sectop("X");
    X_END=(char *)__secend("X");
    Y_BGN=(char *)__sectop("Y");

    for (p=X_BGN,q=Y_BGN;p<X_END;p++,q++)
        *q = *p;
}
```

（编译的汇编语言扩展代码）

```
_func:
      STM.L       (ER4-ER5),@-SP
      MOV.L       #_X_END:32,ER4
      MOV.L       #STARTOF X:32,ER0
      MOV.L       ER0,@_X_BGN:32
      MOV.L       #STARTOF X+SIZEOF X:32,ER0
      MOV.L       ER0,@ER4
      MOV.L       #STARTOF Y:32,ER0
      MOV.L       ER0,@_Y_BGN:32
      MOV.L       @_X_BGN:32,ER1
      MOV.L       ER0,ER5
      BRA         L12:8
L11:
      MOV.B       @ER1,R0L
      MOV.B       R0L,@ER5
      INC.L       #1,ER1
      INC.L       #1,ER5
L12:
      MOV.L       @ER4,ER0
      CMP.L       ER0,ER1
      BCS         L11:8
      LDM.L       @SP+,(ER4-ER5)
      RTS
```

说明

若由段地址运算符所指定的段不存在，运算符将创建一个大小为 0 的段。这个段的属性为*数据*，其边界对齐为 2。

3.4 C++ 语言设定

除了 C 语言的设定之外，C++ 语言还需要下列设定：

3.4.1 设定 EC++ 类程序库

在 HEW1.2 中，除了标准程序库之外，C++ 语言还需要连接 EC++ 类程序库。和标准程序库的情形一样，取决于所使用的 CPU 类型、优化的目的，及所使用的参数传递寄存器数量，EC++ 类程序库必须如以下所示般选定。与标准程序库或编译程序选项的规格不相符的 EC++ 类程序库将无法被连接。

在 HEW2.0 或以上的版本中，标准程序库生成程序工具 (Standard Library Generator Tool) 应被用以创建 EC++ 类程序库。

从标准程序库 (Standard Library) 标签选取类别 (Category)：[标准程序库 (Standard Library)] EC++ 以进行设定。

CPU 系列：	操作模式：	程序库内容：	更改参数数量...	EC++ 类程序库
H8S/2600	普通	代码大小	2	ec226n.lib
		速度	2	ec226ns.lib
		代码大小	3	ec226n3.lib
		速度	3	ec226ns3.lib
	高级	代码大小	2	ec226a.lib
		速度	2	ec226as.lib
		代码大小	3	ec226a3.lib
		速度	3	ec226as3.lib
H8S/2000	普通	代码大小	2	ec226n.lib
		速度	2	ec226ns.lib
		代码大小	3	ec226n3.lib
		速度	3	ec226ns3.lib
	高级	代码大小	2	ec226a.lib
		速度	2	ec226as.lib
		代码大小	3	ec226a3.lib
		速度	3	ec226as3.lib
H8/300H	普通	代码大小	2	ec2hn.lib
		速度	2	ec2hns.lib
		代码大小	3	ec2hn3.lib
		速度	3	ec2hns3.lib
	高级	代码大小	2	ec2ha.lib
		速度	2	ec2has.lib
		代码大小	3	ec2ha3.lib
		速度	3	ec2has3.lib
H8/300	-	代码大小	2	ec2reg.lib
		速度	2	ec2regs.lib
		代码大小	3	ec2reg3.lib
		速度	3	ec2regs3.lib

CPU 系列：	操作模式：	程序库内容：	更改参数数量...	EC++ 类程序库
H8/300L	-	代码大小	2	ec2reg.lib
		速度	2	ec2regs.lib
		代码大小	3	ec2reg3.lib
		速度	3	ec2regs3.lib

3.4.2 更改初始化方法

在 C++ 语言中，初始设定必须依据下列所示进行修改：

下列描述阐释了使用 resetprg.c 文件，在 2.1.1 节，创建新的工作空间 2 (HEW2.0) 中所创建的工作空间内进行修改的方法：

```
#include <machine.h>
#include "stacksct.h"

#pragma entry PowerON_Reset
extern void main(void);
#ifdef __cplusplus
extern "C" {
#endif
extern void _INITSCT(void);
#ifdef   USES_SIMIO
extern void _INIT_IOLIB(void);
extern void _CLOSEALL(void);
#endif
#ifdef OTHERLIB
extern void _INIT_OTHERLIB(void);
#endif
#ifdef   HWSETUP
extern void HardwareSetup(void);
#endif
#ifdef __cplusplus
}
#endif

#pragma section ResetPRG
void PowerON_Reset(void);
void PowerON_Reset(void)
{
    set_imask_ccr(1);
    _INITSCT();
#ifdef   USES_SIMIO
   _INIT_IOLIB();
#endif
#ifdef OTHERLIB
    _INIT_OTHERLIB();
#endif
#ifdef   HWSETUP
   HardwareSetup();
#endif
    _call_init();
   main();
#ifdef   USES_SIMIO
   _CLOSEALL();
#endif

    _call_end();
   sleep();
}
```

：已添加

若静态数据存在于 C++ 程序中，这些函数将被调用。

_call_init 函数初始化存储了构造函数地址的 C++ 初始化数据区域，该地址的调用与全局类目标有关。

_call_end 函数初始化存储了析构函数地址的 C++ 后处理数据区域，该地址的调用与全局类目标有关。

这两项函数都在标准程序库中提供。

3.4.3 更改结构边界对齐

描述

可使用包选项或 #pragma pack1/#pragma pack2/#pragma unpack 来更改结构的边界对齐。

这些规格将如下所示更改边界对齐：

规格	#pragma pack1	#pragma pack2	#pragma unpack 或无
[无符号 (unsigned)]char	1	1	1
[无符号 (unsigned)] 短 (short)、[无符号 (unsigned)]int、[无符号 (unsigned)] 长 (long)、浮点类型、指针类型	1	2	指定的包选项
边界对齐值为 1 的结构、联合，及类。	1	1	1
边界对齐值为 2 的结构、联合，及类。	1	2	指定的包选项

更改边界对齐

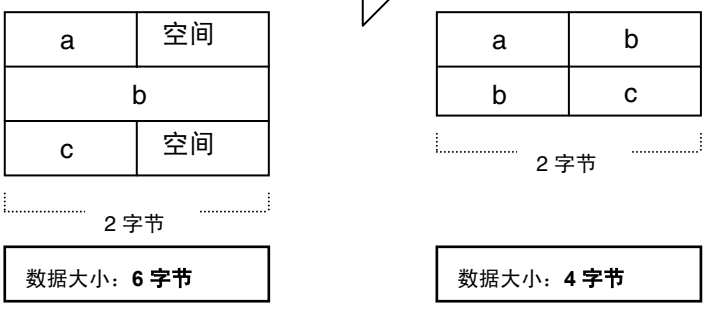
当指定了 #pragma pack1 时，除了 1 字节，数据可在奇数地址被分配以不为边界对齐制造空间。因此数据大小可能缩减。

（C/C++ 程序）

```
struct S1{
    char a;
    int b;
    char c;
}
```

```
#pragma pack 1
struct S1{
    char a;
    int b;
    char c;
}
```

（数据分配）



说明

由于更改边界对齐将缩减数据大小，因此对进行块转移等有用。然而，当指定了 #pragma pack1 时，它可能加大必需的存取代码，即分别以一字节存取结构的字或长字成员。

当 CPU 为 H8SX 时，在奇数地址对字或长字成员的字存取因设备规格而未发生地址错误。 因此这些成员可通过字或长字指令来进行存取。

结果，它不会加大必须的存取代码。
当 CPU 不是 H8SX 时，结构的成员不可通过指针存取，如下例所示。

（C/C++ 程序）

```
struct S {
          char x;
          int y;
} s;                    ← s.y 的地址可以是奇数
int *p=&s.y;
void test()
{
          s.y=1;        ← 已正确存取

          *p =7;        ← 可被不正确存取
}
```

3.5 编译程序版本 4.0 的新扩展函数

本节将说明新添加到编译程序版本 4.0 的扩展函数。

3.5.1 向量表自动生成函数

描述

通过指定 #pragma interrupt、#pragma inderect，及 #pragma entry 的向量号，函数的向量表将自动生成。

[格式]

#pragma interrupt（<函数名称 (function name)>[（vect=<向量号 (vector number)>）]）

#pragma inderect（<函数名称 (function name)>[（vect=<向量号 (vector number)>）]）

#pragma entry（<函数名称 (function name)>[（vect=<向量号 (vector number)>）]）

实例

要指定一个向量号以创建向量表。

（C/C++ 程序）

(CPU=2600a)

```
#pragma entry f1(vect=0)
void f1(){
}
#pragma interrupt (f2(vect=4))
void f2(void){
}
#pragma indirect (f3(vect=5))
unsigned char f3(void){
}
```

← 分配项目函数 f1 至向量号 0。

← 分配中断函数 f2 至向量号 4。

← 分配间接存储器存取函数 f3 至向量号 5。

（存储器映像内容）

```
$VECT0    00000000  00000003
$VECT4    00000010  00000013
$VECT5    00000014  00000017
```

说明与备注

(i) 向量表规格 “vect=” 必须始终使用小写字符指定。
(ii) 必须谨慎所分配的向量号不与其他向量表重复。

3.5.2 指定参数传递寄存器的数量

描述

可为每项函数指定参数传递寄存器的数量。

以 _ _regparam2 指定的函数使用 ER0、ER1（在 H8/300 为 R0 和 R1），及以 _ _regparam3 指定的函数使用 ER0、ER1、ER2（在 H8/300 为 R0、R1，和 R2）。

[格式]

<类型说明符 (type specifier)> _ _ regparam2 <函数名称 (function name)>

<类型说明符 (type specifier)> _ _ regparam3 <函数名称 (function name)>

实例

这项函数指定一个变量以将它存储到堆栈或分配到 ER2。

（C/C++ 程序）

```
void _ _ regparam2 func1(long a, int b, int c, long d);
void _ _ regparam3 func2(long a, int b, int c, long d);
void main(void)
{
          ::
      func1(a,b,c,d);
          :
          :
          :
      func2(a,b,c,d);
          :
          :
          :
}
```

```
Variable allocation patterns
(CPU=2600a)
func1
      long a   :ER0
      int b    :E1
      int c    :R1
      long d   :stack
func2
      long a   :ER0
      int b    :E1
      int c    :R1
      long d   :ER2
```

说明与备注

(i) 这项函数仅支持关键字规格。

(ii) 使用编译程序 CPU 选项 regparam=3 时，参数传递寄存器对所有函数使用 ER0、ER1、ER2（在 H8/300 为 R0、R1，和 R2）。

3.5.3 偶数字节存取规格函数

描述

这项函数允许 2 或 4 字节的标量类型变量/常量始终以偶数字节存取（不以字节存取）。

[格式]

_ _ evenaccess <类型说明符 (type specifier)> <变量名称 (variable name)>

<类型说明符 (type specifier)> _ _ evenaccess <变量名称 (variable name)>

实例

（C/C++ 程序）

```
#define A (*(volatile unsigned short __evenaccess
*)0xff01178)
void test(void)
{
   A &= ~0x2000 ;
}
```

（编译的汇编语言扩展代码）

_ _ evenaccess 未指定

```
_test:
       BCLR.B     #5,@15733112:32
       RTS
```

_ _ evenaccess 已指定

```
_test:
       MOV.W      @15733112:32,R0
       BCLR.B     #5,R0H
       MOV.W      R0,@15733112:32
       RTS
```

在字指令内存取

说明与备注

(i) 在 H8/300 中，函数允许以 2 字节存取。
(ii) 这项函数仅支持关键字规格。

3.6 编译程序版本 6.0 的新扩展函数

本节将说明新添加到编译程序版本 6.0 的扩展函数。

3.6.1 位字段顺序规格功能

描述

#pragma bit_order、bit_order 选项可指定位字段成员的顺序。

有时候不同的 CPU 有不同的位字段顺序规则 (Bit Field Order Rules)，这项函数将提升程序在不同 CPU 之间的兼容性。当此选项被省略时，BIt_order = Left 被选定。

规格方法

(1) 扩展函数格式

#pragma bit_order（左 (left)| 右 (right)）

(2) 选项

BIt_order = {左 (Left)| 右 (Right)}

实例

切换位字段赋值的顺序，如下例所示。

当指定了左时，位字段成员将从最高有效位侧分配。

当指定了右时，位字段成员将从最低有效位侧分配。

若 #pragma bit_order 的指定不包含左或右说明符，bit_order 选项的解释在行下有效。

（C/C++ 程序）

分配自最高有效位

```
#pragma bit_order left
struct {
        unsigned char a:2;
        unsigned char b:3;
}x;
void func(void)
{
        x.a = 3;
        x.b = 5;
}
```

分配自最低有效位

```
#pragma bit_order right
struct {
        unsigned char a:2;
        unsigned char b:3;
}x;
void func(void)
{
        x.a = 3;
        x.b = 5;
}
```

（分配的数据顺序）

3.7 编译程序版本 6.1 的新扩展函数

本节将说明新添加到编译程序版本 6.1 的选项与扩展函数。

3.7.1 legacy=v4

描述

当指定了此选项时，C/C++ 编译程序版本 6.1 将输出使用版本 4.0 的相同方法来进行优化的目标。

这对依赖时序的处理有用，因为目标不会不同于版本 4.0。

当此选项未被指定时，目标具有比版本 4.0 程度更高的优化。

指定方法

命令行： *legacy = v4*

备注与说明

当 CPU 类型为 2600A、2600N、2000A 或 2000N 时，此选项有效。

当指定了 **legacy=v4** 时，下列选项不可用。

opt_range, del_vacant_loop, max_unroll, infinite_loop, global_alloc, struct_alloc, const_var_propagate, volatile_loop, scope, strict_ansi, file_inline, file_inline_path, enable_register

3.7.2 cpuexpand=v6

描述

cpuexpand 选项通过扩展 ANSI 标准的解释来为变量生成乘法及除法代码。

因此，当 CPU 类型为 2600A、2600N、2000A 或 2000N 时，C/C++ 编译程序版本 4.0 与 6.0 或以上版本之间通过指定 **cpuexpand** 选项所生成的目标可能不同。

若这项区别造成一些不合要求的结果，请使用 **cpuexpand=v6** 选项。 **cpuexpand=v6 选项**的目标没有任何区别，因此不会造成不合要求的结果。

指定方法

命令行： *cpuexpand = [v6]*

受影响的表达式

(a) signed long	=	signed int	<<	constant	
(b) signed long	=	unsigned int	<<	constant	
(c) unsigned long	=	signed int	<<	constant	
(d) unsigned long	=	unsigned int	<<	constant	
(e) signed int	= (	signed int	<<	constant)	/ signed int
(f) signed int	= (unsigned int		<<	constant)	/ signed int
(g) signed int	= (unsigned int		<<	constant)	/ unsigned int
(h) unsigned int	= (	signed int	<<	constant)	/ signed int

(i) unsigned int = (unsigned int << constant) / signed int

(j) unsigned int = (unsigned int << constant) / unsigned int

代码例子

(unsigned signed long = unsigned signed int << constant) 的例子

-cpuexpand- legacy=v4

```
MOV.W       @_i1:32,R0
MOV.W       #1024,E0
MULXS.W     E0,ER0
MOV.L       ER0,@_l1:32
```

移位结果被存储到 unsigned long。

-cpuexpand=V6 -legacy=v4

```
MOV.B       @_i1+1:32,R0H
SUB.B       R0L,R0L
SHLL.W      #2,R0
EXTU.L      ER0
MOV.L       ER0,@_l1:32
```

移位结果被零扩展，并存储到 unsigned long。

备注与说明

当 CPU 类型为 2600A、2600N、2000A 或 2000N，并指定了 **legacy=v4** 时，此选项有效。

3.7.3 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。

指定方法

-enable_register

使用的实例：

```
int g_i1;
void func()
{
register long Reg_l1 = 999;
long l2 = 126;
long l3 = 248;

        switch(g_i1){
        case 2:
                Reg_l1++;
        break;
        case 3:
            l2 += 5;
        break;
        case 4:
            l2 += 7;
        break;
        case 9:
                l3 -= 11;
        break;
        case 10:
                l3 -= 19;
        break;
        }
    printf("%d,%d,%d\n",Reg_l1,l2,l3); //由于‘Reg_l1’的值通过 ER1 传递到 printf，将 ER1 分配到‘Reg_l1’将可改进效率。
}
```

代码例子

由于变量 Reg_l1 具有较高的优先级，ER1 被分配。

-enable_register not specified

```
_func:
     STM.L       (ER4-ER6),@-SP
     SUB.W       #8:16,R7
     MOV.L       #H'000003E7,ER5
     SUB.L       ER6,ER6
     MOV.B       #H'7E:8,R6L
     SUB.L       ER4,ER4
     MOV.B       #H'F8:8,R4L
     MOV.W       @_g_i1:16,R0
     MOV.W       R0,R1
     MOV.B       R0H,R0H
     BNE         L26:8
     CMP.B       #2:8,R1L
     BEQ         L27:8
     CMP.B       #3:8,R1L
     BEQ         L28:8
     CMP.B       #4:8,R1L
     BEQ         L29:8
     CMP.B       #9:8,R1L
     BEQ         L30:8
     CMP.B       #H'0A:8,R1L
     BNE         L26:8
     MOV.B       #H'E5:8,R4L
     BRA         L26:8
L30:
     MOV.B       #H'ED:8,R4L
     BRA         L26:8
L29:
     MOV.B       #H'85:8,R6L
     BRA         L26:8
L28:
     MOV.B       #H'83:8,R6L
     BRA         L26:8
L27:
     MOV.B       #H'E8:8,R5L
L26:
     MOV.W       #LWORD L45:16,R0
     MOV.L       ER6,@SP
     MOV.L       ER4,@(4:16,SP)
     MOV.L       ER5,ER1
     JSR         @_printf:16
     ADD.W       #8:16,R7
     LDM.L       @SP+,(ER4-ER6)
     RTS
```

-enable_register

```
_func:
     STM.L       (ER4-ER6),@-SP
     SUB.W       #8:16,R7
     MOV.L       #H'000003E7,ER1
     SUB.L       ER4,ER4
     MOV.B       #H'7E:8,R4L
     SUB.L       ER6,ER6
     MOV.B       #H'F8:8,R6L
     MOV.W       @_g_i1:16,R0
     MOV.W       R0,R5
     MOV.B       R0H,R0H
     BNE         L26:8
     CMP.B       #2:8,R5L
     BEQ         L27:8
     CMP.B       #3:8,R5L
     BEQ         L28:8
     CMP.B       #4:8,R5L
     BEQ         L29:8
     CMP.B       #9:8,R5L
     BEQ         L30:8
     CMP.B       #H'0A:8,R5L
     BNE         L26:8
     MOV.B       #H'E5:8,R6L
     BRA         L26:8
L30:
     MOV.B       #H'ED:8,R6L
     BRA         L26:8
L29:
     MOV.B       #H'85:8,R4L
     BRA         L26:8
L28
     MOV.B       #H'83:8,R4L
     BRA         L26:8
L27:
     MOV.B       #H'E8:8,R1L
L26:
     MOV.W       #LWORD L45:16,R0
     MOV.L       ER4,@SP
     MOV.L       ER6,@(4:16,SP)

     JSR         @_printf:16
     ADD.W       #8:16,R7
     LDM.L       @SP+,(ER4-ER6)
     RTS
```

备注与说明

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。

此选项在 CPU 类型为 H8SX 或 H8S 时有效。

3.7.4 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

```
_main:
          MOV.L       #H'0A:8,@_io:16
          RTS
          .SECTION    $ADDRESS$B100,DATA,LOCATE=H'100
_io:
          .RES.L      1
          .END
```

重要信息

此选项在 CPU 类型为 H8SX 或 H8S 时有效。

(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。
(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 abs8/abs16

#pragma global_register

3.7.5 文件间内联扩展

描述

对每个文件执行 C/C++ 编译程序。 因此，若函数被跨文件调用，内联扩展将不会被应用到该函数，即使在函数中指定了用于内联扩展的 **–speed=inline** 选项、**#pragma inline** 或 **inline** 关键字。

当指定了文件间内联扩展选项时，内联扩展则可被应用到该函数，即使函数是跨文件被调用。

若要进行内联扩展的函数位于其他目录的文件上，可通过指定要进行文件间内联扩展的函数所在目录，将内联扩展应用到该函数。

指定方法

文件间内联扩展

对话框菜单： **C/C++** 标签**类别：(C/C++** Tab **Category:) [优化 (Optimize)] [详细资料… (Details ...)][内联 (Inline)][内联文件路径 (Inline file path)]**

命令行： *FILe_inline=<file name>[,...]*

文件间内联扩展目录指定

对话框菜单： **C/C++** 标签**类别：(C/C++** Tab **Category:) [源 (Source)] [显示有关项目：(Show entries for :)][文件内联路径 (File inline path)]**

命令行： *file_inline_path=<path name> [,...]*

文件间内联扩展的文件按照 [文件内联路径] 目录和当前目录的顺序搜索。

使用的实例

在下列例子中，指定了关键字 **inline** 的函数 **func** 从其他文件被调用。

为进行内联扩展，在编译调用源函数 **test_1.c** 时指定文件间内联扩展选项。

（C/C++ 程序）

ch38 –cpu=h8sxa **–file_inline=test_2.c** test_1.c

ch38 –cpu=h8sxa test_2.c

```
[test_1.c]
void main(void);
void func(void);
int si1,si2;
void main(void)
{
    func();
}
```

```
[test_2.c]
extern int si1,si2;
void func(void);
__inline void func(void)
{
    si1 = 10;
    si2 = 20;
}
```

代码例子

当指定了文件间内联扩展选项时，**test_2.c** 的代码在调用源函数中被扩展。

未指定

```
[test_1.c]
_main:
    JMP         @_func:24
```

```
[test_2.c]
_func:
    MOV.W       #H'A:4,@_si1:32
    MOV.W       #H'14:8,@_si2:32
    RTS
```

已指定

```
[test_1.c]
_main:
    MOV.W       #H'A:4,@_si1:32
    MOV.W       #H'14:8,@_si2:32
    RTS
```

```
[test_2.c]
_func:
    MOV.W       #H'A:4,@_si1:32
    MOV.W       #H'14:8,@_si2:32
    RTS
```

备注与说明

此选项仅在 CPU 类型为 H8SX 或 H8S 时有效（不包含 legacy=v4 选项）。

(1) 当指定了此选项时，内联扩展仅应用到在 <文件名> 的指定文件中指定了 **#pragma inline** 或关键字 **inline** 的函数。 若同时指定了 **–speed=inline** 选项，内联扩展将被应用到文件中所有可能的函数。

(2) 若一个全局函数在此选项所指定的一些文件中被定义了两次或以上，将无法保证操作。
（为内联扩展使用任意选定的单一函数定义）

(3) 不可省略使用 <文件名> 指定的文件名之扩展名。

(4) 不可为 <文件名> 指定通配符（* 或 ?）。

(5) 若文件具有 #pragma asm-endasm、#pragma inline_asm 或 __asm，它将不会被扩展。

3.7.6 分割优化范围

描述

当指定了分割优化范围选项时，编译程序将把大型函数的优化范围分割成数块。当分割优化范围选项未被指定时，编译程序不会分割优化范围。

当优化范围被扩展时，虽然编译时间较长，但整体目标性能获得增进。不过，若寄存器不足，目标性能可能不会被增进。

使用此选项进行性能调谐，因为它视程序而定，会对目标性能造成影响。

指定方法

对话框菜单： 无

命令行：: *SCOpe*
: *NOSCope*

使用的实例

下列例子显示具有 1000 个变量声明，同时已设定每个变量的 C 程序 ROM 大小。 当指定了 **noscope** 选项时，ROM 大小缩减了 6 字节。

当指定了消息选项* 及分割优化范围的范围选项时，将输出下列消息。

注意： * 在 HEW 中，**C/C++** 标签**类别：(C/C++** Tab **Category:) [源 (Source)] [显示有关项目：(Show entries for :)][消息 (Messages)][显示信息级别消息 (Display information level messages)]**

（C/C++ 程序）

cpu=h8sxa

scope 被指定　　8010 字节
noscope 被指定　8004 字节

（消息）

C0101 (I) Optimizing range divided in function "function name"（函数“函数名称”的优化范围被分割）

备注与说明

此选项仅在 CPU 类型为 H8SX 或 H8S 时有效（不包含 legacy=v4 选项）。

3.8 H8SX 的功能

3.8.1 地址空间

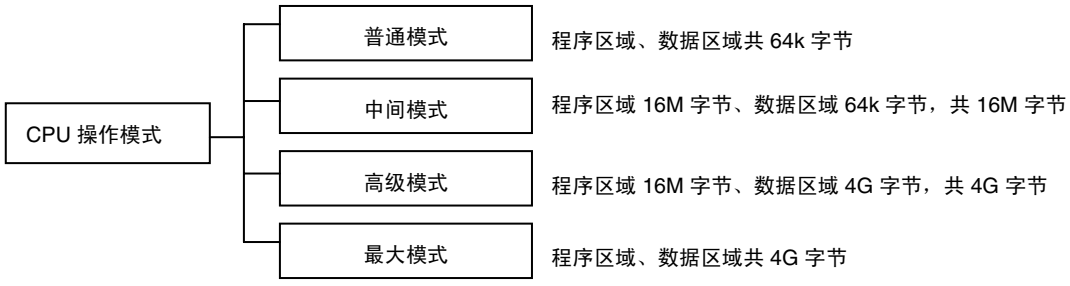
描述

H8/300、H8/300H、H8S/2000、H8S/2600 最多具有两种 CPU 模式，H8SX 具有下列四种 CPU 操作模式。

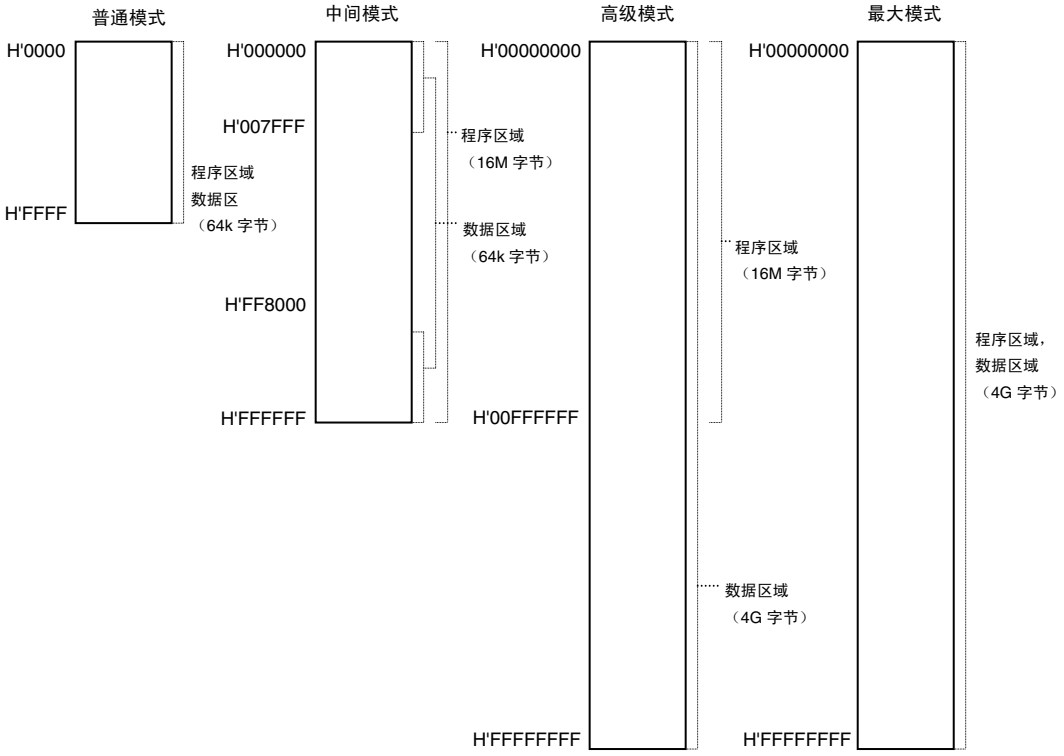
每种模式以 LSI 的模式选择引脚或其他源来选定。

在进行编译时，则将通过指定 CPU 类型和操作模式选项来选定。

由于可用的模式和区域因产品而异，请在指定 CPU 模式时参考硬件手册。



地址空间



3.8.2 指定 8 位绝对地址空间

描述

若数据以 8 位绝对地址空间来进行分配及存取，则对象可以是小型和高速的。

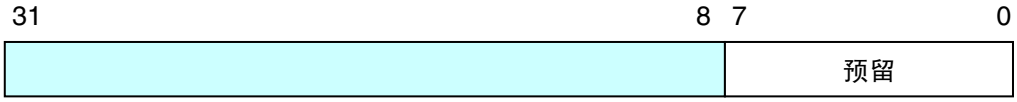
当 CPU 是 H8SX 时，用户可以修改这个 8 位绝对地址空间的存取范围。

在旧的 H8 系列中，8 位绝对地址空间固定在 H’FFFF00 至 H’FFFFFF 之间，并重复内部 I/O 空间。

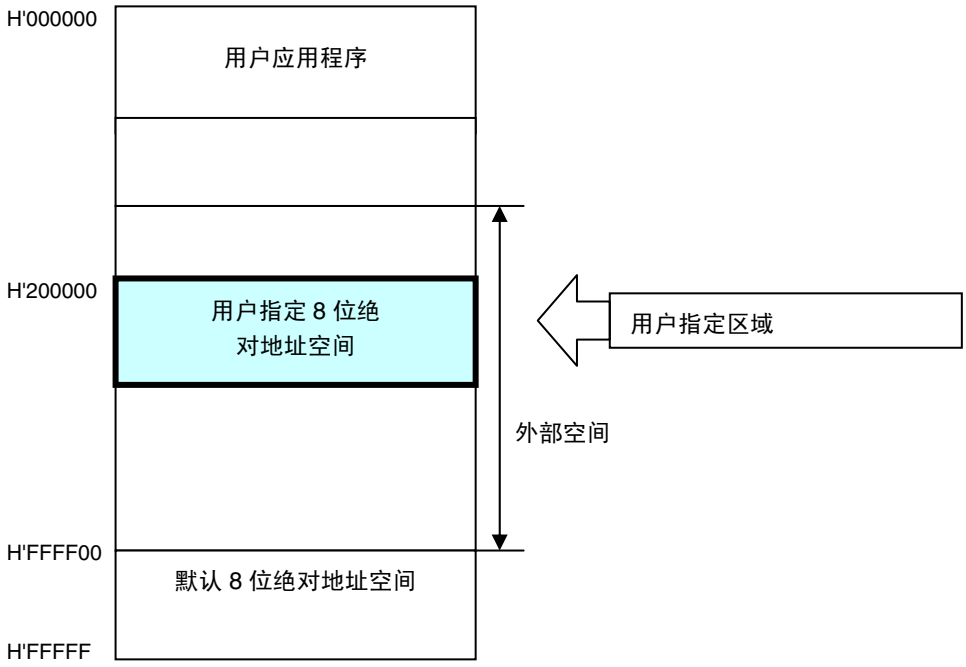
在 H8SX 中，任何 SBR（短地址基址寄存器，Short Address Base Register）指定地址的 256 字节区域，被设定为 8 位绝对地址空间。

寄存器格式

SBR（短地址基址寄存器，Short Address Base Register）是上端 24 位有效的 32 位寄存器。下端八位为预留且读取为 0。



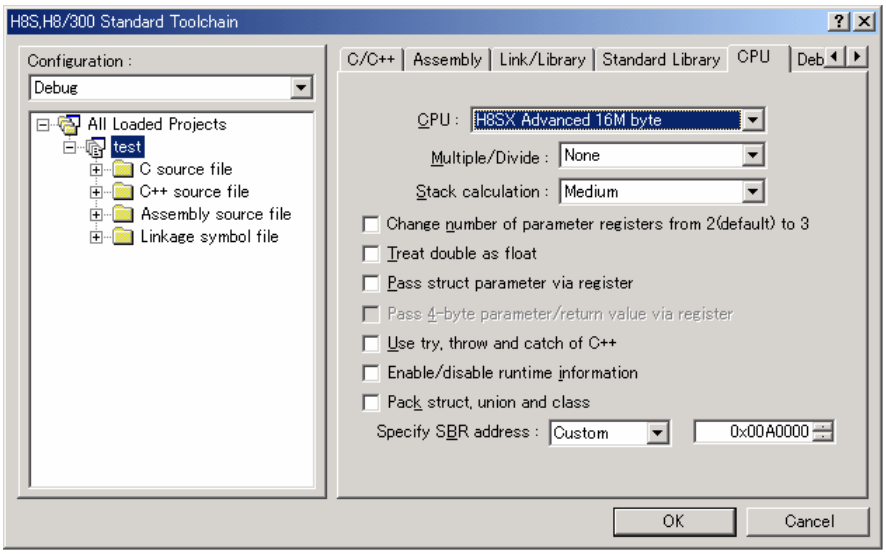
8 位绝对地址空间



规格方法

对话框菜单： **CPU 标签，指定 SBR 地址**

命令行：*sbr = <地址 (address)>*



实例

由于 SBR（短地址基址寄存器，Short Address Base Register）无法从 C/C++ 语言直接存取，SBR 应以汇编指令编写。

但通过如下所示使用编译程序扩展函数 **_asm**，汇编语言指令可以使用 C/C++ 语言编写。

虽然汇编语言指令可在旧版本的编译程序中以 pragma_asm 进行编写，但应在编译后将它们转换为汇编源代码。

以 **_asm** 块编写的汇编程序可被直接编译成目标文件，以便符号可在源代码级调试程序中被援引。

要获取有关 **_asm** 的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”10.2.1(3) 节。

（C/C++ 程序）

```
__abs8 unsigned char a,b;
void main(void)
{
__asm{
    mov.l #0xa0000,er0
ldc.l er0,sbr
}
a = 7;
b = -1;
}
```

← _abs8 的 8 位绝对地址

← 设定 0xa0000 至 SBR

← 从 0xa0000 存取 8 位绝对地址

段分配

8 位绝对地址空间以 _abs8 声明，输出为 $ABS8B 段。上例中，优化连接编辑程序在 0xa0000 分配 $ABS8B 段。

C/C++ 程序的例子

当使用 HEW 时，在 **resetprg.c** 中删除下列粗体部分的注解，以使它们可用。

当不使用 HEW 时，在初始例程中添加相同表达式。

```
__entry(vect=0) void PowerON_Reset(void)
{
//在您进行 H8SX 的 SBR/VBR 初始设定时移除注解
        __asm{
                mov.l   #0xa0000,er0
                ldc.l   er0,sbr
                mov.l   #0x00000000,er0
                ldc.l   er0,vbr
        }
         set_imask_ccr(1);
        _INITSCT();
            :
```

（8 位绝对地址空间**未使用**）

```
unsigned char c1,c2;
void main(void)
{
        c1 = 7;
        c2 = -1;
}
```

（8 位绝对地址空间**已使用**）

```
__abs8 unsigned char c1,c2;
void main(void)
{
        c1 = 7;
        c2 = -1;
}
```

汇编扩展代码的例子

H8SX 高级模式 16M 的例子

（8 位绝对地址空间**未使用**）

```
_main:
    MOV.B       #7:4,@_c1:32
    MOV.B       #255:8,@_c2:32
    RTS
```

（8 位绝对地址空间**已使用**）

存取 0xa0000 - 0xa00FF

```
_main:
MOV.B       #7:8,R0L
MOV.B       R0L,@_c1:8
MOV.B       #255:8,R0L
MOV.B       R0L,@_c2:8
RTS
```

目标大小 [字节]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
增进前	16	16	12
增进后	10	10	10

执行速度 [周期]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
增进前	12	12	11
增进后	11	11	11

3.8.3 切换向量表地址

描述

H8SX 具有可在任何地址为异常处理分配向量区域的函数。

在 H8/300、H8/300H、H8S 系列中，异常处理的向量区域固定为从零起。

当 CPU 为 H8SX 时，用户可通过指定向量基址寄存器 (VBR) 来修改用于异常处理的向量区域分配地址。

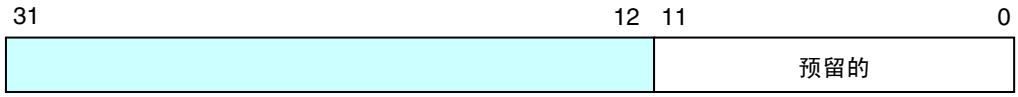
向量基址寄存器的优点

向量基址寄存器 (VBR) 可在任何地址查找异常处理向量区域，向量表可使用快速内部 RAM 制造，即使是没有内部 ROM 的芯片。

这将可增进异常处理的响应。

寄存器格式

向量基址寄存器 (VBR) 是高 20 位有效的 32 位寄存器。 这个寄存器的低 12 位为预留且读取为 0。



使用编译程序版本 6.1 设定向量基址寄存器

内建函数 **set_vbr** 的向量基址寄存器可使用 C/C++ 语言设定。

有关详细资料，请参考 3.2.3 节，设定向量基址寄存器。

编译程序版本 6.0 中的例子

在编译程序版本 6.0 中，向量基址寄存器 (VBR) 无法从 C/C++ 语言直接存取，因此它应以汇编指令编写。

但通过如下所示使用编译程序扩展函数 **_asm**，汇编语言指令可使用 C/C++ 语言编写。

虽然汇编语言指令可在旧版本的编译程序中以 **#pragma_asm** 进行编写，但应在编译后将它们转换为汇编源代码。

以 **_asm** 块编写的汇编程序可直接编译成目标文件，以使符号可在源代码级调试程序中被援引。

要获取有关 **_asm** 的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”10.2.1(3) 节。

切换向量基址寄存器 (VBR) 应在中断屏蔽 (interrupt mask) 状态中完成。
若不在中断屏蔽状态中进行，当切换向量基址寄存器 (VBR) 期间发生中断处理时，将无法保证异常处理的正确运行。

（C/C++ 程序）

```
void main(void)
{
__asm{
        orc.b #0x80,ccr
        mov.l #0xffa000,er0
        ldc.l er0,vbr
        andc.b #0x7F,ccr
        RTS
}
}
```

← 设定中断屏蔽位

← 清除中断屏蔽位

H8S, H8/300 系列 C/C++编译程序应用笔记

HEW

第 4 节 HEW

本节描述当使用 HEW1.2 或 2.0 或以上版本时，受 C/C++ 编译程序、汇编程序、模块间优化器、目标转换器，及库管理程序支持的选项画面与命令选项之间的关系。

要获取各个选项的详细资料，请参考适当的用户手册。（要获取模块间优化器所支持的选项的详细资料，请参考 H8S，H8S/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8S/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 中的描述。）

HEW1.2 中的各个选项画面通过下列方法选取。

工具名称	选取方法
C/C++ 编译程序	**[选项 (Options) -> H8S,H8/300 C/C++ 编译程序 (H8S,H8/300 C/C++ Compiler) …]**
交叉汇编程序	**[选项 (Options) -> H8S,H8/300 汇编程序 (H8S,H8/300 Assembler) …]**
模块间优化器	**[选项 (Options) -> H8S,H8/300 IM Optlinker...]**
目标转换器	**[选项 (Options) -> H 系列 S 类型转换器 (H Series Stype Converter) …]**
库管理程序	**[选项 (Options) -> H 系列库管理程序 (H Series Librarian) …]**

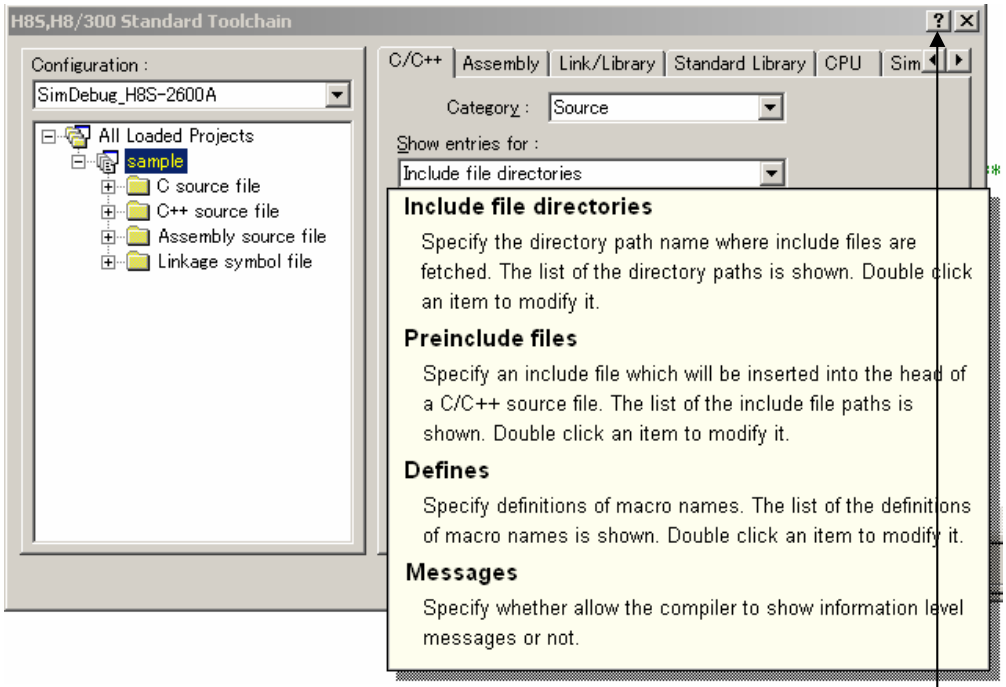
注意： 若在选项 (Options) 菜单内未检测到适当的工具，则可使用 [选项 (Options) -> 创建阶段 (Build Phases) …] 来添加工具。

在 HEW2.0 或以上版本内，从选项 (options) 菜单选取 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) …。

在 HEW4.0 或以上版本内，从创建 (build) 菜单选取 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) …。

工具名称	选取方法
C/C++ 编译程序	**[选项 (Options) -> H8S,H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) … -> C/C++ 标签 (C/C++ Tab)]**
交叉汇编程序	**[选项 (Options) -> H8S,H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) … -> 汇编标签 (Assembly Tab)]**
优化连接编辑程序	**[选项 (Options) -> H8S,H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) … -> 连接/程序库标签 (Link/Library Tab)]**
标准程序库生成程序	**[选项 (Options) -> H8S,H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) … -> 标准程序库标签 (Standard Library Tab)]**
CPU 选项	**[选项 (Options) -> H8S,H8/300 标准工具链 (H8S,H8/300 Standard Toolchain) … -> CPU 标签 (CPU Tab)]**

另外，“帮助”可从每个选项画面上参考。



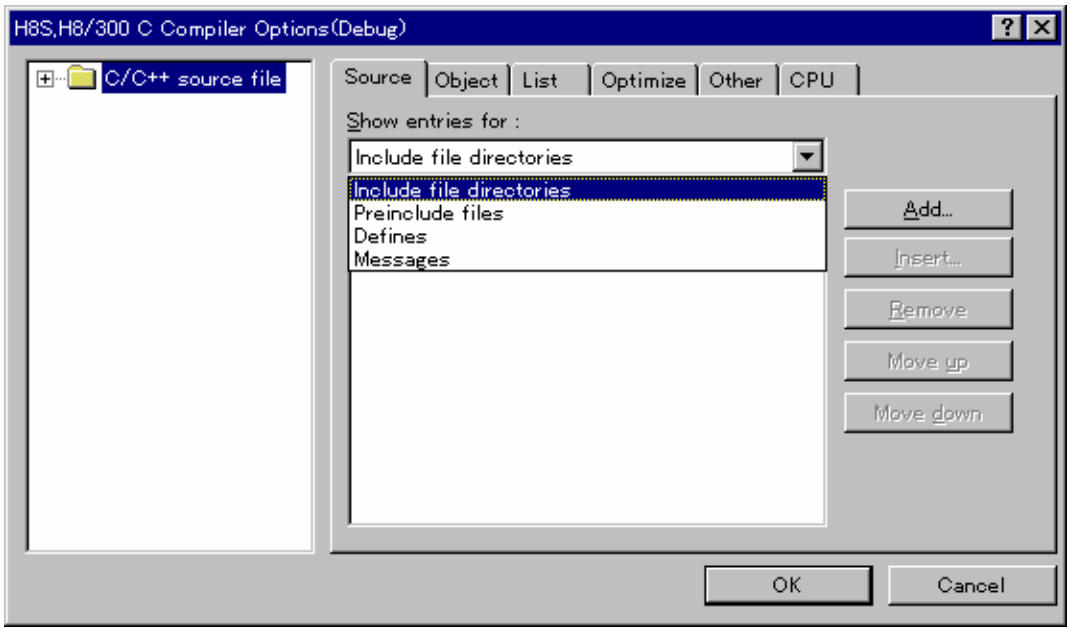
若单击右上角的 [?]，然后再单击想要参考的项目，则类似以上所示的描述将显示。
请使用这项帮助功能以获取快速参考。

4.1 在 HEW1.2 中指定选项

要获取在 HEW2.0 或以上版本中指定选项的详细资料，请参考 4.2 节，在 HEW2.0 或以上版本中指定选项。

4.1.1 C/C++ 编译程序选项

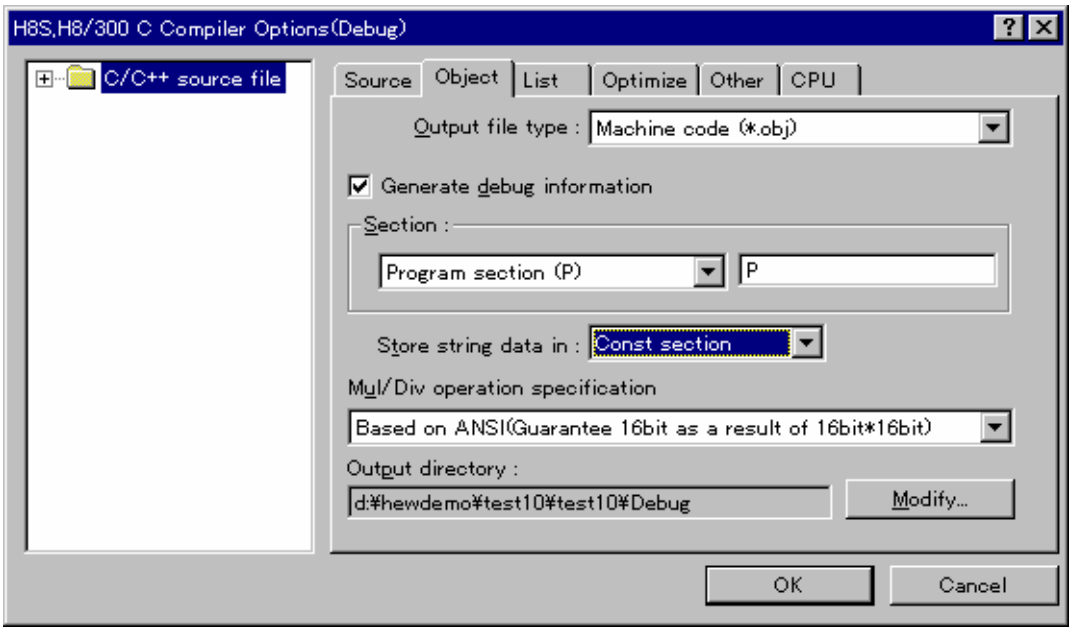
(1) 源 (Source) 标签



显示有关项目 (Show entries for)：

对话框菜单	命令选项	功能
包含文件目录 (Include file directories)	*include*	指定包含文件目录的路径名称
预包含文件 (Preinclude files)	*preinclude*	在编译单元的初始，将文件内容指定为包含文件
定义 (Defines)	*define*	定义宏名称
消息 (Messages)	*message*	输出信息消息

(2) 目标 (Object) 标签



输出文件类型 (Output file type)：

对话框菜单	命令选项	功能
机器码 (*.obj) **(Machine code (*.obj))**	*code=machinecode*	输出机器语言程序
汇编源代码 (*.src) **(Assembly source code (*.src))**	*code=asmcode*	输出汇编语言程序
预处理的源文件 (*.p/*.pp) **(Preprocessed source file (*.p/*.pp))**	*preprocessor*	在预处理程序扩展后输出源程序

生成调试信息 (Generate debug information)

复选框	命令选项	功能
√	*debug*	输出调试信息
	nodebug	不输出调试信息

段 (Section)：

对话框菜单	命令选项	功能
-	*section*	更改默认段名称

将字符串数据存储在 (Store string data in)：

对话框菜单	命令选项	功能
常数段 (Const section)	*string=const*	将字符串文字输出到常数区
数据段 (Data section)	*string=data*	将字符串文字输出到初始化数据区

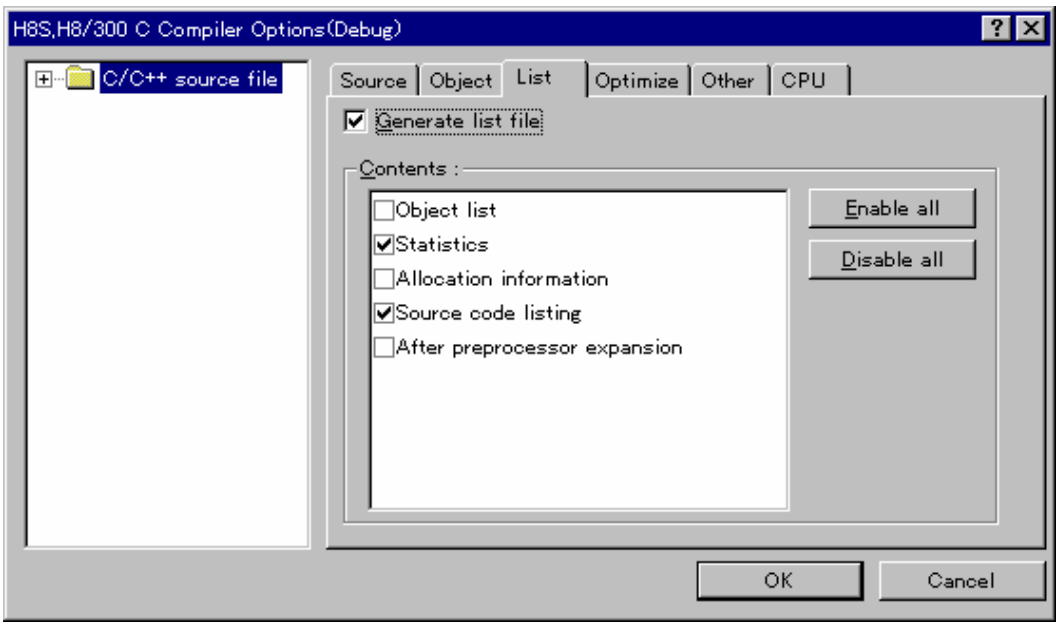
乘除运算规格 (Mul/Div operation specifications)

对话框菜单	命令选项	功能
基于 ANSI（保证 16 位为 16 位* 16 位的结果） (Based on ANSI (Guarantee 16bit as a result of 16bit*16bit))	*nocpuexpand*	根据 ANSI C 语言规格，以代码开发乘法或除法
非 ANSI（保证 32 位为 16 位* 16 位的结果） (Non ANSI (Guarantee 32bit as a result of 16bit*16bit))	*cpuexpand*	根据 CPU 指令规格，以代码开发乘法或除法

输出目录 (Output directory)

对话框菜单	命令选项	功能
-	*object*	指定目标文件输出目录

(3) 列表 (List) 标签



生成列表文件 (Generate list file)

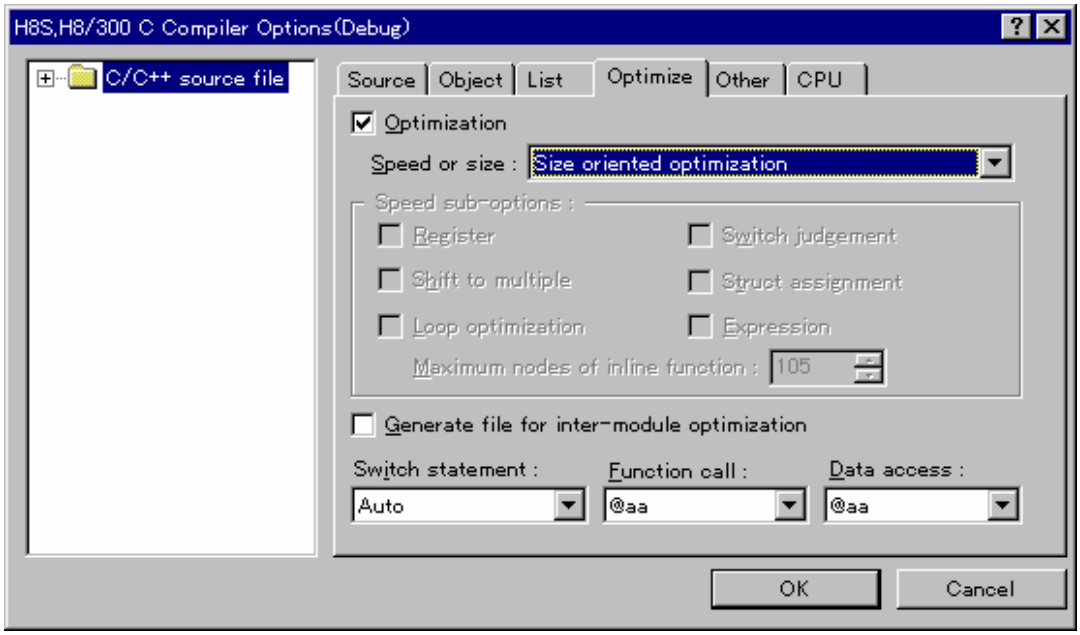
复选框	命令选项	功能
√	*list*	输出目标列表文件
	nolist	不输出目标列表文件

内容 (Contents)：指定要输出到目标文件列表的数据。

对话框菜单	命令选项	功能
目标列表 (Object list)	*show=object*	输出目标列表
统计数据 (Statistics)	*show=statictics*	输出统计数据信息
分配信息 (Allocation information)	*show=allocation*	输出符号分配列表
源代码列表 (Source code listing)	*show=source*	输出源列表
预处理程序扩展后 (After preprocessor expansion)	*show=expand*	输出宏扩展后的列表

若按下 [全部允许 (Enable all)] 按钮，所有数据项目将被输出。 否则，若按下 [全部禁止 (Disable all)] 按钮，所有数据项目将被禁止，且不输出任何数据项目到目标列表文件。

(4) 优化 (Optimize) 标签



优化 (Optimization)

复选框	命令选项	功能
☑	*optimize=1*	指定优化
☐	*optimize=0*	不指定优化

速度或大小 (Speed or size)： 指定优化的格式。

对话框菜单		命令选项	功能
面向大小的优化 (Size oriented optimization)		-	对大小执行优化
面向速度的优化 (Speed oriented optimization)		*speed*	对速度执行优化
速度子选项 (Speed sub-options)	**寄存器 (Register)**	*speed=register*	通过 PUSH 和 POP 指令以更快的速度执行寄存器存储/恢复扩展
	切换判断 (Switch judgement)	*speed=switch*	以更快的速度开发切换语句
	转移到多个 (Shift to multiple)	*speed=shift*	以更快的速度开发转移操作
	结构赋值 (Struct assignment)	*speed=struct*	以更快的速度执行结构扩展及替换表达式
	循环优化 (Loop optimization)	*speed=loop*	以更快的速度开发循环语句
	表达式 (Expression)	*speed=expression*	以更快的速度执行算术操作、比较，及替换表达式处理
	内联函数的最大节点 (Maximum nodes of inline function)	*speed=inline [=<data>]*	以更快的速度自动执行内联扩展

为模块间优化生成文件 (Generate file for inter-module optimization)

复选框	命令选项	功能
☑	*goptimize*	输出模块间优化的附加信息文件
☐	-	不输出模块间优化的附加信息文件

切换语句 (Switch statement)： 指定切换语句扩展方法。

对话框菜单	命令选项	功能
自动 (Auto)	*case=auto*	根据速度选项规格来决定切换语句扩展方法
如果…就 (If then)	*case=ifthen*	以“如果…就”的方法来执行切换语句扩展
表 (Table)	*case=table*	以表跳转 (table jump) 的方法来执行切换语句扩展

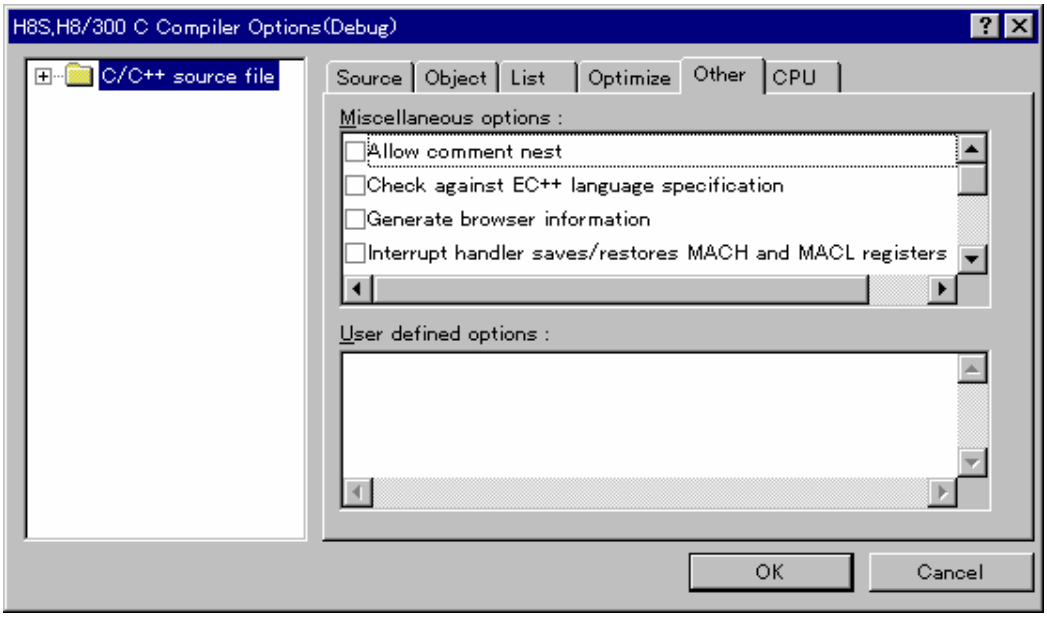
函数调用 (Function call)： 选择函数调用的方法。

对话框菜单	命令选项	功能
@aa	-	选取普通函数调用
@@aa:8	*indirect*	选取存储器间接函数调用

数据存取 (Date access)： 选择数据存取模式。

对话框菜单	命令选项	功能
@aa	-	选取普通数据存取
@aa:8	*abs8*	选取 8 位绝对地址存取
@aa:16	*abs16*	选取 16 位绝对地址存取

(5) 其他 (Other) 标签

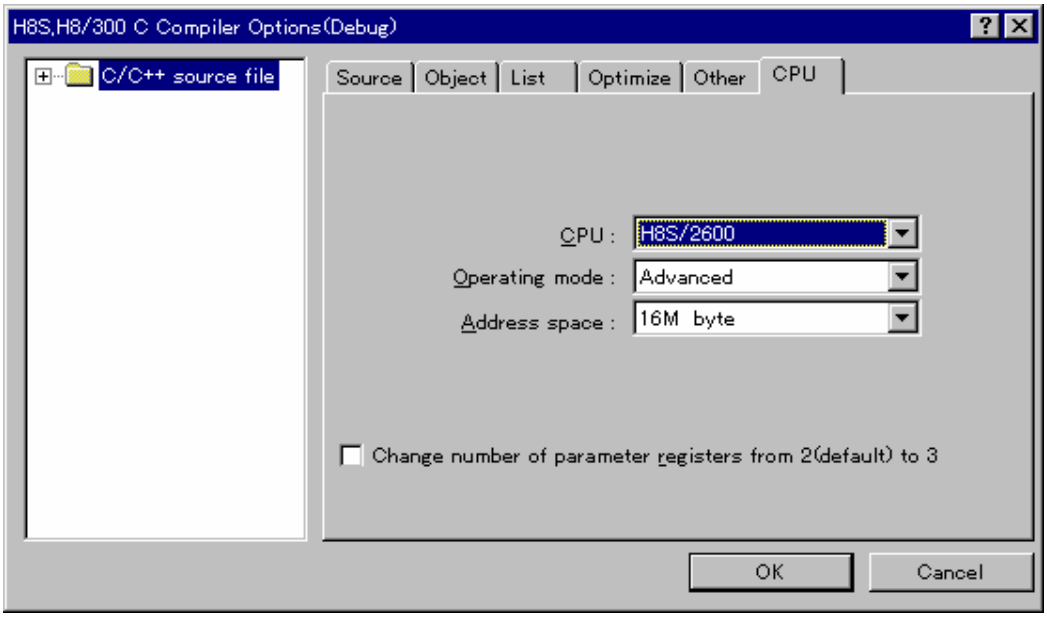


杂项 (Miscellaneous options)：

对话框菜单	命令选项	功能
允许注解嵌套 (Allow comment nest)	*comment*	允许注解嵌套
对照 EC++ 语言规格 (Check against EC++ language specification)	*ecpp*	根据 EC++ 语言规格检查语法
生成浏览器信息 (Generate browser information)	*browser*	输出浏览器信息
中断处理器保存/恢复 MACH 及 MACL 寄存器，若使用 (Interrupt handler saves/restores MACH and MACL registers if used)	*macsave*	保证 MAC 寄存器
包结构、联合及类 (Pack struct, union and class)	*pack=1 | 2*	指定对齐
避免将外部符号视为易失来优化(Avoid optimizing external symbols treating them as volatile)	*volatile*	允许或禁止外部变量的优化
将枚举当作字符处理，若它处于字符的范围内 (Treat enum as char if it is in the range of char)	*byteenum*	将枚举类型的数据当作字符来处理
为寄存器变量增加寄存器 (Increase a register for register variable)	*regexpansion | noregexpansion*	将变量分配寄存器的数量指定为 2 或 3
将公用子表达式暂时放置在寄存器上 (Put common subexpression on a register temporarily)	*cmncode*	增强删除公用表达式的优化功能
以块副本的形式使用 EEPMOV (Use EEPMOV in block copy)	*eepmov*	使用 EEPMOV 指令执行结构替换

用户定义的选项 (User defined options)：指定命令选项。

(6) CPU 标签


H8S,H8/300 C Compiler Options(Debug)
C/C++ source file
Source
Object
List
Optimize
Other
CPU
CPU :
H8S/2600
Operating mode :
Advanced
Address space :
16M byte
Change number of parameter registers from 2(default) to 3
OK
Cancel

CPU： 指定 CPU 类型。

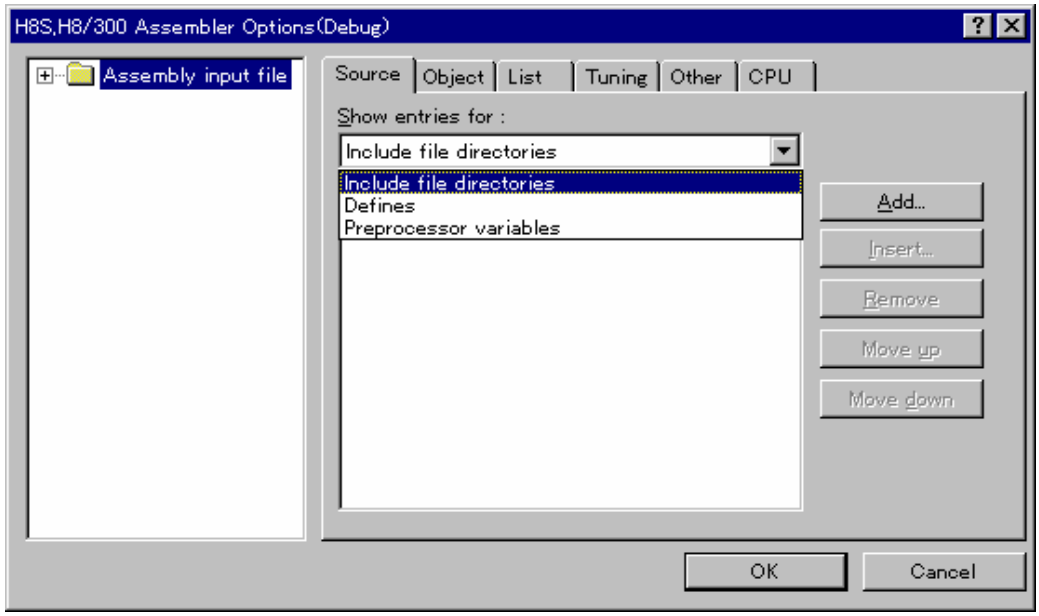
CPU:	操作模式 (Operating Mode) :	地址空间 (Address space) :	规格
环境变量 (Environment variable)	-	-	取决于环境变量 H38CPU
H8S/2600	普通 (Normal)		*cpu=2600N*
	高级 (Advanced)	1 Mbytes	*cpu=2600A:20*
		16 Mbytes	*cpu=2600A:24*
		256 Mbytes	*cpu=2600A:28*
		4 Gbytes	*cpu=2600A:32*
H8S/2000	普通 (Normal)		*cpu=2000N*
	高级 (Advanced)	1 Mbytes	*cpu=2000A:20*
		16 Mbytes	*cpu=2000A:24*
		256 Mbytes	*cpu=2000A:28*
		4 Gbytes	*cpu=2000A:32*
H8/300H	普通 (Normal)		*cpu=300HN*
	高级 (Advanced)	1 Mbytes	*cpu=300HA:20*
		16 Mbytes	*cpu=300HA:24*
H8/300	-	-	*cpu=300*
H8/300L	-	-	*cpu=300*

将参数传递寄存器的数量从 2（默认）更改至 3 (Change number of parameter-passing registers from 2 (default) to 3)

复选框	命令行	功能
☑	*regparam=3*	将参数传递寄存器的数量指定为 3
☐	*regparam=2*	将参数传递寄存器的数量指定为 2

4.1.2 汇编程序选项

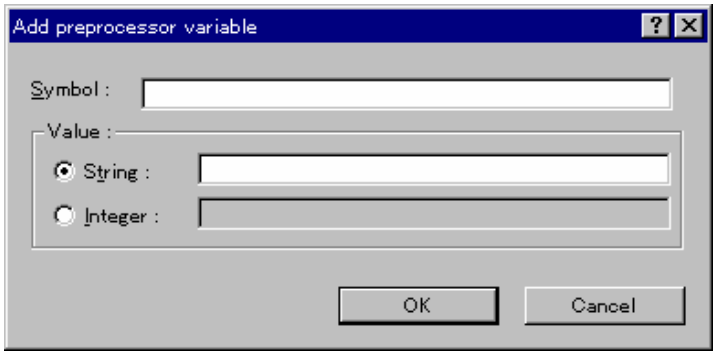
(1) 源 (Source) 标签



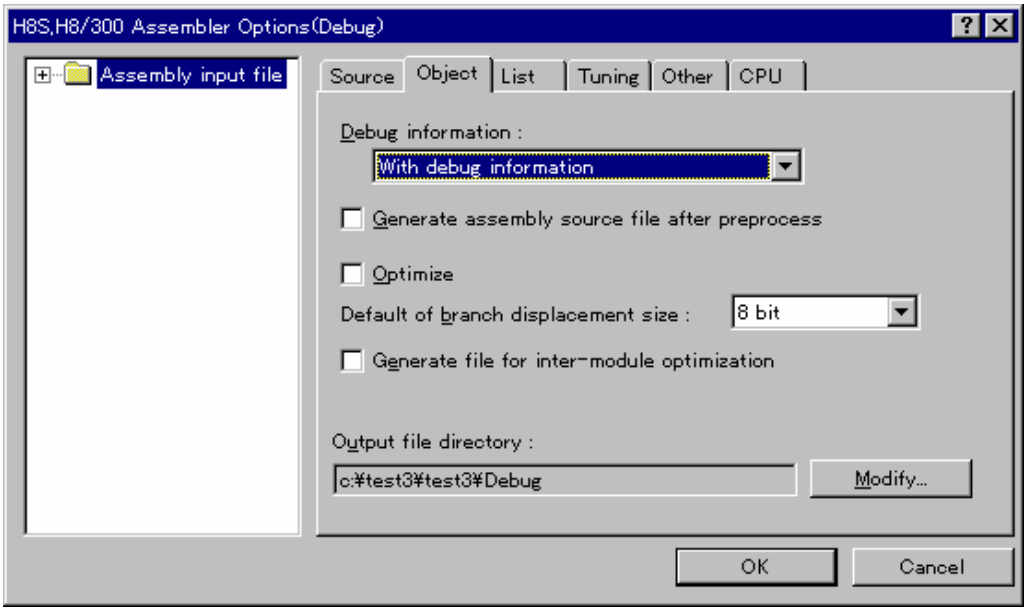
显示有关项目 (Show entries for)：

对话框菜单	命令选项	功能
包含文件目录 (Include file directories)	*include*	指定包含文件的目录
定义 (Defines)	*define*	定义字符串文字的替换
预处理程序变量* (Preprocessor variable*)	*assigna*	定义整数类型的预处理程序变量
	assignc	定义字符类型的预处理程序变量

注意： * 使用下列对话框指定。



(2) 目标 (Object) 标签



调试信息 (Debug information)：

对话框菜单	命令选项	功能
默认 (Default)	-	确认。仅限于调试指令
具有调试信息 (With debug information)	*debug*	允许调试信息的输出
没有调试信息 (Without debug information)	*nodebug*	禁止调试信息的输出

在预处理后生成汇编源文件 (Generate assembly source file after preprocess)

复选框	命令选项	功能
☑	*expand*	输出预处理程序扩展结果
☐	-	不输出预处理程序扩展结果

优化 (Optimize)

复选框	命令选项	功能
☑	*optimize*	指定优化
☐	*nooptimize*	不指定优化

转移位移大小的默认值 (Default of branch displacement size)：

对话框菜单	命令选项	功能
8 位 (8bit)	*br_relative=8*	若为转移指令选取了向前引用位移，则将位移大小指定为 8 位
16 位 (16bit)	*br_relative=16*	若为转移指令选取了向前引用位移，则将位移大小指定为 16 位

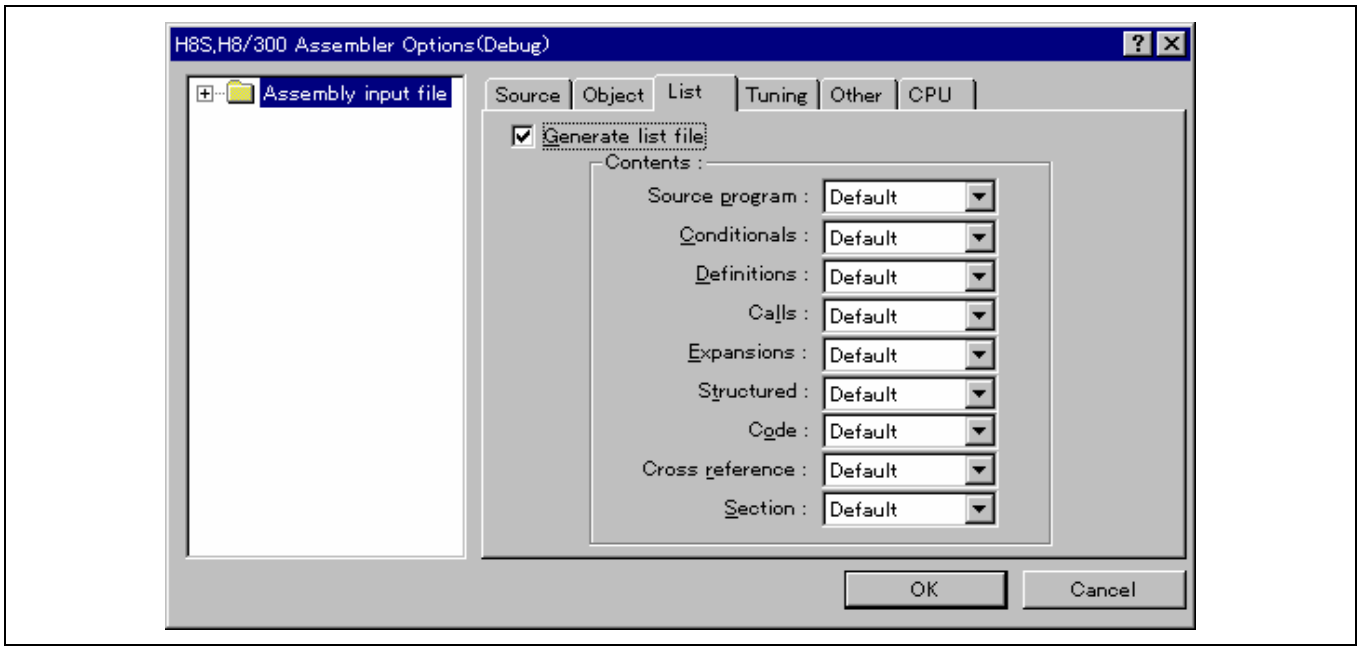
为模块间优化生成文件 (Generate file for inter-module optimization)

复选框	命令选项	功能
√	*goptimize*	输出模块间优化信息
		不输出模块间优化信息

输出目录 (Output directory)

对话框菜单	命令选项	功能
-	*object[=<file name>]*	指定目标输出目录

(3) 列表 (List) 标签



生成列表文件 (Generate list file)

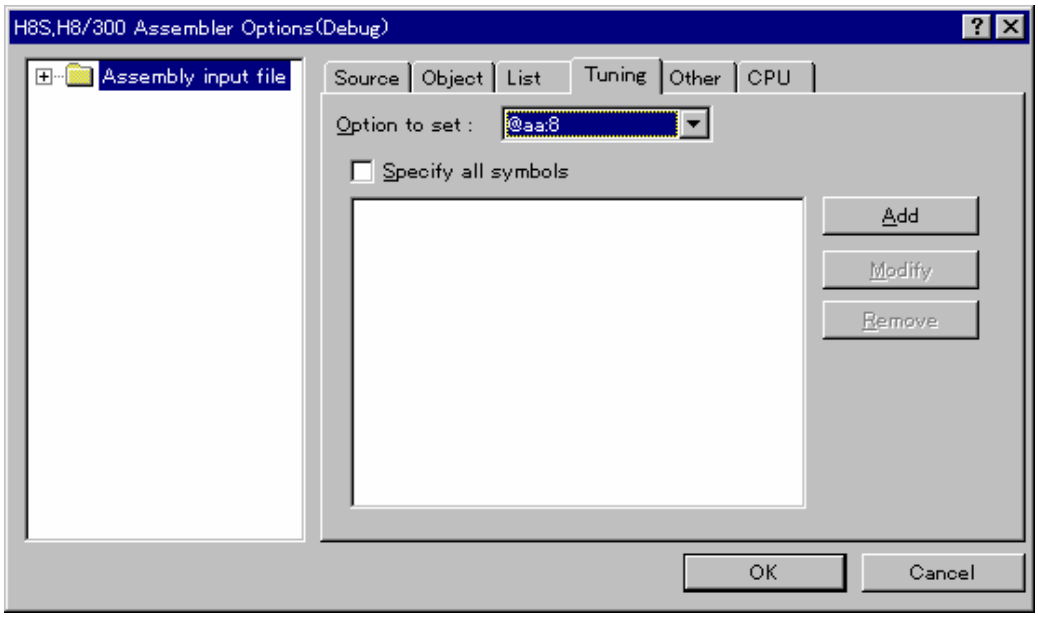
复选框	命令选项	功能
√	*list*	输出汇编列表
	nolist	不输出汇编列表

内容 (Contents)：指定要在列表文件上输出的内容。

对话框菜单	命令选项	功能
源程序 (Source program)	*source*	输出与汇编列表对应的源程序列表
条件 (Conditionals)	*show=conditionals*	输出未满足在 .AIF 或 .AIFDEF 内所指定的条件的部分
定义 (Definitions)	*show=definitions*	输出宏定义、.AREPEAT 和 .AWHILE 定义，及 .INCLUDE、.ASSIGNA 和 .ASSIGNC 指令
调用 (Calls)	*show=calls*	输出宏调用语句和 .AIF、.AIFDEF，及 .AENDI 指令
扩展 (Expansions)	*show=expansions*	输出宏扩展及 .AREPEAT → .AWHILE 扩展
结构的 (Structured)	*show=structured*	输出结构的汇编扩展
代码 (Code)	*show=code*	输出超过源语句行数的行，以显示在目标码显示内
交叉参考 (Cross reference)	*cross_refernce*	输出交叉参考列表
段 (Section)	*section*	输出段信息列表

注意： 若为每个选项选取了默认值，则源列表中的指令将被指定。

(4) 调谐 (Tuning) 标签



要设定的选项 (Option to set)：

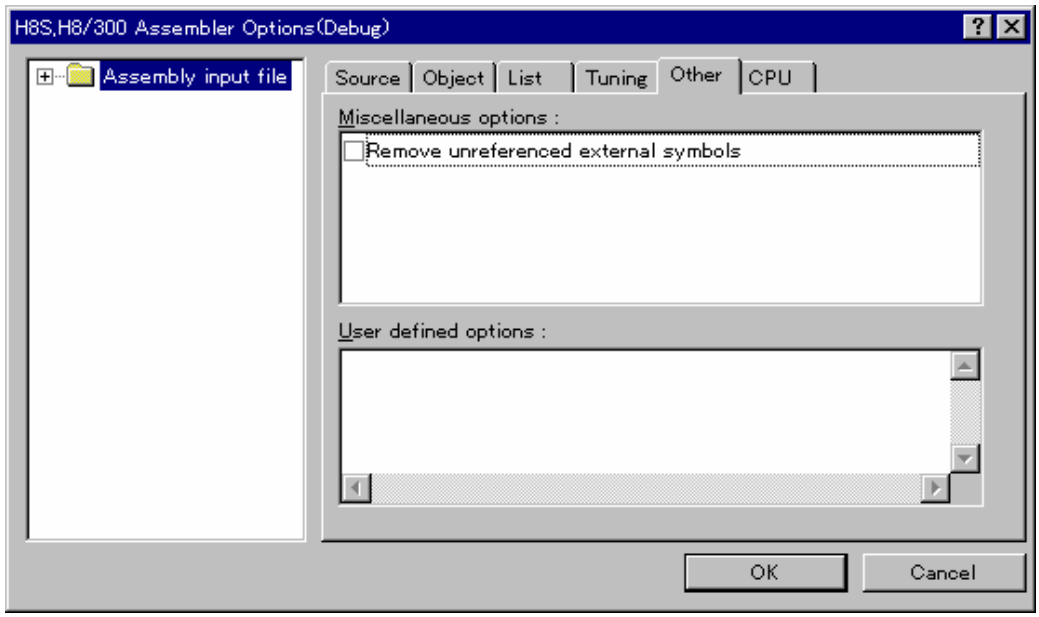
对话框菜单	命令选项	功能
@aa:8	*abs8*	指定 8 位绝对地址符号
@aa:16	*abs16*	指定 16 位绝对地址符号

注意： 选择外部参考符号或外部定义符号。

指定全部符号 (Specify all symbols)

复选框	功能
√	将指定的大小分配到外部参考符号和外部定义符号
_	分配指定的大小到每个符号或不分配大小

(5) 其他 (Other) 标签



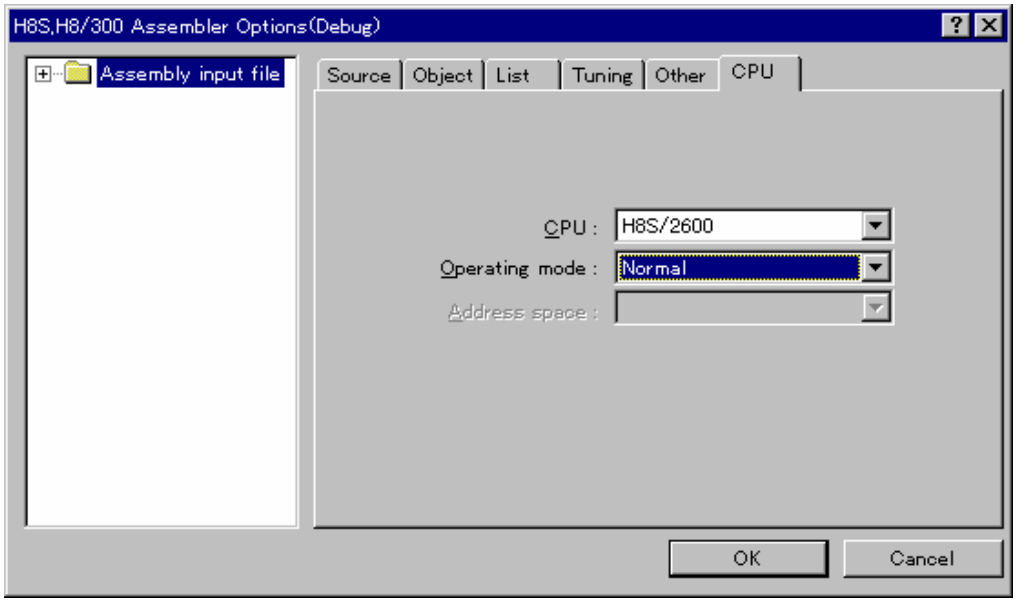
杂项 (Miscellaneous options)：

对话框菜单	复选框	命令选项	功能
移除未被参考的外部符号 (Remove unreferenced external symbols)	√	*exclude*	禁止输出未被参考的外部参考符号信息
	_	*noexclude*	允许输出未被参考的外部参考符号信息

用户定义的选项 (User defined options)：

描述命令选项。

(6) CPU 标签

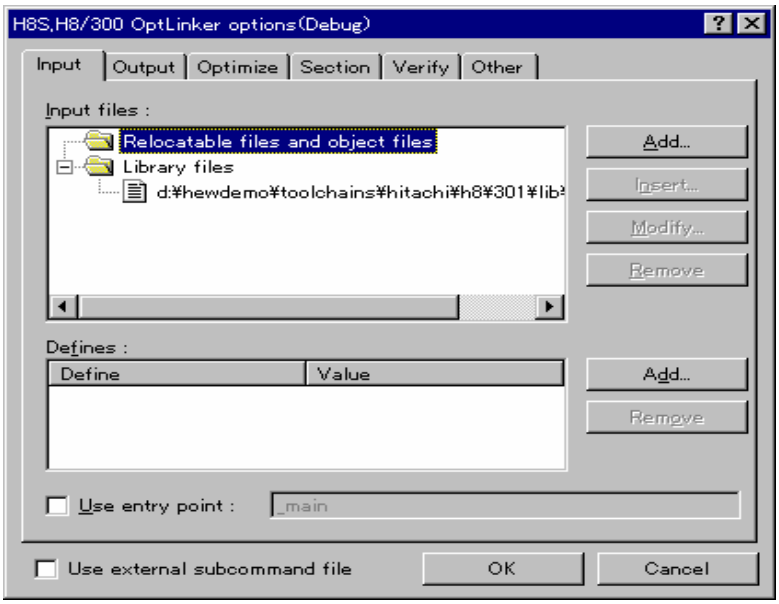


对话框菜单

CPU:	**操作模式 (Operating Mode) :**	**地址空间 (Address space) :**	**命令选项**	**功能**
默认 (default)	-	-	-	确认。CPU 指令规格
H8S/2600	**普通 (Normal)**	-	*cpu=2600n*	H8S/2600 普通模式
	高级 (Advanced)	**1M byte**	*cpu=2600a:20*	H8S/2600 高级模式
		16M bytes	*cpu=2600a[:24]*	H8S/2600 高级模式
		256M bytes	*cpu=2600a:28*	H8S/2600 高级模式
		4G bytes	*cpu=2600a:32*	H8S/2600 高级模式
H8S/2000	**普通 (Normal)**	-	*cpu=2000n*	H8S/2000 普通模式
	高级 (Advanced)	**1M byte**	*cpu=2000a:20*	H8S/2000 高级模式
		16M bytes	*cpu=2000a[:24]*	H8S/2000 高级模式
		256M bytes	*cpu=2000a:28*	H8S/2000 高级模式
		4G bytes	*cpu=2000a:32*	H8S/2000 高级模式
H8/300H	**普通 (Normal)**	-	*cpu=300hn*	H8/300H 普通模式
	高级 (Advanced)	**1M byte**	*cpu=300ha:20*	H8/300H 高级模式
		16M bytes	*cpu=300ha[:24]*	H8/300H 高级模式
H8/300	-	-	*cpu=300*	H8/300
H8/300L	-	-	*cpu=300l*	H8/300L

4.1.3 模块间优化器选项

(1) 输入 (Input) 标签



输入文件 (Input files)：指定要连接的装入模块及程序库。

对话框菜单	子命令	功能
可再定位的文件和目标文件 (Relocatable files and object files)	*input*	指定输入文件*
程序库文件 (Library files)	*library*	指定程序库文件

注意： * 此选项在输入工程文件以外的 .obj 或更改工程文件输入顺序时指定。

定义 (Defines)：

对话框菜单	子命令	功能
-	*define*	强行定义外部参考符号

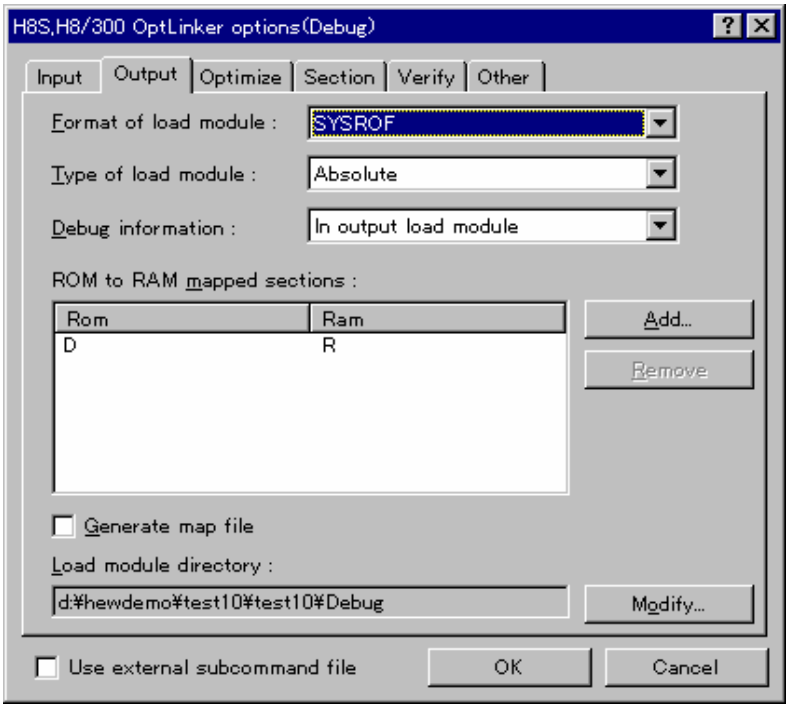
使用入口点 (Use entry point)：

对话框菜单	子命令	功能
-	*entry*	指定执行的起始地址

使用外部子命令文件 (Use external subcommand file)

对话框菜单	子命令	功能
-	*subcommand*	指定现有的子命令文件

(2) 输出 (Output) 标签



装入模块的格式 (Format of load module)：指定装入模块的输出格式。

对话框菜	子命令	功能
ELF	*elf*	以 ELF 格式输出
SYSROF	*sysrof*	以 sysrof 格式输出
SYSROFPLUS	*sysrofplus*	以 sysrof 格式输出 dwarf 调试信息

装入模块的类型 (Type of load module)：指定装入模块的文件输出格式。

对话框菜单	子命令	功能
绝对 (Absolute)	*form∆abs*	以绝对格式输出
可再定位的 (Relocatable)	*form∆rel*	以可再定位的格式输出

调试信息 (Debug information)：指定调试信息的输出选项。

对话框菜单	子命令	功能
无 (None)	*nodebug*	不输出调试信息
以输出到装入模块的形式 (In output load module)	*debug*	将调试信息输出到装入模块
以个别调试文件 (*.dbg) 的形式 (In separate debug file (*.dbg))	*sdebug*	将调试信息输出到文件

ROM 到 RAM 的映像段 (ROM to RAM mapped sections)：

对话框菜单	子命令	功能
-	*rom*	在 ROM 和 RAM 内定义初始化数据区

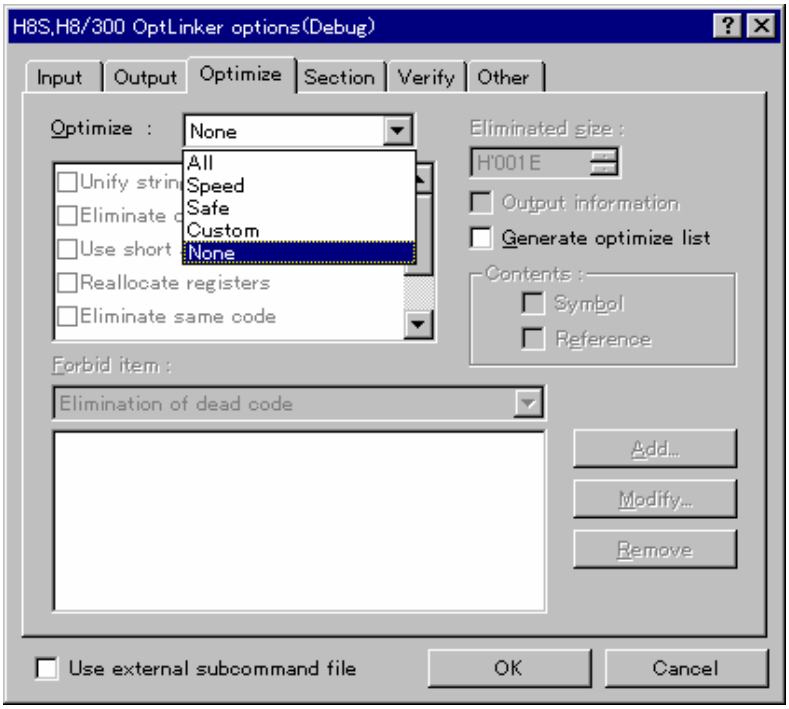
生成映像文件 (Generate map file)

复选框	子命令	功能
√	*print [Δfile name]*	输出连接列表文件
	-	不输出连接列表文件

装入模块目录 (Load module directory)：

对话框菜单	子命令	功能
-	*output*	选取装入模块的输出目录

(3) 优化 (Optimize) 标签



优化 (Optimize)：指定优化项目。

对话框菜单		子命令	功能
全部 (All)		*Optimize*	允许所有优化项目
速度 (Speed)		*Optimize⊿speed*	对速度执行优化
安全 (Safe)		*Optimize⊿safe*	执行安全优化
定制 (Custom)		*Optimize⊿*	允许优化项目的选择
	统一字符串 (Unify strings)	*String_unify*	统一常数或字符串的文字
	删除死码 (Eliminate dead code)	*Symbol_delete*	删除未被参考的符号
	使用短寻址 (Use short addressing)	*Variable_access*	使用短的绝对寻址模式
	再分配寄存器 (Reallocate registers)	*Register*	再分配寄存器
	删除相同的代码 (Eliminate same code)	*Same_code*	删除相同的代码
	删除的大小 (Eliminated size)：	*Samesize*	指定删除相同代码的目标大小
	使用间接调用/跳转 (Use indirect call/jump)	*Function_call*	使用间接寻址模式
	优化转移 (Optimize branches)	*Branch*	优化转移指令
无 (None)		*Nooptimize*	禁止优化

输出信息 (Output information)

复选框	子命令	功能
☑	*information*	显示优化的函数名称
☐		不显示优化的函数名称

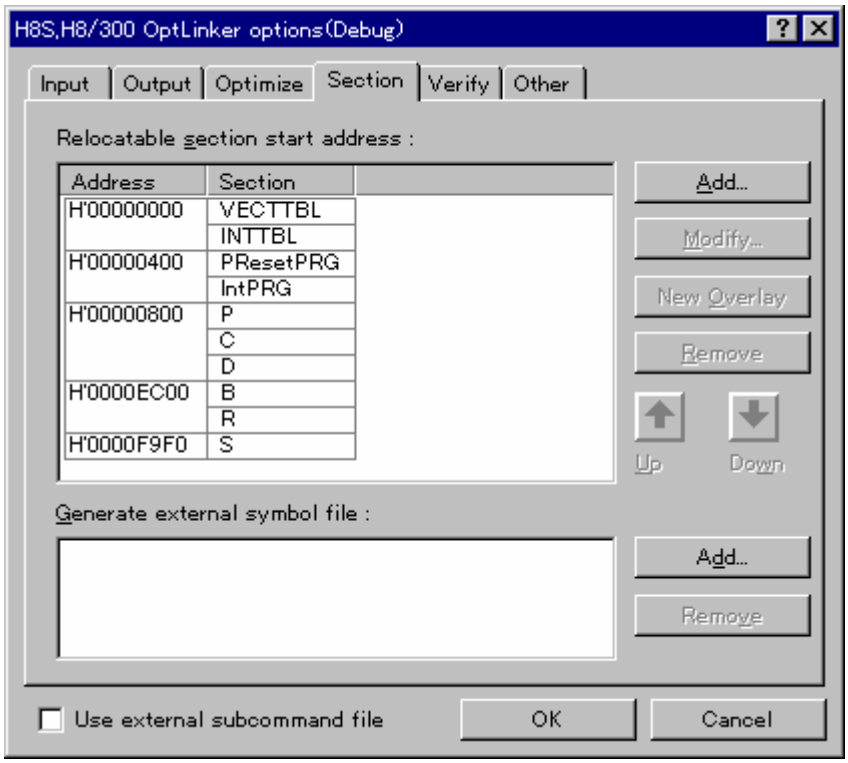
生成优化列表 (Generate optimize list)

对话框菜单		子命令	功能
-		*Mlist [⊿file name]*	输出优化信息列表
内容 (Contents)：	**符号 (Symbol)**	*show⊿symbol*	输出符号优化信息
	参考 (Reference)	*show⊿reference*	输出符号参考计数

禁止项目 (Forbid item)：

对话框菜单	子命令	功能
死码的删除 (Elimination of dead code)	*Symbol_forbid*	指定禁止以删除未被参考的符号来达到优化的变量或函数名称
相同代码的删除 (Elimination of same code)	*Samecode_forbid*	指定禁止以删除相同代码来达到优化的函数名称
将短寻址使用到 (Use of short addressing to)	*Variable_forbid*	指定禁止以使用短绝对寻址模式来达到优化的变量名称
将间接调用/跳转使用到 (Use of indirect call/jump to)	*Function_forbid*	指定禁止以使用间接寻址模式来达到优化的函数名称
存储器分配在 (Memory allocation in)	*Absolute_forbid*	指定不执行地址分配的地址区

(4) 段 (Section) 标签



可再定位的段起始地址 (Relocatable section start address)：

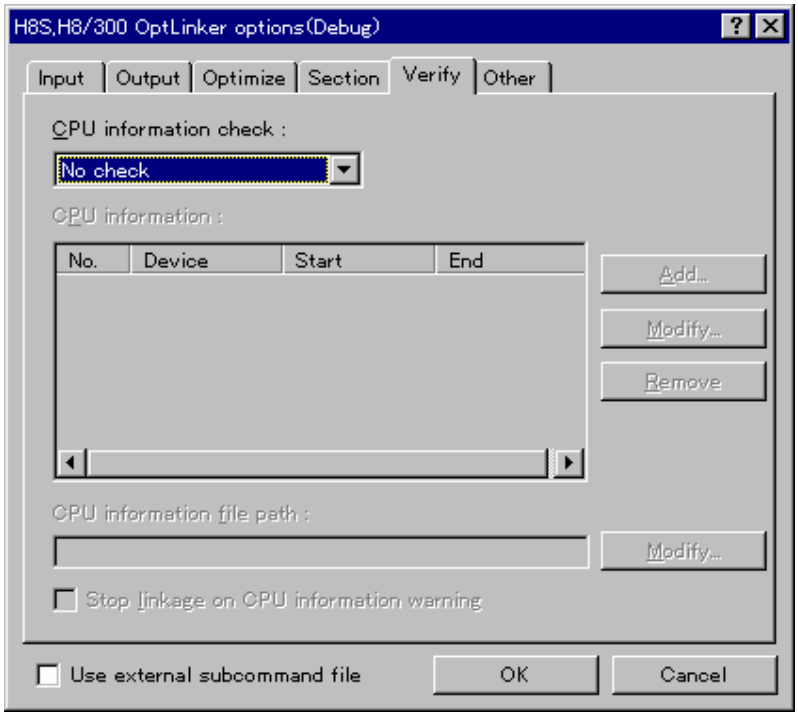
对话框菜单	子命令	功能
-	*start*	指定每个段的起始地址和连接顺序

生成外部符号文件 (Generate external symbol file)：

对话框菜单	子命令	功能
-	*fsymbol*	将由连接函数处理的外部定义符号，以汇编程序指令格式输出到文件

注意： 输出文件名称为 <工程名称>.fsy。

(5) 验证 (Verify) 标签



CPU 信息检查 (CPU information check)：

对话框菜单	子命令	功能
不检查 (No check)		不检查 CPU 分配
检查 (Check)		根据 CPU 信息文件检查存储器分配
使用 CPU 信息文件 (Use CPU information file)	*CPU*	根据现有的 CPU 信息文件检查存储器分配

CPU 信息 (CPU information)

对话框菜单	子命令	功能
-	-	创建或修改 CPU 信息文件 CPU 指定存储器类型，然后再指定每个存储器的地址

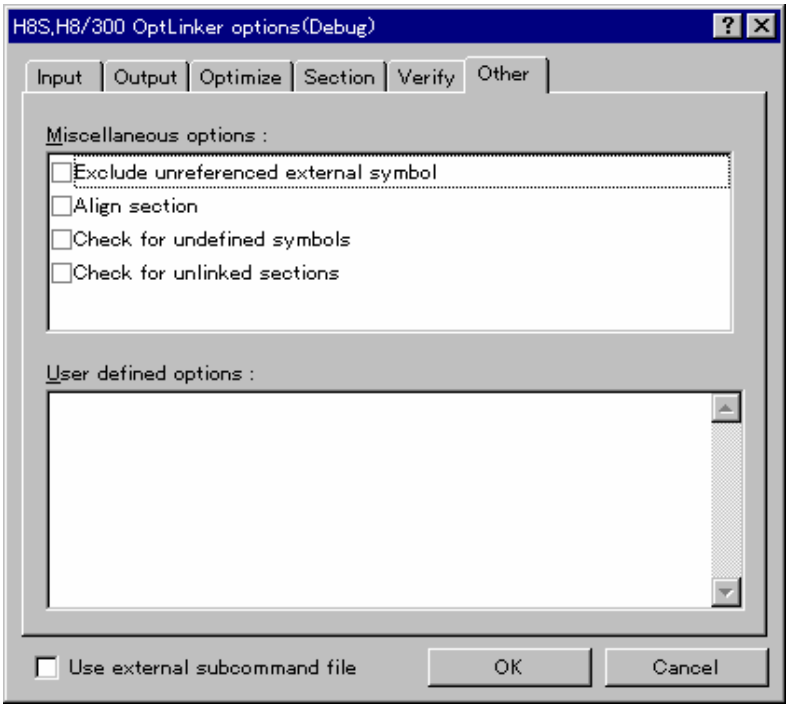
CPU 信息文件路径 (CPU information file path)

对话框菜单	子命令	功能
-	-	指定现有的 CPU 信息文件

在 CPU 信息警告发出时停止连接 (Stop linkage on CPU information warning)

复选框	子命令	功能
☑	CPUCheck	在根据 CPU 信息文件进行存储器分配检查期间输出错误信息
☐	-	在存储器分配检查期间不输出错误信息

(6) 其他 (Other) 标签

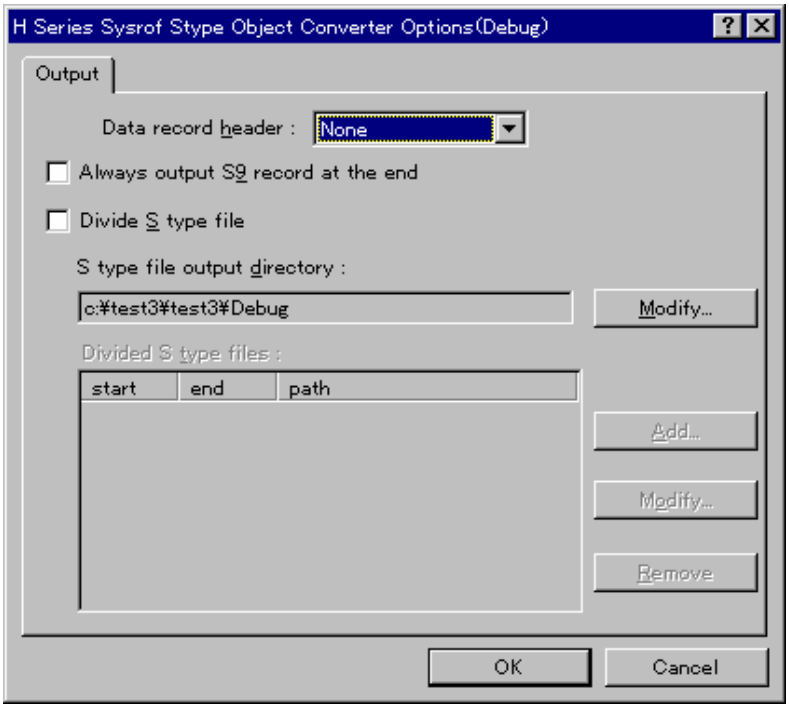


杂项 (Miscellaneous options)：指定其他功能。

对话框菜单	子命令	功能
排除未被参考的外部符号 (Exclude unreferenced external symbols)	*Exclude*	禁止未被参考的程序库连接
对齐段 (Align section)	*Align_section*	检查具有不同对齐方式的段
检查未定义的符号 (Check for undefined symbols)	*Udfcheck*	在检测到未定义的符号时输出错误信息
检查未连接的段 (Check for unlinked sections)	*Check_section*	检查未被分配地址的段

4.1.4 S 类型转换器选项

(1) 输出 (Output) 标签



数据记录标头 (Data record header):

对话框菜单	命令选项	功能
无 (None)	-	-
S1	*record=s1*	以 S1 的数据记录输出
S2	*record=s2*	以 S2 的数据记录输出
S3	*record=s3*	以 S3 的数据记录输出

始终在结束部分输出 S9 记录 (Always output S9 record at the end)

复选框	命令选项	功能
☑	s9	在结束部分输出 S9 记录，即使项目地址已超过 H'10000
☐	-	始终输出

分隔 S 类型文件 (Divide S type file)

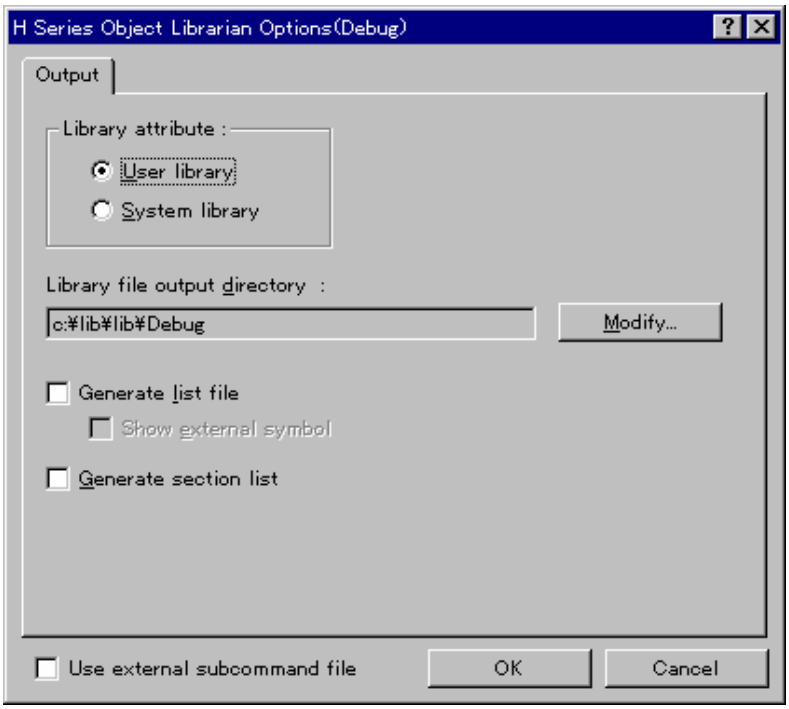
复选框	命令选项	功能
☑	-	通过分隔入任意地址区来输出 S 类型文件
☐	-	不使用分隔来输出 S 类型文件

S 类型文件的输出目录 (S type file output directory)

对话框菜单	命令选项	功能
-	-	指定 S 类型文件的输出目录

4.1.5 库管理程序选项

(1) 输出 (Output) 标签



程序库属性 (Library attribute)：指定所要输出的程序库的属性。

对话框菜单	选项/子命令	功能
用户程序库 (User library)	*output*	将程序库属性指定为用户程序库
系统程序库 (System library)	*output*	将程序库属性指定为系统程序库

程序库文件的输出目录 (Library file output directory)：

对话框菜单	选项/子命令 (Option/Subcommand)	功能
-	output	指定程序库的输出目录

生成列表文件 (Generate list file)：指定是否输出程序库列表文件。

复选框	选项/子命令	功能
√	*list*	显示程序库文件的内容
_	-	不显示程序库文件的内容

显示外部符号 (Show external symbol)：指定在模块中定义的外部定义符号名称的输出。

复选框	选项/子命令	功能
☑	*list*	显示在模块中定义的外部定义符号名称
☐	-	不显示外部定义符号名称

生成段列表 (Generate section list)：指定段名称列表文件的输出。

复选框	选项/子命令	功能
☑	*slist*	显示段的内容
☐	-	不显示段的内容

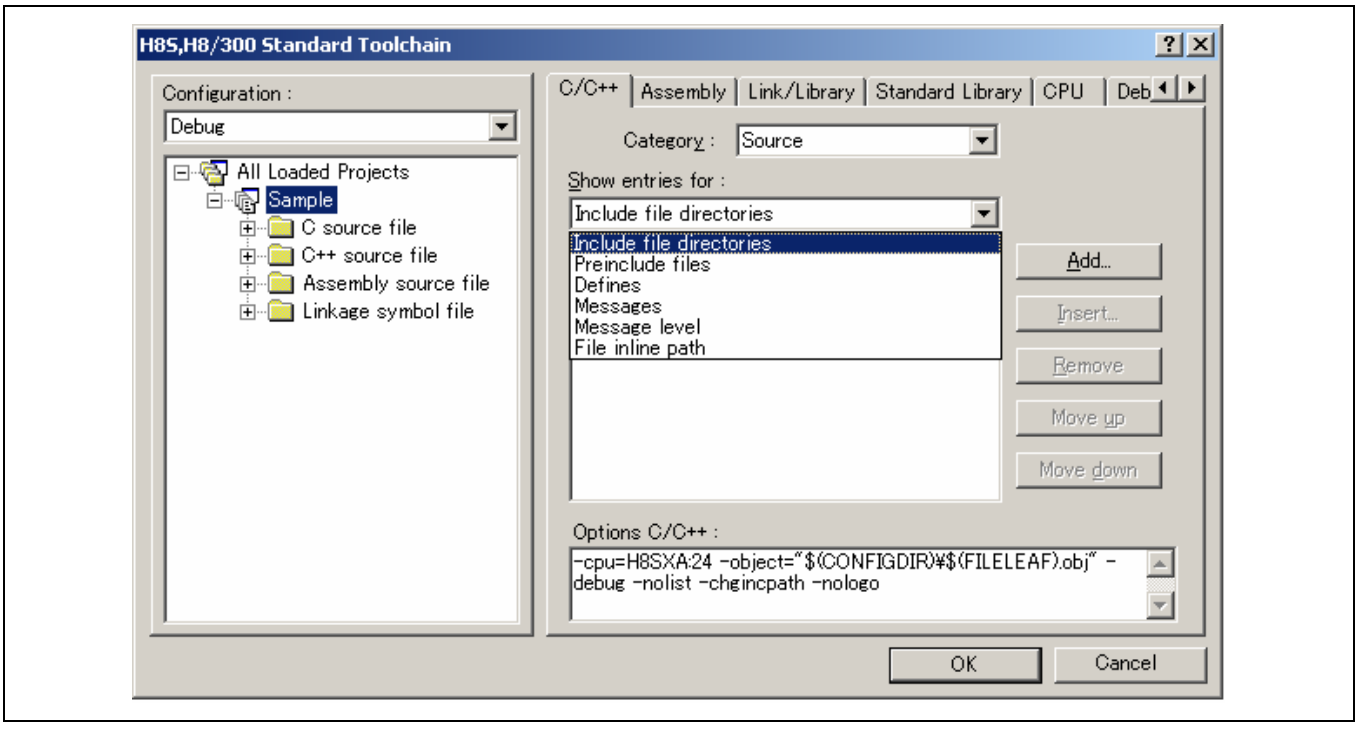
4.2 在 HEW2.0 或以上版本中指定选项

要获取在 HEW1.2 中指定选项的详细资料，请参考 4.1 节，在 HEW1.2 中指定选项。

4.2.1 C/C++ 编译程序选项

从 H8S，H8300 标准工具链 (H8S, H8300 Standard Toolchain) 对话框选取 C/C++ 标签。

(1) **类别 (Category)：**[源 (Source)]



显示有关项目 (**Show entries for**)：

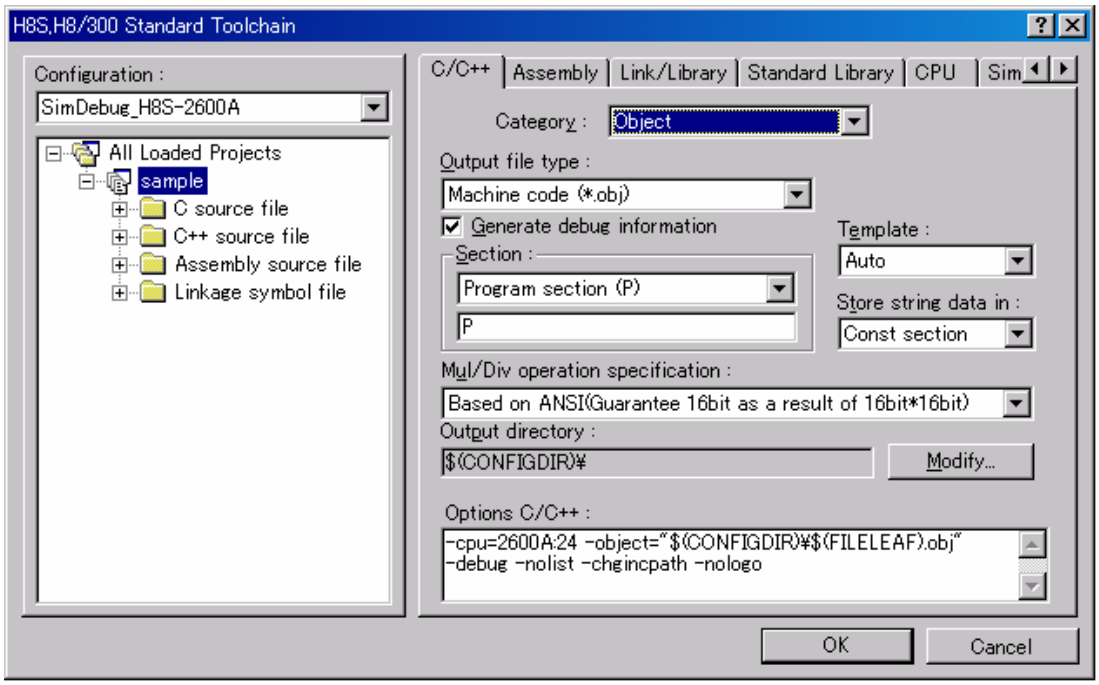
对话框菜单	命令选项	功能
包含文件目录 (Include file directories)	*include*	指定包含文件目录的路径名称
预包含文件 (Preinclude files)	*preinclude*	在编译单元的初始，将文件内容指定为包含文件
定义 (Defines)	*define*	定义宏名称
消息 (Messages)	*message*	输出信息消息
消息级 (Message level)	*charge_message*	更改消息级
文件内联路径 (File inline path)	*file_inline_path*	指定其函数定义要被扩展为内联函数的文件路径名称

(2) **类别 (Category)**：[目标 (Object)]

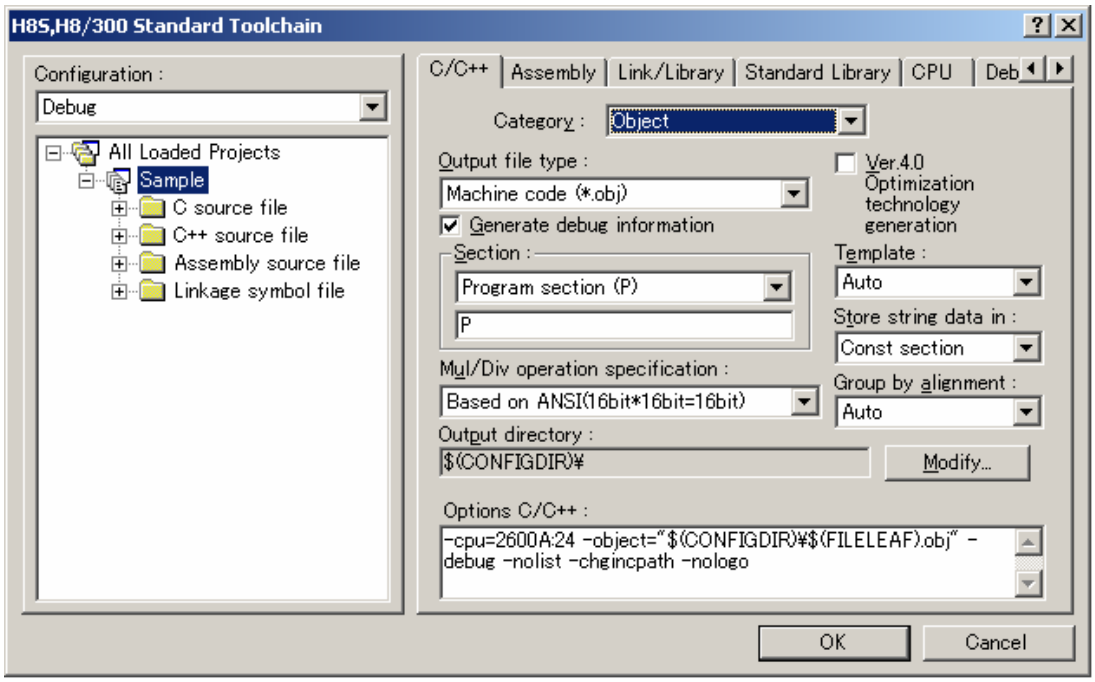
[目标] HEW 4.0 中的类别与之前版本（HEW 3.0 或以上版本）不同。

它们都在下表中显示。

<HEW2.0 至 HEW3.0>



<HEW4.0>



输出文件类型 (Output file type)：

对话框菜单	命令选项	功能
机器码 (*.obj) (Machine code (*.obj))	*code=machinecode*	输出机器语言程序
汇编源代码 (*.src) (Assembly source code (*.src))	*code=asmcode*	输出汇编语言程序
预处理源文件 (*.p/*.pp) (Preprocessed source file (*.p/*.pp))	*preprocessor*	在预处理程序扩展后输出源程序

生成调试信息 (Generate debug information)

复选框	命令选项	功能
√	*debug*	输出调试信息
_	*nodebug*	不输出调试信息

段 (Section)：

对话框菜单	命令选项	功能
-	*section*	更改默认段名称

将字符串数据存储在 (Store string data in)：

对话框菜单	命令选项	功能
常数段 (Const section)	*string=const*	将字符串文字输出到常数区
数据段 (Data section)	*string=data*	将字符串文字输出到初始化数据区

乘除运算规格 (Mul/Div operation specifications)

对话框菜单	命令选项	功能
基于 ANSI（保证 16 位为 16 位*16 位的结果） (Based on ANSI (Guarantee 16bit as a result of 16bit*16bit))	*nocpuexpand*	根据 ANSI C 语言规格，以代码开发乘法或除法
非 ANSI（保证 32 位为 16 位*16 位的结果） (Non ANSI (Guarantee 32bit as a result of 16bit*16bit))	*cpuexpand*	根据 CPU 指令规格，以代码开发乘法或除法

输出目录 (Output directory)

对话框菜单	命令选项	功能
-	*object*	指定目标文件的输出目录

模板 (Template)

对话框菜单	命令选项	功能
无 (None)	*template=none*	不生成示例
静态 (static)	*template=static*	将示例生成为内部连接仅限于参考的模板
已使用的 (Used)	*template=used*	将示例生成为外部连接仅限于参考的模板
全部 (All)	*template=all*	为声明或参考的模板生成示例
自动 (Auto)	*template=auto*	在连接时生成示例

位字段分配顺序（HEW4.0 或以上版本指定 CPU 标签）

对话框菜单	命令选项	功能
左 (Left)	*bit_order=left*	从上端位存储成员
右 (Right)	*bit_order=right*	从下端位存储成员

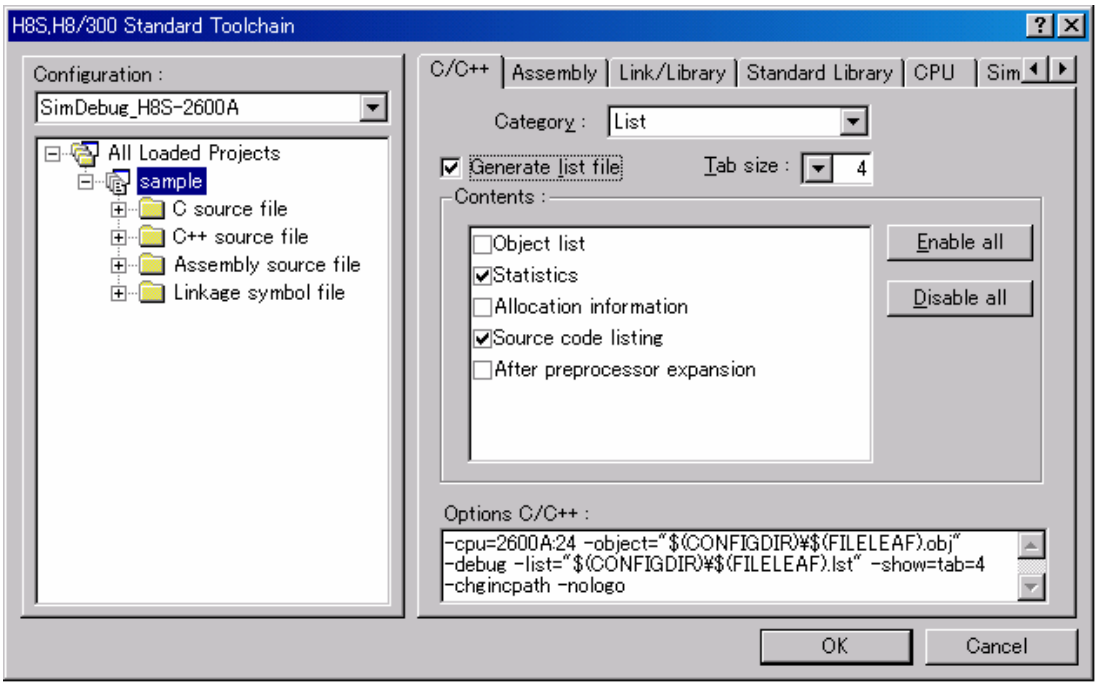
以对齐方式分组 (Group by alignment)

对话框菜单	命令选项	功能
无 (None)	*noalign*	以定义的顺序分配定义的变量
自动 (Auto)	*align*	分配变量以缩减边界对齐所形成的空间
4 字节 (4byte)	*align=4*	将数据段分隔为 4、2、1 字节的边界对齐段，然后分配入多个 4、2、1 地址，以增进存取速度

目标代码的输出兼容性（HEW4.0 或以上版本）

复选框	命令选项	功能
√	*legacy=v4*	输出 H8S 版本 4.0 优化技术生成的目标
_	-	输出 H8S 版本 6.1 优化技术生成的目标

(3) **类别 (Category)：**[列表 (List)]



生成列表文件 (Generate list file)

复选框	命令选项	功能
√	*list*	输出目标列表文件
_	*nolist*	不输出目标列表文件

内容 (Contents)：指定要输出至目标文件列表的数据。

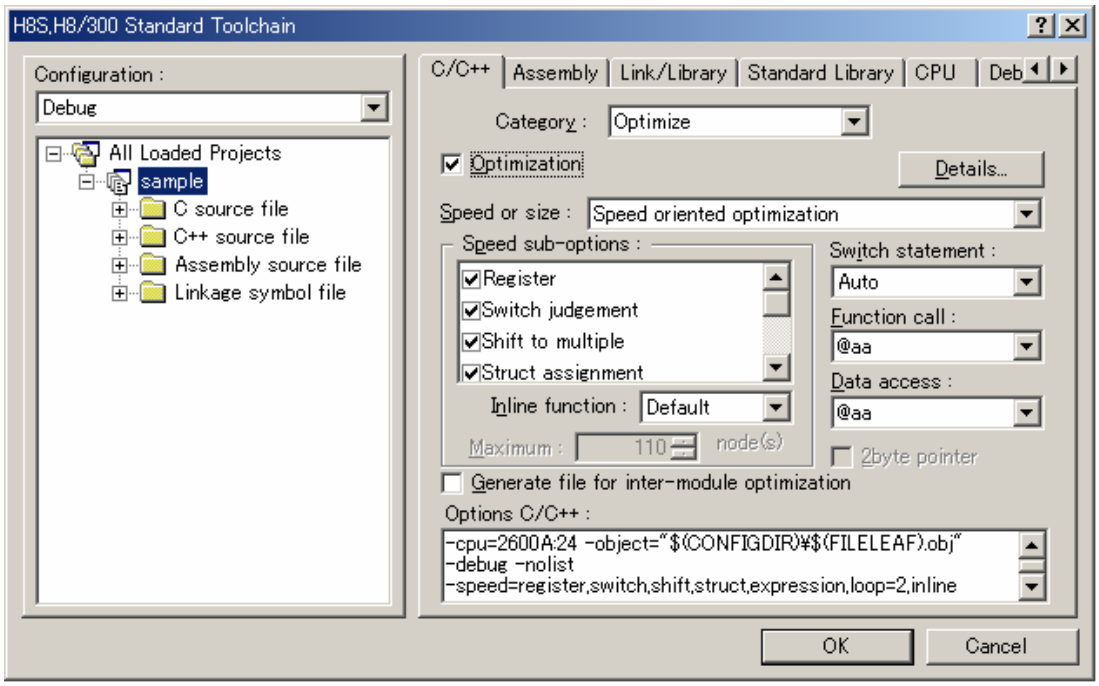
对话框菜单	命令选项	功能
目标列表 (Object list)	*show=object*	输出目标列表
统计数据 (Statistics)	*show=statistics*	输出统计数据信息
分配信息 (Allocation information)	*show=allocation*	输出符号分配列表
源代码列表 (Source code listing)	*show=source*	输出源列表
预处理程序扩展后 (After preprocessor expansion)	*show=expansion*	在宏扩展后输出源程序列表

若按下 [全部允许 (Enable all)] 按钮，所有数据项目将被输出。 否则，若按下 [全部禁止 (Disable all)] 按钮，所有数据项目将被禁止，且不输出任何数据项目到目标列表文件。

制表符大小 (Tab size)

对话框菜单	命令选项	功能
4	*show=tab=4*	将显示在列表中的制表符大小指定为 4
8	*show=tab=8*	将显示在列表中的制表符大小指定为 8

(4) **类别 (Category)**：[优化 (Optimize)]



优化 (Optimization)

复选框	命令选项	功能
√	optimize=1	指定优化
	optimize=0	不指定优化

速度或大小 (Speed or size)：指定优化的格式。

对话框菜单		命令选项	功能
面向大小的优化 (Size oriented optimization)		-	对大小执行优化
面向速度的优化 (Speed oriented optimization)		*speed*	对速度执行优化
速度子选项 (Speed sub-options)	**寄存器 (Register)**	*speed=register*	通过 PUSH 和 POP 指令以更快的速度执行寄存器存储/恢复扩展
	切换判断 (Switch judgement)	*speed=switch*	以更快的速度开发切换语句
	转移到多个 (Shift to multiple)	*speed=shift*	以更快的速度开发转移操作
	结构赋值 (Struct assignment)	*speed=struct*	以更快的速度执行结构扩展及替换表达式
	表达式 (Expression)	*speed=expression*	以更快的速度执行算术操作、比较，及替换表达式处理
	循环优化 (Loop optimization)	*speed=loop1*	感应变量的删除
	解开循环 (Loop Unrolling)	*speed=loop2*	感应变量和循环扩展的删除
	内联函数的最大节点 (Inline function Maximum:node(s))	*speed=inline [=<data>]*	执行或不执行自动内联扩展

为模块间优化生成文件 (Generate file for inter-module optimization)

复选框	命令选项	功能
√	*goptimize*	输出模块间优化的加载信息文件
	-	不输出模块间优化的加载信息文件

切换语句 (Switch statement)：指定切换语句扩展方法。

对话框菜单	命令选项	功能
自动 (Auto)	*case=auto*	根据速度选项规格来决定切换语句扩展方法
如果…就 (If then)	*case=ifthen*	以“如果…就”的方法来执行切换语句扩展
表 (Table)	*case=table*	以表跳转 (table jump) 的方法来执行切换语句扩展

函数调用 (Function call)：选择函数调用的方法。

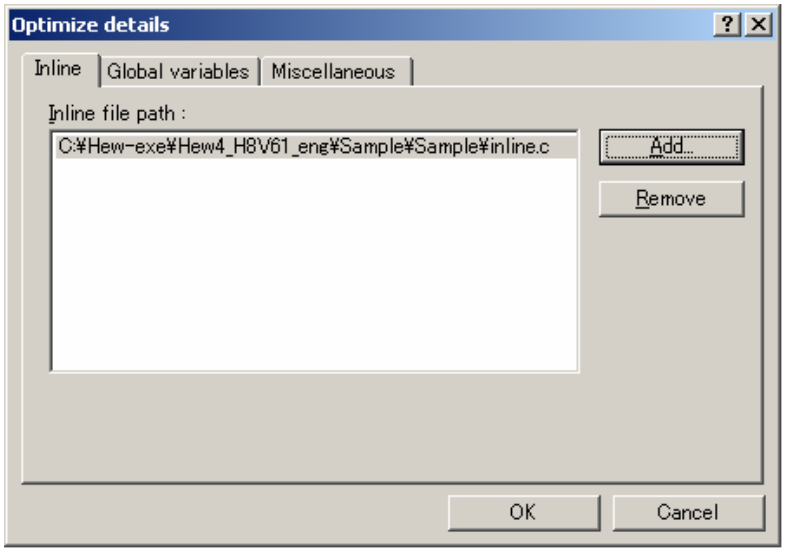
对话框菜单	命令选项	功能
@aa	-	选取普通函数调用
@@aa:8	*idirect=normal*	选取存储器间接函数调用
@@vec:7	*idirect=extended*	选取扩展的存储器间接函数调用

数据存取 (Date access)：选取数据存取模式。

对话框菜单	命令选项	功能
@aa	-	选取普通数据存取
@aa:8	*abs8*	选取 8 位绝对地址存取
@aa:16	*abs16*	选取 16 位绝对地址存取

(a) **[详细资料 (Details)] 按钮：**[内联 (Inline)] 标签

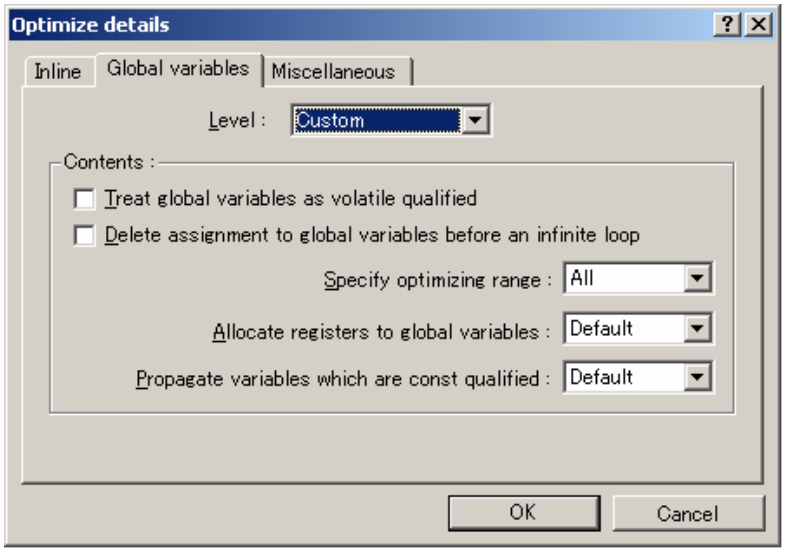
（HEW 4.0 之后支持）



指定优化的范围 (Specify optimizing range)

对话框菜单	命令选项	功能
内联文件路径 (Inline file path)	*file_inline*	指定进行文件间内联扩展的文件。

(b) **[详细资料 (Details)] 按钮：**[全局变量 (Global Variables)] 标签



级别 (Level)：指定外部变量优化的级别

对话框菜单	命令选项	功能
1 级 (Level 1)		禁止所有外部变量优化。
	volatile	[处理全局… (Treat global…)] = [选中 (checked)]
	infinite_loop=0	[删除赋值… (Delete assignment…)] = [未选中 (Not checked)]
	opt_range=noblock	[指定优化… (Specify optimizing) …] = [没有块 (No block)]
	global_alloc=0	[分配寄存器… (Allocate registers) …] = [禁止 (Disable)]
	const_var_propagate=0	[传播变量… (Propagate variables) …] = [禁止 (Disable)]
2 级 (Level 2)		优化不具有易失性说明符的外部变量。 禁止优化跨越循环或转移进行扩展的外部变量。
	novolatile	[处理全局… (Treat global…)] = [未选中 (Not checked)]
	infinite_loop=0	[删除赋值… (Delete assignment…)] = [未选中 (Not checked)]
	opt_range=noblock	[指定优化… (Specify optimizing) …] = [没有块 (No block)]
	global_alloc=0	[分配寄存器… (Allocate registers) …] = [禁止 (Disable)]
	const_var_propagate=0	[传播变量… (Propagate variables) …] = [禁止 (Disable)]
3 级 (Level 3)		优化在整个函数内不具有易失性说明符的外部变量。
	novolatile	[处理全局… (Treat global…)] = [未选中 (Not checked)]
	infinite_loop=0	[删除赋值… (Delete assignment…)] = [未选中 (Not checked)]
	opt_range=all	[指定优化… (Specify optimizing) …] = [全部 (All)]
	global_alloc=1	[分配寄存器… (Allocate registers) …] = [允许 (Enable)]
	const_var_propagate=1	[传播变量… (Propagate variables) …] = [允许 (Enable)]
定制 (Custom)	-	根据用户指定的选项来优化外部变量

将全局变量视为符合易失性标准来处理 (Treat global variables as volatile qualified)

复选框	命令选项	功能
√	*volatile*	禁止外部变量的优化。
_	*novolatile*	优化不具有易失性说明符的外部变量。

在无穷循环之前删除到全局变量的赋值 (Delete assignment to global variables before an infinite loop)

复选框	命令选项	功能
√	*infinite_loop=1*	删除紧随于无穷循环之前的赋值表达式，该赋值是不在无穷循环中使用的对外部变量的赋值。
_	*infinite_loop=0*	禁止删除无穷循环之前的外部变量之赋值表达式。

指定优化的范围 (Specify optimizing range)

对话框菜单	命令选项	功能
全部 (All)	*opt_range=all*	优化整个函数内的外部变量。
没有循环 (No loop)	*opt_range=noloop*	循环中的外部变量及循环迭代条件中所使用的外部变量将不被优化。
没有块 (No block)	*opt_range=noblock*	跨越转移（包括循环）扩展的外部变量将不被优化。

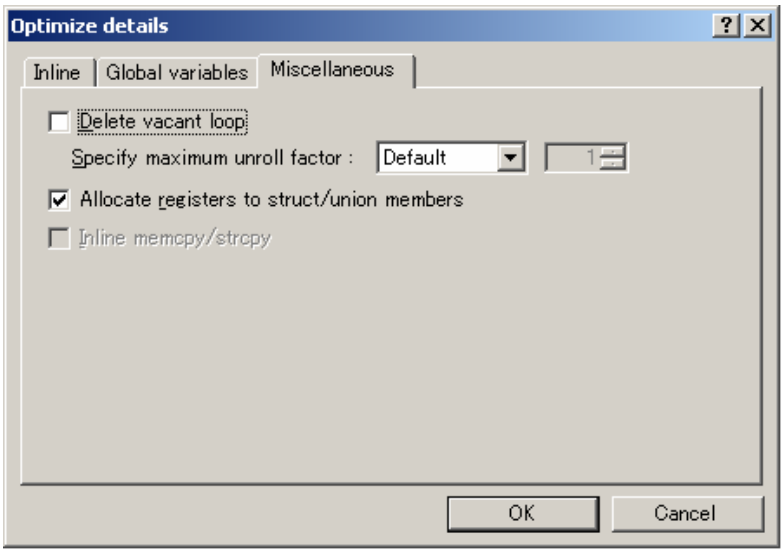
将寄存器分配到全局变量 (Allocate registers to global variables)

对话框菜单	命令选项	功能
禁止 (Disable)	*global_alloc=0*	禁止将外部变量分配到寄存器。
允许 (Enable)	*global_alloc=1*	将外部变量分配到寄存器。
默认 (Default)	*global_alloc=1*	将外部变量分配到寄存器。

传播符合常数标准的变量 (Propagate variables which are const qualified)

对话框菜单	命令选项	功能
禁止 (Disable)	*const_var_propagate=0*	禁止以 **const** 声明的外部常数的常数传播。
允许 (Enable)	*const_var_propagate=1*	执行以 **const** 声明的外部常数的常数传播。
默认 (Default)	*const_var_propagate=1*	执行以 **const** 声明的外部常数的常数传播。

(c) **详细资料 (Details)] 按钮：**[杂项 (Miscellaneous)] 标签



删除空的循环 (Delete vacant loop)

复选框	命令选项	功能
√	*del_vacant_loop=1*	删除其中不含语句的循环。
	del_vacant_loop=0	禁止删除空的循环，即使循环内不含语句。

指定最大的解开因数 (Specify maximum unroll factor)

对话框菜单	复选框	命令选项
默认 (Default)	*max_unroll=2 or 1*	2 或 1 被假设为将要被扩展的最大循环数。
定制 (Custom)	*max_unroll=* *< numeric value >*	指定将要被扩展的最大循环数。 可为 <数值> 指定从 1 到 32 的整数。

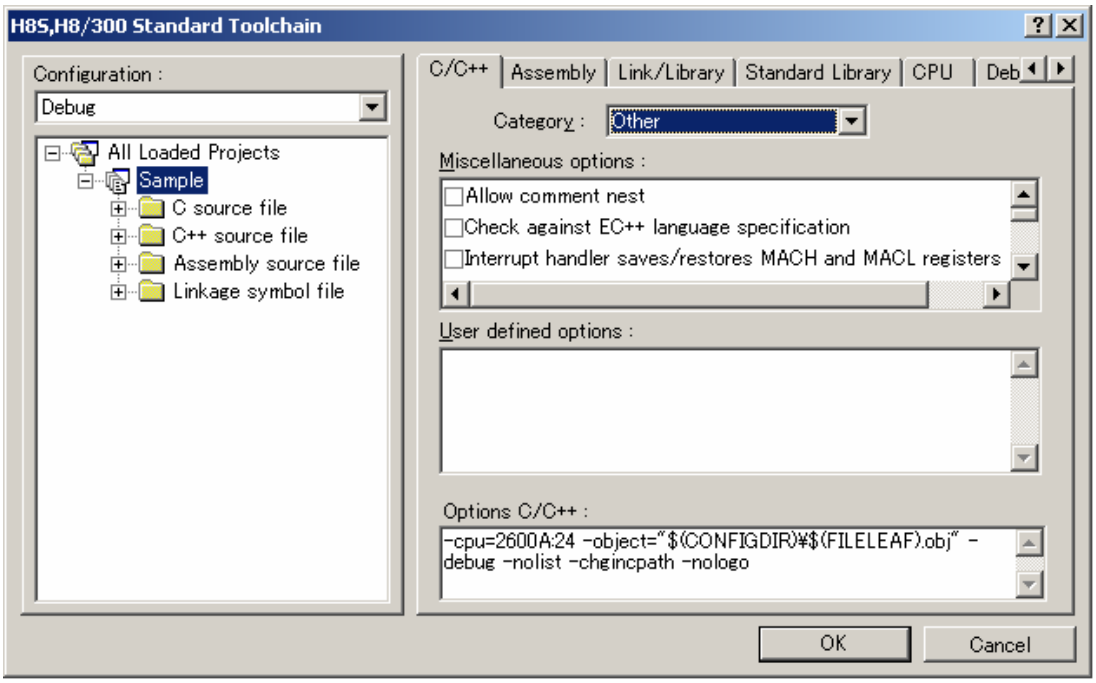
将寄存器分配到结构/联合成员 (Allocate registers to struct/union members)

复选框	命令选项	功能
√	*struct_alloc=1*	将结构/联合成员分配到寄存器。
	struct_alloc=0	禁止将结构/联合成员分配到寄存器。

内联 memcpy/strcpy (Inline memcpy/strcpy)

复选框	命令选项	功能
√	*library=intrinsic*	为 **memcpy** 和 **strcpy** 执行内联扩展
	library=function	为 **memcpy** 和 **strcpy** 产生函数调用。

(5) **类别 (Category)：**[其他 (Other)]



杂项 (Miscellaneous options)：

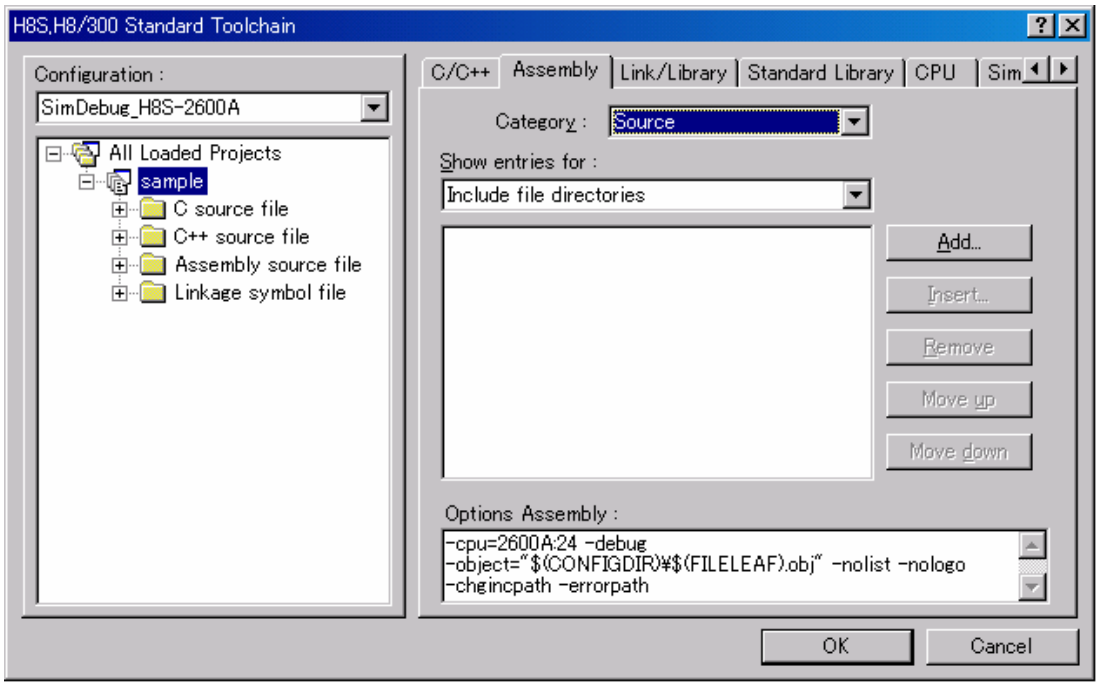
对话框菜单	命令选项	功能
允许注解嵌套 (Allow comment nest)	*comment*	允许注解嵌套
对照 EC++ 语言规格 (Check against EC++ language specification)	*ecpp*	根据 EC++ 语言规格检查语法
中断处理器保存/恢复 MACH 及 MACL 寄存器，若使用 (Interrupt handler saves/restores MACH and MACL registers if used)	*macsave*	保证 MAC 寄存器
将循环条件视为符合易失性标准来处理 (Treat loop condition as volatile qualified)	*volatile_loop*	禁止循环迭代条件的优化
将枚举当作字符处理，若它处于字符的范围内 (Treat enum as char if it is in the range of char)	*byteenum*	将枚举类型的数据当作字符来处理
为寄存器变量增加寄存器 (Increase a register for register variable)	*Regexpansion* *noregexpansion*	将变量分配寄存器的数量指定为 2 或 3
将公用子表达式暂时放置在寄存器上 (Put common subexpression on a register temporarily)	*cmncode*	增强删除公用表达式的优化功能
以块副本的形式使用 EEPMOV (Use EEPMOV in block copy)	*eepmov*	使用 EEPMOV 指令执行结构替换
将循环条件视为符合易失性标准来处理 (Treat loop condition as volatile qualified)	*volatile_loop*	禁止循环迭代的优化
在预处理的源文件中禁止 #line (Suppress #line in preprocessed source file)	*noline*	禁止 **#line** 在预处理程序扩展的输出
允许寄存器声明 (Enable register declaration)	*enable_register*	优先将具有指定寄存器存储类的变量分配给寄存器。
更严格遵循 ANSI 规格 (Obey ANSI specifications more strictly)	*strict_ansi*	下列处理符合 ANSI 标准。 - 浮点运算的联合规则

用户定义的选项 (User defined options)：指定命令选项。

4.2.2 汇编程序选项

从 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) 对话框选取汇编 (Assembly) 标签。

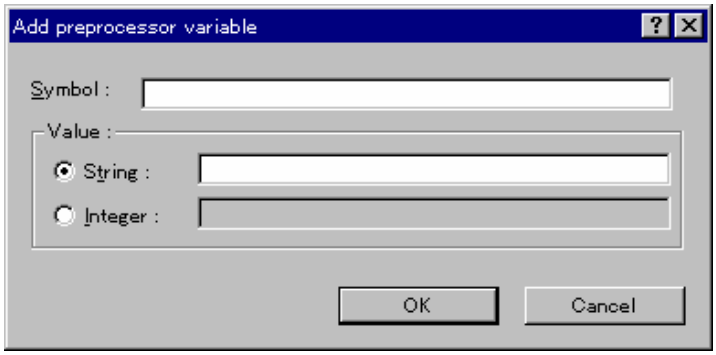
(1) **类别 (Category)：**[源 (Source)]



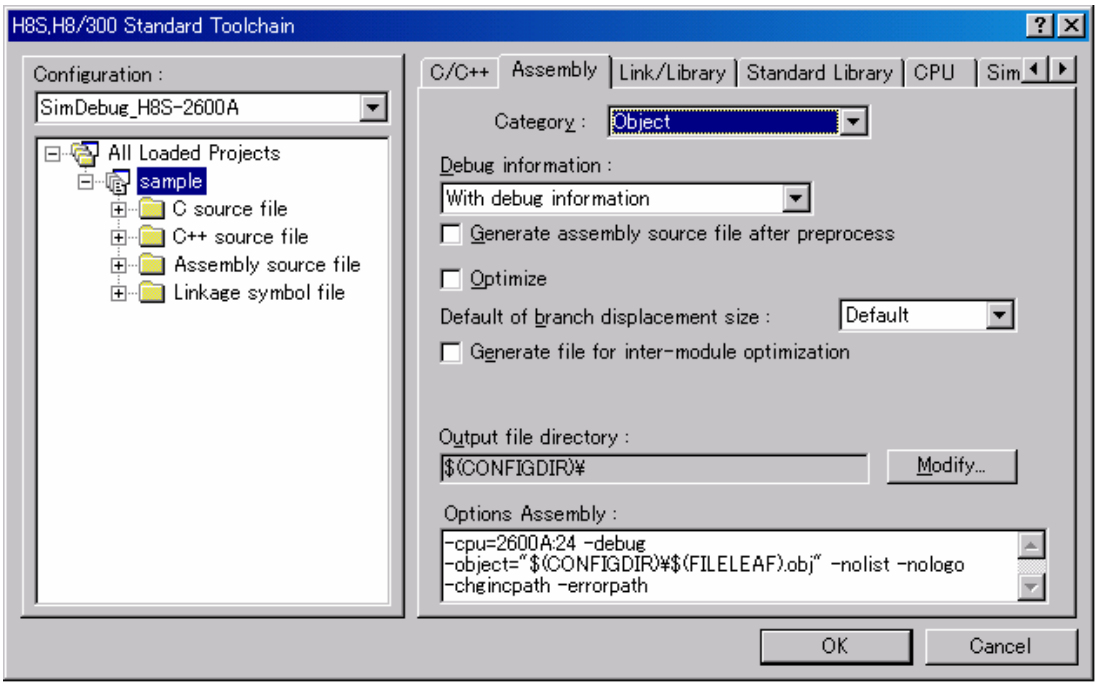
显示有关项目 (Show entries for)：

对话框菜单	命令选项	功能
包含文件目录 (Include file directories)	*include*	指定包含文件的目录
定义 (Defines)	*define*	定义字符串文字的替换
预处理程序变量* (Preprocessor variable*)	*assigna*	定义整数类型的预处理程序变量
	assignc	定义字符类型的预处理程序变量

注意： * 使用下列对话框指定。



(2) **类别 (Category)：**[目标 (Object)]



调试信息 (Debug information)：

对话框菜单	命令选项	功能
默认 (Default)	-	确认。仅限于调试指令
具有调试信息 (With debug information)	*debug*	允许调试信息的输出
没有调试信息 (Without debug information)	*nodebug*	禁止调试信息的输出

在预处理后生成汇编源文件 (Generate assembly source file after preprocess)

复选框	命令选项	功能
☑	*expand*	输出预处理程序扩展结果
☐	-	不输出预处理程序扩展结果

优化 (Optimize)

复选框	命令选项	功能
☑	*optimize*	指定优化
☐	*nooptimize*	不指定优化

转移位移大小的默认值 (Default of branch displacement size)：

对话框菜单	命令选项	功能
默认		由源文件中的指令描述指定。
8 位 (8 bit)	*br_relative=8*	若为转移指令选取了向前引用位移，则将位移大小指定为 8 位
16 位 (16 bit)	*br_relative=16*	若为转移指令选取了向前引用位移，则将位移大小指定为 16 位

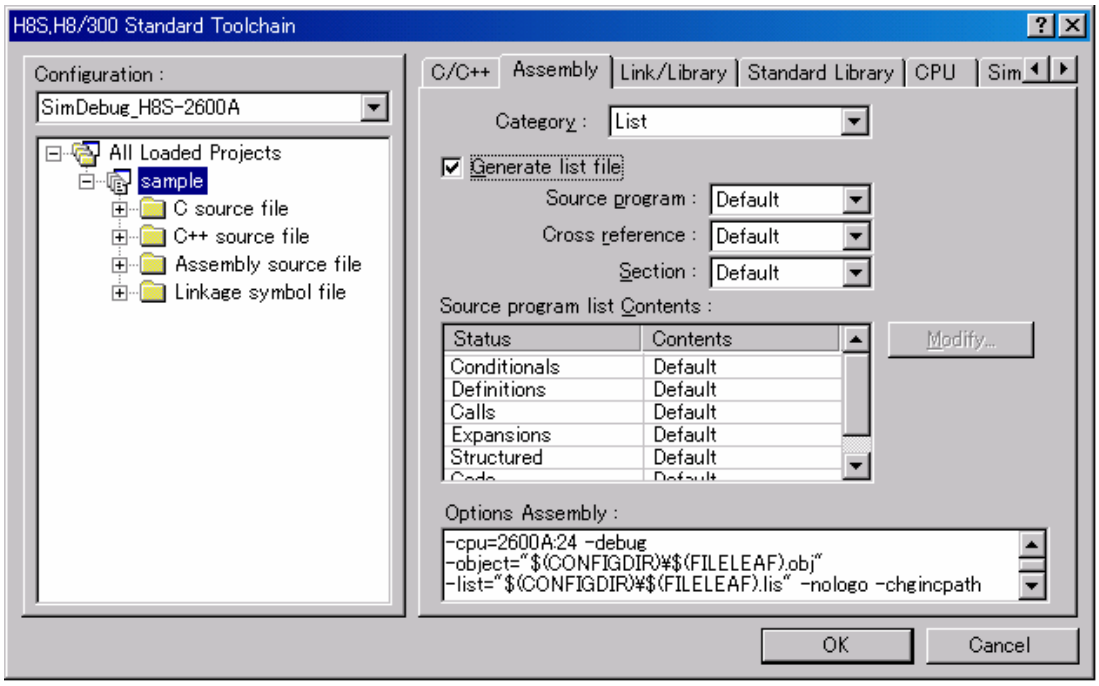
为模块间优化生成文件 (Generate file for inter-module optimization)

复选框	命令选项	功能
√	*goptimize*	输出模块间优化信息
		不输出模块间优化信息

输出目录 (Output directory)

对话框菜单	命令选项	功能
-	*object*	指定目标的输出目录

(3) **类别 (Category)：**[列表 (List)]



生成列表文件 (Generate list file)

复选框	命令选项	功能
√	*list*	输出汇编列表
	nolist	不输出汇编列表

源程序 (Source program)：

对话框菜单	命令选项	功能
显示 (Shown)	*source*	输出源程序列表
不显示 (Not shown)	*nosource*	不输出源程序列表

交叉参考 (Cross reference)：

对话框菜单	命令选项	功能
显示 (Shown)	*cross_refernce*	输出交叉参考列表
不显示 (Not shown)	*nocross_refernce*	不输出交叉参考列表

段 (Section)：

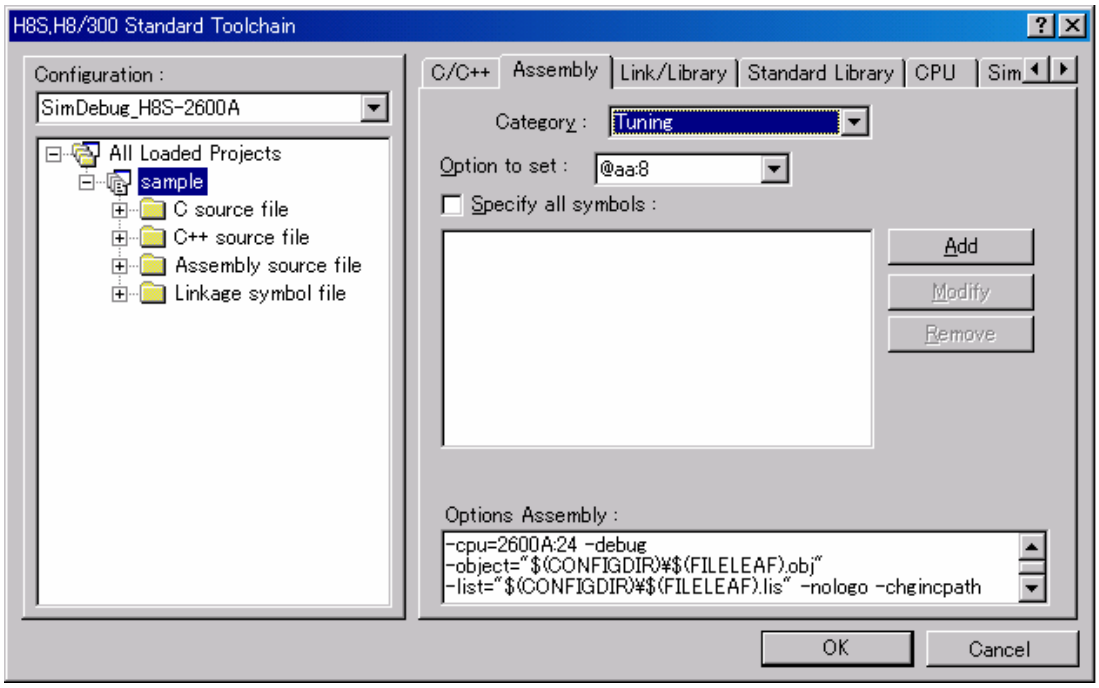
对话框菜单	命令选项	功能
显示 (Shown)	*section*	输出段信息列表
不显示 (Not shown)	*nosection*	不输出段信息列表

源程序列表的内容 (Source program list Contents)：指定要在列表文件上输出的内容。

对话框菜单	命令选项	功能
条件 (Conditionals)	*show=conditionals*	输出未满足在 .AIF 或 .AIFDEF 内所指定的条件的部分
定义 (Definitions)	*show=definitions*	输出宏定义、.AREPEAT 和 .AWHILE 定义，及 .INCLUDE、.ASSIGNA 和 .ASSIGNC 指令
调用 (Calls)	*show=calls*	输出宏调用语句和 .AIF、.AIFDEF，及 .AENDI 指令
扩展 (Expansions)	*show=expansions*	输出宏扩展及 .AREPEAT .AWHILE 扩展
结构的 (Structured)	*show=structured*	输出结构的汇编扩展
代码 (Code)	*show=code*	输出超过源语句行数的行，以显示在目标码显示内

注意： 若为每个选项选取了默认值，则源列表中的指令将被指定。

(4) **类别 (Category)：**[调谐 (Tuning)]



要设定的选项 (Option to set)：

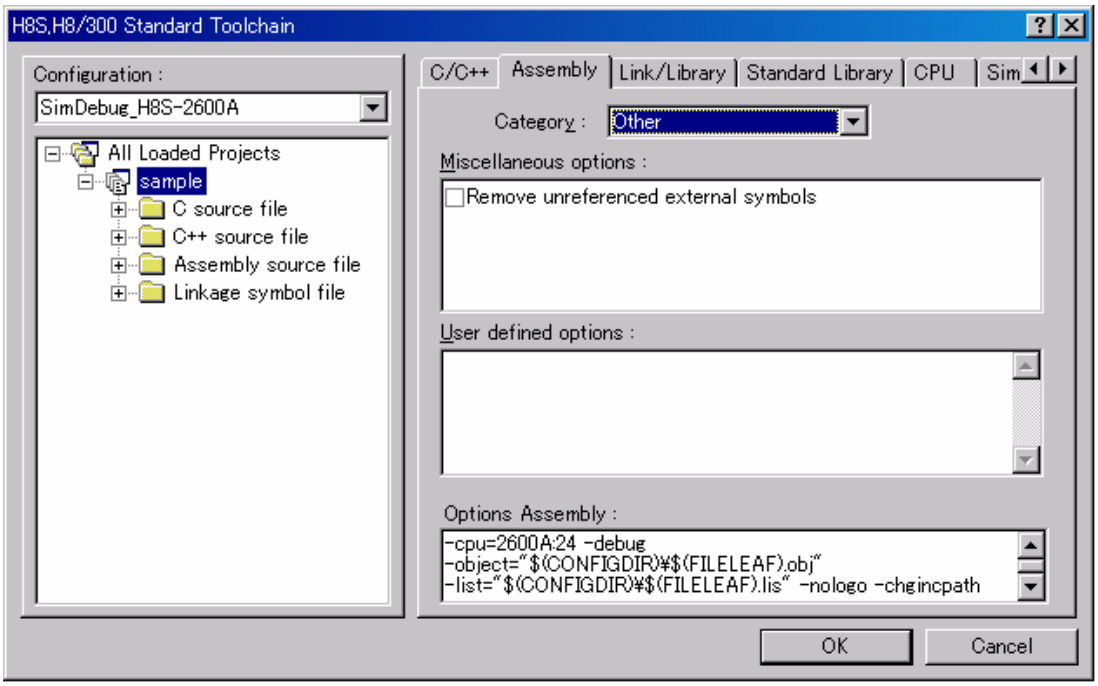
对话框菜单	命令选项	功能
@aa:8	*abs8*	指定 8 位绝对地址符号
@aa:16	*abs16*	指定 16 位绝对地址符号

注意： 选择外部参考符号或外部定义符号。

指定全部符号 (Specify all symbols)

复选框	命令选项
√	将指定的大小分配到外部参考符号和外部定义符号
	分配指定的大小到每个符号或不分配大小

(5) **类别 (Category)：**[其他 (Other)]



杂项 (Miscellaneous options)：

对话框菜单	复选框	命令选项	功能
移除未被参考的外部符号 (Remove unreferenced external symbols)	√	exclude	禁止输出未被参考的外部参考符号信息
		noexclude	允许输出未被参考的外部参考符号信息

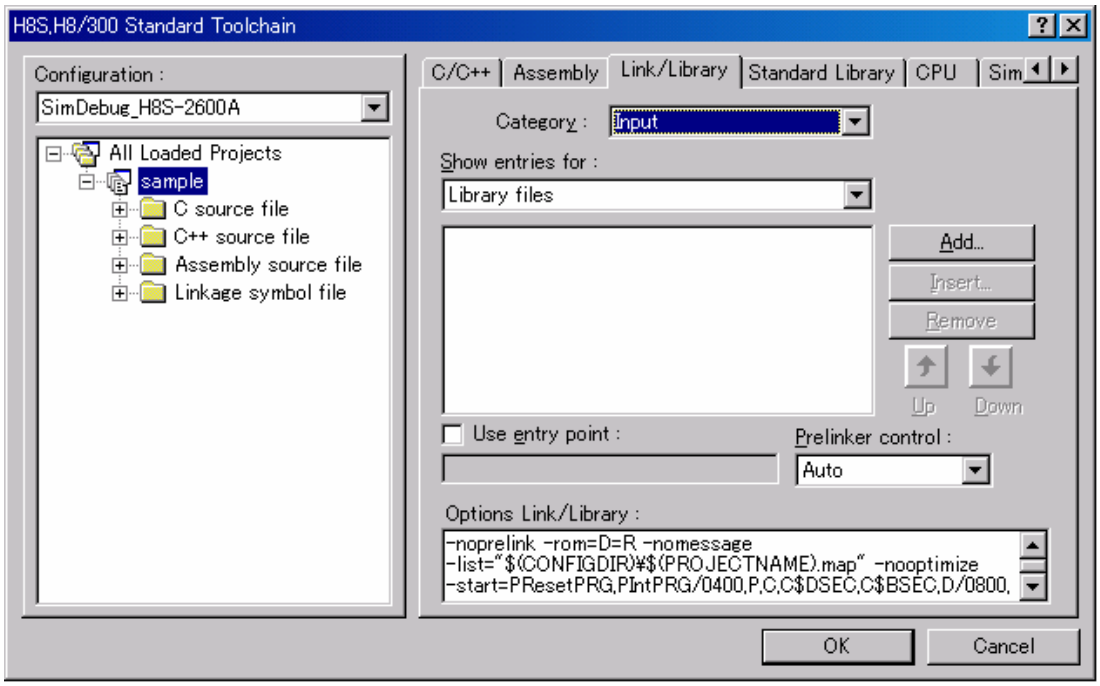
用户定义的选项 (User defined options)：

描述命令选项。

4.2.3 优化连接编辑程序选项

从 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) 对话框选取连接/程序库 (Link/Library) 标签。

(1) **类别 (Category)：**[输入 (Input)]



显示有关项目 **(Show entries for)：**

对话框菜单	命令选项	功能
程序库文件 (Library files)	*library*	指定输入程序库的名称
可再定位的文件和目标文件 (Relocatable files and object files)	*input*	指定输入文件
二进制文件 (Binary files)	*binary*	指定输入二进制文件
定义 (Defines)	*define*	强行定义未定义的符号

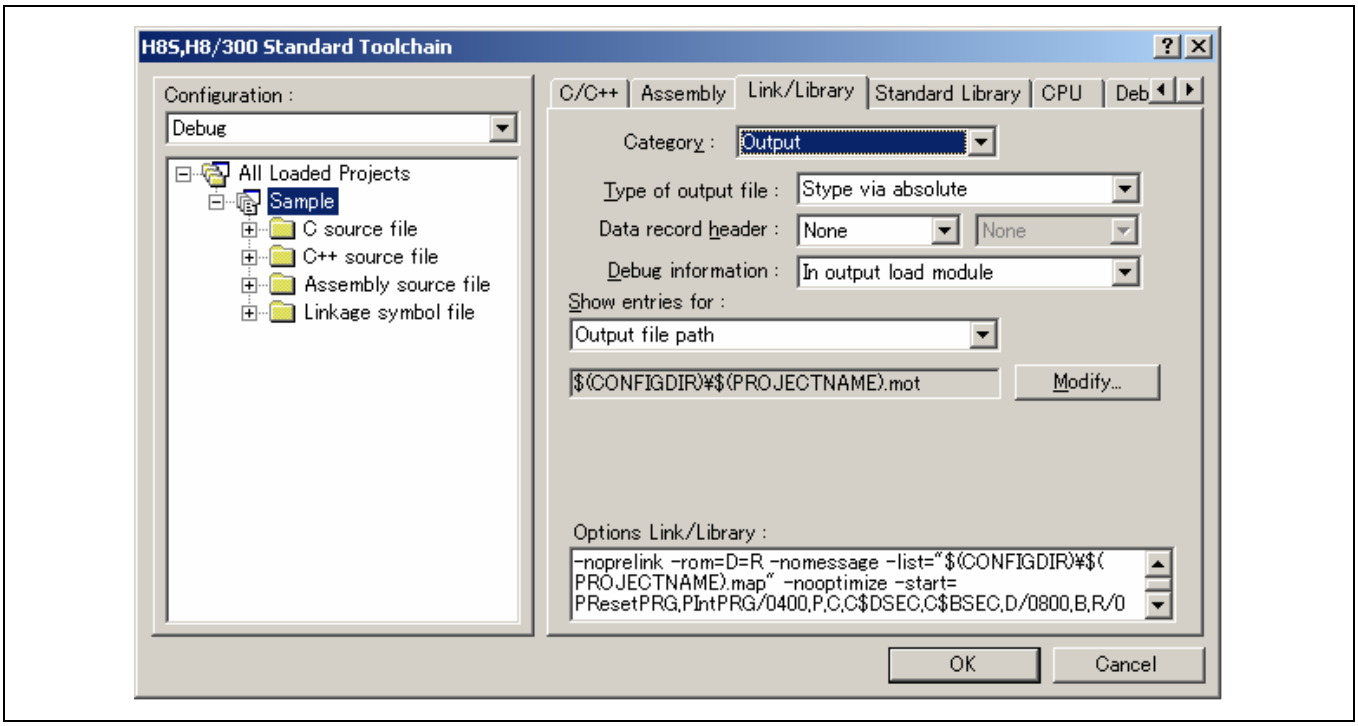
使用入口点 **(Use entry point)：**

复选框	命令选项	功能
√	*entry*	指定入口符号和入口地址
_	-	不指定入口符号和入口地址

预连接程序控制项 (Prelinker control)：

对话框菜单	命令选项	功能
自动 (Auto)	-	若没有示例信息文件，则不运行预连接程序
跳过预连接程序 (Skip prelinker)	noprelink	不运行预连接程序
运行预连接程序 (Run prelinker)	-	运行预连接程序

(2) **类别 (Category)：**[输出 (Output)]



输出文件的类型 (Type of output file)：

对话框菜单	命令选项	功能
绝对 (ELF/DWARF) (Absolute (ELF/DWARF))	*form=absolute*	以 ELF/DWARF 格式输出绝对装入模块
绝对 (SYSROF) (Absolute (SYSROF))	*form=absolute helfcnv.exe*	以 SYSROF 格式输出绝对装入模块
可再定位的 (Relocatable)	*form=relocate*	输出可再定位的装入模块
系统程序库 (System library)	*form=library=s*	输出系统程序库
用户程序库 (User library)	*form=library=u*	输出用户程序库
通过绝对来输出十六进制 (Hex via absolute)	*form=hexadecimal*	输出十六进制文件
通过绝对来输出 S 类型 (Stype via absolute)	*form=stype*	输出 S 类型文件
通过绝对来输出二进制 (Binary via absolute)	*form=binary*	输出二进制文件

数据记录标头 (Data record header)：

对话框菜单	命令选项	功能
-		根据各个地址输出一个记录
H16	*record=h16*	输出十六进制记录
H20	*record=h20*	输出扩展的十六进制记录
H32	*record=h32*	输出 32 位的十六进制记录
S1	*record=s1*	输出 S1 记录
S2	*record=s2*	输出 S2 记录
S3	*record=s3*	输出 S3 记录

调试信息 (Debug information)：

对话框菜单	命令选项	功能
无 (None)	*nodebug*	不输出调试信息
以输出到装入模块的形式 (In output load module)	*debug*	将调试信息输出到装入模块
以分隔调试文件的形式 (In Separate Debug File)	*sdebug*	将调试信息输出到文件

显示有关项目 (Show entries for)：

对话框菜单	命令选项	功能
输出文件的路径 (Output file path)	-	为输出文件指定路径
ROM 到 RAM 的映像段 (ROM to RAM mapped sections)	*rom*	预留 RAM 区域以解决 RAM 内地址的符号再定位
分隔的输出文件 (Divided output files)	-	设定或不设定输出范围
指定要填入未使用区的值 (Specify value filled in unused area)	-	指定要输出到未使用区的值
输出消息 (Output messages)	-	指定信息级别消息是否被输出
缩小空区域 (Reduce empty areas)	-	缩小编译后生成为段的边界对齐的空区域

压制的信息级别消息 (Repressed information level messages)：

复选框	命令选项	功能
☑	*nomessage*	不输出信息级别消息
☐	*message*	输出信息级别消息

通知未使用的符号 (Notify unused symbol)：

复选框	命令选项	功能
☑	*msg_unused*	通知用户从未被参考的定义符号
☐	-	不通知用户从未被参考的定义符号

分隔的输出文件 (Divided output files)：

复选框	命令选项	功能
☑	*output*	指定输出文件名称并设定输出范围
☐	-	指定输出文件名称但不设定输出范围

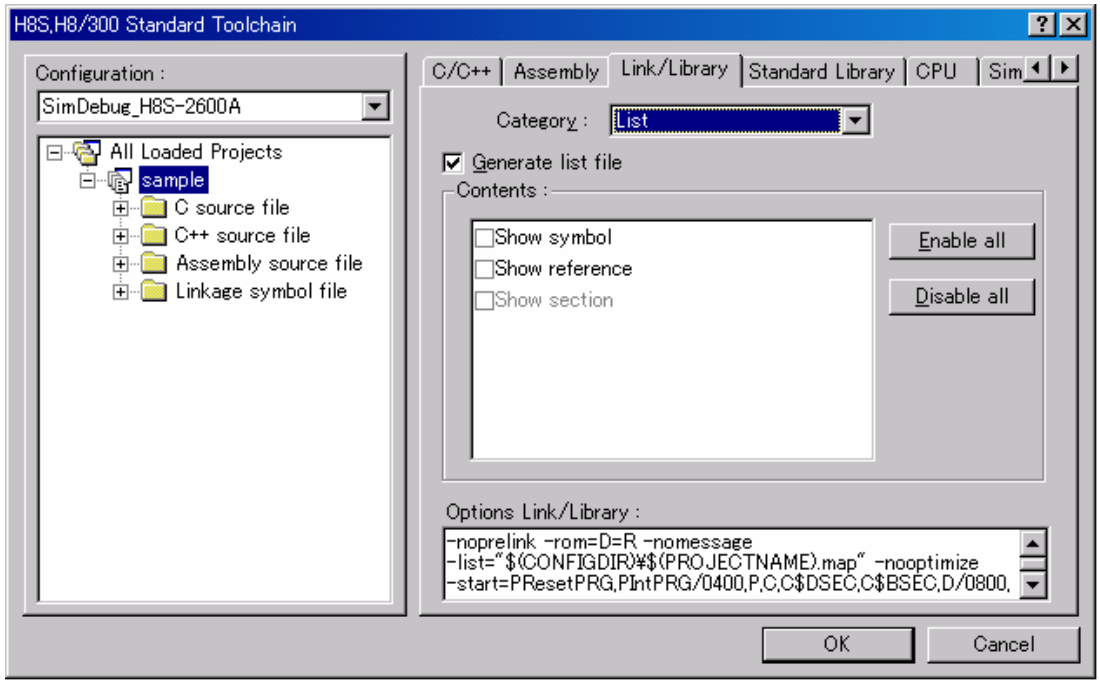
输出填充数据 (Output padding data)：

复选框	命令选项	功能
☑	*space=* *<numerical value>*	指定要输出到未使用区的值
☐	-	不指定要输出到未使用区的值

缩小边界对齐的空区域 (Reduce empty areas of boundary alignment)：

复选框	命令选项	功能
☑	*data_stuff*	缩小编译后生成为段的边界对齐的空区域
☐	-	不缩小编译后生成为段的边界对齐的空区域

(3) **类别 (Category)：** [列表 (List)]



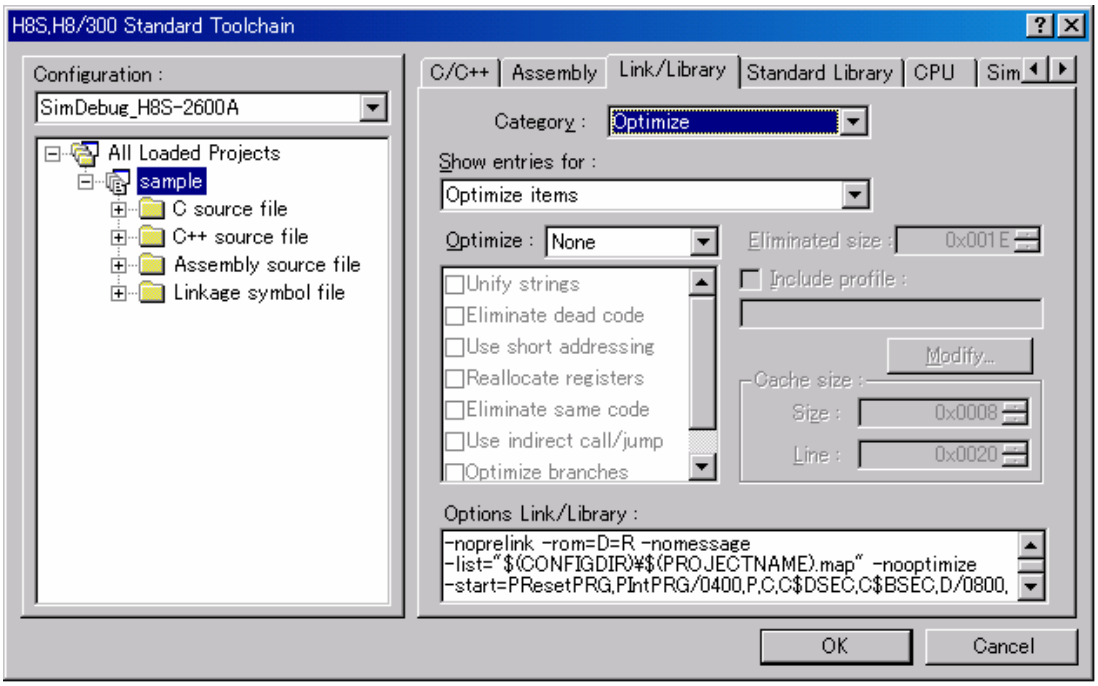
生成列表文件 (Generate list file)：

复选框	命令选项	功能
√	*list*	输出列表文件
	-	不输出列表文件

内容 (Contents)：

对话框菜单	命令选项	功能
显示符号 (Show symbol)	*show=symbol*	输出符号名称的列表
显示参考 (Show reference)	*show=reference*	输出符号参考的数量
显示段 (Show secion)	*sher=section*	输出段的列表
显示交叉参考 (Show cross reference)	*show=xreference*	输出交叉参考信息

(4) **类别 (Category)：**[优化 (Optimize)]



显示有关项目 **(Show entries for)：**

对话框菜单	**命令选项**	**功能**
优化项目 (Optimize items)	-	指定优化
禁止项目 (Forbid item)	-	禁止指定符号和地址区的优化
死码的删除 (Elimination of dead code)	*Symbol_forbid*	指定禁止了未被参考符号的删除优化的变量或函数名称
相同代码的删除 (Elimination of same code)	*Samecode_forbid*	指定禁止了相同代码的删除优化的函数名称
将短寻址使用到 (Use of short addressing to)	*Variable_forbid*	指定禁止以使用短绝对寻址模式来达到优化的变量名称
将间接调用/跳转使用到 (Use of indirect call/jump to)	*Function_forbid*	指定禁止以使用间接寻址模式来达到优化的函数名称
存储器分配在 (Memory allocation in)	*Absolute_forbid*	指定不执行地址分配的地址区

优化 (Optimize)：

对话框菜单		命令选项	功能
全部 (All)		*Optimize*	允许所有优化项目
速度 (Speed)		*Optimize∆speed*	对速度执行优化
安全 (Safe)		*Optimize∆safe*	执行安全优化
定制 (Custom)		*Optimize∆*	允许优化项目的选择
	统一字符串 (Unify strings)	*String_unify*	统一常数或字符串的文字
	删除死码 (Eliminate dead code)	*Symbol_delete*	删除未被参考的符号
	使用短寻址 (Use short addressing)	*Variable_access*	使用短的绝对寻址模式
	再分配寄存器 (Reallocate registers)	*Register*	再分配寄存器
	删除相同的代码 (Eliminate same code)	*Same_code*	统一指令码
	使用间接调用/跳转 (Use indirect call/jump)	*Function_call*	使用间接寻址模式
	优化转移 (Optimize branches)	*Branch*	优化转移指令
	删除的大小 (Eliminated size)：	*Samesize*	指定删除相同代码的目标大小
无 (None)		*Nooptimize*	禁止优化

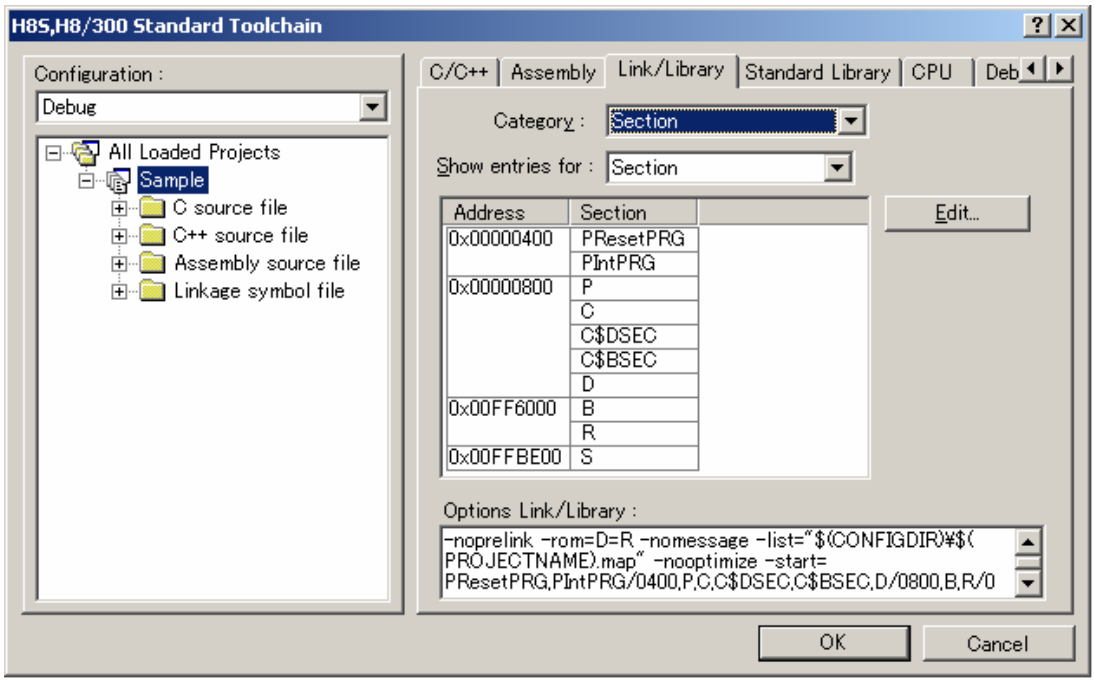
包含配置文件 (Include profile)

复选框	命令选项	功能
√	profile	指定配置文件的信息文件
_	-	不指定配置文件的信息文件

高速缓存大小 (Cache size)：

对话框菜单	命令选项	功能
大小 (Size)	cachesize =sized	指定高速缓存大小
行 (Line)	cachesize =align	指定高速缓存对齐大小

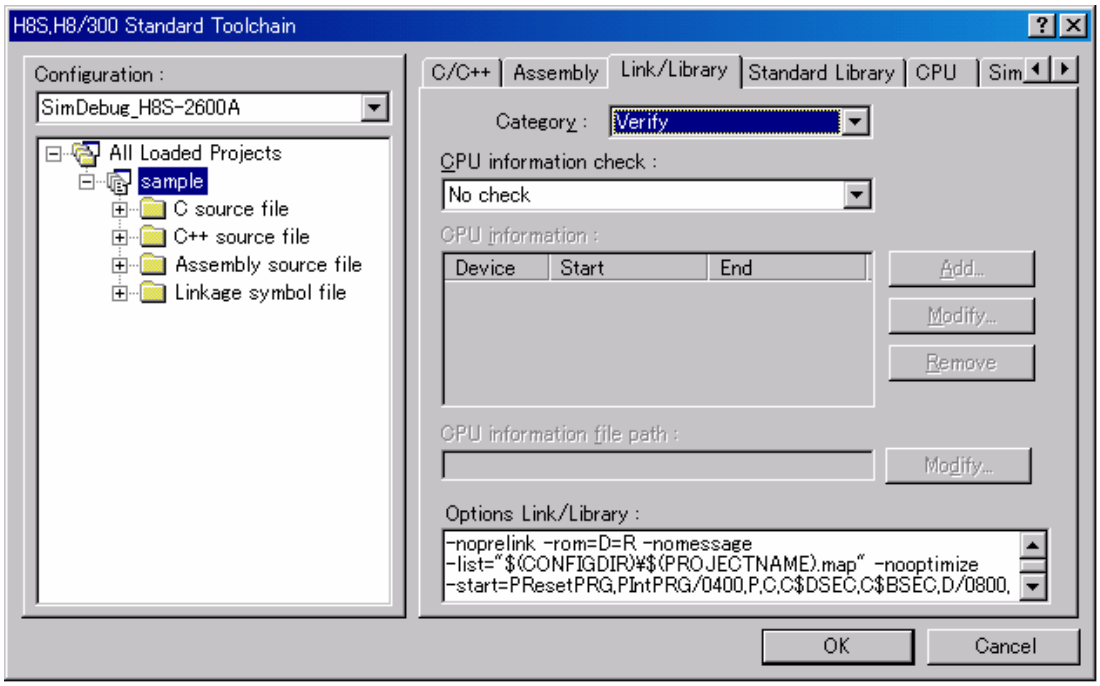
(5) **类别 (Category)：**[段 (Section)]



显示有关项目 (Show entries for)：

对话框菜单	命令选项	功能
段 (Section)	*start-*	指定每个段的起始地址和连接顺序
符号文件 (Symbol file)	*fsymbol*	将由连接函数处理的外部定义符号以汇编程序指令格式输出到文件

(6) **类别 (Category)：**[验证 (Verify)]



CPU 信息检查 (CPU information check)：

对话框菜单	命令选项	功能
不检查 (No check)	-	不检查 CPU 分配
检查 (Check)	*CPU*	根据 CPU 信息文件检查存储器分配
使用 CPU 信息文件 (Use CPU information file)	*CPU*	根据现有的 CPU 信息文件检查存储器分配

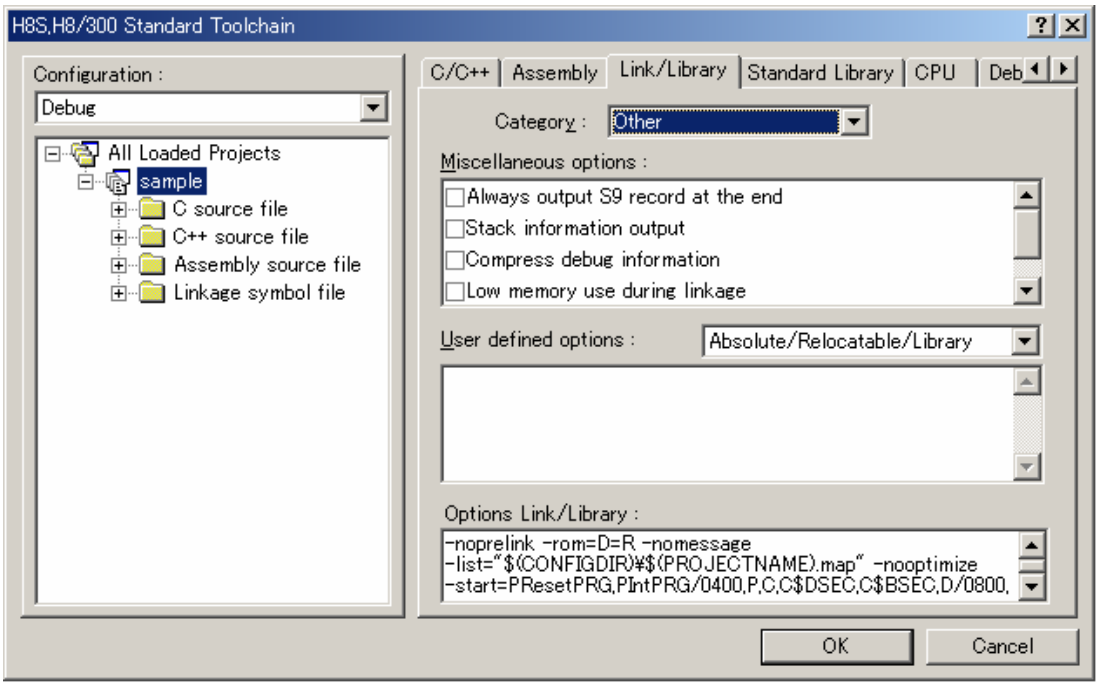
CPU 信息 (CPU information)

对话框菜单	命令选项	功能
-	*{ROm | RAm}= <address range>*	创建或修改 CPU 信息文件 CPU 指定存储器类型，然后再指定每个存储器的地址

CPU 信息文件路径 (CPU information file path)

对话框菜单	命令选项	功能
-	*<File name>*	指定现有的 CPU 信息文件

(7) **类别 (Category)：**[其他 (Other)]

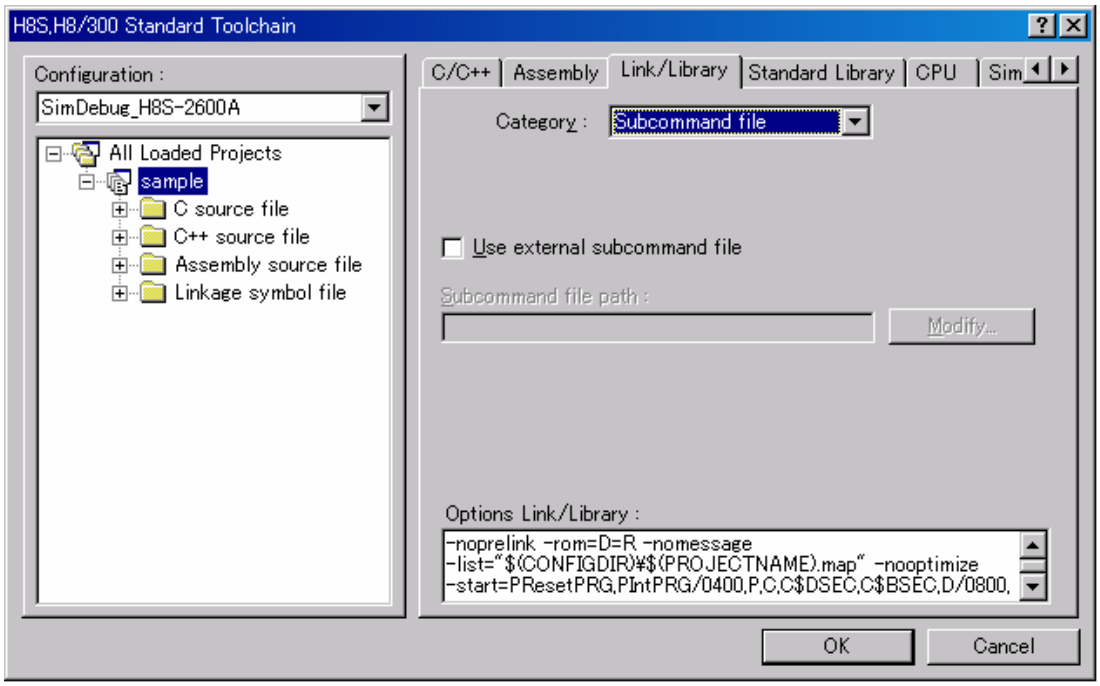


杂项 (Miscellaneous options)：指定其他功能。

对话框菜单	命令选项	功能
始终在结束部分输出 S9 记录 (Always output S9 record at the end)	*S9*	固定输出 S9 记录
堆栈信息输出 (Stack information output)	*stack*	输出堆栈使用信息文件
压缩调试信息 (Compress debug information)	*compress*	压缩调试信息
	nocompress	不压缩调试信息
连接期间使用低存储 (Low memory use during linkage)	*Memory=high*	所占用的存储大小和平常一样
	Memory=low	所占用的存储大小被缩减

用户定义的选项 (User defined options)：指定命令选项。

(8) **类别 (Category)：**[子命令文件 (Subcommand file)]



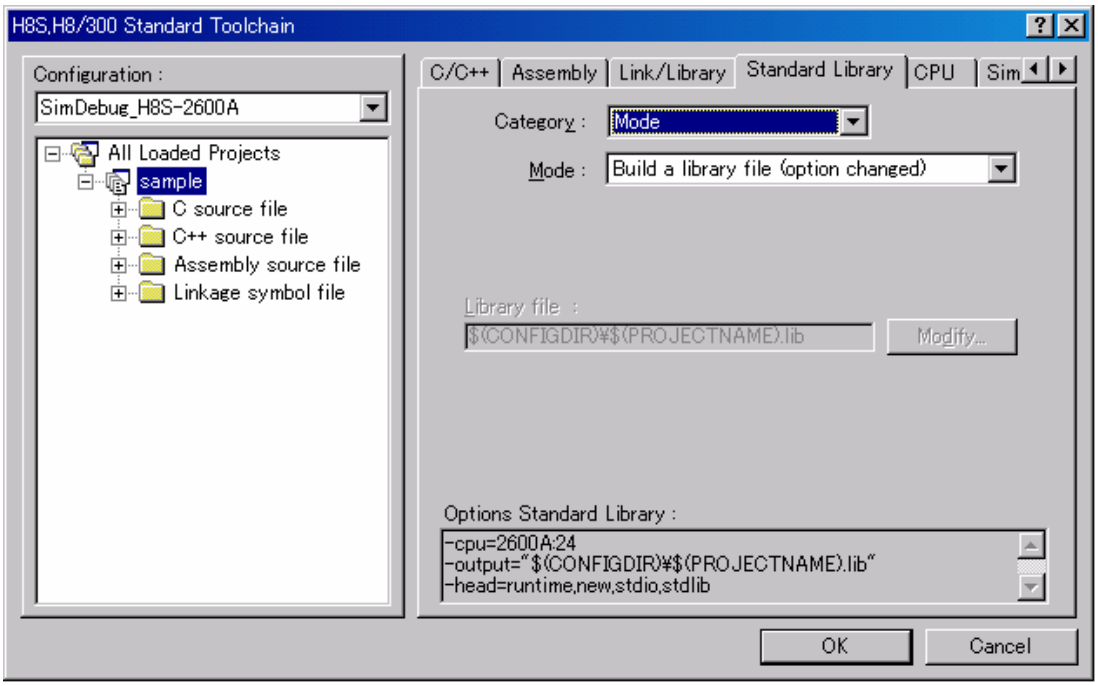
使用外部子命令文件 (Use external subcommand file)

复选框	命令选项	功能
√	Subcommand	指定子命令文件的选项
	-	不指定子命令文件

4.2.4 标准程序库生成程序选项

从 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) 对话框选取标准程序库 (Standard Library) 标签。

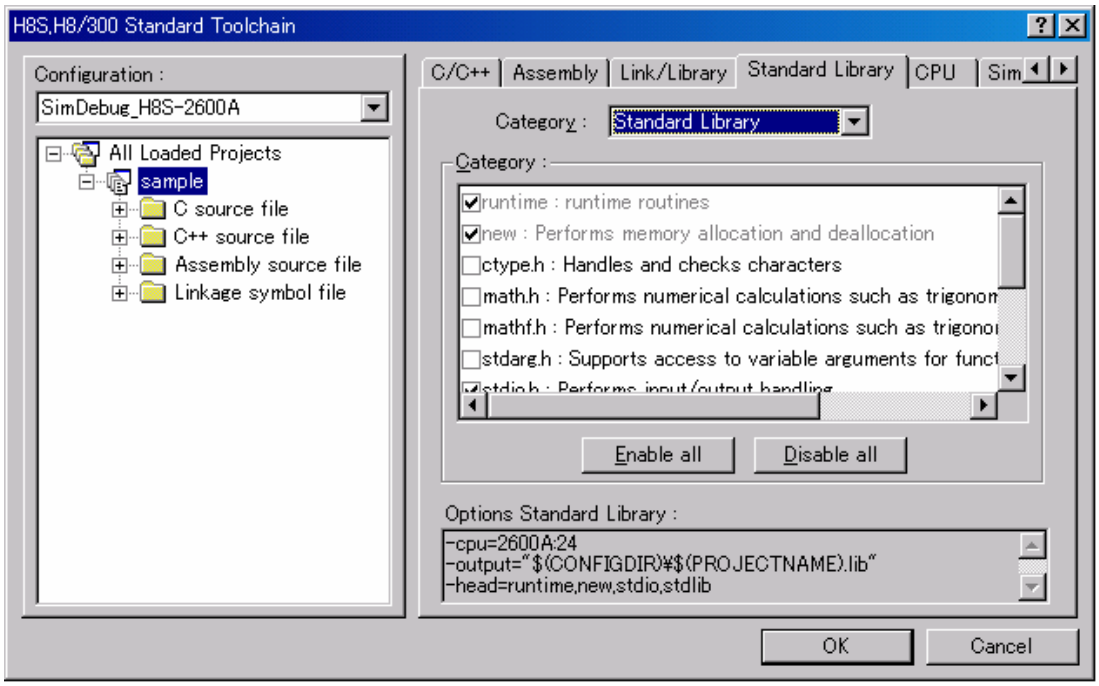
(1) **类别 (Category)：**[模式 (Mode)]



模式 (Mode)：

对话框菜单	命令选项	功能
创建程序库文件（任何时候） (Build a library file (anytime))	-	创建新的标准程序库
创建程序库文件（选项更改） (Build a library file (Option Changed))	-	当选项更改时，创建新的标准程序库
使用现有的程序库文件 (Use an existing library file)	-	连接现有的标准程序库
不添加程序库文件 (Do not add a library file)	-	不连接标准程序库

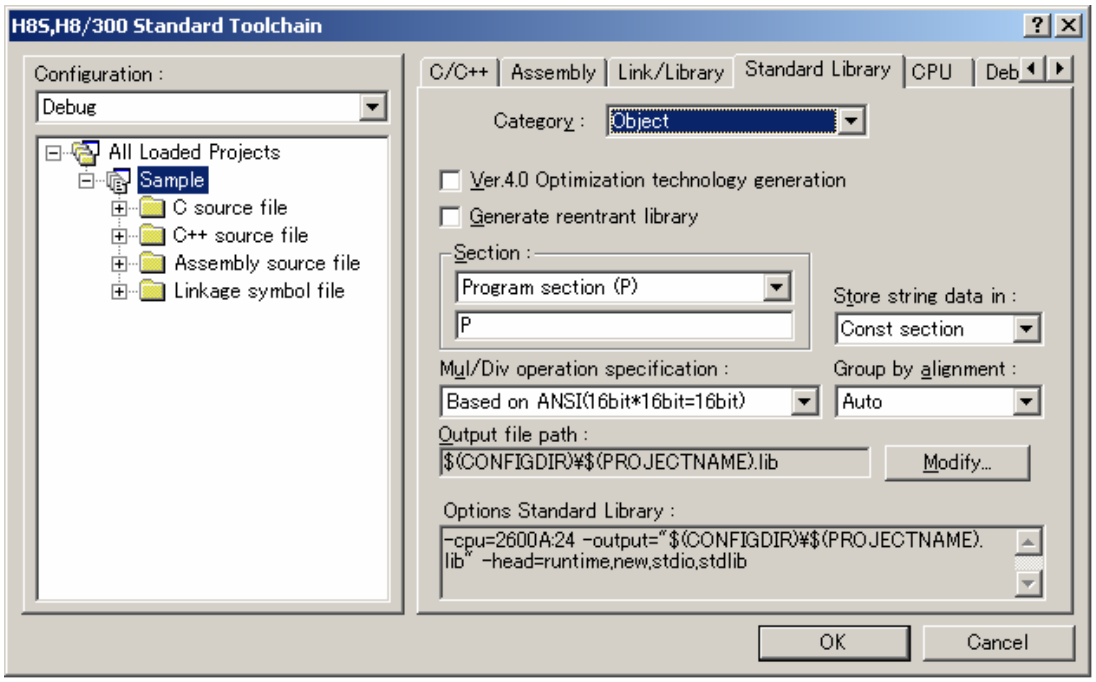
(2) **类别 (Category)：**[标准程序库 (Standard Library)]



类别 (Category)：

对话框菜单	命令选项	功能
运行时 (runtime)	*Head=RUNTIME*	指定运行时例程
新建 (new)	*Head=NEW*	指定由新建所声明的 EC++
ctype.h	*Head=CTYPE*	指定 ctype.h
math.h	*Head=MATH*	指定 math.h
mathf.h	*Head=MATHF*	指定 mathf.h
stdarg.h	*Head=STDARG*	指定 stdarg.h
stdio.h	*Head=STDIO*	指定 stdio.h
stdlib.h	*Head=STDLIB*	指定 stdlib.h
string.h	*Head=STRING*	指定 string.h
ios(EC++)	*Head=IOS*	指定 ios(EC++)
complex(EC++)	*Head=COMPREX*	指定 complex(EC++)
string(EC++)	*Head=CPPSTRING*	指定 string(EC++)

(3) **类别 (Category)：**[目标 (Object)]



版本 4.0 优化技术生成（受 HEW 版本 4.0 或以上 版本支持）

复选框	命令选项	功能
√	*legacy=v4*	输出与 H8S 版本 4.0 优化技术的生成兼容的目标
	-	输出 H8S 版本 6.1 优化技术生成的目标

生成可重入程序库 (Generate reentrant library)

复选框	命令选项	功能
√	*reent*	创建可重入的函数
	-	不创建可重入的函数

段 (Section)：

对话框菜单	命令选项	功能
-	section	更改默认段名称

将字符串数据存储在 (Store string data in)：

对话框菜单	命令选项	功能
常数段 (Const section)	*string=const*	将字符串文字输出到常数区
数据段 (Data section)	*string=data*	将字符串文字输出到初始化数据区

乘除运算规格 (Mul/Div operation specifications)

对话框菜单	命令选项	功能
基于 ANSI（保证 16 位为 16 位*16 位的结果）(Based on ANSI (Guarantee 16bit as a result of 16bit*16bit))	*nocpuexpand*	根据 ANSI C 语言规格，以代码开发乘法或除法
非 ANSI（保证 32 位为 16 位*16 位的结果）(Non ANSI (Guarantee 32bit as a result of 16bit*16bit))	*cpuexpand*	根据 CPU 指令规格，以代码开发乘法或除法

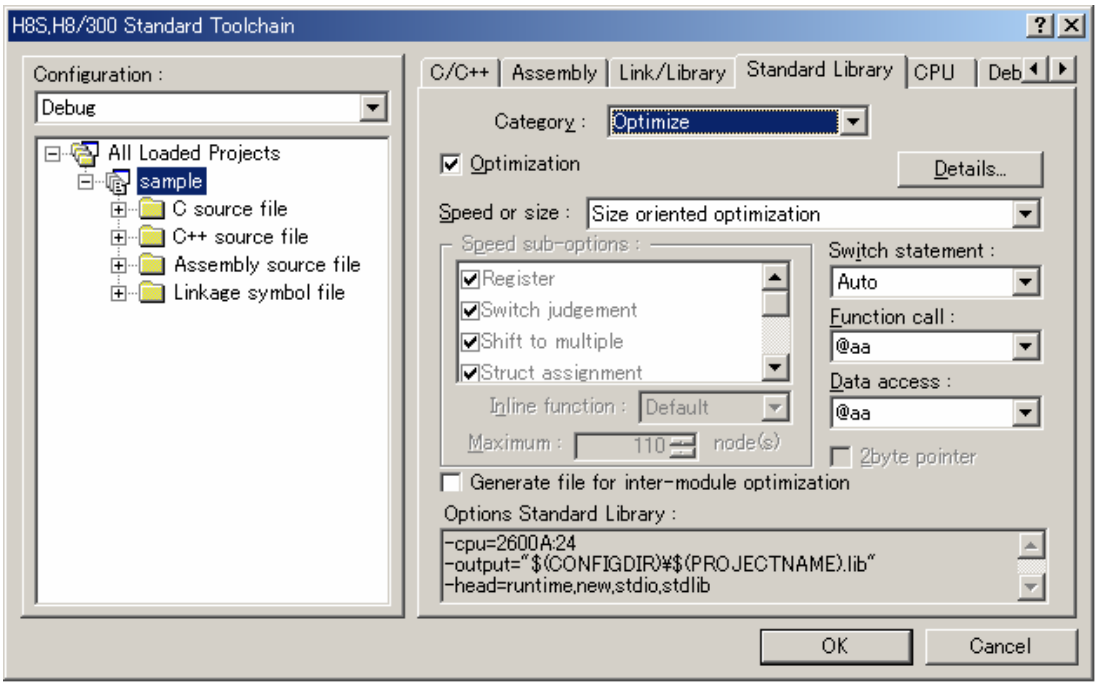
输出文件的路径 (Output file path)

对话框菜单	命令选项	功能
-	output	指定程序库文件的输出目录

以对齐方式分组 (Group by alignment)

对话框菜单	命令选项	功能
无 (None)	*noalign*	以定义的顺序分配定义的变量
自动 (Auto)	*align*	分配变量以缩减边界对齐所形成的空间
4 字节 (4byte)	*align=4*	将数据段分隔为 4、2、1 字节的边界对齐段，然后分配入多个 4、2、1 地址，以增进存取速度

(4) **类别 (Category)：**[优化 (Optimize)]



优化 (Optimization)

复选框	命令选项	功能
☑	optimize=1	指定优化
☐	optimize=0	不指定优化

速度或大小 (Speed or size)：指定优化的格式。

对话框菜单		命令选项	功能
面向大小的优化 (Size oriented optimization)		-	对大小执行优化
面向速度的优化 (Speed oriented optimization)		*speed*	速度的优化
速度子选项 (Speed sub-options)	**寄存器 (Register)**	*speed=register*	通过 PUSH 和 POP 指令以更快的速度执行寄存器存储/恢复扩展
	切换判断 (Switch judgement)	*speed=switch*	以更快的速度开发切换语句
	转移到多个 (Shift to multiple)	*speed=shift*	以更快的速度开发转移操作
	结构赋值 (Struct assignment)	*speed=struct*	以更快的速度执行结构扩展及替换表达式
	表达式 (Expression)	*speed=expression*	以更快的速度执行算术操作、比较，及替换表达式处理
	循环优化 (Loop optimization)	*speed=loop1*	感应变量的删除
	解开循环 (Loop unrolling)	*speed=loop2*	感应变量和循环扩展的删除
	内联函数的最大节点 (Inline function Maximum:node(s))	*speed=inline [=<data>]*	自动内联扩展

为模块间优化生成文件 (Generate file for inter-module optimization)

复选框	命令选项	功能
√	*goptimize*	输出模块间优化的加载信息
	-	不输出模块间优化的加载信息

切换语句 (Switch statement)：指定切换语句扩展方法。

对话框菜单	命令选项	功能
自动 (Auto)	*case=auto*	根据速度选项规格来决定切换语句扩展方法
如果…就 (If then)	*case=ifthen*	以“如果…就”的方法来执行切换语句扩展
表 (Table)	*case=table*	以表跳转 (table jump) 的方法来执行切换语句扩展

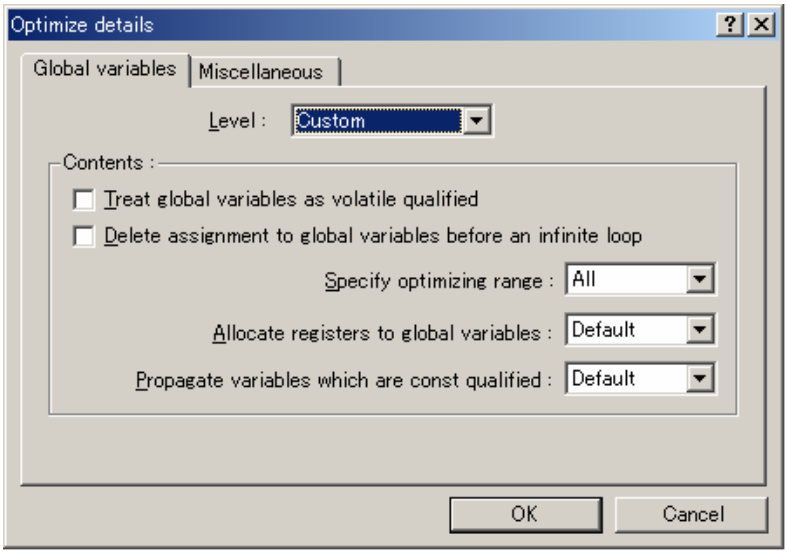
函数调用 (Function call)：选择函数调用的方法。

对话框菜单	命令选项	功能
@aa	-	选取普通函数调用
@@aa:8	*idirect=normal*	选取存储器间接函数调用
@@vec:7	*idirect=extended*	选取扩展存储间接函数调用

数据存取 (Date access)：选取数据存取模式。

对话框菜单	命令选项	功能
@aa	-	选取普通数据存取
@aa:8	*abs8*	选取 8 位绝对地址存取
@aa:16	*abs16*	选取 16 位绝对地址存取

(a) **[详细资料 (Details)] 按钮：**[全局变量 (Global Variables)] 标签



级别 (Level)：指定外部变量优化的级别

对话框菜单	命令选项	功能
1 级 (Level 1)		禁止所有外部变量优化。
	volatile	[处理全局… (Treat global…] = [选中 (Checked)]
	infinite_loop=0	[删除赋值… (Delete assignment…] = [未选中 (Not Checked)]
	opt_range=noblock	[指定优化… (Specify optimizing) …] = [没有块 (No block)]
	global_alloc=0	[分配寄存器… (Allocate registers) …] = [禁止 (Disable)]
	const_var_propagate=0	[传播变量… (Propagate variables) …] = [禁止 (Disable)]
2 级 (Level 2)		优化不具有易失性说明符的外部变量。 禁止优化跨越循环或转移进行扩展的外部变量。
	novolatile	[处理全局… (Treat global…)] = [未选中 (Not Checked)]
	infinite_loop=0	[删除赋值… (Delete assignment…] = [未选中 (Not Checked)]
	opt_range=noblock	[指定优化… (Specify optimizing) …] = [没有块 (No block)]
	global_alloc=0	[分配寄存器… (Allocate registers) …] = [禁止 (Disable)]
	const_var_propagate=0	[传播变量… (Propagate variables) …] = [禁止 (Disable)]
3 级 (Level 3)		优化在整个函数内不具有易失性说明符的外部变量。
	novolatile	[处理全局… (Treat global…)] = [未选中 (Not Checked)]
	infinite_loop=0	[删除赋值…(Delete assignment…] = [未选中 (Not Checked)]
	opt_range=all	[指定优化… (Specify optimizing) …] = [全部 (All)]
	global_alloc=1	[分配寄存器… (Allocate registers) …] = [允许 (Enable)]
	const_var_propagate=1	[传播变量… (Propagate variables) …] = [允许 (Enable)]
定制 (Custom)	-	根据用户指定的选项来优化外部变量

将全局变量视为符合易失性标准来处理 (Treat global variables as volatile qualified)

复选框	命令选项	功能
√	*volatile*	禁止外部变量的优化。
_	*novolatile*	优化不具有易失性说明符的外部变量。

在无穷循环之前删除到全局变量的赋值 (Delete assignment to global Jariables before an infinite loop)

复选框	命令选项	功能
√	*infinite_loop=1*	删除紧随于无穷循环之前的赋值表达式，该赋值是不在无穷循环中使用的对外部变量的赋值。
_	*infinite_loop=0*	禁止删除无穷循环之前的外部变量之赋值表达式。

指定优化的范围 (Specify optimizing range)

对话框菜单	命令选项	功能
全部 (All)	*opt_range=all*	优化整个函数内的外部变量。
没有循环 (No loop)	*opt_range=noloop*	循环中的外部变量及循环迭代条件中所使用的外部变量将不被优化。
没有块 (No block)	*opt_range=noblock*	跨越转移扩展的外部变量（包括循环）将不被优化。

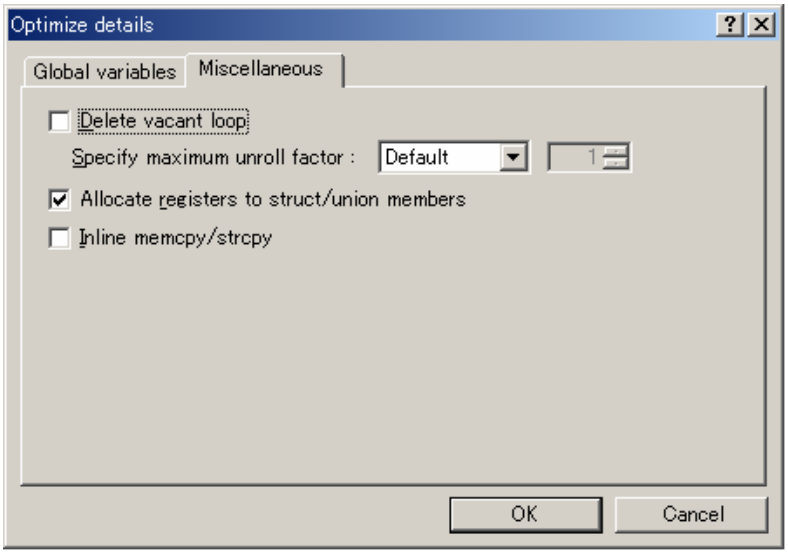
将寄存器分配到全局变量 (Allocate registers to global variables)

对话框菜单	命令选项	功能
禁止 (Disable)	*global_alloc=0*	禁止将外部变量分配到寄存器。
允许 (Enable)	*global_alloc=1*	将外部变量分配到寄存器。
默认 (Default)	*global_alloc=1*	将外部变量分配到寄存器。

传播符合常数标准的变量 (Propagate variables which are const qualified)

对话框菜单	命令选项	功能
禁止 (Disable)	*const_var_propagate=0*	禁止以 **const** 声明的外部常数的常数传播。
允许 (Enable)	*const_var_propagate=1*	执行以 **const** 声明的外部常数的常数传播。
默认 (Default)	*const_var_propagate=1*	执行以 **const** 声明的外部常数的常数传播。

(b) **[详细资料 (Details)] 按钮：**[杂项 (Miscellaneous)] 标签



删除空的循环 (Delete vacant loop)

复选框	命令选项	功能
√	*del_vacant_loop=1*	删除其中不含语句的循环。
	del_vacant_loop=0	禁止删除空的循环，即使循环内不含语句。

指定最大的解开因数 (Specify maximum unroll factor)

对话框菜单	命令选项	功能
默认 (Default)	*max_unroll=2 or 1*	2 或 1 被假设为将要被扩展的最大循环数。
定制 (Custom)	*max_unroll=* *< numeric value >*	指定将要被扩展的最大循环数。可为 <数值> 指定从 1 到 32 的整数。

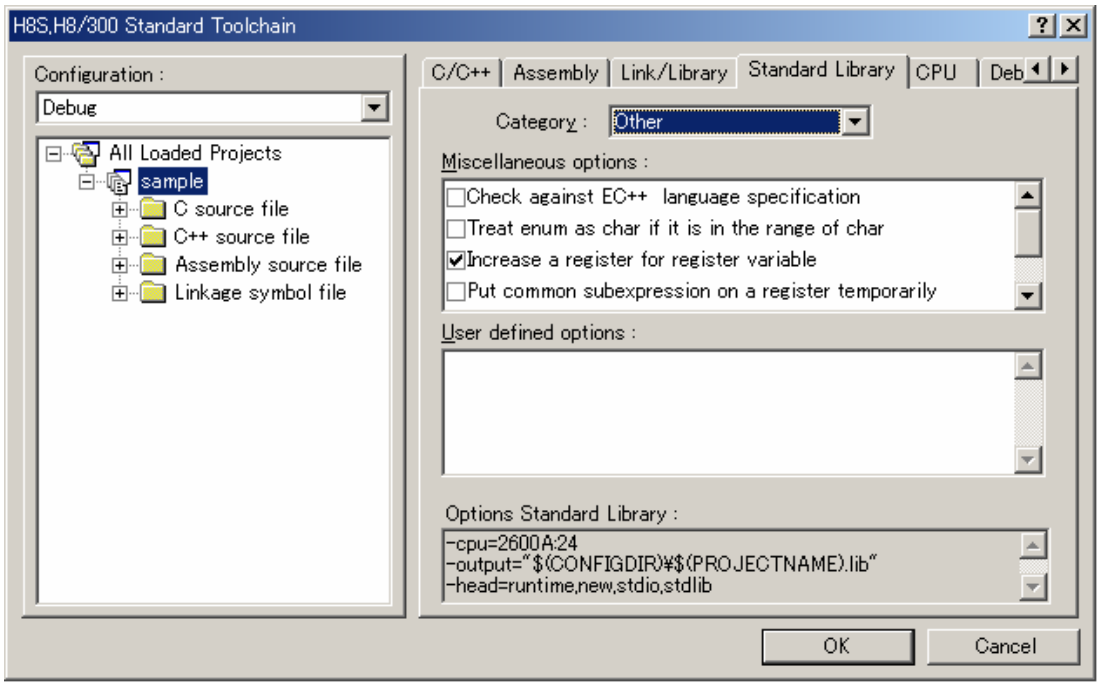
将寄存器分配到结构/联合成员 (Allocate registers to struct/union members)

复选框	命令选项	功能
√	*struct_alloc=1*	将结构/联合成员分配到寄存器。
	struct_alloc=0	禁止将结构/联合成员分配到寄存器。

内联 memcpy/strcpy (Inline memcpy/strcpy)

复选框	命令选项	功能
√	*library=intrinsic*	为 **memcpy** 和 **strcpy** 执行内联扩展。
	library=function	为 **memcpy** 和 **strcpy** 产生函数调用。

(5) **类别 (Category)**：[其他 (Other)]



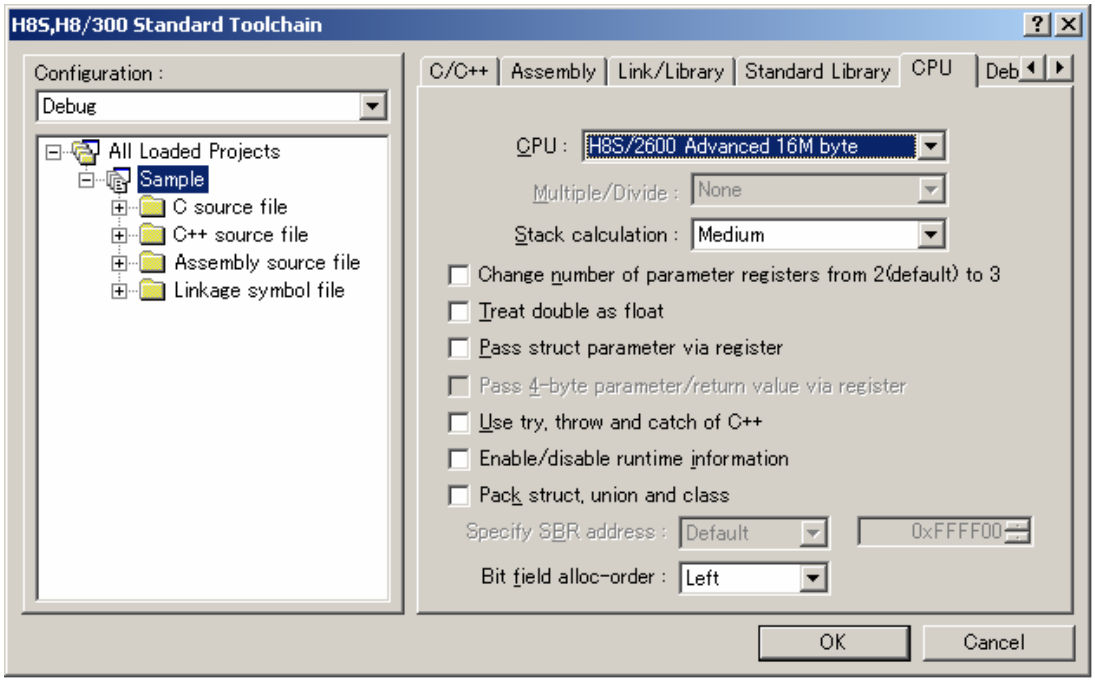
杂项 (Miscellaneous options)：

对话框菜单	命令选项	功能
对照 EC++ 语言规格 (Check against EC++ language specification)	*ecpp*	根据 EC++ 语言规格检查语法
将循环条件视为符合易失性标准来处理 (Treat loop condition as volatile qualified)	*volatile_loop*	禁止循环迭代条件的优化
将枚举当作字符处理，若它处于字符的范围内 (Treat enum as char if it is in the range of char)	*byteenum*	将枚举类型的数据当作字符来处理
为寄存器变量增加寄存器 (Increase a register for register variable)	*Regexpansion* *noregexpansion*	将变量分配寄存器的数量指定为 2 将变量分配寄存器的数量指定为 3
将公用子表达式暂时放置在寄存器上 (Put common subexpression on a register temporarily)	*cmncode*	增强删除公用表达式的优化功能
以块副本的形式使用 EEPMOV (Use EEPMOV in block copy)	*eepmov*	使用 EEPMOV 指令执行结构替换
将循环条件视为符合易失性标准来处理 (Treat loop condition as volatile qualified)	*volatile_loop*	禁止循环迭代的优化
允许寄存器声明 (Enable register declaration)	*enable_register*	优先将具有指定寄存器存储类的变量分配给寄存器。
更严格遵循 ANIS 规格 (Obey ANSI specifications more strictly)	*strict_ansi*	下列处理符合 ANSI 标准。 - 浮点运算的联合规则

用户定义的选项 (User defined options)：指定命令选项。

4.2.5 CPU 选项

从 H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) 对话框选取 CPU 标签。



CPU：指定 CPU 类型。

CPU	规格
环境变量 (Environment variable)	取决于环境变量 H38CPU
H8SX 最大 4GB (H8SX Maximum 4G byte)	*cpu=h8sxx:32*
H8SX 最大 256MB (H8SX Maximum 256M byte)	*cpu=h8sxx:28*
H8SX 高级 4GB (H8SX Advanced 4G byte)	*cpu=h8sxa:32*
H8SX 高级 256MB (H8SX Advanced 256M byte)	*cpu=h8sxa:28*
H8SX 高级 16MB (H8SX Advanced 16M byte)	*cpu=h8sxa:24*
H8SX 高级 1MB (H8SX Advanced 1M byte)	*cpu=h8sxa:20*
H8SX 中间 16MB (H8SX Middle 16M byte)	*cpu=h8sxm:24*
H8SX 中间 1MB (H8SX Middle 1M byte)	*cpu=h8sxm:20*
H8SX 普通 (H8SX Normal)	*cpu=h8sxn*
H8S/2600 高级 4GB (H8S/2600 Advanced 4G byte)	*cpu=2600A:32*
H8S/2600 高级 256MB (H8S/2600 Advanced 256M byte)	*cpu=2600A:28*
H8S/2600 高级 16MB (H8S/2600 Advanced 16M byte)	*cpu=2600A:24*
H8S/2600 高级 1MB (H8S/2600 Advanced 1M byte)	*cpu=2600A:20*
H8S/2600 普通 (H8S/2600 Normal)	*cpu=2600N*
H8S/2000 高级 4GB (H8S/2000 Advanced 4G byte)	*cpu=2000A:32*
H8S/2000 高级 256MB (H8S/2000 Advanced 256M byte)	*cpu=2000A:28*
H8S/2000 高级 16MB (H8S/2000 Advanced 16M byte)	*cpu=2000A:24*

CPU	规格
H8S/2000 高级 1MB (H8S/2000 Advanced 1M byte)	*cpu=2000A:20*
H8S/2000 普通 (H8S/2000 Normal)	*cpu=2000N*
H8S/300H 高级 16MB (H8S/300H Advanced 16M byte)	*cpu=300HA:24*
H8S/300H 高级 1MB (H8S/300H Advanced 1M byte)	*cpu=300HA:20*
H8/300H 普通 (H8/300H Normal)	*cpu=300HN*
H8/300	*cpu=300*
H8/300L	*cpu=300l*

乘除 (Multiple/Divide)：

对话框菜单	命令选项	功能
无 (None)	*cpu=[...][][]*	没有乘法器与除法器
乘除 (Multiple and Divide)	*cpu=[...][][MD]*	指定乘法器与除法器
乘 (Multiple)	*cpu=[...][][M]*	指定乘法器
除 (Divide)	*cpu=[...][][D]*	指定除法器

堆栈计算 (Stack calculation)：

对话框菜单	命令选项	功能
小 (Small)	*STAck=Small*	以 1 字节计算堆栈地址
中 (Medium)	*STAck=Medium*	以 2 字节计算堆栈地址
大 (Large)	*STAck=Large*	以 4 字节计算堆栈地址

将参数传递寄存器的数量从 2（默认）更改至 3 (Change number of parameter-passing registers from 2 (default) to 3)

复选框	命令选项	功能
√	*regparam=3*	将参数传递寄存器的数量指定为 3
	regparam=2	将参数传递寄存器的数量指定为 2

将复式当作浮动处理 (Treat double as float)

复选框	命令选项	功能
√	*DOuble=Float*	将复式类型的变量/值当作浮动类型处理
	-	-

通过寄存器传递结构参数 (Pass struct parameter via register)

复选框	命令选项	功能
√	*STRUctreg*	将结构参数分配到寄存器
	NOSTRUctreg	不将结构参数分配到寄存器

通过寄存器传递 4 字节参数/返回值 (Pass 4-byte parameter/return value via register)

复选框	命令选项	功能
√	*LONgreg*	将 4 字节参数/返回值分配到寄存器
	NOLONgreg	不将 4 字节参数/返回值分配到寄存器

使用 C++ 的 try、throw 和 catch (Use try, throw and catch of C++)

复选框	命令选项	功能
√	*EXception*	允许异常处理函数
	NOEXception-	禁止异常处理函数

允许/禁止运行时信息 (Enable/disable runtime information)

复选框	命令选项	功能
√	*RTti=ON*	允许 dynamic_cast、typeid
	RTti=OFF	禁止 dynamic_cast、typeid

包结构、联合及类 (Pack struct, union and class)

复选框	命令选项	功能
√	PAck=1	将结构、联合，及类的边界对齐指定到 1
	PAck=2	跟随数据的边界对齐号

指定 SBR 地址 (Specify SBR address)：

对话框菜单	命令选项	功能
默认 (Default)	-	被假设为默认 8 位绝对地址
定制 (Custom)	*sbr=<address>*	指定 8 位绝对区的起始地址

位字段分配顺序 (Bit field allocation order)

对话框菜单	命令选项	功能
左 (Left)	*bit_order=left*	从高位存储成员
右 (Right)	*bit_order=right*	从低位存储成员

4.3 使用 HEW 创建现有的文件

本节将说明如何处理一系列不使用 HIM 而已事先准备的装入模块创建程序，以注册为 HEW 工程。

在 HEW1.2 中，样品程序在 HEW 目录 \Tools\HITACHI\H8\3_0a_0\sample 中提供。

编号	HEW1.2 文件	描述
1	init.c	初始化例程
2	vectbl.c	向量表设定
3	scttbl.c	段初始化例程
4	cmain.c	主要函数文件
5	c2600a.sub	模块间优化器的子命令文件

HEW 2.0 或以上的版本未提供样品程序。因此必须准备用户自制的样品程序或将创建样品程序时所要生成的下列文件用作样品程序。

根据 2.1.2 节，创建新的工作空间 2（HEW2.0 或以上版本），选取**演示 (Demonstration)** 为工程类型设定，以创建样品工程。

编号	HEW 2.0 或以上版本文件	描述
1	resetprg.c	初始化例程
2	intprg.c	向量表设定
3	dbsct.c	段初始化例程
4	main.c	主要函数文件
5	2600a.sub（用户自制）	子命令文件

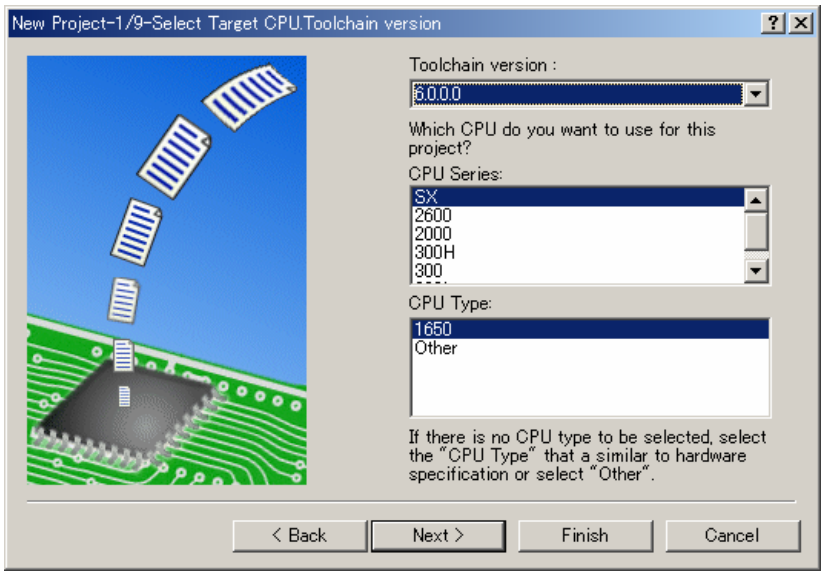
(1) 创建新的工程

根据 2.1.1 节，创建新的工作空间，来创建新的工程。

选取空的应用程序 (Empty Application) 为工程类型。

(2) 选择 CPU

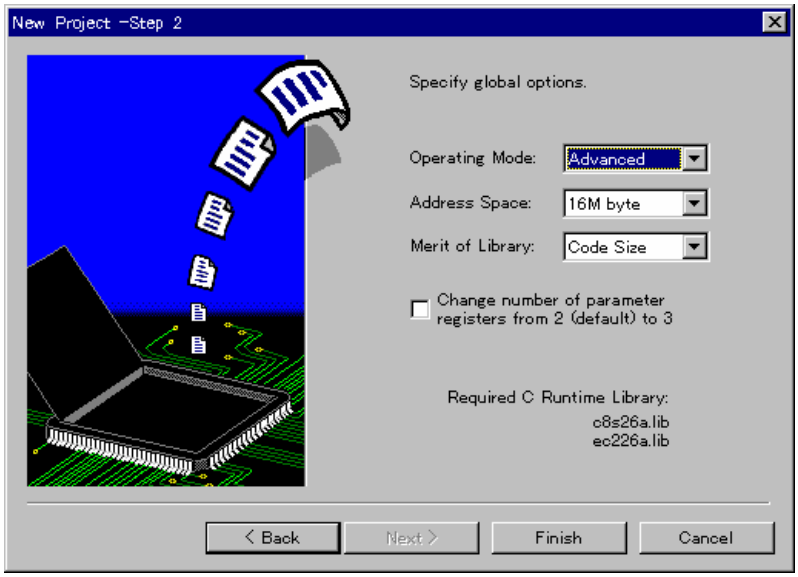
在 1/9 画面上选择 CPU 类型。



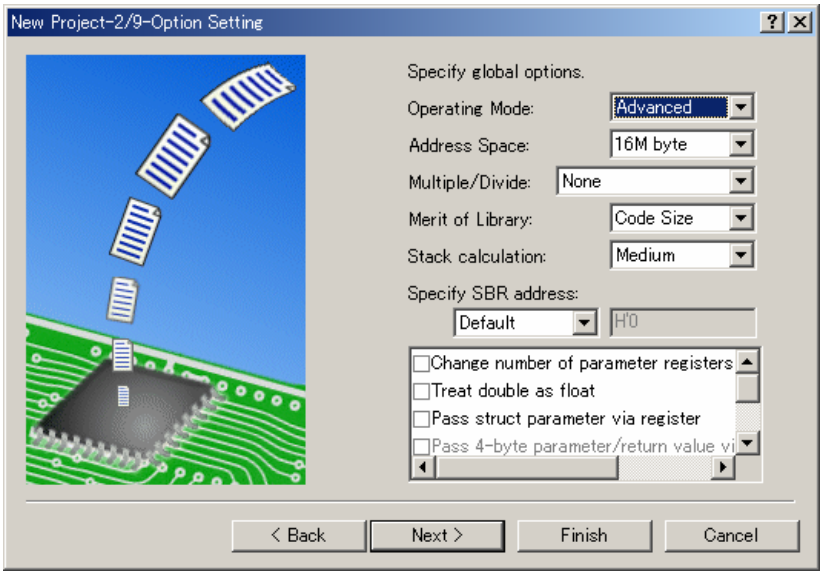
(3) 选择全局选项

在 2/9 画面上选择全局选项。

<HEW1.2>



<HEW 2.0 或以上版本>

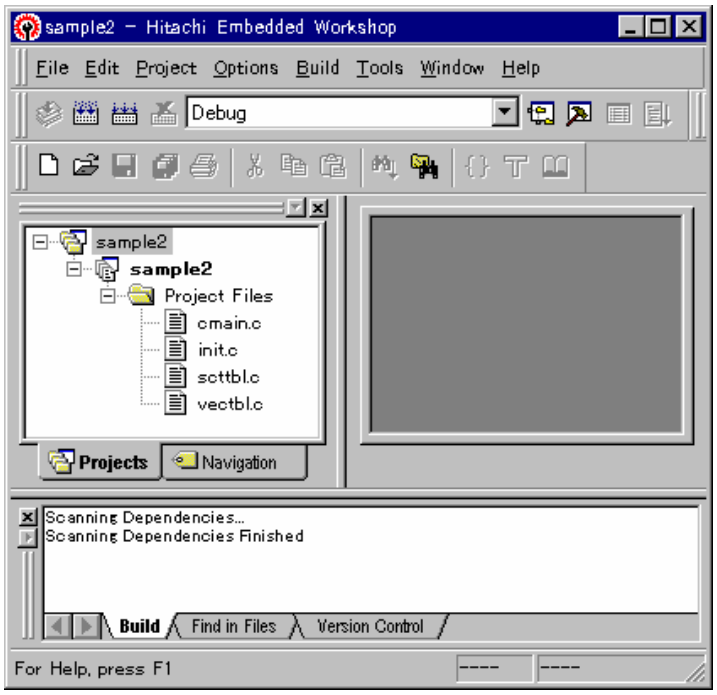


要获取在初始化后更改全局选项的详细资料，请参考 11.2.1 节，“未定义外部符号”的输出。

(4) 将文件添加到工程

在下一个步骤中，使用 [工程 (Project) --> 添加文件 (Add Files) …] 以指定要添加到工程的 C 源文件。

为 HEW1.2 添加文件 init.c、vectbl.c、scttbl.c 和 cmain.c 及为 HEW2.0 添加 resetprg.c、intprg.c、dbsct.c 和 main.c。

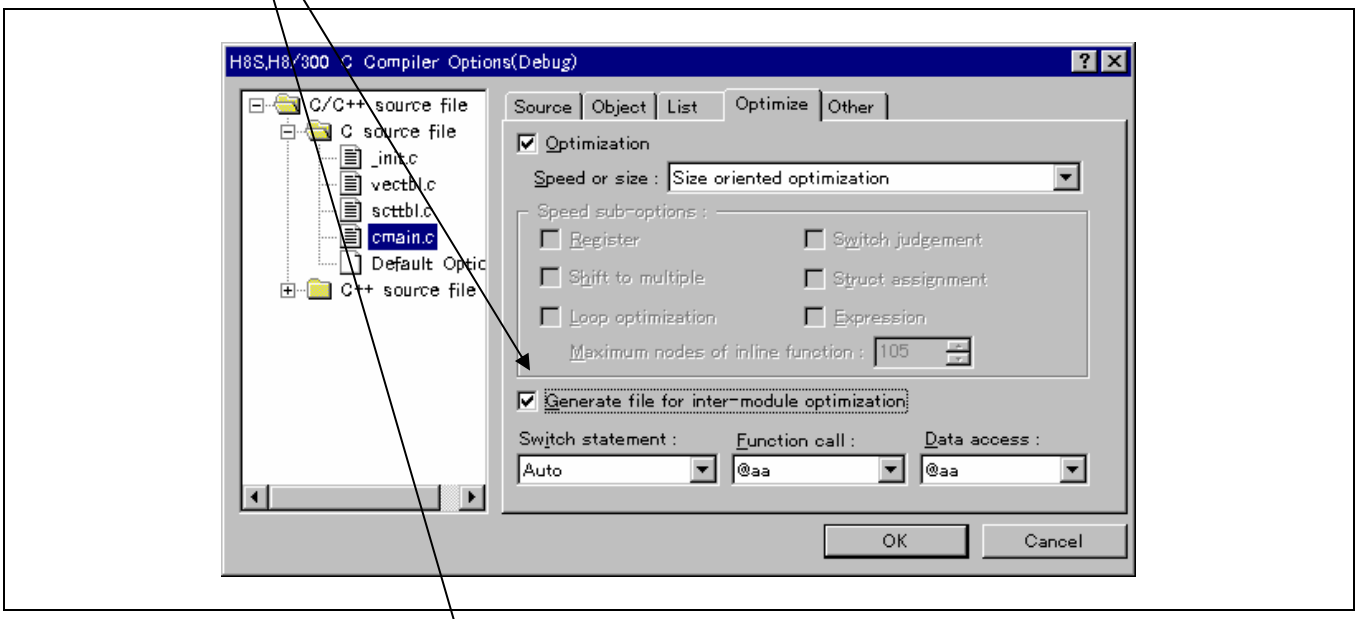


(5) 选择编译程序选项

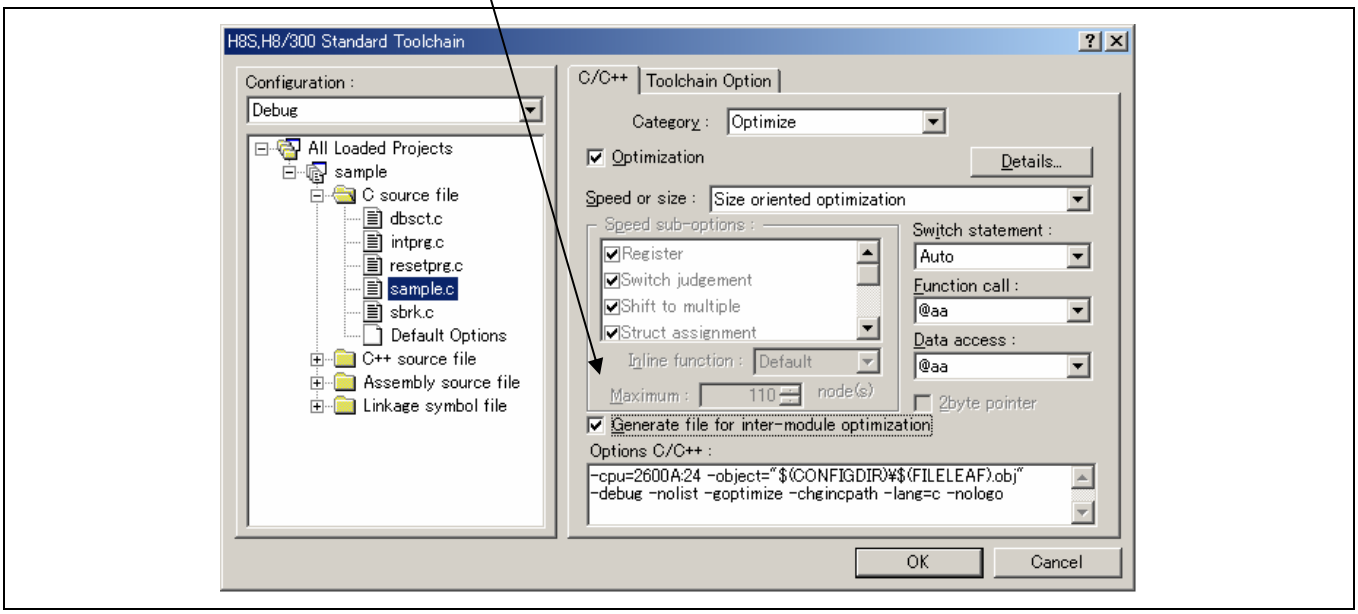
在 HEW1.2 中使用 [选项 (Options) -> H8S，H8/300 C/C++ 编译程序 (H8S, H8/300 C/C++ Compiler…)] 及在 HEW2.0 或以上版本中使用 [C/C++ 标签] [类别 (Category)/优化 (Optimize)] 以指定编译程序选项。

在这个步骤中，在这里指定模块间优化加载信息工具的输出。

<HEW1.2>



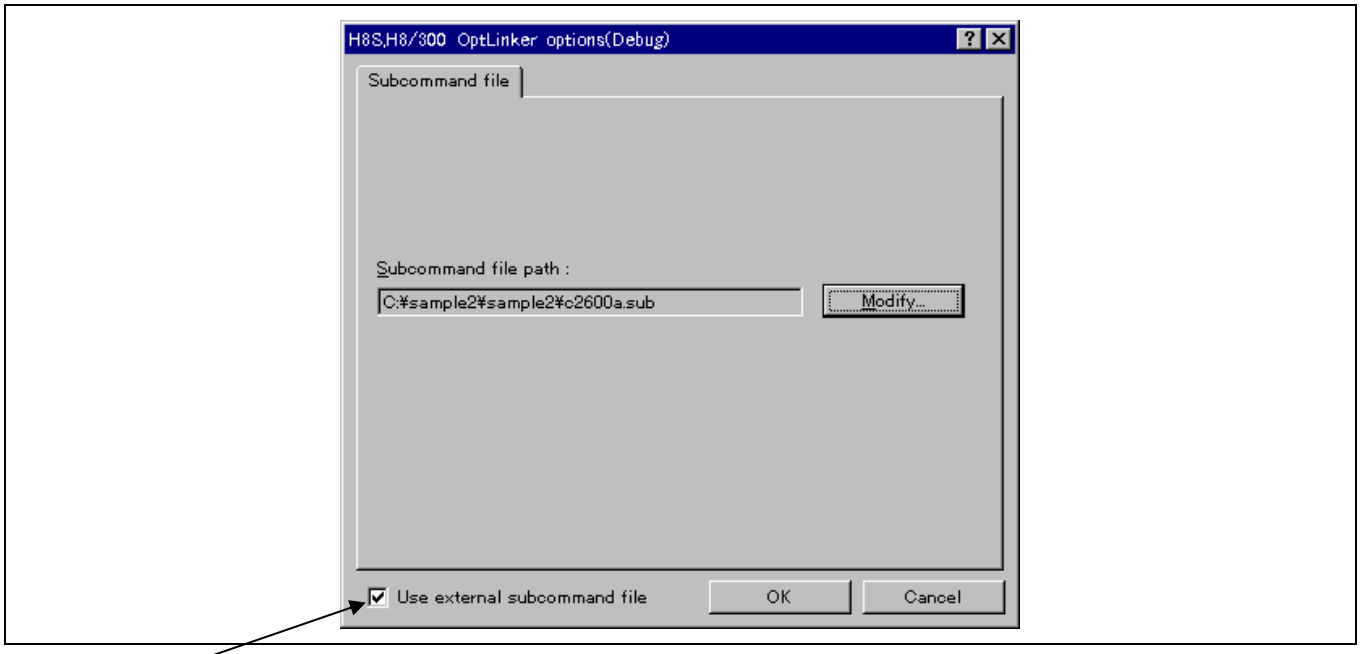
<HEW 2.0 或以上版本>



(6) 为模块间优化器指定子命令文件

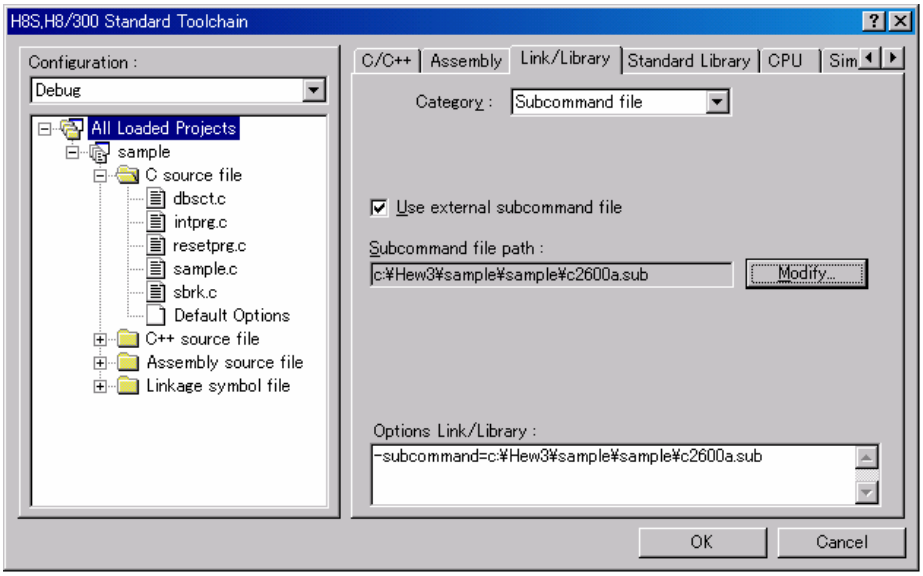
在 HEW1.2 中使用 [选项 (Options) -> H8S，H8/300 IM Optlinker…] 及在 HEW2.0 或以上版本中使用 [连接/程序库 (Link/Library) 标签] [类别 (Category)/子命令文件 (Subcommand file)] 以为模块间优化器调用指定子命令文件的 HEW 选项对话框。

<HEW1.2>



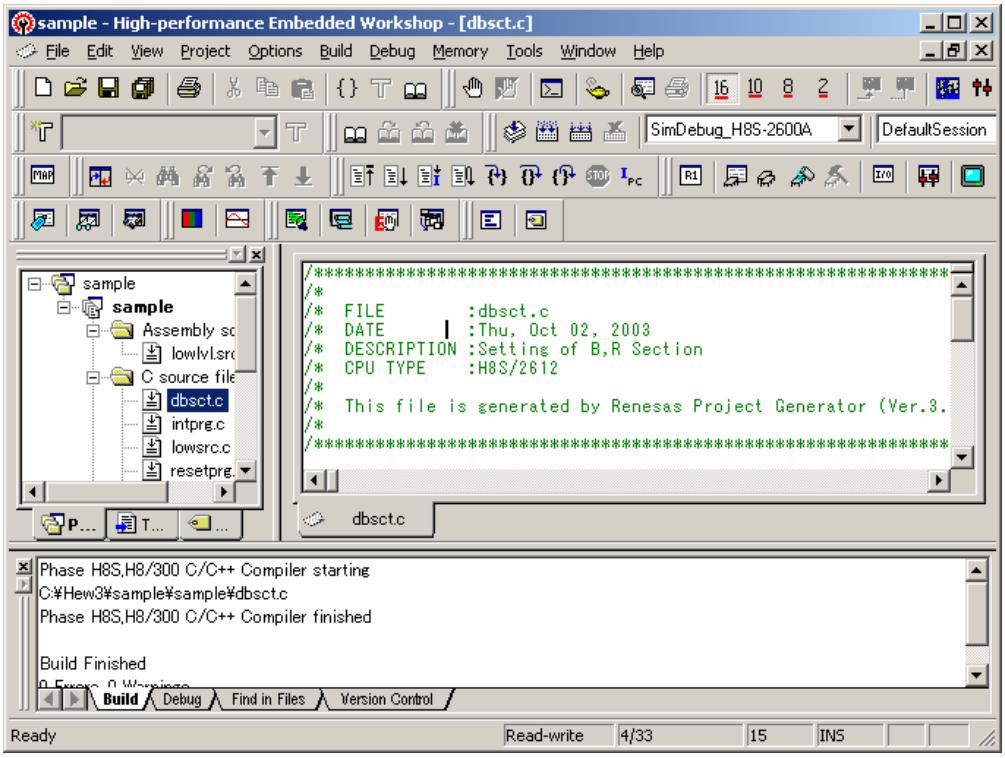
选中这里以显示子命令文件 (Subcommand file) 标签。

<HEW 2.0 或以上版本>



(7) 执行创建过程

执行创建过程将生成装入模块。



注意： 当用户使用本产品提供为样品的模块间优化器子命令文件时，可能会发生错误。

这是因为标准程序库未被指定，及 CPU 信息检查文件未在适当的目录中被定义。

要避免这项错误，请复制子命令文件和 CPU 信息检查文件并指定标准程序库。

H8S, H8/300 系列 C/C++编译程序应用笔记

使用优化功能

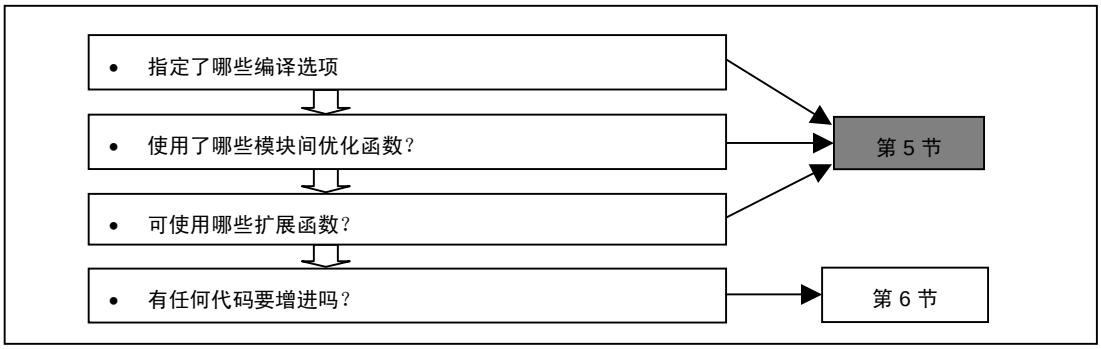
第 5 节　使用优化功能

在准备好操作的装入模块后，就有必要增强目标程序的性能，以使装入模块更有效率且效果卓著。

共有四种方法可增强目标程序的性能：

(i)　使用各种选项执行优化。
(ii)　使用模块间优化器来执行优化。
(iii)　使用扩展函数来执行优化。
(iv)　通过修改代码来执行有效率的编程。

使用下列步骤：



本节说明在创建装入模块时所要指定的选项、所要使用的扩展函数，及所要使用的模块间优化器选项。

下表列出编译程序所支持的优化函数：

编号	优化函数	规格模式	大小	速度
1	使用 1 字节枚举类型	选项	O	O
2	乘法/除法规格的扩展解释	选项	O	O
3	指定参数传递寄存器的数量	选项	Δ	Δ
4	增加变量分配寄存器的数量	选项	Δ	Δ
5	外部变量的优化	选项	-	-
6	块转移指令	选项	X	O
7	速度 (SPEED) 选项	选项	X	O
8	将全局变量分配到寄存器	扩展函数	Δ	Δ
9	控制寄存器保存/恢复代码在函数入口及出口点的输出	扩展函数	O	O
10	指定函数的内联扩展	扩展函数	X	O
11	汇编语言函数的内联扩展	扩展函数	X	O
12	使用 8 位绝对地址区	选项/扩展函数	O	O
13	使用 16 位绝对地址区	选项/扩展函数	O	O
14	分配到存储器间接区	选项/扩展函数	O	X

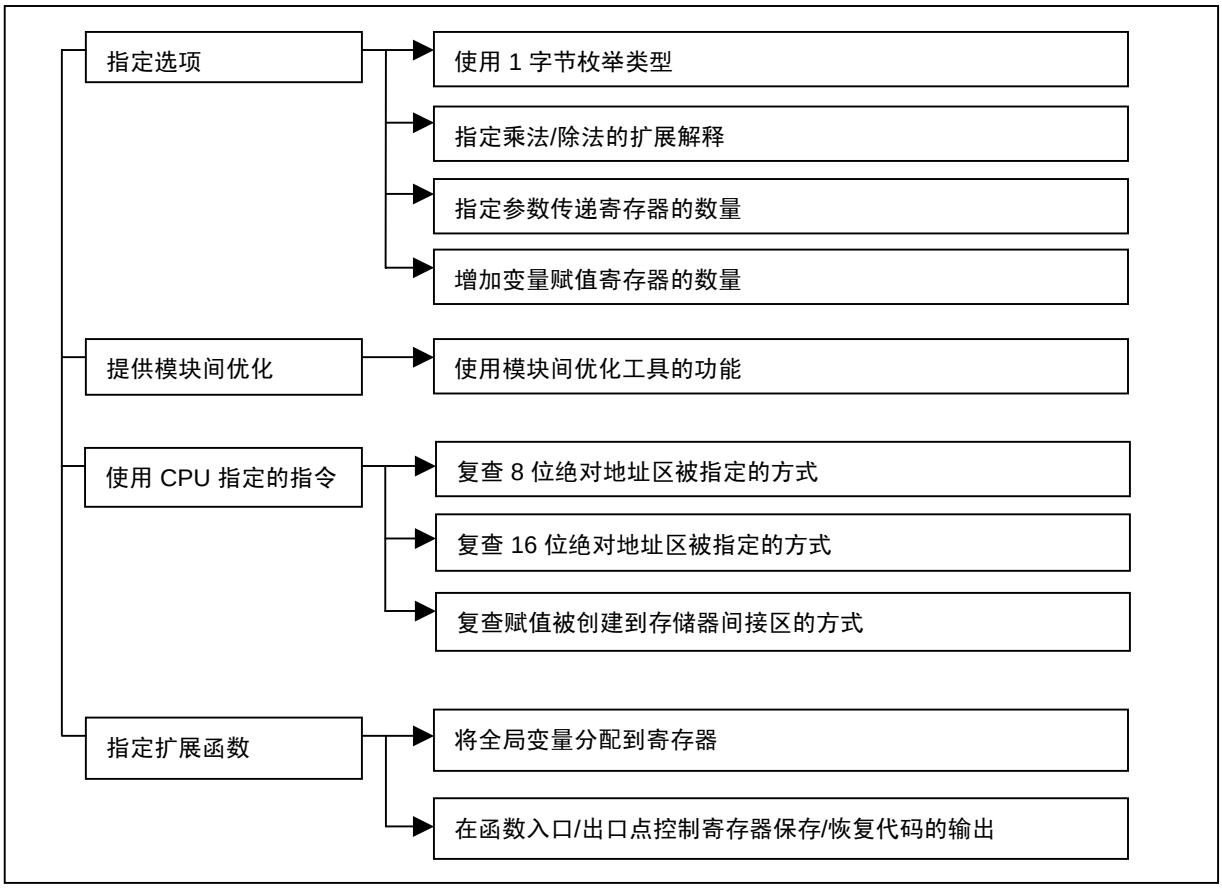
图例：
O: 有效率的；Δ:对一些程序有效率；X: 减低效率

下表列出模块间优化工具所支持的优化函数：

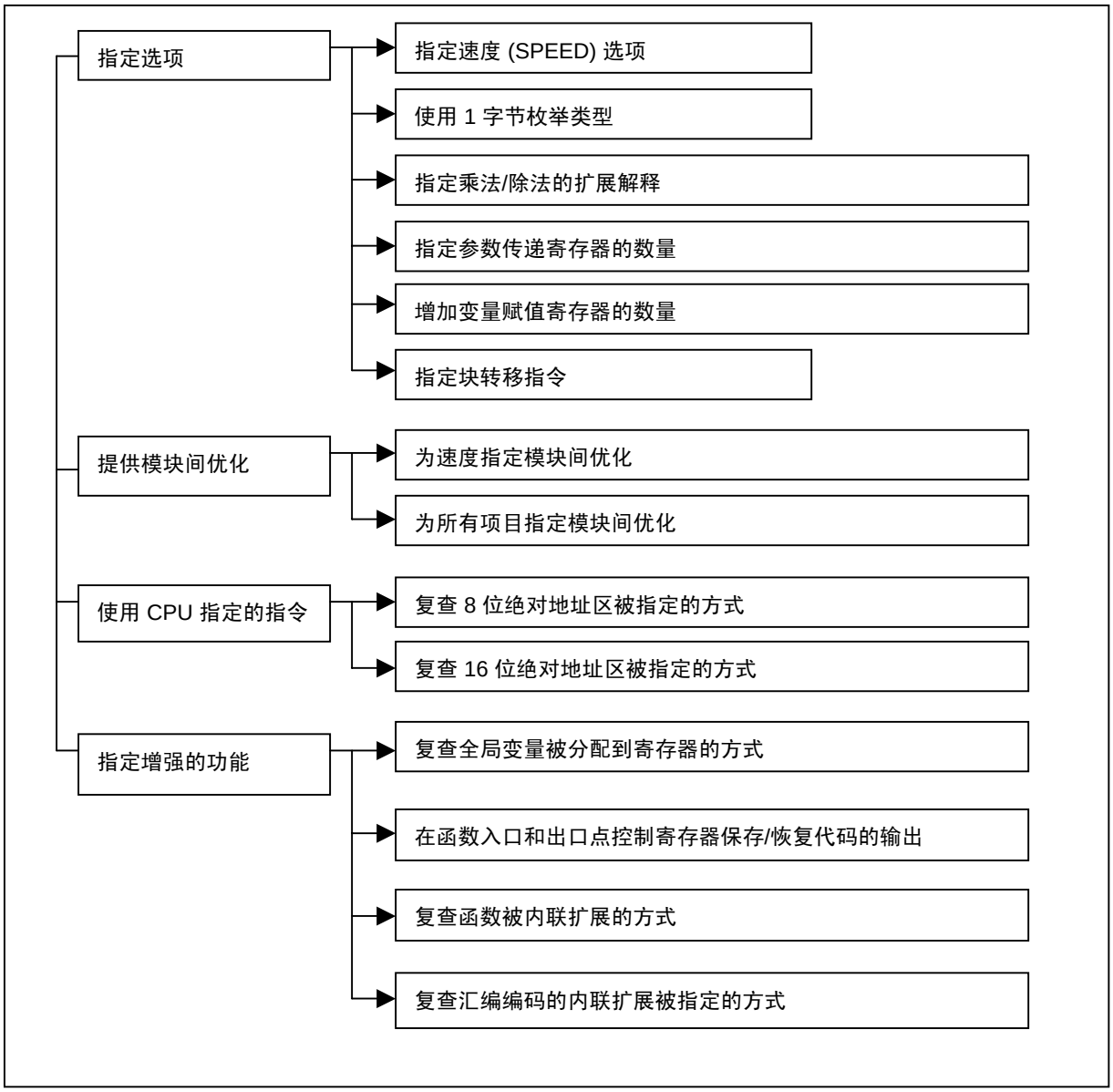
编号	描述	规格模式
1	统一常数和字符串	选项
2	删除未参考的变量和函数	选项
3	优化对变量的存取	选项
4	优化对函数的存取	选项
5	再分配寄存器	选项
6	删除相同的代码	选项
7	优化转移指令	选项

本节将分开两组描述这些优化函数，即对大小的优化及对速度的优化。每项函数的指定根据下列流程表进行检查：

<在优化大小时的指定步骤>



<在优化大小时的指定步骤>



5.1 对大小的优化

在本节中，一个普遍的基准程序，Dhrystone 版本 2.1 将被用作样品程序。

下列所提供的大小和速度数据，反映了 H8S/2600 高级模式中的编译结果。

5.1.1 默认编译

首先，不使用任何优化选项来编译程序。

下表显示目标大小和执行周期计数的结果：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580

即使未指定优化选项，编译程序仍然执行基本的优化任务，因为有些优化选项是被默认允许的。

5.1.2 无优化规格

当未指定优化时，其结果如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
无优化	3582	1713

[指定方法]

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，取消选中**优化 (Optimization)** 复选框。

命令行：*optimize=0*

5.1.3 优化调谐

(1) 指定 1 字节枚举类型

此选项仅对包含枚举类型数据的程序有效，然而，我们建议您始终予以指定。

目标大小和执行周期计数如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
1 字节枚举类型规格	3050	1580

[指定方法]

对话框菜单： **C/C++** 标签**类别：[其他 (Other)]**, 对**杂项 (Miscellaneous options)** 选择**将枚举当作字符处理，若它处于字符的范围内 (Treat enum as char if it is in the range of char)。**

命令行： *byteenum*

要进一步获取详细资料，请参考 5.4.1 节，使用 1 字节枚举类型。

(2) 指定参数传递寄存器的数量

将参数传递寄存器的数量从 2 增加至 3，所形成的性能特征如下：

优化函数	大小 (ROM)	执行周期数
2 个参数传递寄存器	3048	1580
3 个参数传递寄存器	3034	1528

[指定方法]

对话框菜单： 在 **CPU** 标签上，选取**将参数传递寄存器的数量从 2（默认）更改至 3 (Change number of parameter-passing registers from 2(default) to 3)**。

命令行： *regparam=3*

这些性能特征和程序内函数参数的数量有关。通过决定分配给寄存器的参数数量、可用的寄存器数量，及可用的寄存器类型，在 2 和 3 的选项之间选择。

若无法检查所有参数，则可尝试不同的选项，然后选取其中产生最小的目标大小的选项。

当和 1 字节枚举类型的规格结合时，此项目形成下列的性能特征：

优化函数	大小 (ROM)	执行周期数
默认 (Default)	3048	1580
1 字节枚举类型 +3 个参数传递寄存器	3034	1527

要进一步获取详细资料，请参考 5.4.3 节，指定参数传递寄存器的数量。

(3) 扩展变量分配寄存器的数量

默认情形下，编译程序使用寄存器 [E]R3 至 [E]R6 为变量分配寄存器。

当此选项被禁止时，编译程序使用寄存器 [E]R4 至 [E]R6 为变量分配寄存器。

下面列出这两种规格的性能特征：

优化函数	大小 (ROM)	执行周期数
寄存器变量 [E]R3 至 [E]R6	3048	1580
寄存器变量 [E]R4 至 [E]R6	3048	1580

[指定方法]

对话框菜单： **C/C++** 标签**类别： [其他 (Other)]**, 对**杂项 (Miscellaneous option)** 选择**为寄存器变量增加寄存器 (Increase a register for register variable)**。

命令行： *regexpansion*

在此程序中，这并无区别。然而，除非表达式语句太过复杂，否则变量分配寄存器的数量越大，编译程序在目标大小上的性能越高。
在 H8S V6.01 中，此选项不被支持，因此在性能上没有区别。

要进一步获取详细资料，请参考 5.4.4 节，增加变量分配寄存器的数量。

(4) 外部变量的优化

下表比较当外部变量的优化被指定或被禁止时的结果：

优化函数	大小 (ROM)	执行周期数
启用了外部变量优化 (novolatile)	3048	1580
禁用了外部变量优化 (volatile)	3076	1592

[指定方法]

对话框菜单： **C/C++** 标签**类别： [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： *volatile*

注意有些外部变量不应被优化：

(实例 1)

```
int a;
void f()
{
    a=1;
    a=2;
}
```

←删除了第一个替换

(实例 2)

```
volatile int a;
void f()
{
    a=1;
    a=2;
}
```

在实例 1 当中，对变量 *a* 连续做出两次替换，使第一个替换语句因为优化的结果被删除。然而，若两个替换语句中间发生了中断且 *a* 的值被参考了，将使结果出现错误。

当 *volatile* 被指定时，优化被禁止了且第一个替换语句的代码被生成，使这个问题可被避免。然而，这个方法禁止了所有外部变量的优化，使目标性能明显降低。

要仅仅对适当的外部变量禁止优化，就必须在源程序中对中断功能内所使用的变量和 I/O 寄存器指定 *volatile* 声明，如实例 2 所示。如此即可关闭此选项以编译程序。

要进一步获取详细资料，请参考 5.4.5 节，外部变量的优化。

(5) 乘法/除法规格的扩展解释

对乘法/除法代码从 ANSI 标准扩展的扩展解释将形成下列性能特征：

优化函数	大小 (ROM)	执行周期数
服从 ANSI 的	3048	1580
扩展的解释	3048	1580

[指定方法]

对话框菜单： **C/C++** 标签**类别：[目标 (Object)]**，对**乘除运算规格 (Mul/Div operation specification)** 选择**非 ANSI（保证 32 位为 16 位*16 位的结果）(Non ANSI (Guarantee 32bit as a result of 16bit*16bit))**。

命令行： *cpuexpand*

在本程序中，扩展的解释未造成任何显著的性能差异。

但是，乘法/除法代码的扩展解释可产生不一样的计算结果，因为该扩展解释的范围和语言规格中所保证的不同。因此，扩展的解释应该只在被认为适当时使用。

要进一步获取详细资料，请参考 5.4.2 节，乘法/除法规格的扩展解释。

5.1.4 使用模块间优化功能

通过使用模块间优化器，将可更有效的创建有大小效率的目标。

在使用模块间优化器指定优化之前，先在编译程序或交叉汇编程序内指定模块间优化加载信息文件的输出。

[指定方法]

C/C++ 编译程序

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**为模块间优化生成文件 (Generate file for inter-module optimization)**。

命令行： *goptimize*

交叉汇编程序

对话框菜单： **汇编 (Assembly)** 标签**类别：[目标 (Object)]**，选择**为模块间优化生成文件 (Generate file for inter-module optimization)。**

命令行： *goptimize*

在 HEW1.2 中，模块间优化加载信息文件也被提供给进行模块间优化时所连接的标准程序库。由于此文件在 Windows 版本中是以压缩形式提供，因此在使用前必须先解压缩。

通过双击和所要使用的程序库同名的压缩文件 (*.exe)，文件将自行解压并创建一个包含了信息文件的目录。

要获取此程序库之模块间优化的详细资料，请参考 H8S，H8/300 系列 C/C++ 编译程序的补遗。

在 HEW2.0 或以上版本中，标准程序库生成程序的模块间优化功能应被使用来创建程序库。通过选中**标准程序库 (Standard Library)** 标签**类别：[优化 (Optimize)] 为模块间优化生成文件 (Generate file for inter-module optimization)**，模块间优化加载信息文件将被输出。

(1) 默认优化

模块间优化器支持下列优化函数：

编号	描述	对话框菜单	子命令选项
1	统一常数/字符串	统一字符串 (Unify strings)	String_Unify
2	删除未参考的变量/函数	删除相同的代码 (Eliminate same code)	Symbol_delete
3	优化对变量的存取	使用短寻址 (Use short addressing)	Variable_access
4	优化对函数的存取	使用间接调用/跳转 (Use indirect call/jump)	Funcation_call
5	再分配寄存器	再分配寄存器 (Reallocate registers)	Register
6	删除死码	删除死码 (Eliminate dead code)	Same_code
7	优化转移指令	优化转移 (Optimize branches)	Branch

下面显示用于编译程序的最具效率优化规格：

优化函数	大小 (ROM)	执行周期数
有效的编译程序优化选项： 1 字节枚举类型规格 +3 个参数传递寄存器	3034	1527

在模块间优化器中，若使用了 HEW 则默认为不执行优化，且将产生一个简单连接的模块。因此，默认设定产生和编译程序优化相同的结果。

(2) 指定模块间优化项目

(a) 为模块间优化器逐一指定优化项目：

优化函数	模块间优化	大小 (ROM)	执行周期数
指定了编译程序优化选项	-	3034	1527
	统一常数/字符串	3034	1527
	删除未参考的变量/函数	3034	1527
	优化对变量的存取	2970	1513
	优化对函数的存取	3024	1538
	再分配寄存器	3018	1535
	删除死码	3034	1527
	优化转移指令	3034	1527

(b) 允许所有模块间优化功能

允许所有模块间优化功能。

优化函数	模块间优化	大小 (ROM)	执行周期数
指定了编译程序优化选项	-	3034	1527
	全部优化	2946	1517

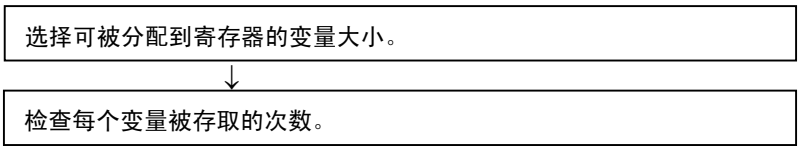
当指定了此函数时，优化将对即使不应被优化的项目执行。这时，指定 _list 选项（在 HEW1.2 为 _mlist 选项）将输出所有已被优化程序删除或再定位的符号。以 symbol_forbid_xxxx 的格式指定优化应被禁止的符号。这项指定必须经过谨慎考虑符号后才做出。

5.1.5 选择扩展函数

(1) 将寄存器分配到全局变量

通过使用 #pragma global_register 以将外部变量分配到固定的寄存器，可缩减用以存取变量的程序大小。

要分配到寄存器的变量如下选取：



这可通过在连接编辑程序（在 HEW1.2 为模块间优化器）中指定优化信息列表的输出来选中。

[指定方法]

对话框菜单： **连接/程序库 (Link/Library)** 标签**类别：[列表 (List)] 生成列表文件 (Generate list file)**
连接/程序库 (Link/Library) 标签**类别：[列表 (List)] 内容 (Contents)：显示参考 (Show reference)**

子命令： *list*
show reference

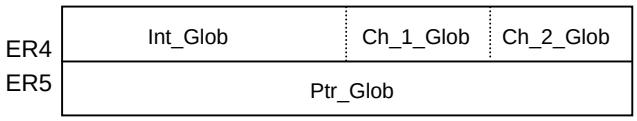
结果将创建下列文件：

```
*** Variable Accessible with Abs8 ***
SYMBOL                                  SIZE    COUNTS  OPTIMIZE
_Ch_1_Glob
                                           1         4
_Ch_2_Glob
                                           1         2
*** Variable Accessible with Abs16 ***
SYMBOL                                  SIZE    COUNTS  OPTIMIZE
_Ptr_Glob
                                           4         4
_Next_Ptr_Glob
                                           4         2
_Int_Glob
                                           2         6
_Bool_Glob
                                           2         2
_Arr_2_Glob
                                        1388         1
_flmod
                                           3         2
_brk
                                           4         2
```

ER4 和 ER5 作为全局寄存器，共有 8 字节的数据可被分配。

下列是对最常被存取的变量 Int_Glob 和 Ptr_Glob 的寄存器分配的说明。即使在这些变量已被分配时，寄存器可容纳 2 个附加字节。

将变量 Ch_1_Glob 和 Ch_2_Glob 分配到剩余的 2 字节：



指定如下：

```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L,Ptr_Glob=ER5)
```

下面将结果与默认设定相比：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
#pragma global_register	2940	1512

在上例中，目标大小被缩减了，同时执行速度也增强了。然而，在一些情况下，外部变量对寄存器的分配可能造成工作寄存器的短缺，且其他变量可能被分配到存储器，因而使目标性能降级。

因此，在使用此函数时必须小心谨慎。

下表显示有大小效率的编译程序选项（指定了 1 字节枚举类型与 3 个参数传递寄存器）与变量到寄存器 (variable-to-register) 赋值选项的结合成果。

优化函数	大小 (ROM)	执行周期数
默认 (Default)	3048	1580
#pragma global_register	3010	1487

要进一步获取详细资料，请参考 5.4.8 节，将寄存器分配到全局变量。

注意当程序库在模块间优化程序中被指定为模块间优化的目标时，全局寄存器将无法被指定。因此，这项指定未在本节中做出。

(2) 在函数入口和出口点控制寄存器保存/恢复代码的输出

#pragma regsave 语句声明可保存/恢复所有寄存器的函数。它也生成一个不在函数调用以外分配被保证的寄存器（[E]R2 到 [E]R6）的目标。

#pragma noregsave 语句声明不保存/恢复任何寄存器的函数。此语句也被用作无须经由其他函数调用而首先启动的函数；它也被用作由指定 #pragma regsave 的函数所调用的函数。

要使用这项功能，必须创建一个函数调用关系图示。

在 HEW2.0 或以上版本中，可通过在创建关系图示中输出可用的堆栈空间信息文件，及将信息文件读取入模拟程序调试程序，来检验函数之间的调用关系。

要获取如何输出可用的堆栈空间信息文件的说明，请选择 [选项 (Options) -> H8S，H8/300 标准工具链 (H8S, H8/300 Standard Toolchain) … -> 连接/程序库 (Link/Library) 标签] 类别：[其他 (Other)] 堆栈信息输出 (Stack information output)。

在 Dhrystone 版本 2.1 的情况下，下列关系有效：

```
main:
     malloc:
     strcpy:
     Proc_1:
          Proc_3:
               Proc_7:
          Proc_6:
               Func_3:
          Proc_7:
     Func_2:
          Func_1:
          strcmp:
     Func_1:
     Proc_8:
     Proc_7:
     Proc_6:
          Func_3:
     Proc_5:
     Proc_4:
     Func_2:
```

由 #pragma noregsave 所声明的函数：

因为 main() 是执行首次处理的函数，所以无需保存/恢复任何在该函数之前所使用的寄存器。因此，此函数在 #pragma noregsave 语句中被声明。

若 main() 仅包含函数调用，所有从 main() 调用的函数可在 #pragma noregsave 语句中声明。

<inc.h>

```
#pragma noregsave (main)
```

<dhrystone21.c>

```
#include "inc.h"
             :
```

执行成果如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
#pragma noregsave specified	3030	1580

#pragma regsave/noregsave 语句中所指定的函数：

检查若有 *main* 函数以外的其他函数，仅执行函数调用。

假定中断函数 *intr1* 仅执行与下列调用关系有关的函数调用：

```
intr1:
    proc1:
        func1:
    proc2:
    proc3:
```

假定 proc1、proc2 和 proc3 不被任何其他函数所调用。

在 #pragma regsave 语句中声明 intr1()。

同样的，在 #pragma noregsave 语句中声明 proc1()、proc2()，及 proc3()。

如此一来，三个函数的寄存器保存/恢复，可被一个函数的寄存器保存/恢复所替代。

下表显示当使用模块间优化和之前 Dhrystone 版本 2.1 程序中的 #pragma noregsave 语句，所指定的有效选项组合（指定了 1 字节枚举类型与 3 个参数传递寄存器）的执行成果：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
指定了 #pragma noregsave 编译程序选项模块间优化	2944	1517

要进一步获取详细资料，请参考 5.4.9 节，在功能的入口/出口点控制寄存器保存/恢复代码的输出。

5.1.6 使用 CPU 指定的指令

(1) 分配到短 8 位绝对区

下面显示字符/无符号字符类型的数据以 8 位绝对地址存取时的结果。

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
指定了 8 位绝对地址	3014	1566

注意： 在此情形下，当选项被指定为短 8 位绝对地址区赋值，并超出了 8 位绝对地址区时，模拟程序将因为区域不足而无法正确操作，且执行周期数无法被测量。

[指定方法]

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**数据存取 (Data access) @aa:8**

命令行： *abs8*

短 8 位绝对地址区必须在 H’FFFF00 至 H’FFFFFF 的存储范围内被分配。若超出了此范围，所有 $ABS8 中的段将无法被分配到短 8 位绝对地址区。

因此，不要将 abs8 选项指定到整个文件，但选取要分配到短 8 位绝对地址区的变量。

条件是能适合于区域并且接收频繁存取的变量。

变量的大小可使用由编译程序输出的目标列表来检查，而存取的次数则可使用由模块间优化器所产生的优化信息列表来检查。

以模块间优化器的优化选项来指定短绝对地址区的使用，并检查所形成的优化信息列表：

```
*** Variable Accessible with Abs8 ***
SYMBOL                                  SIZE    COUNTS  OPTIMIZE
_Ch_1_Glob
                                           1         4
_Ch_2_Glob
                                           1         2
```

此文件表示变量被参考的次数。

基于此信息，在 #pragma abs8 语句中做出适当的指定：

```
#pragma abs8 (Ch_1_Glob,Ch_2_Glob)
```

执行成果如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
指定了 #pragma abs8	3014	1566

若有许多可被分配到 8 位绝对地址区的其他变量，则检查存取的次数并分配接收到最大存取数的变量。

另外，指定一些选项以缩减目标大小。

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
#pragma abs8(Char1Glob,Char2Glob) +指定了 #pragma noregsave +编译程序选项 +模块间优化	2944	1517

很明显的上述指定可产生较佳的成果。

要进一步获取有关 abs8 的详细资料，请参考 5.4.11 节，使用 8 位绝对地址区。

(2) 分配到短 16 位绝对地址区

生成代码以使用 16 位绝对地址执行存取。

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
abs 16 选项	2988	1558

[指定方法]

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**数据存取 (Data access) @aa:16**

命令行： *abs16*

16 位绝对地址区必须在 H’000000 到 H’007FFF 和 H’FF8000 到 H’FFFFFF 的存储范围内被分配。

在初始化阶段将选项指定为 abs16，从而揭露可被分配到 ABS16 段的变量。若变量能适合于范围，则它们可在选项中直接指定。然而，因为 16 位绝对地址和许多其他区域重叠，若任何变量超出了范围，则在程序的主体中指定 #pragma abs16 语句。

使用模块间优化器所生成的优化信息列表来为符号检查存取计数，并将具有大存取数的变量分配到 ABS16 段。

当短绝对地址模式的使用在模块间优化器的优化选项中被指定时，存取计数可通过以下方法来检查：

```
*** Variable Accessible with Abs16 ***

SYMBOL                                  SIZE    COUNTS  OPTIMIZE
_Ptr_Glob
                                           4         4
_Next_Ptr_Glob
                                           4         2
_Int_Glob
                                           2         6
_Bool_Glob
                                           2         2
_Arr_2_Glob
                                        1388         1
_flmod
                                           3         2
_brk
                                           4         2
```

基于以上结果，在 #pragma abs16 语句中指定要分配到 16 位绝对地址区的变量：

```
#pragma abs16 (Int_Glob,Bool_Glob,Arr_2_Glob,Ptr_Glb,Next_Ptr_Glob)
```

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
abs 16 选项	2998	1558
指定了 #pragma	3012	1575

当添加了上述的 #pragma abs8 指定时，执行的结果如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
abs 16 选项	2988	1558
指定了 #pragma abs16	3012	1575
#pragma abs16+ 指定了 #pragma abs8	2980	1561

接下来，检查变量 Int_Glob 和 Ptr_Glob 应被分配到 16 位绝对地址区或全局寄存器。当结合了证实有效的选项时，将提供下列结果：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
#pragma abs16 (Bool_Glob, Arr_2_Glob, Next_Ptr_Glob Ptr_Glob,Int_Glob) +#pragma abs8(Ch_1_Glob,Ch_2_Glob) +指定了 #pragma noregsave +编译程序选项	2920	1484
#pragma global_register(Int_Glob=E4,Ptr_Glob=ER5) +#pragma abs16 (Bool_Glob, Arr_2_Glob, Next_Ptr_Glob) +#pragma abs8(Ch_1_Glob,Ch_2_Glob) +指定了 #pragma noregsave +编译程序选项	2958	1486

根据结果表示，将变量 Int_Glob 和 Ptr_Glob 分配到全局寄存器比分配到 16 位绝对地址区更有效。

要进一步获取有关 abs16 指定的详细资料，请参考 5.4.12 节，使用 16 位绝对地址区。

变量可根据 CPU 功能，由模块间优化器分配给 8 位或 16 位绝对地址区。

(3) 分配到存储间接区

函数调用以此规格在存储间接格式中被执行。

要参考输出目标，同时指定列表输出。

下表显示当存储间接区赋值选项被默认（有大小效率）指定时的结果：

优化函数	大小 (ROM)	执行周期数
默认 (Default)	3048	1580
存储间接区赋值指定	2994	1599

[指定方法]

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**函数调用 (Function call)：@@aa:8**

命令行： *indirect*

运行时例程也可被分配到此区。

当指定了 #include <indirect.h> 语句时，运行时例程调用将作为存储器间接调用执行。

若在使用编译程序指定目标列表的输出时指定堆栈帧信息的输出，则将显示函数中所调用的运行时例程。

```
Function (File hv21_dhry_, Line   309): Proc_1

  Optimize Option Specified : No Allocation Information Available

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000004 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x0000000c Byte(s)
Total Frame Size        : 0x00000010 Byte(s)

Used Runtime Library Name
$MVN$3
```

结果，调用 $MVN$3 遂成为存储器间接调用。

要个别指定函数，则指定 #pragma indirect $MVN$3。

由于存储间接区在从 0x00000000 到 0x000000ff 的范围内被分配，所有函数可在此区内分配。

这时，请注意此区与异常处理向量区重叠。

将有必要区隔段，以避免赋值上的重叠。

在这种情形下，函数能适合于区域，然而，若 $INDIRECT 段超出了存储间接区，这些接收到频繁存取的函数应使用 #pragma indirect 语句个别分配。另外，使用 #pragma indirect section 语句以在赋值上区隔段。

此选项指定和下列相同：

```
#pragma
indirect(main,malloc,Proc1,Proc2,Proc3,Proc4,Proc5,Proc6,Proc7,Proc8,Func1,
Func2,Func3)
#pragma indirect $MVN$3
```

当结合了证实有效的选项时，将提供下列结果：

优化函数	大小 (ROM)	执行周期数
无（默认）	3048	1580
#pragma abs8(Ch_1_Glob,Ch_2_Glob) +#pragma abs16 (Bool_Glob, Arr_2_Glob, Ptr_Glob, Next_Ptr_Glob) +指定了 #pragma noregsave +编译程序选项 +模块间优化函数 +指定了 #pragma indirect	2902	1496

要进一步获取详细资料，请参考 5.4.13 节，使用间接存储格式。

函数调用可根据 CPU 功能，以模块间优化功能在存储间接格式中执行，即使此选项未被指定。

5.2 速度的优化

5.2.1 指定速度 (SPEED) 选项

要为速度提供优化，则指定速度 (SPEED) 选项。

执行成果如下：

优化函数	大小 (ROM)	执行周期数
默认 (Default)	3048	1580
速度 (SPEED) 选项	3420	1325

[指定方法]

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**速度或大小(Speed or size) 面向速度的优化 (Speed oriented optimization)。**

命令行： *speed*

结果，速度增进了 255 个执行周期，虽然目标大小在程序 Dhrystone 版本 2.1 中增加了 372 字节。

(1) 选择子选项

当指定了速度 (SPEED) 选项时，对速度的优化将被执行，从而可能造成大小的增加。

要避免这个问题，可能需要使用调谐步骤提供详细的指定。建议的方法是使用各种子选项的有效函数。所要指定的子选项可通过结合它们的效果来决定，以便程序大小可以符合目标 ROM 大小。参考从程序 Dhrystone 版本 2.1 获取的下列数据：

优化函数	大小 (ROM)	执行周期数
全部指定	3420	1325
寄存器	3048	1580
转移到多个	3048	1580
结构赋值	3074	1527
切换判断值	3048	1580
内联函数的最大节点（105）	3314	1437
循环优化	3048	1580
表达式	3080	1526

注意： 在 H8/300 和 H8/300H 上，当未指定**寄存器 (Register)** 时，编译程序使用函数调用来执行寄存器保存/恢复任务（使用运行时例程程序库）。当指定了**寄存器 (Register)** 时，编译程序生成 PUSH/POP 指令而非使用函数调用。

在 H8S/2000 和 H8S/2600 系列上，寄存器保存/恢复任务始终由 STM/LDM 指令执行。（或 PUSH/POP 指令，根据所涉及的寄存器而定）。因此，在此情形下**寄存器 (Register)** 指定将无效。

下面显示当在 H8/300H 高级模式中指定了**寄存器 (Register)** 的执行结果：

优化函数	大小 (ROM)	执行周期数
全部指定	3422	1598
寄存器	3262	1721

应被自动内联扩展的函数节点数将由**内联函数的最大节点 (Maximum nodes of inline function)** 来指定。

节点的数量表示编译程序内部处理的单元，它是无法被准确检查的。然而，一般上若函数的大小越大，则节点的数量也越大。默认的节点计数是 105。

要禁止内联扩展（节点计数为 0），请将指定关闭。

下面显示当节点数量被设定至 0，即默认值，及最大值时的执行结果：（可在 1 至 65535 的范围内选择节点数量。）

优化函数	大小 (ROM)	执行周期数
内联函数的最大节点（1）	3052	1549
内联函数的最大节点（105）	3314	1437
内联函数的最大节点（65535）	3314	1437

有时候，指定内联扩展到所有函数可能会减低大小的效率及速度的效率。这是因为函数大小的增加禁止了优化。

当使用自动内联扩展时，必须提防尽量不要增加节点的数量。若指定的函数必须被内联扩展，则在 #pragma inline 语句中将它指定以实现效率。

要进一步获取有关速度 (SPEED) 选项的详细资料，请参考 5.4.7 节，速度选项。

5.2.2 调谐优化选项

(1) 使用块转移指令 (eepmov)

为结构的替换使用块转移指令 (EEPMOV)。

下面显示执行结果：

优化函数	大小 (ROM)	执行周期数
速度 (SPEED) 选项	3420	1325
速度 (SPEED) 选项+块转移指令	3366	1285

[指定方法]

对话框菜单： **C/C++** 标签**类别：[其他 (Other)]**，对**杂项 (Miscellaneous option)** 选择**以块副本的形式使用 EEPMOV (Use EEPMOV in block copy)。**

命令行： *eepmov*

EEPMOV 指令包含对 CPU 规格的下列限制：

EEPMOV.B → 不检测除了 NMI 以外的中断。

EEPMOV.W → 不检测除了 NMI 以外的中断。

若 NMI 中断在此命令执行期间发生，转移结果将不被保证。

打开编译列表以在目标列表中搜索 EEPMOV 指令。确保 EEPMOV 指令不受以上使用限制所影响。

当 EEPMOV 指令被用于特定结构的数据转移而非整个文件时，则使用内建函数 eepmov()；

要进一步获取详细资料，请参考 5.4.6 节，块转移指令。

(2) 其他优化选项的调谐

下面描述结合了经证实有大小效率的选项的指定。

首先，指定 1 字节枚举类型。

优化函数	大小 (ROM)	执行周期数
速度 (SPEED) 选项	3420	1325
速度 (SPEED) 选项 +块转移指令	3366	1285
速度 (SPEED) 选项 +块转移指令 +1 字节枚举类型	3392	1296

执行速度将稍微降低。

接下来，指定三个参数传递寄存器：

优化函数	大小 (ROM)	执行周期数
速度 (SPEED) 选项	3420	1325
速度 (SPEED) 选项 +块转移指令	3366	1285
速度 (SPEED) 选项 +块转移指令 +3 个参数传递寄存器	3348	1249

执行速度将会增进。

然后，指定变量分配寄存器计数：

优化函数	大小 (ROM)	执行周期数
速度 (SPEED) 选项	3420	1325
速度 (SPEED) 选项 +块转移指令	3366	1285
速度 (SPEED) 选项 +块转移指令 +无变量分配寄存器计数的扩展名	3366	1285

基于这些结果，块转移指令规格和三个参数传递寄存器规格的选项可被适当决定。要进一步获取详细资料，请参考 5.4.3 节，指定参数传递寄存器的数量。

5.2.3 使用模块间优化功能

本节描述使用模块间优化器的优化，以获取执行效率更高的目标程序。

在使用模块间优化器执行优化之前，先在编译程序或交叉汇编程序内指定模块间优化加载信息文件的输出。

[指定方法]

C/C++ C 编译程序

对话框菜单： **C/C++** 标签**类别：[优化 (Optimize)]**，选择**为模块间优化生成文件 (Generate file for inter-module optimization)。**

命令行： *goptimize*

交叉汇编程序

对话框菜单： **汇编 (Assembly)** 标签**类别：[目标 (Object)]**，选择**为模块间优化生成文件 (Generate file for inter-module optimization)。**

命令行： *goptimize*

在 HEW1.2 中，模块间优化加载信息文件也被提供给进行模块间优化时所连接的标准程序库。由于此文件在 Windows 版本中是以压缩形式提供，因此在使用前必须先解压缩。

通过双击和所要使用的程序库同名的压缩文件 (*.exe)，文件将自行解压然后生成一个包含了信息文件的目录。

要获取此程序库之模块间优化的详细资料，请参考 H8S，H8/300 系列 C/C++ 编译程序的补遗。

在 HEW2.0 或以上版本中，标准程序库生成程序的模块间优化功能应被使用来创建程序库。通过选中**标准程序库 (Standard Library)** 标签**类别：[优化 (Optimize)] 为模块间优化生成文件 (Generate file for inter-module optimization)**，将输出模块间优化加载信息文件。

(1) 默认优化

模块间优化器支持下列优化函数：

编号	描述	对话框菜单	子命令选项
1	统一常数/字符串	**统一字符串 (Unify strings)**	*String_Unify*
2	删除未参考的变量/函数	**删除死码 (Eliminate dead code)**	*Symbol_delete*
3	优化对变量的存取	**使用短寻址 (Use short addressing)**	*Variable_access*
4	优化对函数的存取	**使用间接调用/跳转 (Use indirect call/jump)**	*Funcation_call*
5	再分配寄存器	**再分配寄存器 (Reallocate registers)**	*Register*
6	删除相同的代码	**删除相同的代码 (Eliminate same code)**	*Same_code*
7	优化转移指令	**优化转移 (Optimize branches)**	*Branch*

编译程序最有效的优化显示如下：

优化函数	大小 (ROM)	执行周期数
有效的编译程序优化选项 （速度 (SPEED) 选项 +块转移指令 +3 个参数传递寄存器）	3348	1249

在模块间优化器中，默认设定为不执行优化，且将产生一个简单连接的模块。因此，执行结果和编译程序优化的结果是相同的。

(2) 指定模块间优化项目

在模块间优化器中逐一指定优化项目：

优化函数	模块间优化函数	大小 (ROM)	执行周期数
指定了编译程序优化选项	-	3348	1249
	统一常数/字符串	3348	1249
	删除未参考的变量/函数	2984	1249
	优化对变量的存取	3258	1232
	优化对函数的存取	3332	1250
	再分配寄存器	3332	1249
	删除死码	3348	1249
	优化转移指令	3348	1249

(3) 允许对速度的模块间优化

执行下列函数：统一常数/字符串、删除未参考的变量/函数、优化对变量的存取、再分配寄存器，及优化及转移指令。

优化函数	模块间优化函数	大小 (ROM)	执行周期数
指定了编译程序优化选项	-	3348	1249
	速度的优化	2906	1232

(4) 允许所有模块间优化函数

允许所有模块间优化函数：

优化函数	模块间优化函数	大小 (ROM)	执行周期数
指定了编译程序优化选项	-	3348	1249
	全部优化	2902	1232

当指定了此函数时，优化可能被应用到应禁止优化的部分。在指定此选项前请小心检查列表。

5.2.4 选择扩展函数

(1) 将寄存器分配到全局变量

将在大小效率的优化期间所指定的变量分配到全局寄存器：

(A)
```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L,Ptr_Glob=ER5)
```

同时如下指定以增加工作寄存器的数目：

(B)
```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L)
```

(C)
```
#pragma global_register(Ptr_Glob=ER5)
```

优化函数	大小 (ROM)	执行周期数
速度 (SPEED) 选项	3420	1325
速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器	3348	1249
速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器+指定了 #pragma global_register (A)	3318	1246
速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器+指定了 #pragma global_register (B)	3370	1262
速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器+指定了 #pragma global_register (C)	3296	1246

要进一步获取详细资料，请参考 5.4.8 节，将寄存器分配到全局变量。

(2) 在函数入口和出口点控制寄存器保存/恢复代码的输出

根据对大小的优化结果如下指定：

```
#pragma noregsave (main)
```

执行成果如下：

优化函数	大小 (ROM)	执行周期数
编译程序选项 模块间优化函数	2906	1232
编译程序选项 模块间优化函数 +指定了 #pragma noregsave	2906	1232

执行速度将会增进。要进一步获取详细资料，请参考 5.4.9 节，在函数入口和出口点控制寄存器保存/恢复代码的输出。

5.2.5 使用内联扩展功能

(1) 指定函数的内联扩展

#pragma inline 声明执行内联扩展的函数，而非函数调用。当提供了内联扩展时，虽然目标大小增加，执行速度却获得了提高。

然而，如有关速度 (SPEED) 选项的自动内联扩展一节中所描述，指定所有函数的内联扩展不单降低目标大小的性能，同样的也降低执行速度的性能。

#pragma inline 语句应被用以声明调用自嵌套深层的函数，从而有效提高执行速度。

程序 Dhrystone 版本 2.1 中的嵌套关系显示如下：

```
main:
     malloc:
     strcpy:
     Proc_1:
         Proc_3:
             Proc_7:
         Proc_6:
             Func_3:
         Proc_7:
     Func_2:
         Func_1:
         strcmp:
     Func_1:
     Proc_8:
     Proc_7:
     Proc_6:
         Func_3:
     Proc_5:
     Proc_4:
     Func_2:
```

在此情形下，函数开始从嵌套的最深层被指定。与自动内联扩展（选项中所指定的内联扩展）的 105 节点计数的比较显示如下：

优化函数	大小 (ROM)	执行周期数
无（默认）	3052	1549
自动内联扩展	3306	1445
指定了内联扩展 (Proc7, Func3)	3048	1589
指定了内联扩展 (Proc7, Func3, Func1, strcmp, Proc3, Proc6)	3048	1589
指定了内联扩展 (Proc7, Func3, Func1, strcmp, Proc3, Proc6, malloc, strcpy, Proc5, Proc4, Proc1)	3048	1589
对所有函数指定了内联扩展	3322	1445

注意： 函数 Proc8 和 Func2 不被内联扩展。

这些结果表示，自动内联扩展选项制造了快速的目标程序。

因此，高性能目标可通过考虑函数调用关系、执行周期数，及目标大小后，适当结合指定来创建。

#pragma inline 声明只在当函数本身及有关的函数调用包含在同一个文件中时有效。若对要内联扩展的函数指定了“静态(static)”，将不输出实际代码，且代码只在调用目标函数上被扩展，从而使到目标大小被缩减。建议始终使用此选项。

要进一步获取详细资料，请参考 5.4.10 节，指定函数的内联扩展。

(2) 指定汇编语言函数的内联扩展

在 C/C++ 中编码程序时，特别要求增强性能的段有时会以汇编语言编写。在这种情形下，若以汇编语言编写的函数以 #pragma inline_asm 指定，函数可在调用的位置被内联扩展。

要进一步获取详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”10.2.1 节，#pragma 扩展名和关键字 (#pragma Extension and Keywords)。

5.2.6 使用 CPU 指定的指令

(1) 分配到短 8 位绝对地址区

将对大小进行优化时所选取的变量分配到 8 位绝对地址区：

```
#pragma abs8 (Ch_1_Glob,Ch_2_Glob)
```

下面显示执行结果：

优化函数	大小 (ROM)	执行周期数
编译程序选项 模块间优化函数 +指定了 #pragma noregsave	2906	1232
编译程序选项 模块间优化函数 +指定了 #pragma noregsave +指定了 #pragma abs8	2906	1232

不同于面向大小的优化 (5.1.6)，这在 Dhrystone 版本 2.1 程序中并无区别。 然而，用户被建议使用此选项。

要进一步获取详细资料，请参考 5.4.11 节，使用 8 位绝对地址区。

(2) 分配到短 16 位绝对地址区

将对大小进行优化时所选取的变量分配到 16 位绝对地址区：

```
#pragma abs16 (Int_Glob,Bool_Glob,Arr_2_Glob,Ptr_Glob,Ptr_Glb_Next)
```

下面显示执行结果：

优化函数	大小 (ROM)	执行周期数
编译程序选项 模块间优化函数 +指定了 #pragma noregsave	2906	1232
编译程序选项 模块间优化函数 +指定了 #pragma noregsave +指定了 #pragma abs8 +指定了 #pragma abs16	2894	1231

于是，速度和大小均获得增进。

要进一步获取详细资料，请参考 5.4.12 节，使用 16 位绝对地址区。

(3) 分配到存储间接区

将对大小进行优化时所选取的变量分配到存储间接区：

```
#pragma
indirect(Proc0,main,malloc,Proc1,Proc2,Proc3,Proc4,Proc5,Proc6,Proc7,Proc8,
Func1,Func2,Func3)
```

下面显示执行结果：

优化函数	大小 (ROM)	执行周期数
编译程序选项 模块间优化函数 +指定了 #pragma noregsave +指定了 #pragma abs8 +指定了 #pragma abs16	2894	1231
编译程序选项 模块间优化函数 +指定了 #pragma noregsave +指定了 #pragma abs8 +指定了 #pragma abs 16 +指定了 #pragma indirect	2892	1232

结果，执行速度降低了，故此规格不应被使用。

5.3 大小与速度的效率结合

如前面的小节中所述，编译程序优化支持可缩减大小和增进执行速度的函数。对于各个函数和指定方法，请参考前面的小节。创建高性能程序的最好方法是将需要高密度的函数和需要高速性能的函数分隔入不同的文件内，然后可选择对个别文件进行大小和速度的优化。

即使函数不能被完全的分隔，能清楚了解整个程序中的哪一部分需要高速性能是非常重要的。通过对需要高速的文件（或函数）指定（选项 + 扩展函数 + 编码 + 模块间优化），并向其他部分提供大小优化，将可有效提高目标性能。

到目前为止所执行的调查结果可摘要如下：

下面列出以选项和扩展函数规格实行最佳的大小效率的检查结果：

		大小		速度	
规格	**描述**	**字节**	**%**	**周期**	**%**
默认	–	3048	100	1580	100
编译程序选项	指定了 1 字节枚举类型 +3 个参数传递寄存器	3034	99	1527	97
编译程序选项 +模块间优化函数	指定了 1 字节枚举类型 +3 个参数传递寄存器 +全部模块间优化函数	2946	97	1517	96
编译程序选项 +模块间优化函数 +扩展函数	指定了 1 字节枚举类型 +3 个参数传递寄存器 +全部模块间优化函数 +指定了 #pragma abs8 +指定了 #pragma abs16 +指定了 #pragma noregsave +指定了 #pragma indirect	2902	95	1496	95

下面列出以选项和扩展函数规格实行最佳的速度效率的检查结果：

		大小		执行速度	
规格	**描述**	**字节**	**%**	**周期**	**%**
默认	–	3048	100	1580	100
编译程序选项	速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器	3348	110	1249	79
编译程序选项 +模块间优化函数	速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器 +速度优先的模块间优化功能	2906	95	1232	78
编译程序选项 +模块间优化函数 +扩展函数	速度 (SPEED) 选项 +指定了块转移指令 +3 个参数传递寄存器 +速度优先的模块间优化功能 +指定了 #pragma noregsave +指定了 #pragma abs8 +指定了 #pragma abs16	2894	95	1231	78

因此，和没有指定选项的情形相比，使用了选项和扩展函数的程序 Dhrystone 版本 2.1，其性能在大小上提高了最多 5% 及在执行周期上提高了 22%。

选项和扩展函数的指定，比修改代码更为轻易和有效的增进了程序性能。充分利用这些项目以创建高性能的目标程序。

5.4 优化函数的详细资料

编译程序提供下列的优化函数。项目 1 至 23 代表编译程序的函数，而项目 24 至 30 则代表模块间优化器的函数。

性能是在下列条件下被衡量。

[测量的交叉工具]

H8S，H8/300 C/C++ 程序库生成程序（版本 2.01.00.001）

H8S，H8/300 C/C++ 编译程序（版本 6.01.00.009）

H8S，H8/300 汇编程序（版本 6.01.01.000）

优化连接编辑程序（版本 9.00.02.000）

[选项指定]

当选项指定的方法未在各节中描述时，将使用默认选项。

[测量条件]

条件	H8/300, H8/300H	H8S/2600,H8S/2000	H8SX
总线宽度	16	16	32
存储器的存取状态	2	1	1
取指令大小	-	-	32

编号	优化函数	大小缩减	速度增进	参考的节
1	使用 1 字节枚举类型	O	O	5.4.1
2	乘法/除法规格的扩展解释	O	O	5.4.2
3	指定参数传递寄存器的数量	Δ	Δ	5.4.3
4	增加变量分配寄存器的数量	Δ	Δ	5.4.4
5	优化外部变量	–	–	5.4.5
6	块转移指令	X	O	5.4.6
7	速度 (SPEED) 选项			5.4.7
8	寄存器保存/恢复代码的增进速度扩展	X	O	5.4.7(1)
9	移位表达式的增进速度代码扩展	X	O	5.4.7(2)
10	结构和复式数据的替换代码扩展	X	O	5.4.7(3)
11	切换语句的速度效率代码扩展	X	O	5.4.7(4)
12	小型函数的内联扩展	X	O	5.4.7(5)
13	循环表达式的速度效率代码扩展	Δ	O	5.4.7(6)
14	禁止运行时例程调用	X	O	5.4.7(7)
15	将寄存器分配到全局变量	Δ	Δ	5.4.8
16	在函数入口/出口点控制寄存器保存/恢复代码的输出	O	O	5.4.9
17	指定函数的内联扩展	X	O	5.4.10
18	使用 8 位绝对地址区	O	O	5.4.11
19	使用 16 位绝对地址区	O	O	5.4.12
20	分配到间接存储区	O	X	5.4.13
21	扩展的存储间接	O	X	5.4.14
22	2 字节指针	O	O	5.4.15
23	边界对齐	O	O	5.4.16
24	统一常数/字符串	–	–	5.4.17(1)
25	删除未参考的变量/函数	–	–	5.4.17(2)
26	优化对变量的存取	–	–	5.4.17(3)
27	优化对函数的存取	–	–	5.4.17(4)
28	优化寄存器保存/恢复代码	–	–	5.4.17(5)
29	统一公用代码	–	–	5.4.17(6)
30	优化转移指令	–	–	5.4.17(7)

图例：

O: 获得增进

Δ: 增进在一些程序中达到

X: 效率减低

5.4.1 使用 1 字节枚举类型

大小	O	速度	O

描述

若枚举类型成员的值介于 –128 和 127 之间，则可以使用此选项来指定 1 字节类型的操作。

根据语言规格，枚举类型的值一般会占用 2 字节，然而，当指定了枚举选项时，枚举类型成员的值将被视为 1 字节的数据进行操作。

因为此选项并非基于语言规格，它在编译程序的默认状态中被设定为“未指定”。然而，建议用户始终指定此选项。

指定方法

对话框菜单： **C/C++ 标签类别：[其他 (Other)] 将枚举当作字符处理，若它处于字符的范围内 (Treat enum as char if it is in the range of char)**

命令行： *byteenum*

实例

要将枚举类型数据 E1 设定为 1：

（C/C++ 程序）

```
enum EN1 {a=0,b,c,d,e}E1;
void func(void)
{
    E1=1;
}
```

◀枚举成员的值介于以一字节代表的数据范围内。

（汇编扩展代码）

未指定

```
_func:
      MOV.W       #1,R0
      MOV.W       R0,@_E1:32
      RTS
      .SECTION    B,DATA,ALIGN=2
_E1:
      .RES.W      1
```

2 字节数据

已指定

```
_func:
      MOV.B       #1,R0L
      MOV.B       R0L,@_E1:32
      RTS
      .SECTION    B,DATA,ALIGN=2
_E1:
      .RES.B      1
```

1 字节数据

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	12	10	12	10	10
已指定	10	8	10	8	8

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	8	8	6
已指定	8	8	6

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	11	9	22	18	18
已指定	10	8	20	16	16

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	8	8	7
已指定	8	8	7

5.4.2 乘法/除法规格的扩展解释

大小	O	速度	O

描述

乘法/除法运算的代码扩展由 ANSI 标准的扩展解释输出。

当指定了此选项时，计算结果可能和没有此选项时有所不同，因其解释有如下所示的差异：

操作数	操作中 us1*us2 的大小（在 H8S/2600）	
	扩展的解释	**ANSI 标准解释**
Unsigned short us1,us2; unsigned long ul; ul=us1*us2;	us1*us2 作为无符号长型操作 输出实例： MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd MOV.L ERd,@_ul us1*us2 的结果以 4 字节分配到 u1。	us1*us2 作为无符号短型操作 输出实例： MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L ERd,@_ul us1*us2 结果的下端 2 字节以零扩展分配到 u1。
Unsigned short us1,us2,us3 Unsigned short us; us=us1*us2/us3;	us1*us2 作为无符号长型计算 输出实例： MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd MOV.L @_us3.Rs DIVXU.W Rs,ERd MOV.L Rd,@_us us1*us2 结果的 4 字节作为操作指令的被除数分配。	us1*us2 作为无符号短型计算 输出实例： MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L @_us3.Rs DIVXU.W Rs,ERd MOV.L Rd,@_us us1*us2 结果的下端 2 字节被零扩展，并作为除法运算的被除数分配。

指定方法

对话框菜单： **C/C++ 标签类别：[目标 (Object)] 乘除运算规格 (Mul/Div operation specification) 非 ANSI（保证 32 位为 16 位*16 位的结果）(Non ANSI (Guarantee 32bit as a result of 16bit*16bit))。**

命令行： *cpuexpand*

实例

要将两个 2 字节数据的乘法结果保存在 4 字节类型数据内：

（C/C++ 程序）

```
unsigned long ll;
unsigned short a,b;
void func()
{
    ll=a*b;
}
```

（汇编扩展代码）

未指定

```
_func:
    MOV.W       @_a:32,R0
    MOV.W       @_b:32,E0
    MULXU.W     E0,ER0
    EXTU.L      ER0
    MOV.L       ER0,@_ll:32
    RTS
```

已指定

```
_func:
    MOV.W       @_a:32,R0
    MOV.W       @_b:32,E0
    MULXU.W     E0,ER0

    MOV.L       ER0,@_ll:32
    RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	26	20	26	20	24
已指定	24	18	24	18	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	26	26	20
已指定	24	24	18

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	24	20	62	54	136
已指定	23	19	60	52	184

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	14	14	13
已指定	14	14	12

5.4.3 指定参数传递寄存器的数量

大小	Δ	速度	Δ

描述

分配参数的寄存器数量可使用此规格设定。当参数被分配到寄存器时，存取的大小要比分配到堆栈时来得低。另一方面，当参数传递寄存器的数量增加时，工作寄存器的区域将会缩减，且有时如在复杂的操作上，数据不被分配到寄存器。在此情形下，目标程序的效率将会降低。

参数传递寄存器的数量也可用选项来指定。

比较两种规格的执行结果，然后采用其中较佳的。

指定方法

对话框菜单： **CPU 标签，将参数传递寄存器的数量从 2（默认）更改至 3 (Change number of parameter-passing registers from 2(default) to 3)。**

命令行： *regparam=3*

实例

在下面的实例中，当指定了三个参数传递寄存器时，效率将被提高。

（C/C++ 程序）

```
extern short ee;
void func(short a,short b,short c,short d,long e)
{
    ee=a*b*c*d/e;
}
```

（汇编扩展代码的编译结果）

未指定

```
_func:
        PUSH.L      ER2
        SUBS.L      #2,SP
        MOV.W       R0,R2
        MULXU.W     E0,ER2
        MULXU.W     R1,ER2
        MOV.W       R2,R1
        MULXU.W     E1,ER1
        EXTS.L      ER1
        MOV.W       R0,@SP
        MOV.L       ER1,ER0
        MOV.L       @(10:16,SP),ER1
        JSR         @$DIVL$3:24
        MOV.W       R0,@_ee:32
        POP.L       ER2
        RTS
```

已指定

```
_func:
        PUSH.L      ER3
        SUBS.L      #2,SP
        MOV.W       R0,R3
        MULXU.W     E0,ER3
        MULXU.W     R1,ER3
        MOV.W       R3,R1
        MULXU.W     E1,ER1
        EXTS.L      ER1
        MOV.W       R0,@SP
        MOV.L       ER1,ER0
        MOV.L       ER2,ER1
        JSR         @$DIVL$3:24
        MOV.W       R0,@_ee:32
        ADDS.L      #2,SP
        POP.L       ER3
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	42	40	46	44	72
已指定	38	36	42	40	70

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	36	36	36
已指定	34	34	32

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	144	138	294	282	686
已指定	140	134	284	272	682

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	44	43	44
已指定	39	39	41

说明与备注

此规格被应用到所有文件和连接的程序库。它不能对每个文件个别指定。因此，当修改此规格时，记得要在所有文件和连接的程序库中更改选项的规格。

另外，若被优化的程序连接到汇编程序，函数调用的界面也需要被修改。

要获取有关 C/C++ 程序和汇编语言程序之间连接的描述，请参考 H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户指南 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual) 9.3 节，连接 C/C++ 程序和汇编程序 (Linking C/C++ Programs and Assembly Programs)。

5.4.4 增加变量分配寄存器的数量

大小	Δ	速度	Δ

描述

分配变量的寄存器数量可使用此选项来设定（4 或 3 个寄存器）。

大部分程序在指定了四个寄存器时可以更好的执行。然而，若程序包含复杂的表达式而造成了寄存器短缺，指定三个寄存器可有更好的执行性能。

对普通执行指定四个变量分配寄存器，且在需要时比较两种规格的执行结果，比如在 ROM 的程序存储上进行比较。

指定方法

对话框菜单： **C/C++ 标签类别：[其他 (Other)] 为寄存器变量增加寄存器 (Increase a register for register variable)**

命令行： *regexpansion*

实例

在下面的实例中，当指定了三个变量赋值寄存器时，效率将被提高。

（C/C++ 程序）

```
long func(short a,long b,short c,char d,long e)
{
    long x,y,z;
    x=a+b;
    y=b*c;
    z=a/e;
    return (a*x*(z+y)*b*d+e*z-e/x*c/(x*y*a*z));
}
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
变量寄存器： 4	202	202	190	190	416
变量寄存器： 3	202	202	190	190	416

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
变量寄存器： 4	150	150	150
变量寄存器： 3	150	150	150

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
变量寄存器： 4	783	752	2158	2082	4836
变量寄存器： 3	783	752	2158	2082	4836

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
变量寄存器： 4	174	187	169
变量寄存器： 3	174	187	169

5.4.5 外部变量的优化

大小	–	速度	–

描述

```
:
a=0;  //(1)
a=1;  //(2)
:
```

编译程序通过删除上面 (1) 的替换来优化上述的表达式。若替换 (1) 不应被删除，如可能用作 I/O 端口或中断处理中的一个变量，则对变量声明易失性。

通过使用该选项，可对所有指定文件中的外部变量禁止优化。

然而，这可能减低目标的效率。当使用该选项时，在源程序中对不应被优化的变量声明易失性，如中断函数或 I/O 寄存器中的变量，然后，编译禁止优化外部变量的形成程序。

指定方法

对话框菜单： **C/C++ 标签类别：[其他 (Other)] 避免将外部符号视为易失来优化 (Avoid optimizing external symbols treating them as volatile)**

命令行： *volatile*

实例

要将数值 0、1 和 2 以此顺序分配给外部变量 a：

（C/C++ 程序）

```
unsigned int a;
void func()
{
    a=0;
    a=1;
    a=2;
}
```

（汇编扩展代码）

未指定

```
_func:
                    仅限 a=2 代码
      MOV.W       #2,R0
      MOV.W       R0,@_a:32
      RTS
```

已指定

```
_func:
      SUB.W       R0,R0          a=0
      MOV.W       R0,@_a:32
      MOV.B       #1,R0L
      MOV.W       R0,@_a:32      a=1
      MOV.B       #2,R0L
      MOV.W       R0,@_a:32
      RTS                        a=2
```

说明与备注

当禁止了外部变量的优化时，文件中的所有外部变量被更改为易失性变量。 个别为每个变量设定易失性，指定如下：

```
volatile unsigned int a;
void func()
{
    a=0;
    a=1;
    a=2;
}
```

以易失性选项输出如以上所示的相同代码

默认情况下，外部变量的优化通过编译程序选项来允许。

5.4.6 块转移指令

大小	X	速度	O

描述

结构替换一般上通过调用运行时例程来处理。当使用了此选项时，块转移指令在结构替换表达式上输出，同时执行速度也被提高。

然而，若 NMI 中断在 EEPMOV.W 指令执行期间发生，转移结果将不被保证。

在指定此选项前先检查这项条件。

要仅在结构数据转移的一部分中输出 EEPMOV 指令，则指定 eepmov() 内建函数。

指定方法

对话框菜单： **C/C++ 标签类别：[其他 (Other)] 以块副本的形式使用 EEPMOV (Use EEPMOV in block copy)**

命令行： *eepmov*

实例

要以结构 s1 替换 s2：

（C/C++ 程序）

```
struct S{
    char  cc;
    short ss;
    long  ll;
    long  ll2;
}s1,s2;
void main()
{
    s1=s2;
}
```

（汇编扩展代码的编译结果）

未指定

```
_main:
        PUSH.L      ER2
        MOV.L       #_s2,ER0
        MOV.L       #_s1,ER1
        SUB.L       ER2,ER2
        MOV.B       #12,R2L
        JSR         @$MVN$3:24
```

通过运行时例程调用来处理

```
        POP.L       ER2
        RTS
```

已指定

```
_main:
        STM.L       (ER4-ER6),@-SP
        MOV.L       #_s2,ER5
        MOV.B       #12,R4L
        MOV.L       #_s1,ER6
        EEPMOV.B
```

扩展入 EEPMOV 指令

```
        LDM.L       @SP+,(ER4-ER6)
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	30	22	30	22	22
已指定	28	24	26	22	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	28	26	22
已指定	22	22	18

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	117	102	270	224	256
已指定	58	55	226	210	168

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	66	66	57
已指定	24	24	24

5.4.7 速度选项

描述

编译程序通常输出代码有大小效率的目标。当指定了此选项时，将输出有执行速度效率的目标。

下列项目指定的是具速度效率的目标输出，而不是具大小效率的目标输出：

描述	**参考**
寄存器保存/恢复代码的速度效率代码扩展	5.4.7(1)
移位表达式的速度效率代码扩展	5.4.7(2)
结构和复式数据的赋值代码扩展	5.4.7(3)
函数的内联扩展	5.4.7(4)
循环表达式的速度效率代码扩展	5.4.7(5)
切换语句的速度效率代码扩展	5.4.7(6)
对算术操作禁止运行时例程调用	5.4.7(7)

这些项目可以个别指定。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 面向速度的优化 (Speed oriented optimization)**

命令行： *speed*

(1) 寄存器保存/恢复代码的速度效率代码扩展

大小	X	速度	O

描述

在函数的入口或出口点，函数中所使用的寄存器被保存或恢复。在 H8/300H 或 H8/300 系列上，当要保存/恢复的寄存器数量为三个或以上时，寄存器通过调用运行时例程而被保存/恢复。

当使用了运行时例程时，目标的大小被缩减了，然而，也因为函数调用的处理或无必要的寄存器保存/恢复而使执行速度降低。若仅保存/恢复必须的寄存器，且不调用运行时例程，将可提高执行速度，虽然目标大小因此增加。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：寄存器 (Register)**

命令行： *speed=register*

实例

要在指定 300HA CPU/操作模式的同时定义函数 *sub*：

（C/C++ 程序）

```
long a,b;
long sub(char c1,short s2,short s3)
{
    s3=a+b;
    return (c1+s2+s3);
}
```

在寄存器保存/恢复时调用的运行时例程，根据是否指定了优化及参数传递寄存器的数量而异。

（汇编扩展代码）

未指定

```
_sub:
       JSR         @$sp_regsv$3:24

       MOV.B       R0L,R5L
       MOV.W       @_a+2:24,R1
       MOV.W       @_b+2:24,R2
       ADD.W       R2,R1
       EXTS.W      R5
       ADD.W       E0,R5
       ADD.W       R1,R5
       EXTS.L      ER5
       MOV.L       ER5,ER0
       JMP         @$spregld2$3:24
```

已指定

```
_sub:
       PUSH.L      ER5
       PUSH.W      R2
       MOV.B       R0L,R5L
       MOV.W       @_a+2:24,R1
       MOV.W       @_b+2:24,R2
       ADD.W       R2,R1
       EXTS.W      R5
       ADD.W       E0,R5
       ADD.W       R1,R5
       EXTS.L      ER5
       MOV.L       ER5,ER0
       POP.W       R2
       POP.L       ER5
       RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	24	20	28	24	38
已指定	24	20	28	24	44

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	24	24	20
已指定	24	24	20

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	18	15	48	42	134
已指定	18	15	48	42	94

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	17	15	14
已指定	17	15	14

(2) 移位表达式的速度效率代码扩展

大小	X	速度	O

描述

移位操作所生成的目标代码具有增进的速度，而非缩减的大小。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：转移到多个 (Shift to multiple)**

命令行：*speed=shift*

实例

要多次移位变量 *a*：

（C/C++ 程序）

```
unsigned char a=0x80;
int dat;
void main(void)
{
    a>>=dat;
}
```

（汇编扩展代码的编译结果）

未指定

```
_main:
        MOV.L       #_a,ER0
        MOV.W       @_dat:32,R1
        JSR         @$DSRUC$3:24
```

通过调用运行时例程来处理

```
        RTS
```

已指定

```
_main:
            MOV.B       @_a:32,R0L
            MOV.B       @_dat+1:32,R0H
L5:
            DEC.B       R0H
            BMI         L8:8
            SHLR.B      R0L
            BRA         L7:8
L6:
            MOV.B       R0L,@_a:32
            RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	25	19	19	15	15
已指定	49	43	29	23	23

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	25	25	19
已指定	25	25	19

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	39	33	78	64	64
已指定	29	25	40	32	32

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	14	14	12
已指定	14	14	12

(3) 结构和复式数据的赋值代码扩展

大小	Δ	速度	O

描述

当结构或复式数据被分配时，通常将输出被用以调用运行时例程的代码（小型结构的情形除外）。因此，若执行的处理没有调用运行时例程，执行速度将被提高。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：结构赋值 (Struct assignment)**

命令行： *speed=struct*

实例

要将结构 s2 分配到 s1：

（C/C++ 程序）

```
struct S{
    unsigned char cc;
    short ss;
    long ll;
}s1,s2;
void main(void)
{
    s1=s2;
}
```

（汇编扩展代码的编译结果）

未指定

```
_main:
        PUSH.L      ER2
        MOV.L       #_s2,ER0
        MOV.L       #_s1,ER1
        SUB.L       ER2,ER2
        MOV.B       #8,R2L
        JSR         @$MVN$3:24

        POP.L       ER2
        RTS
```

运行时例程处理

已指定

```
_main:
        PUSH.L      ER2
        MOV.L       #_s2,ER0
        MOV.L       #_s1,ER1
        MOV.L       @ER0+,ER2
        MOV.L       ER2,@ER1
        MOV.L       @ER0,ER2
        MOV.L       ER2,@(4:16,ER1)
        POP.L       ER2
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	18	14	30	22	22
已指定	40	32	40	36	34

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	22	22	18
已指定	22	22	18

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	49	44	244	198	220
已指定	39	32	78	72	124

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	18	14	14
已指定	18	14	14

(4) 函数的内联扩展

大小	X	速度	O

描述

当使用选项指定了内联扩展时，小型函数将被内联扩展，进而提高执行速度。然而，若符合下列任何一项条件，内联扩展将不被执行：

- 函数在指定 #pragma inline 之前被定义。
- 包含变量参数。
- 参数的地址被参考。
- 实际参数和虚设参数的类型不同。
- 超出了内联扩展的大小限制。

内联扩展的大小限制表示所指定函数的节点数量。

节点数量表示编译程序内部处理中所使用的处理单元，可从 1 到 65535 的范围内选择。若指定了小的节点数，只有小函数会被内联扩展，然而，若指定了大的节点数，大型函数也可被内联扩展。

默认数是 105。

若对函数指定了 #pragma inline，函数将不受内联扩展的大小限制所影响，而被内联扩展。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：内联函数的最大节点 (Maximum nodes of inline function)**

命令行：*speed=inline[=(node)]*

实例

要调用命名为 *func* 的函数：

（C/C++ 程序）

```
extern long a;
void func(void);
void sub(void)
{
    func();
    a+=2;
}
void func(void)
{
    a++;
}
```

（汇编扩展代码的编译结果）

未指定

```
_sub:
        BSR         _func:8
        MOV.L       #_a,ER0
        MOV.L       @ER0,ER1
        INC.L       #2,ER1
        MOV.L       ER1,@ER0
        RTS
_func:
        MOV.L       #_a,ER0
        MOV.L       @ER0,ER1
        INC.L       #1,ER1
        MOV.L       ER1,@ER0
        RTS
```

函数调用

已指定

```
_sub:
        MOV.L       @_a:32,ER0
        INC.L       #1,ER0
        INC.L       #2,ER0
        MOV.L       ER0,@_a:32
        RTS
_func:
        MOV.L       #_a,ER0
        MOV.L       @ER0,ER1
        INC.L       #1,ER1
        MOV.L       ER1,@ER0
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	44	36	40	36	38
已指定	58	46	52	46	42

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	30	28	24
已指定	34	34	28

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	41	34	78	68	182
已指定	31	26	58	52	166

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	23	23	21
已指定	15	15	14

说明

若调用目标函数包含在调用方的相同文件内，且函数不被任何其他文件所调用，函数的外部定义将不被生成，而当在函数声明中指定了静态时，函数大小将被缩减。

(5) 循环表达式的速度效率代码扩展

大小	Δ	速度	O

描述

满足文件中下列所有条件的循环，被以扩展代码输出：

- 循环的初始值是一个常数。
- 循环的最终判断值是一个常数。
- 循环的重复数是 3 的倍数或者偶数。
- 循环中没有包含转至 (goto) 标签。
- 循环只包含表达式，且表达式的数量为 10 或以下。
- 已指定了优化。

当循环被扩展时，程序大小将增加。要增进特定循环的执行速度，则在程序中提供被循环扩展的编码。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：循环优化 (Loop optimization)**

命令行： *speed=loop*

实例

要零清除数组 *a* 的内容：

（C/C++ 程序）

```
int a[10];

void f(void)
{
    int i;

    for (i=0;i<10;i++)
        a[i]=0;
}
```

（汇编扩展代码的编译结果）

未指定

```
_f:
        PUSH.L      ER6
        SUB.W       R6,R6
        SUB.W       R1,R1
L6:
        EXTS.L      ER6
        MOV.L       ER6,ER0
        SHLL.L      ER0
        MOV.W       R1,@(_a:32,ER0)
        INC.W       #1,R6
        CMP.W       #10:16,R6
        BLT         L6:8
        POP.L       ER6
        RTS
```

已指定

```
_f:
        MOV.L       #_a,ER1
        SUB.L       ER0,ER0

L6:
        MOV.W       R0,@ER1
        INC.W       #1,E0
        INC.L       #2,ER1
        MOV.W       R0,@ER1
        INC.W       #1,E0
        INC.L       #2,ER1
        CMP.W       #10,E0
        BLT         L6:8
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	34	26	36	22	28
已指定	38	28	40	30	40

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	20	20	18
已指定	36	36	32

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	132	103	294	212	244
已指定	88	72	162	138	244

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	95	87	87
已指定	75	71	71

说明

此规格有时缩减代码的大小。

在调谐选项时不时尝试此选项。

(6) 切换语句的速度效率代码扩展

大小	Δ	速度	O

描述

切换语句通过此规格，使用输出较少执行周期数的方法来进行扩展。

共有两种方法可用以扩展切换语句，即表 (table) 法和如果－就 (if-then) 法。

通常，编译程序决定使用哪一种方法缩减大小较佳。

在如果－就 (if-then) 法中，切换语句评估表达式的值将与 case 标签的值相比。若它们是一样的，则将根据 case 标签被包含的次数重复跳转到 case 标签。因此，在此方法中，目标代码的大小根据 case 标签被包含在切换语句中的次数而增加。

在表 (table) 法中，case 标签跳转的目的地被存储在跳转表内，且与切换语句评估表达式匹配的 case 标签语句，将通过跳转表的单一参考而被执行。在此情形下，在常数区内分配的跳转表大小，将根据切换语句内所包含的 case 标签数酌量增加，然而，执行速度始终为常数。

当指定了速度 (SPEED) 选项时，增进执行速度的处理方法根据上述条件来选取。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：切换判断值 (Switch judgement)**

命令行： *speed=switch*

实例

要替换变量 *a* 的值：

（C/C++ 程序）

```
extern unsigned a;
void sub(void)
{
    switch(a){
    case 0: a=1;break;
    case 2: a=2;break;
    case 4: a=3;break;
    case 6: a=4;break;
    case 8: a=5;break;
    case 10: a=6;break;
    case 12: a=7;break;
    case 14: a=8;break;
    case 16: a=9;break;
    case 18: a=10;break;
    case 20: a=11;break;
    }
}
```

（汇编扩展代码的编译结果）

未指定

```
_sub:MOV.L      #_a:32,ER1
      MOV.W      @ER1,R0
      MOV.B      R0H,R0H
      BNE        L16:8
      CMP.B      #0:8,R0L
      BEQ        L5:8
      CMP.B      #2:8,R0L
      BEQ        L6:8
      CMP.B      #4:8,R0L
      BEQ        L7:8
      CMP.B      #6:8,R0L
      BEQ        L8:8
      CMP.B      #8:8,R0L
      BEQ        L9:8
      CMP.B      #10:8,R0L
      BEQ        L10:8
      CMP.B      #12:8,R0L
      BEQ        L11:8
      CMP.B      #14:8,R0L
      BEQ        L12:8
      CMP.B      #16:8,R0L
      BEQ        L13:8
      CMP.B      #18:8,R0L
      BEQ        L14:8
      CMP.B      #20:8,R0L
      BEQ        L15:8
      RTS
L5:   MOV.W      #1:16,R0
      BRA        L26:8
L6:   MOV.W      #2:16,R0
      BRA        L26:8
L7:   MOV.W      #3:16,R0
      BRA        L26:8
L8:   MOV.W      #4:16,R0
      BRA        L26:8
L9:   MOV.W      #5:16,R0
      BRA        L26:8
L10:  MOV.W      #6:16,R0
      BRA        L26:8
L11:  MOV.W      #7:16,R0
      BRA        L26:8
L12:  MOV.W      #8:16,R0
      BRA        L26:8
L13:  MOV.W      #9:16,R0
      BRA        L26:8
L14:  MOV.W      #10:16,R0
      BRA        L26:8
L15:  MOV.W      #11:16,R0
L26:  MOV.W      R0,@ER1
L16:  RTS
```

已指定

```
_sub:MOV.L      #_a,ER1
      MOV.W      @ER1,R0
      CMP.W      #20,R0
      BHI        L18:8
      EXTU.L     ER0
      MOV.B      @(L19:32,ER0),R0L
      EXTU.W     R0
      EXTU.L     ER0
      ADD.L      #L7,ER0
      JMP        @ER0
L5:   MOV.W      #1,R0
      BRA        L27:8
L6:   MOV.W      #2,R0
      BRA        L27:8
L7:   MOV.W      #3,R0
      BRA        L27:8
L8:   MOV.W      #4,R0
      BRA        L27:8
L9:   MOV.W      #5,R0
      BRA        L27:8
L10:  MOV.W      #6,R0
      BRA        L27:8
L11:  MOV.W      #7,R0
      BRA        L27:8
L12:  MOV.W      #8,R0
      BRA        L27:8
L13:  MOV.W      #9,R0
      BRA        L27:8
L14:  MOV.W      #10,R0
      BRA        L27:8
L15:  MOV.W      #11,R0
L27:  MOV.W      R0,@ER1
L16:  RTS
      .SECTION   C,DATA,ALIGN=2
L17:  .DATA.B    L5-L5
      .DATA.B    L16-L5
      .DATA.B    L6-L5
      .DATA.B    L16-L5
      .DATA.B    L7-L5
      .DATA.B    L16-L5
      .DATA.B    L8-L5
      .DATA.B    L16-L5
      .DATA.B    L9-L5
      .DATA.B    L16-L5
      .DATA.B    L10-L5
      .DATA.B    L16-L5
      .DATA.B    L11-L5
      .DATA.B    L16-L5
      .DATA.B    L12-L5
      .DATA.B    L16-L5
      .DATA.B    L13-L5
      .DATA.B    L16-L5
      .DATA.B    L14-L5
      .DATA.B    L16-L5
      .DATA.B    L15-L5
      .DATAB.B   1,0
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	136	120	126	114	120
已指定	136	120	126	114	120

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	118	118	108
已指定	118	118	108

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	44	33	66	52	66
已指定	44	33	66	52	66

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	23	24	18
已指定	23	24	18

说明

在上面的实例中，增进了速度和缩减了大小的代码，因为表 (table) 法的采用而被生成。然而，根据 *a* 的值，若未指定此选项，可能输出更佳的代码。

(7) 对算术操作禁止运行时例程调用

大小	X	速度	O

描述

当选定了此选项时，算术操作、比较表达式，或赋值表达式被扩展入代码，而不使用运行时例程（此选项被一些表达式所禁止）。

指定方法

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options) 表达式 (expression)**

命令行： *speed=expression*

实例

要执行乘法：

（C/C++ 程序）

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

（汇编扩展代码的编译结果）

未指定

```
_main:

        MOV.B       @_c:32,R0L
        EXTS.W      R0
        EXTS.L      ER0
        MOV.L       @_b:32,ER1
        JSR         @$MULL$3:24









        MOV.L       ER0,@_a:32

        RTS
```

已指定

```
_main:
        STM.L       (ER2-ER3),@-SP
        MOV.B       @_c:32,R0L
        EXTS.W      R0
        EXTS.L      ER0
        MOV.L       @_b:32,ER1
        MOV.W       E0,R2
        MULXU.W     R1,ER2
        MOV.W       E1,R3
        MULXU.W     R0,ER3
        MULXU.W     R1,ER0
        ADD.W       R2,E0
        ADD.W       R3,E0
        MOV.L       ER0,@_a:32
        LDM.L       @SP+,(ER2-ER3)
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	32	26	32	26	38
已指定	52	46	48	42	46

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	30	30	24
已指定	30	30	24

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	63	57	180	168	410
已指定	54	50	266	248	366

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	18	18	17
已指定	18	18	17

5.4.8 将寄存器分配到全局变量

大小	Δ	速度	Δ

描述

当常用的外部变量被分配到寄存器时，存取代码将被缩短。

注意未被优化的外部变量，如 I/O 变量，不能被分配到寄存器。

外部变量可被分配到如下所示的寄存器：

ER4	E4	R4H	R4L
ER5	E5	R5H	R5L

对于 CPU 300，R4 和 R5 可被使用。

[格式]

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

实例

要将 1 字节和 2 字节数据分配到寄存器：

（C/C++ 程序）

```
#pragma global_register (a=R4,b=R5L)
int a; char b;
void func();
void main()
{
    a=10;
    b=20;
    func();
}
void func()
{
    a++;
    b-=2;
}
```

（汇编扩展代码）

未指定

```
_main:
    MOV.W       #10,R0
    MOV.W       R0,@_a:32
    MOV.B       #20,R0L
    MOV.B       R0L,@_b:32
_func:
    MOV.L       #_a,ER0
    MOV.W       @ER0,R1
    INC.W       #1,R1
    MOV.W       R1,@ER0
    MOV.L       #_b,ER0
    MOV.B       @ER0,R1L
    ADD.B       #-2,R1L
    MOV.B       R1L,@ER0
    RTS
    .SECTION    B,DATA,ALIGN=2
_a:.RES.W      1
_b:.RES.B      1
```

已指定

```
_main:
    MOV.W       #10,R4
    MOV.B       #20,R5L
_func:
    INC.W       #1,R4
    ADD.B       #-2,R5L
    RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	52	40	48	40	40
已指定	20	20	16	16	16

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	38	36	28
已指定	18	16	16

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	37	30	70	60	60
已指定	15	14	26	24	24

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	21	21	18
已指定	12	12	12

注意

(a) 此选项可在 #pragma global_register 被声明后，用于变量定义和变量声明。
(b) 此选项可用于简单类型或指针类型的全局变量。它不能被用于复式变量。
(c) 初始值无法被指定。另外，地址无法被参考。
(d) 从连接目标参考特定变量（在文件中没有寄存器规格）不被保证。
(e) 中断函数中的规格或参考不被保证。
(f) 变量和寄存器无法被重复指定。此选项无法和 #pragma abs8 或 #pragma abs16 声明一同指定。

当指定了此选项时，模块间优化无法对程序库执行。除了下面所述，出自模块间优化目标的所有程序库函数：

[在 PC]

<HEW1.2>

删除程序库进行模块间优化时，所解压的程序库同名目录。

<HEW2.0 或以上版本>

使用标准程序库生成程序在标准程序库标签内的预包含选项，对含有 #pragma global_register 声明的标头文件指定包含。

[在 UNIX]

修改和程序库同名的目录名称。

5.4.9 在函数入口/出口点控制寄存器保存/恢复代码的输出

大小	O	速度	O

描述

对于所有函数，编译程序在函数入口点保存将要在函数中使用的寄存器，然后在函数出口点将它们恢复。

当寄存器的保存/恢复处理通过此选项来控制时，寄存器保存/恢复代码的大小可为主要函数或仅包含函数调用的函数缩减。

当指定了 #pragma regsave 时，所有寄存器将被保存/恢复。寄存器在函数调用之前和之后的保证值将不被分配。

当指定了 #pragma noregsave 时，寄存器保存/恢复将被禁止，不管寄存器是否在函数中被使用。

[格式]

#pragma regsave（<函数名称>[，…]）

#pragma noregsave（<函数名称>[，…]）

实例

要从函数 *noregf* 调用函数 *regf*:

（C/C++ 程序）

未指定

```
void regf();
void noregf(int);
void func();

extern int X,Y,Z,XX;
void regf(void)
{
    int A=X;
    Y=A;
    noregf(X);
    Z=A;
}
void noregf(int P)
{
    int B=P;
    Y=B;
    func(X);
    Z=B;
}
```

已指定

```
#pragma regsave (regf)
#pragma noregsave (noregf)
void regf();
void noregf(int);
void func();

extern int X,Y,Z,XX;
void regf(void)
{
    int A=X;
    Y=A;
    noregf(X);
    Z=A;
}
void noregf(int P)
{
    int B=P;
    Y=B;
    func(X);
    Z=B;
}
```

（汇编扩展代码）

未指定

```
_regf:
       PUSH.W      R6
       MOV.W       @_X:32,R6
       MOV.W       R6,@_Y:32
       MOV.W       R6,R0
       BSR         _noregf:8
       MOV.W       R6,@_Z:32
       POP.W       R6
       RTS
_noregf:
       PUSH.W      R6
       MOV.W       R0,R6
       MOV.W       R6,@_Y:32
       MOV.W       @_X:32,R0
       JSR         @_func:24
       MOV.W       R6,@_Z:32
       POP.W       R6
       RTS
```

已指定

```
_regf:
       STM.L       (ER2-ER3),@-SP
       STM.L       (ER4-ER6),@-SP
       MOV.W       @_X:32,R6
       MOV.W       R6,@_Y:32
       MOV.W       R6,R0
       PUSH.W      R6
       BSR         _noregf:8
       POP.W       R6
       MOV.W       R6,@_Z:32
       LDM.L       @SP+,(ER4-ER6)
       LDM.L       @SP+,(ER2-ER3)
       RTS
_noregf:
       MOV.W       R0,R6
       MOV.W       R6,@_Y:32
       MOV.W       @_X:32,R0
       JSR         @_func:24
       MOV.W       R6,@_Z:32
       RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	66	52	66	52	52
已指定	80	66	68	54	54

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	68	66	52
已指定	78	76	62

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	66	54	132	108	108
已指定	91	79	266	232	190

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	46	43	38
已指定	60	58	60

5.4.10 指定函数的内联扩展

大小	X	速度	O

描述

当指定了内联扩展时，扩展将在调用源函数中执行，但函数不被调用，且执行速度获得提高。

有下列两种方法可供指定内联扩展：

(1) 以扩展函数来指定

[格式]

#pragma inline（<函数名称>[，…]）

(2) 以选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 速度子选项 (Speed sub-options)：内联函数的最大节点 (Maximum nodes of inline function)**

命令行：*speed=inline[=(node)]*

当函数被调用时，一般上，JSR 或 BSR 指令将被输出。然而，当指定了内联扩展时，代码将直接在函数被调用的位置被扩展。因此，调用函数时的 JSR 或 BSR 指令和从函数返回时的 RTS 指令将不被输出，因而提高了执行速度。

实例

要执行函数 *func* 的内联扩展：

（C/C++ 程序）

#pragma 语句中的指定

```
#pragma inline func
int a,b;
void func()
{
    a+=b;
}
void main()
{
    a=0;
    func();
}
```

（汇编扩展代码）

未指定

```
_func:
        MOV.W       @_b:32,R0
        MOV.L       #_a,ER1
        MOV.W       @ER1,E0
        ADD.W       R0,E0
        MOV.W       E0,@ER1
        RTS
_main:
        SUB.W       R0,R0
        MOV.W       R0,@_a:32
        BRA         _func:88
```

已指定

```
_func:
        MOV.W       @_b:32,R0
        MOV.L       #_a,ER1
        MOV.W       @ER1,E0
        ADD.W       R0,E0
        MOV.W       E0,@ER1
        RTS
_main:
        SUB.W       E0,E0
        MOV.W       @_b:32,R0
        ADD.W       R0,E0
        MOV.W       E0,@_a:32
        RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	32	24	30	24	24
已指定	36	26	38	30	30

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	22	22	16
已指定	28	28	20

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	25	20	48	40	40
已指定	13	10	30	24	24

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	16	16	15
已指定	11	11	10

说明与备注

(1) #pragma inline 语句应在函数被定义前指定。

若优化未被指定，此规格将不可用，然而，#pragma 规格将可用。

内联扩展不对下列函数执行：

- 包含变量参数的函数
- 参考参数地址的函数
- 实际参数与虚设参数的类型不匹配的函数
- 调用被内联扩展的函数的函数
- 超出内联扩展的大小限制的函数

(2) 当将函数指定为静态时，函数仅在被调用方被扩展，从而增进了大小效率。在此情形下，被内联扩展的函数仅在相同的文件内被使用。

5.4.11 使用 8 位绝对地址区

大小	O	速度	O

描述

H8S 或 H8/300 系列提供 8 位绝对地址区。当被频繁存取的字节数据被分配到此区时，那些数据能以 8 位绝对地址格式被存取，与使用绝对地址格式的一般存取相较之下，增进了 ROM 效率、RAM 效率，及执行速度。

共有下列两种方法可指定 8 位绝对地址区：

(1) 以扩展函数来指定

[格式]

#pragma abs8（<变量或结构名称，数组名称>[，…]）

(2) 以选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 数据存取 (Data access) @aa:8**

命令行： *abs8*

当指定了 #pragma abs8 时，可指定要以 8 位绝对地址格式来存取的变量。

当已使用选项格式将此指定时，文件中的所有 1 字节数据被设定为以 8 位绝对地址格式来存取。

下面为每个 CPU/操作模式列出 8 位绝对地址区的范围：

CPU 类型	地址空间大小	8 位绝对地址区
H8SX 最大模式	32	H'FFFFFF00 至 H'FFFFFFFF
H8SX 高级模式 H8SX 中间模式 H8S/2600 高级模式 H8S/2000 高级模式 H8S/300H 高级模式	28	H'FFFFF00 至 H'FFFFFFF
	24	H'FFFF00 至 H'FFFFFF
	20	H'FFF00 至 H'FFFFF
H8SX 普通模式 H8S/2600 普通模式 H8S/2000 普通模式 H8S/300H 普通模式 H8/300	16	H'FF00 至 H'FFFF

实例

要存取在 8 位绝对地址区内分配的变量 *a*、*b* 和 *c*：

（C/C++ 程序）

使用 #pragma 语句的指定

```
#pragma abs8 (a,b,c)

const char a=1;
      char b=1;
      char c;
void func(void)
{
    c=b=a;
}
```

（汇编扩展代码的编译结果）

未指定

```
_func:
    MOV.B       #1,R0L
    MOV.B       R0L,@_b:32
    MOV.B       R0L,@_c:32
    RTS
    .SECTION    C,DATA,ALIGN=2
_a: .DATA.B     H'01
    .SECTION    D,DATA,ALIGN=2
_b: .DATA.B     H'01
    .SECTION    B,DATA,ALIGN=2
_c: .RES.B      1
```

已指定

```
_func:
    MOV.B       #1,R0L
    MOV.B       R0L,@_b:8
    MOV.B       R0L,@_c:8
    RTS
    .SECTION    $ABS8C,DATA,ALIGN=2
_a: .DATA.B     H'01
    .SECTION    $ABS8D,DATA,ALIGN=2
_b: .DATA.B     H'01
    .SECTION    $ABS8B,DATA,ALIGN=2
_c: .RES.B      1
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	18	14	18	14	14
已指定	10	10	10	10	10

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	16	16	12
已指定	10	10	10

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	14	11	28	22	22
已指定	10	9	20	18	18

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	10	10	10
已指定	9	9	9

说明与备注

#pragma abs8 无法为之前已声明过的数据指定。

此选项只对 1 字节外部变量有效。

在连接上，以 $ABS8 起始的段被分配到 8 位绝对地址区。

5.4.12 使用 16 位绝对地址区

大小	O	速度	O

描述

H8S 或 H8/300 系列提供 16 位绝对地址区。当被频繁存取的字节数据被分配到此区时，那些数据将能以 16 位绝对地址的格式来存取，与使用绝对地址格式的一般存取相比之下，增进了 ROM 效率、RAM 效率，和执行速度。

共有下列两种方法可指定 16 位绝对地址区：

(1) 以扩展函数来指定

[格式]

#pragma abs16（<变量或结构名称，数组名称>[，…]）

(2) 以扩展选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 数据存取 (Data access) @aa:16**

命令行：*abs16*

当指定了 #pragma abs16 时，可指定要以 16 位绝对地址格式来存取的变量。当已使用选项格式将此指定时，文件中的所有数据将被设定为以 16 位绝对地址的格式来存取。

下面为每个 CPU/操作模式列出 16 位绝对地址区的范围：

CPU 类型	地址空间大小	16 位绝对地址区
H8SX 最大模式	32	0 至 H'7FFF, H'FFFF0000 至 H'FFFFFFFF
H8SX 高级模式 H8SX 中间模式	28	0 至 H'7FFF, H'FFF0000 至 H'FFFFFFF
H8S/2600 高级模式	24	0 至 H'7FFF, H'FF0000 至 H'FFFFFF
H8S/2000 高级模式 H8S/300H 高级模式	20	0 至 H'7FFF, H'F0000 至 H'FFFFF

实例

要存取在 16 位绝对地址区内分配的变量 *a*、*b* 和 *c*：

（C/C++ 程序）

使用 #pragma 语句的指定

```
#pragma abs16 (a,b,c)
const int a=1;
      int b=1;
      int c;

void func(void)
{
    c=b=a;
}
```

（汇编扩展代码的编译结果）

未指定

```
_main:
    MOV.W     #1,R0
    MOV.W     R0,@_b:32
    MOV.W     R0,@_c:32
    RTS
    .SECTION  C,DATA,ALIGN=2
_a:
    .DATA.W   H'0001
    .SECTION  D,DATA,ALIGN=2
_b:
    .DATA.W   H'0001
    .SECTION  B,DATA,ALIGN=2
_c:
    .RES.W    1
```

已指定

```
_main:
    MOV.W     #1,R0
    MOV.W     R0,@_b:16
    MOV.W     R0,@_c:16
    RTS
    .SECTION  $ABS16C,DATA,ALIGN=2
_a:
    .DATA.W   H'0001
    .SECTION  $ABS16D,DATA,ALIGN=2
_b:
    .DATA.W   H'0001
    .SECTION  $ABS16B,DATA,ALIGN=2
_c:
    .RES.W    1
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	22	18	22	18	18
已指定	18	18	18	18	18

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	18	18	14
已指定	14	14	14

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	15	12	30	24	24
已指定	13	12	26	24	24

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	10	10	9
已指定	9	9	9

说明与备注

此规格只对 CPU/操作模式 H8SXX、H8SXA、H8SXM、2600a、2000a 或 300ha 有效。

#pragma abs16 无法对之前已被声明过的数据指定。

此选项只对外部变量有效。

输出数据的段名称可通过 #pragma 语句来修改。

在连接上，以 $ABS16 起始的段被分配到 16 位绝对地址区。

5.4.13 使用间接存储格式

大小	O	速度	X

描述

当频繁使用的函数被以间接存储格式来存取时，ROM 效率将被提高。若函数地址在连接时被存储在间接存储区内，则当函数被调用时，它是以间接存储的格式被调用。在此情形下，执行速度被降低了，但程序大小也因函数能以短指令来调用而缩减。

有下列两种方法可供指定间接存储格式：

(1) 以扩展函数来指定

[格式]

#pragma indirect（<函数名称>[（vect=<向量号>）][，…])
__indirect[（vect=<向量号>）] <类型说明符> <函数名称>
<类型说明符> __indirect[（vect=<向量号>）] <函数名称>

(2) 以选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 函数调用 (Function call)：@@aa:8**

命令行：*indirect=Normal*

间接存储器地址区是从 00 到 FF 的范围。

将包含指定到包含文件 indirect.h，所有要使用的运行时例程将以间接存储的格式被调用。

另外，每个运行时例程能以间接存储的格式被个别调用。

实例

要以间接存储格式来调用函数 *func*：

（C/C++ 程序）

在 #pragma 语句中指定

```
#pragma indirect func
extern void func(int, int);
extern int a,b,c;
int d;
void main(void)
{
    b=0;
    func(a,b);
    func(b,c);
    func(c,a);
    d=c;
}
```

← **指定 #pragma indirect**

（汇编扩展代码的编译结果）

未指定

```
_main:
    PUSH.L      ER6
    SUB.W       R0,R0
    MOV.W       R0,@_b:32
    MOV.W       R0,E0
    MOV.W       @_a:32,R0
    JSR         @_func:24
    MOV.L       #_c,ER6
    MOV.W       @ER6,E0
    MOV.W       @_b:32,R0
    JSR         @_func:24
    MOV.W       @_a:32,E0
    MOV.W       @ER6,R0
    JSR         @_func:24
    MOV.W       @ER6,R6
    MOV.W       R6,@_d:32
    POP.L       ER6
    RTS
```

已指定

```
_main:
    PUSH.L      ER6
    SUB.W       R0,R0
    MOV.W       R0,@_b:32
    MOV.W       R0,E0
    MOV.W       @_a:32,R0
    JSR         @@$func:8
    MOV.L       #_c,ER6
    MOV.W       @ER6,E0
    MOV.W       @_b:32,R0
    JSR         @@$func:8
    MOV.W       @_a:32,E0
    MOV.W       @ER6,R0
    JSR         @@$func:8
    MOV.W       @ER6,R6
    MOV.W       R6,@_d:32
    POP.L       ER6
    RTS
```

目标大小的比较 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	58	52	70	54	54
已指定	66	48	68	50	50

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	70	64	50
已指定	62	62	46

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	75	58	152	118	118
已指定	78	58	158	118	118

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	49	46	43
已指定	50	50	44

说明与备注

$INDIRECT 段应被分配到可在连接时以间接存储格式来存取的存储区 00 至 FF。

间接存储区被输出到“$INDIRECT”段。段名称可使用 #pragma indirect section 语句来进行修改。

5.4.14 使用扩展的间接存储格式

大小	O	速度	X

描述

当频繁使用的函数被以间接存储格式来存取时，ROM 效率将被提高。当 CPU 是 H8SX 时，扩展的存储器间接寻址模式可被附加使用。这也将提高 ROM 的效率。

有下列两种方法可供指定扩展的间接存储格式：

(1) 以扩展函数来指定

[格式]

__indirect_ex[（vect=<向量号>）] <类型说明符> <函数名称>
<类型说明符> __indirect_ex[（vect=<向量号>）] <函数名称>

(2) 以选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 函数调用 (Function call)：@@vec:7**

命令行：*indirect=Extended*

[扩展的间接存储器寻址的地址范围]

H8SX 普通模式：从 0x0100 至 0x01FF 的区域
H8SX 其他模式：从 0x0200 至 0x03FF 的区域

实例

要以扩展的间接存储格式来调用函数 *func*：

当向量号不被 **vect** 所指定时，函数地址将在段“$EXINDIRECT”内存储为地址表。

当指定了向量号时，段“$VECT***”作为地址表被存储。在连接时，优化连接编辑程序自动将段分配到对应的地址。

（C/C++ 程序）

在关键字中指定

```
__indirect_ex void func(int, int);
extern int a,b,c;
int d;
void main(void)
{
    b=0;
    func(a,b);
    func(b,c);
    func(c,a);
    d=c;
}
```

← **指定 _indirect_ex**

（汇编扩展代码的编译结果）

未指定

```
_main:
    STM.L       (ER2-ER3),@-SP
    SUB.W       E0,E0
    MOV.W       E0,@_b:32
    MOV.L       #_func,ER2
    MOV.W       @_a:32,R0
    JSR         @ER2
    MOV.W       @_b:32,R0
    MOV.L       #_c,ER3
    MOV.W       @ER3,E0
    JSR         @ER2
    MOV.W       @_a:32,E0
    MOV.W       @ER3,R0
    JSR         @ER2
    MOV.W       @ER3,@_d:32
    RTS/L       (ER2-ER3)
```

已指定

```
_main:
    PUSH.L      ER2
    SUB.W       E0,E0
    MOV.W       E0,@_b:32
    MOV.W       @_a:32,R0
    JSR         @@$$func:7
    MOV.W       @_b:32,R0
    MOV.L       #_c,ER2
    MOV.W       @ER2,E0
    JSR         @@$$func:7
    MOV.W       @_a:32,E0
    MOV.W       @ER2,R0
    JSR         @@$$func:7
    MOV.W       @ER2,@_d:32
    RTS/L       ER2
```

目标大小的比较 [字节]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	82	76	58
已指定	74	74	54

执行速度的比较 [周期]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	61	58	55
已指定	62	62	56

说明与备注

当向量号未被指定时，$EXINDIRECT 段应被分配到在连接时能以扩展的间接存储格式来存取的存储区。

扩展的间接存储区被输出到“$EXINDIRECT”段。段名称可使用 #pragma indirect section 语句来进行修改。

5.4.15 指定 2 字节指针

大小	O	速度	O

描述

当被频繁使用的变量被分配到 16 位绝对地址区时，大小效率和执行速度都获得提高。由 ABS16 选项或 #pragma abs16 选项所指定的 ABS16 选项，将数据分配到 16 位绝对地址区。

此 2 字节指针选项假定指针对于数据的大小为二字节。

有下列两种方法可供指定此函数：

(1) 以扩展函数来指定

[格式]

<类型说明符> __ptr16 * <变量>

(2) 以选项来指定

对话框菜单： **C/C++ 标签类别：[优化 (Optimize)] 2 字节指针 (2byte pointer)**

命令行：*ptr16*

若此选项未被指定，表示数据的指针大小是四字节。若指定了此选项，且数据段在 16 位绝对地址区中被明确的分配，表示数据的指针大小将被设定为二字节。

实例

要通过二字节指针参考变量 *b*：

（C/C++ 程序）

在关键字中指定

```
__abs16 int a;
int __ptr16 *b;
int c;
void func(void);
void func(void)
{
    b = (int __ptr16 *)&a;
    *b = 10;
    c = *b;
}
```

← 指定 __abs16

← 指定 __ptr16

（汇编扩展代码的编译结果）

未指定

```
_func
            MOV.L       #_a:32,@_b:16
            MOV.L       @_b:16,ER0
            MOV.W       #10:8,@ER0
            MOV.L       @_b:16,ER0
            MOV.W       @ER0,@_c:16
            RTS
```

已指定

```
_func:
            MOV.L       #_a,ER1
            MOV.W       R1,@_b:16
            MOV.W       R1,R0
            EXTS.L      ER0
            MOV.W       #10:8,@ER0
            MOV.W       @_b:16,R0
            EXTS.L      ER0
            MOV.W       @ER0,@_c:16
            RTS
```

目标大小的比较 [字节]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	36	36	24
已指定	34	34	24

执行速度的比较 [周期]

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	23	19	16
已指定	22	18	16

说明与备注

此关键字只对 H8SX 高级模式和 H8SX 最大模式有效。

此关键字必须在一个一元运算符“*”之前指定。

5.4.16 边界对齐值和边界对齐

大小	O	速度	O

描述

对齐 (align) 选项可再定位变量，以缩减边界对齐所形成的空间。

align=4 选项将数据段分隔为一个 4 字节的边界对齐段，一个 2 字节的边界对齐段和一个 1 字节的边界对齐段。
（align=4 仅在 H8SX 有效）

于是大小效率和执行速度获得提高。*

指定方法

对话框菜单： **C/C++ 标签类别：[目标 (Object)] 以对齐方式分组 (Group by alignment)**

命令行： *ALign [=4] (Default is ALign)*
NOALign

实例

有关数据分配顺序的说明如下：它们依据选项指定而异。

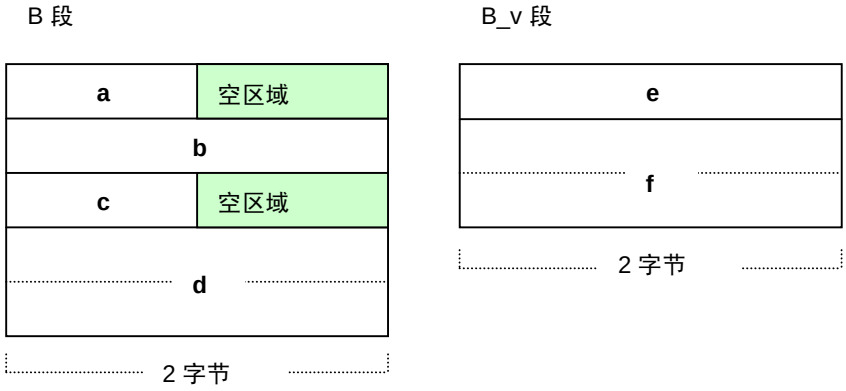
（C/C++ 程序）

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

void func(void)
{
    a = 127;
    b = 0x7fff;
    c = 30;
    d = 0x7fffffff;
    e = 0x1000;
    f = 0x1ffff;
}
```

(1) 指定了 **noalign**

数据以声明到段 B 和段 B_v 的顺序被分配。

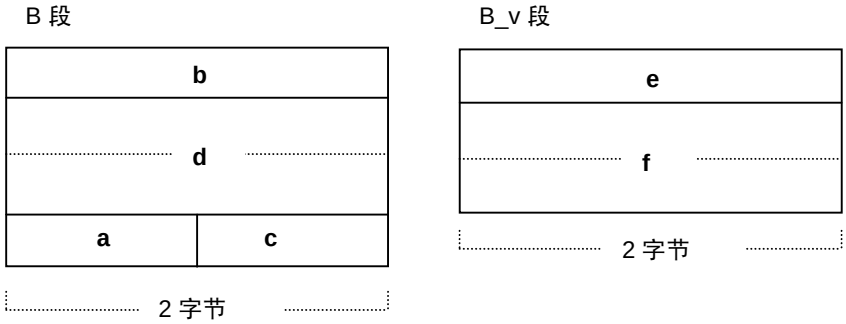
如下所示，应用了 2 字节对齐的数据始终在偶数地址上分配，因此在分配奇数大小的数据时生成了不被使用的空区域。



(2) 指定了 **align**

为了最小化空区域，2 字节对齐的数据（短型、长型、浮点型）先于 1 字节对齐的数据被分配到段 B 和段 B_v。

故没有空区域被生成，如下所示。



(3) 指定了 **align**=4

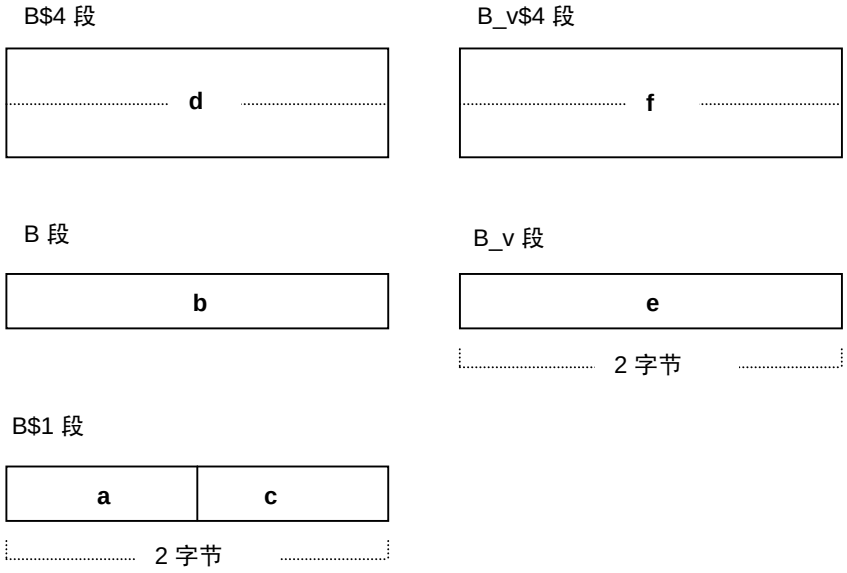
数据被分类为下列 3 组：

(a) 大小是 4 的倍数的数据
(b) 大小是奇数的数据
(c) 其他（大小是偶数，但不是 4 的倍数的数据）

且段名称分别如下所示被更改

(a) 在原段名称之后附加了“$4”
(b) 在原段名称之后附加了“$1”
(c) 段名称不被更改

当 CPU 类型是 H8SX 时，对齐在 4 字节边界地址上的 4 字节数据被存取的速度提高了。*



align=4 的段地址分配

要在指定了 **align=4** 时，在特定的地址定位 1 字节或 4 字节数据段，每个段需要使用优化连接编辑程序的**起始 (start)** 选项来明确指定。

在 HEW 中，对话框菜单：**连接/程序库 (Link/Library) 标签类别：[段 (Section)]** 被使用。

实例

以 $4 将段分配到 4 的倍数的地址。

以 $1 分配段，以便最小化空区域。

目标大小的比较 [字节]（RAM 大小）

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	16	16	16	16	24
已指定	14	14	14	14	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	16	16	16
已指定	14	14	14

执行速度的比较 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
未指定	45	38	90	76	156
已指定	45	38	90	76	156

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
未指定	28	28	24
已指定	27	26	23

说明与备注

此选项 **align=4** 仅在 H8SX 有效。

注意： * 执行速度只在 H8SX 中被提高。
4 字节数据一般上通过字指令的两次存取而被存取。当 4 字节数据以 **align=4** 在 4 字节边界上被对齐，且总线宽度是 32 位时，H8SX 可通过一次存取来存取 4 字节数据。
在 16 位总线宽度中，数据通过字指令的两次存取而被存取。因此，执行速度没有被提高。

5.4.17 模块间优化项目的说明

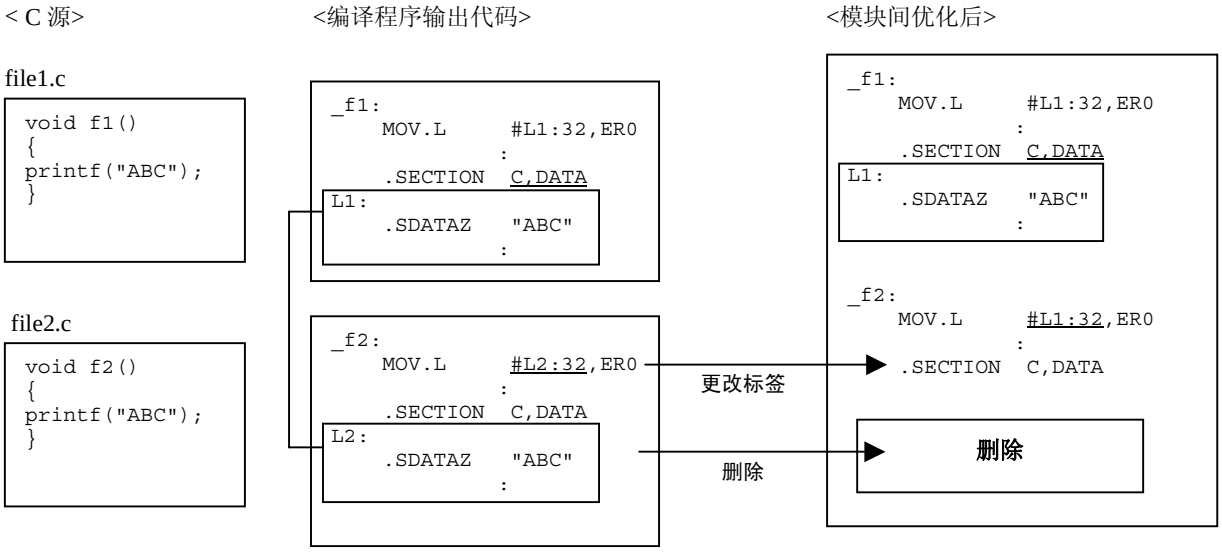
模块间优化器支持下列优化函数：

描述	**对话框菜单**	**子命令**	**参考的节**
统一常数/字符串	**统一字符串 (Unify strings)**	*String_Unify*	5.4.17(1)
删除未参考的变量/函数	**删除死码 (Eliminate dead code)**	*Symbol_delete*	5.4.17(2)
优化对变量的存取	**使用短寻址 (Use short addressing)**	*Variable_access*	5.4.17(3)
优化对函数的存取	**使用间接调用/跳转 (Use indirect call/jump)**	*Funcation_call*	5.4.17(4)
优化寄存器保存/恢复代码	**再分配寄存器 (Reallocate registers)**	*Register*	5.4.17(5)
统一指令代码	**删除相同的代码 (Eliminate same code)**	*Same_code*	5.4.17(6)
优化转移指令	**优化转移 (Optimize branches)**	*Branch*	5.4.17(7)

下面描述各项优化函数。

(1) 统一常数/字符串

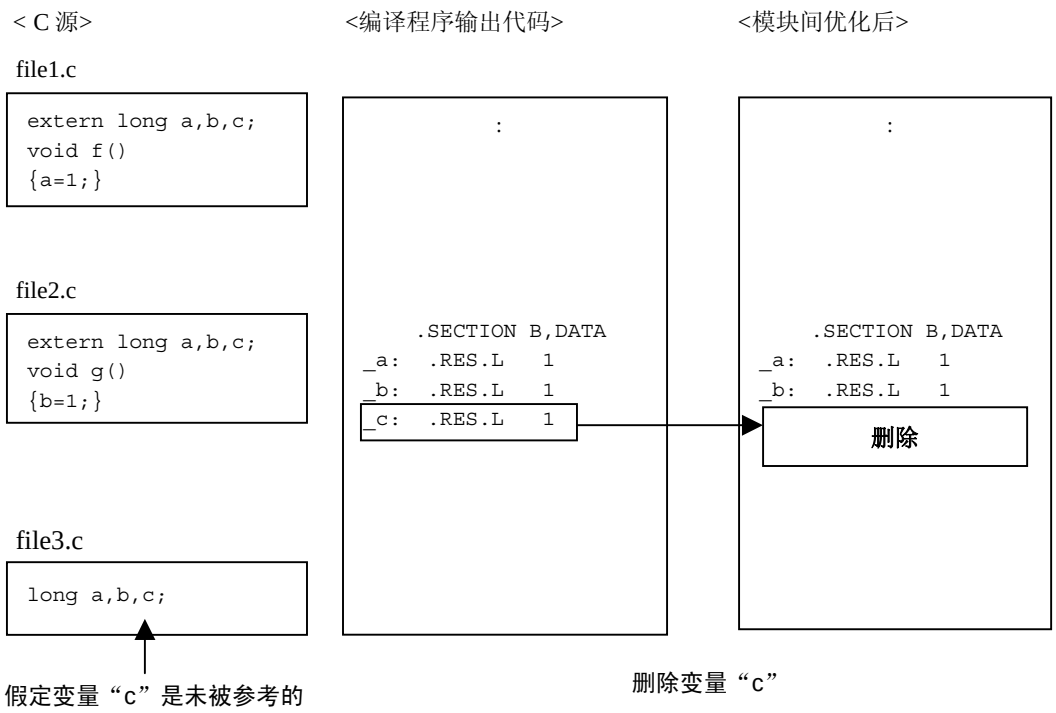
相同值的常数和具有常数属性的相同字符串被跨模块统一。下面显示一个实例：



(2) 删除未被参考的变量/函数

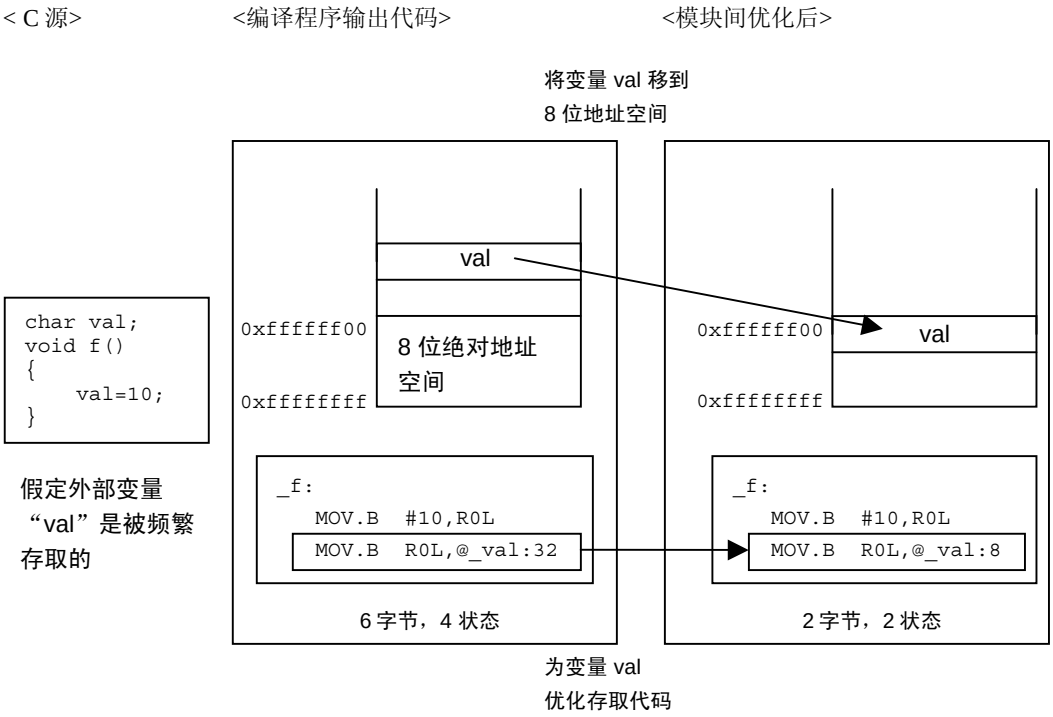
未被参考的变量/函数将通过此规格被删除。当指定这项优化时，请确认指定项目函数。没有项目函数，这项优化将不被执行。

下面显示一个实例：



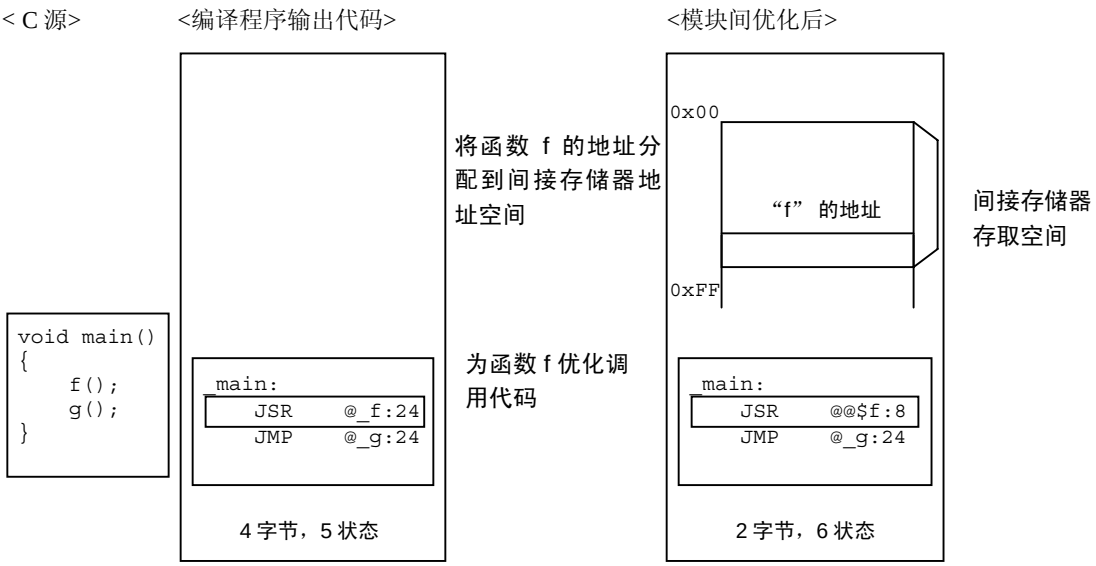
(3) 优化对变量的存取

若在 8 位或 16 位绝对寻址模式中可存取的区域有空间，被频繁存取的变量将被分配，变量存取代码的优化将被分配，同时变量的存取代码通过此规格获得优化。



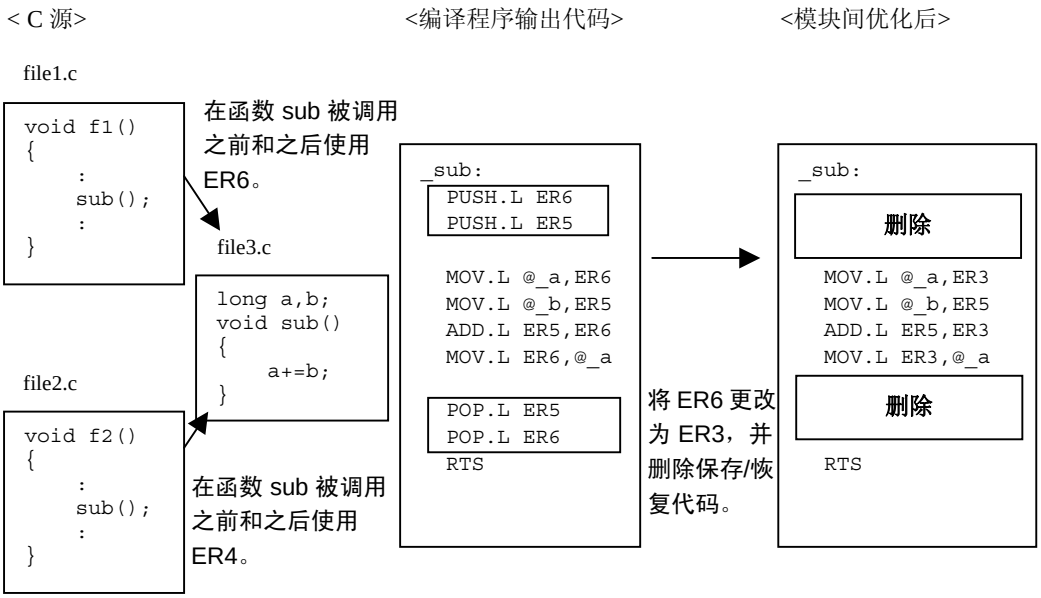
(4) 优化对函数的存取

若从 0 至 0xFF 的存储范围有空间，将对被频繁存取的函数执行分配地址的优化。



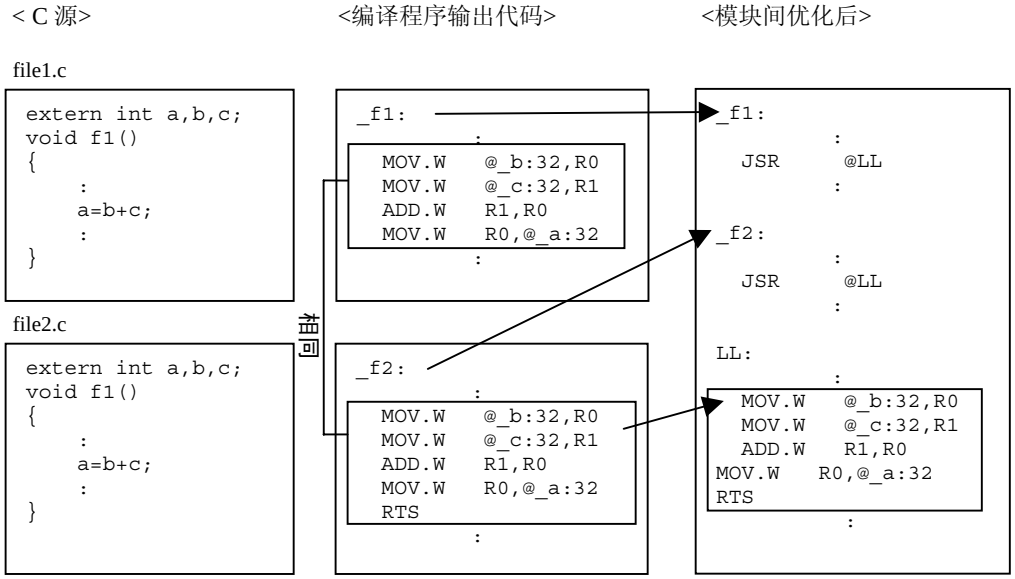
(5) 再定位寄存器

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。



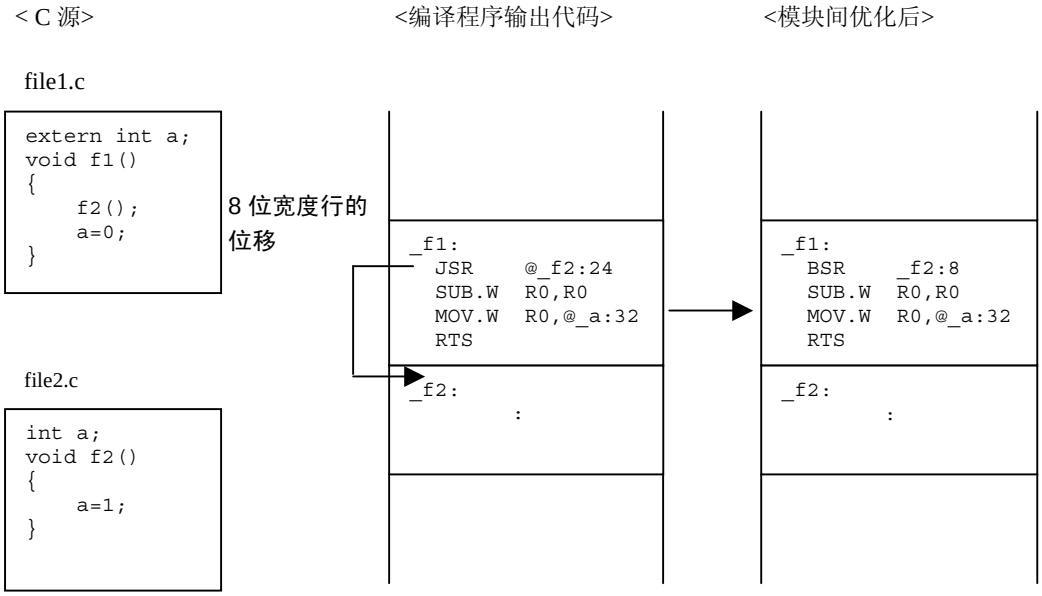
(6) 统一公用代码

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。



(7) 优化转移指令

基于程序分配信息，转移指令的大小获得优化。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。



5.4.18 禁止模块间优化

模块间优化器支持禁止特定优化函数的函数。

当此函数对必须禁止特定优化项目的程序使用时，详细的规格将会提供，然后优化的禁止将被简洁的执行。

模块间优化器支持下列函数以禁止优化项目：

禁止优化的项目	要指定的单元	对话框菜单	子命令
禁止删除未被参考的符号	符号名称	**死码的删除** **(Elimination of dead code)**	symbol_forbid
禁止删除相同的代码	函数名称	**相同代码的删除** **(Elimination of same code)**	samecode_forbid
禁止短绝对地址区的分配	变量名称	**将短寻址使用到** **(Use of short addressing to)**	variable_forbid
禁止间接地址调用	函数名称	**将间接调用/跳转使用到** **(Use of indirect call/jump to)**	function_forbid
禁止寄存器的再分配	地址 [+大小]	**存储器的分配** **(Memory allocation)**	absolute_forbid

H8S, H8/300 系列 C/C++编译程序应用笔记

有效的编程技术

第 6 节　有效的编程技术

除了 H8S 及 H8/300 C/C++ 编译程序所执行的优化之外，程序的性能可通过有效的编程技术进一步提高。

本节描述用户可能会考虑用于创建有效编程的方法。

(i) 缩减程序大小的规则

要缩减程序大小，必须普遍使用相似的处理任务，并且必须复查复杂的函数以取得可能的改进。

(ii) 增进执行速度的规则

执行速度是经常执行的语句和复杂的语句的一个主要函数。用户必须复查这些语句的处理，以便专注于重点来增进程序。

由于编译程序的优化功能，实际的执行速度可能不同于优先基准所决定的性能级别。 性能的增进必须通过使用各种技术，及验证使用编译程序的实际性能来达成。

在本节中，汇编语言扩展代码通过假定所使用的 CPU 类型是运行于高级模式下的 H8S/2600 系列而被提供。本节所提供的汇编语言扩展代码，可能会因在编译程序设计中的进一步增进而有所更改。

其性能可通过以下条件衡量。

[用于进行衡量的交叉工具]

H8S，H8/300 C/C++ 程序库生成程序（版本 2.01.00.001）

H8S，H8/300 C/C++ 编译程序（版本 6.01.00.009）

H8S，H8/300 汇编程序（版本 6.01.01.000）

优化连接编辑程序（版本 9.00.02.000）

[选项指定]

在每节都没有描述选项指定的方法时，默认选项将被使用。

[衡量条件]

条件	H8/300, H8/300H	H8S/2600,H8S/2000	H8SX
总线宽度	16	16	32
存取到存储器的状态	2	1	1
取指令大小	-	-	32

以下是有效编程技术的列表：

编号	类型	项目	大小	速度	参考的节
1	类型	使用 1 字节数据类型（字符/无符号字符）	O	O	6.1.1
2	声明	使用无符号变量	O	O	6.1.2
3		禁止冗余类型转换	O	O	6.1.3
4		使用 const 限定符	O	O	6.1.4
5		使用一致的变量大小	O	O	6.1.5
6		将 in-file 函数指定为静态 (statics)	O	–	6.1.6
7	操作	统一公用表达式	O	O	6.2.1
8		增进条件决定	O	O	6.2.2
9		使用替换值的条件决定	O	Δ	6.2.3
10		使用合适的运算法	O	O	6.2.4
11		使用公式	O	O	6.2.5
12		使用本地变量	O	O	6.2.6
13		将“f”分配给浮点类型常数	O	O	6.2.7
14		指定移位操作中的常数	O	O	6.2.8
15		使用移位操作	O	O	6.2.9
16		统一连续 ADD 指令	O	O	6.2.10
17	循环处理	选择循环计数器	O	O	6.3.1
18		选择重复控制语句	O	O	6.3.2
19		将不变量表达式从循环的内部移到外部	O	O	6.3.3
20		合并循环条件	O	O	6.3.4
21	指针	使用指针变量	O	O	6.4.1
22	数据结构	确保数据兼容性	O	–	6.5.1
23		数据初始化的技术	O	O	6.5.2
24		统一数组元素的初始化	O	O	6.5.3
25		将参数传递为结构地址	O	O	6.5.4
26		将结构分配给寄存器	O	O	6.5.5
27	函数	增进函数被定义的程序位置	–	O	6.6.1
28		宏调用	O	O	6.6.2
29		声明原型	–	–	6.6.3
30		尾递归的优化	O	O	6.6.4
31		增进参数传递的方式	O	O	6.6.5
32	转移指令	将切换 (switch) 语句重写为表	O	O	6.7.1
33		编码 case 语句跳转至相同标签的程序	O	O	6.7.2
34		转移到直接在指定语句下编码的函数	O	O	6.7.3

图例：

O: 更高效率 Δ: 无更改 X: 更低效率 –: 不适用

6.1 类型声明

6.1.1 使用字节数据类型（字符/无符号字符）

大小	O	速度	O	堆栈大小	Δ

重点

为增进 ROM 的效率和执行速度，可用 1 字节大小来代表的数据，应被声明为字符/无符号字符 (char/unsigned char) 类型。

描述

H8S 及 H8/300 系列 CPU 提供可在字节大小的数据上有效操作的指令集。

因此，可在使用数据前通过将任何字节大小的数据声明为字符/无符号字符类型，来增进 ROM 的效率和执行速度。

实例

决定变量 a 和常数 0x80 之间的逻辑产品，并将结果存储在变量 a 中。

（优化前的 C 语言程序）

```
int a;
void func(void)
{
    a&=0x80;
}
```

（优化后的 C 语言程序）

```
char a;
void func(void)
{
    a&=0x80;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
      MOV.L       #_a,ER0
      MOV.W       @ER0,R1
      AND.W       #128,R1
      MOV.W       R1,@ER0
      RTS
_a:
     .RES.W      1
```

（扩展入汇编语言代码：优化后）

```
_func:
      MOV.L       #_a,ER0
      MOV.B       @ER0,R1L
      AND.B       #-128,R1L
      MOV.B       R1L,@ER0
      RTS
_a:
      .RES.B      1
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	18	14	16	14	14
之后	16	12	14	12	12

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	12	12	10
之后	10	10	8

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	15	12	28	24	24
之后	14	11	26	22	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	11	11	10
之后	9	9	9

6.1.2 使用无符号变量

大小	O	速度	O	堆栈大小	Δ

重点

为增进目标的效率和执行速度，任何具有正数值的变量应被声明为无符号。

描述

在将指定数据项目扩展入大型数据类型时，若数据项目是带符号的数据，编译程序将执行符号扩展；若它是无符号数据，编译程序将执行零扩展。 由于 H8/300 系列 CPU 并没有数据扩展指令，CPU 需要决定符号的目标以处理带符号数据。 因此，通过将任何始终具有正数值的变量列为无符号变量，可同时增进 ROM 的效率和执行速度。

请注意，由于 H8S 及 H8/300H CPU 有提供数据扩展指令，将正值变量声明为无符号变量将不会对这些 CPU 的性能产生效果。

实例

将变量扩展入 int 类型；将结果设定到变量 b。

以下是在 300 CPU 上编译程序所得到的结果：

（优化前的 C 语言程序）

```
char a;
int  b;
void func(void)
{
    b=a;
}
```

（优化后的 C 语言程序）

```
unsigned char a;
int  b;
void func(void)
{
    b=a;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
      MOV.B       @_a:16,R0L
      BLD.B       #7,R0L
      SUBX.B      R0H,R0H
      MOV.W       R0,@_b:16
      RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
      MOV.B       @_a:16,R0L
      SUB.B       R0H,R0H
      MOV.W       R0,@_b:16
      RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	16	12	16	12	14
之后	16	12	16	12	12

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	16	16	12
之后	16	16	12

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	14	11	28	22	24
之后	14	11	28	22	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	11	11	10
之后	11	11	10

6.1.3 禁止冗余类型转换

大小	O	速度	O	堆栈大小	Δ

重点

通过确保在相同大小的数据项目之间执行操作，可增进 ROM 的效率和执行速度。

描述

在不同大小的数据项目之间所执行的操作，将生成冗余的符号扩展指令和零扩展指令，导致小型数据类型转换成大型数据类型。通过确保数据项目的大小相同，可增进 ROM 的效率和执行速度。

实例

相加变量 a 和 b；将结果设定到变量 c。

（优化前的 C 语言程序）

```
unsigned char a;
         int  b,c;
void func(void)
{
    c=a+b;
}
```

（优化后的 C 语言程序）

```
int a,b,c;

void func(void)
{
    c=a+b;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        MOV.B       @_a:32,R0L
        EXTU.W      R0
        MOV.W       @_b:32,E0
        ADD.W       E0,R0
        MOV.W       R0,@_c:32
        RTS
_a:
        .RES.B      1
_b:
        .RES.W      1
_c:
        .RES.W      1
```

（扩展入汇编语言代码：优化后）

```
_func:
        MOV.W       @_a:32,R0
        MOV.W       @_b:32,E0
        ADD.W       E0,R0
        MOV.W       R0,@_c:32
        RTS

_a:
        .RES.W      1
_b:
        .RES.W      1
_c:
        .RES.W      1
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	24	18	24	18	18
之后	22	16	22	16	16

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	24	24	18
之后	22	22	16

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	19	15	38	30	30
之后	18	14	36	28	28

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	13	13	12
之后	13	13	12

6.1.4 使用 const 限定符

大小	O	速度	O	堆栈大小	-

重点

保持值不变的初始化数据应被列为常数以节省 RAM 区域。

描述

已初始化的数据项目经常可能在值上变更。 这些数据项目于连接期间在 ROM 上分配，并在程序执行开始时被复制到 RAM，导致它们同时在 ROM 和 RAM 区域内被分配。在程序执行的过程中保持值不变的数据项目可被列为常数，以使它们只在 ROM 区域内被分配。

实例

分配初始化数据的 5 字节。

（优化前的 C 语言程序）

```
unsigned char a[5]=
    {1, 2, 3, 4, 5};
```

（优化后的 C 语言程序）

```
const unsigned char a[5]=
          {1, 2, 3, 4, 5};
```

（扩展入汇编语言代码；优化前）

```
   .SECTION  D,DATA,ALIGN=2
_a:
   .DATA.B   H'01,H'02,H'03,H'04,H'05
```

（扩展入汇编语言代码；优化后）

```
  .SECTION  C,DATA,ALIGN=2
_a:
   .DATA.B   H'01,H'02,H'03,H'04,H'05
```

说明与备注

在优化前，除了 ROM 以外，程序还需要在 RAM 为分配目标大小表中所列出的数据区域大小。

若是字符串数据，其输出目标可在选项中指定。

[指定方法]

对话框菜单： **C/C++ 标签类别：[目标 (Object)] 将字符串数据存储在 (Store string data in)：常数段 (Const section)｜数据段 (Data section)**

命令选项： *string=const | data*

默认设定为将数据输出到常数段。

6.1.5 使用一致的变量大小

大小	O	速度	O	堆栈大小	Δ

重点

在循环语句中作比较时，可使用统一的变量大小来消除对扩展代码的需要，并缩减所产生的代码大小。

描述

在将一个数据项目与另一个作比较时，编译程序将首先确保这些数据项目的大小相同。用户在编码程序前，先确保数据项目的大小相同，可消除对扩展代码的需要并提高速度。

实例

通过循环调用函数 *func1* 。

（优化前的 C 语言程序）

```
extern char tb[5];
void sub(void)
{
    int i;
    for (i=0; i<2L; i++)
       func1(tb[i]);
}
```

（优化后的 C 语言程序）

```
extern char tb[5];
void sub(void)
{
    unsigned int i;
    for (i=0; i<2L; i++)
        func1(tb[i]);
}
```

（扩展入汇编语言代码：优化后）

```
_sub:
        PUSH.L      ER6
        SUB.W       R6,R6
L6:
        EXTS.L      ER6
        MOV.B       @(_tb:32,ER6),R0L
        JSR         @_func1:24
        INC.W       #1,R6
        EXTS.L      ER6
        CMP.L       #2,ER6
        BLT         L6:8
        POP.L       ER6
        RTS
```

（扩展入汇编语言代码：优化前）

```
_sub:
        PUSH.L      ER6
        SUB.W       R6,R6
L6:
        EXTU.L      ER6
        MOV.B       @(_tb:32,ER6),R0L
        JSR         @_func1:24
        INC.W       #1,R6

        CMP.W       #2,R6
        BLO         L6:8
        POP.L       ER6
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	34	30	40	34	50
之后	34	28	36	26	28

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	32	30	26
之后	32	30	24

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	59	50	124	102	378
之后	59	49	116	86	90

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	43	43	41
之后	43	43	37

6.1.6 将 in-file 函数指定为静态函数

大小	O	速度	O	堆栈大小	O

重点

只在一个文件中使用的函数应被指定为*静态 (static)*。

描述

指定为*静态*的函数若不被外部函数所调用，则将被删除。当为内联扩展指定时，这些函数也将被删除，以增进大小效率。

实例

为内联扩展指定函数。

从函数 *main* 调用函数 *func*。

（优化前的 C 语言程序）

```
#pragma inline func
int a,b;
void func()
{
    a+=10;
}
void main()
{
    a=1;
    func();
    b=a;
}
```

（优化后的 C 语言程序）

```
#pragma inline func
int a,b;
static void func()
{
    a+=10;
}
void main()
{
    a=1;
    func();
    b=a;
}
```

（扩展入汇编语言代码；优化前）

```
_func:
    MOV.L     #_a,ER0
    MOV.W     @ER0,R1
    ADD.W     #10,R1
    MOV.W     R1,@ER0
    RTS
_main:
    MOV.W     #11,R0
    MOV.W     R0,@_a:32
    MOV.W     R0,@_b:32
    RTS
```

（扩展入汇编语言代码；优化后）

```
_main:
    MOV.W     #11,R0
    MOV.W     R0,@_a:32
    MOV.W     R0,@_b:32
    RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	36	28	34	28	28
之后	18	14	18	14	14

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	26	26	20
之后	14	14	10

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	15	12	30	24	24
之后	15	12	30	24	24

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	10	10	8
之后	10	10	8

6.2 操作

6.2.1 统一公用表达式

大小	O	速度	O	堆栈大小	O

重点

通过在多个算术表达式中统一公用组件，可增进 ROM 的效率和执行速度。

描述

如果公用表达式出现在多个表达式中，第一个表达式上的操作结果应被使用在第二个和随后的表达式中，以缩减执行算术操作的次数。 这可同时增进 ROM 的效率和执行速度。

如果公用表达式在本地变量中出现三次或以上，编译程序将执行优化。

实例

相加变量 *x*、*y* 和 *z*；将结果存储在变量 *a* 中。同样地，相加变量 *x*、*y* 和 *w*；将结果存储在变量 *b* 中。

（优化前的 C 语言程序）

```
unsigned char a,b,w,x,y,z;
void func(void)
{
    a=x+y+z;
    b=x+y+w;
}
```

（优化后的 C 语言程序）

```
unsigned char a,b,w,x,y,z;
void func(void)
{
    unsigned char tmp;
    tmp=x+y;
    a=tmp+z;
    b=tmp+w;
}
```

（扩展入汇编语言代码；优化前）

```
_func:
      MOV.B       @_x:32,R1L
      MOV.B       @_y:32,R0H
      ADD.B       R1L,R0H
      MOV.B       R0H,R1H
      MOV.B       @_z:32,R0L
      ADD.B       R0L,R0H
      MOV.B       R0H,@_a:32
      MOV.B       @_w:32,R0L
      ADD.B       R0L,R1H
      MOV.B       R1H,@_b:32
      RTS
```

（扩展入汇编语言代码；优化后）

```
_func:
      MOV.B       @_x:32,R0H
      MOV.B       @_y:32,R0L
      ADD.B       R0L,R0H
      MOV.B       @_z:32,R0L
      ADD.B       R0H,R0L
      MOV.B       R0L,@_a:32
      MOV.B       @_w:32,R0L
      ADD.B       R0L,R0H
      MOV.B       R0H,@_b:32
      RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	44	32	46	34	34
之后	46	34	44	32	32

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	44	44	32
之后	46	46	34

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	32	25	66	52	52
之后	33	26	64	50	50

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	21	21	19
之后	22	22	19

说明与备注

尽管编译程序通过为本地变量统一公用表达式来执行优化，但它不可为外部变量执行优化。

6.2.2 增进条件决定

大小	O	速度	O	堆栈大小	Δ

重点

通过评估一次操作中的相似条件表达式来增进 ROM 的效率。

描述

相似的条件表达式必须在操作中被评估，以缩减条件决定和条件表达式被评估的次数。这可同时增进 ROM 的效率和执行速度。

实例

决定变量 *a* 和 *b* 的逻辑产品；将结果返回到调用源函数。

（优化前的 C 语言程序）

```
unsigned char a,b;
unsigned char func(void)
{
    if (!a)    return(0);
    if (a&&!b) return(0);
    return(1);
}
```

（优化后的 C 语言程序）

```
unsigned char a,b;
unsigned char func(void)
{
    if (a&&b) return(1);
    else      return(0);
}
```

（扩展入汇编语言代码；优化前）

```
_func:
      MOV.B       @_a:32,R0H
      BNE         L6:8
      SUB.B       R0L,R0L
      RTS
L6:   MOV.B       R0H,R0H
      BEQ         L7:8
      MOV.B       @_b:32,R0L
      BEQ         L8:8
L7:   MOV.B       #1,R0L
L8:   RTS
```

（扩展入汇编语言代码；优化后）

```
_func:
      MOV.B       @_a:32,R0L
      BEQ         L5:8
      MOV.B       @_b:32,R0L
      BEQ         L5:8
      MOV.B       #1,R0L
      RTS
L5:   SUB.B       R0L,R0L
      RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	26	22	30	24	24
之后	26	22	26	20	20

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	26	26	22
之后	26	26	22

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	18	15	42	36	36
之后	18	15	36	30	30

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	15	15	14
之后	15	15	14

说明与备注

执行速度是通过假设 *a*=1 和 *b*=1 来衡量。

6.2.3 使用替换值的条件决定

大小	O	速度	O	堆栈大小	Δ

重点

当在条件表达式中对决定语句使用替换值时，可通过将赋值语句视为条件决定语句来增进 ROM 的效率。

描述

通过同时执行条件表达式的决定和替换，可有效缩减代码大小。

实例

复制字符串 *s*。

（优化前的 C 语言程序）

```
char *s,*d;
void func(void)
{
    while(*s){
        *d++ = *s++;
    }
    *d++ = *s++;
}
```

（优化后的 C 语言程序）

```
char *s,*d;
void func(void)
{
    while(*d++ = *s++);
}
```

（扩展入汇编语言代码：优化前）

```
_func:
       STM.L       (ER4-ER5),@-SP
       MOV.L       #_s,ER5
       MOV.L       #_d,ER4
       BRA         L7:8
L6:    MOV.B       @ER0+,R1L
       MOV.L       ER0,@ER5
       MOV.L       @ER4,ER0
       MOV.B       R1L,@ER0
       MOV.L       @ER4,ER0
       INC.L       #1,ER0
       MOV.L       ER0,@ER4
L7:    MOV.L       @ER5,ER0
       MOV.B       @ER0,R1L
       BNE         L6:8
       MOV.B       @ER0+,R1L
       MOV.L       ER0,@ER5
       MOV.L       @ER4,ER0
       MOV.B       R1L,@ER0
       MOV.L       @ER4,ER0
       INC.L       #1,ER0
       MOV.L       ER0,@ER4
       LDM.L       @SP+,(ER4-ER5)
       RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
       STM.L       (ER4-ER5),@-SP
       MOV.L       #_s,ER5
       MOV.L       #_d,ER4
L5:    MOV.L       @ER5,ER0
       MOV.B       @ER0+,R1L
       MOV.L       ER0,@ER5
       MOV.L       @ER4,ER0
       INC.L       #1,ER0
       MOV.L       ER0,@ER4
       MOV.B       R1L,@-ER0
       BNE         L5:8
       LDM.L       @SP+,(ER4-ER5)
       RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	80	62	74	54	54
之后	52	32	44	36	34

	H8SX		
CPU 类型	MAX	ADV	NML
之前	70	70	56
之后	52	52	32

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	59	48	232	84	84
之后	45	26	218	74	74

	H8SX		
CPU 类型	MAX	ADV	NML
之前	37	43	31
之后	26	27	20

6.2.4 使用合适的运算法

大小	O	速度	O	堆栈大小	Δ

重点

通过使用数学技术，可增进 ROM 的效率和执行速度。

描述

如果算术表达式包含常用术语，应将这些术语析出以缩减执行算术操作的次数。 这可同时增进 ROM 的效率和执行速度。

实例

解决三阶方程式。

（优化前的 C 语言程序）

```
unsigned char a,b,c,d,x,y;
void func(void)
{
    y=a*x*x*x+b*x*x+c*x+d;
}
```

（优化后的 C 语言程序）

```
unsigned char a,b,c,d,x,y;
void func(void)
{
    y=x*(x*(a*x+b)+c)+d;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        PUSH.W      R6
        MOV.B       @_x:32,R6L
        MOV.B       R6L,R0L
        MULXU.B     R6L,R0
        MOV.B       R0L,R6H
        MULXU.B     R6L,R0
        MOV.B       @_a:32,R0H
        MULXU.B     R0H,R0
        MOV.B       @_b:32,R1L
        MULXU.B     R6H,R1
        ADD.B       R1L,R0L
        MOV.B       @_c:32,R1L
        MULXU.B     R6L,R1
        ADD.B       R1L,R0L
        MOV.B       @_d:32,R0H
        ADD.B       R0H,R0L
        MOV.B       R0L,@_y:32
        POP.W       R6
        RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
        MOV.B       @_x:32,R1L
        MOV.B       @_a:32,R0L
        MULXU.B     R1L,R0
        MOV.B       @_b:32,R0H
        ADD.B       R0H,R0L
        MULXU.B     R1L,R0
        MOV.B       @_c:32,R0H
        ADD.B       R0H,R0L
        MULXU.B     R1L,R0
        MOV.B       @_d:32,R0H
        ADD.B       R0H,R0L
        MOV.B       R0L,@_y:32
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	64	52	62	50	50
之后	56	44	50	38	38

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	64	64	52
之后	58	58	46

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	56	49	150	136	136
之后	48	41	106	92	92

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	33	29	26
之后	27	27	24

6.2.5 使用公式

大小	O	速度	O	堆栈大小	Δ

重点

如果指定的算术表达式中存在适当的数学公式，ROM 的效率和执行速度可通过使用公式来增进。

描述

使用数学公式以缩减面向算法的编码技术中所需的算术操作次数。 这可同时增进 ROM 的效率和执行速度。

实例

计算 1 到 100 的总和。

（优化前的 C 语言程序）

```
unsigned int s;
unsigned int n=100;
void func(void)
{
    unsigned int i;
    for (s=0,i=1;i<=n;i++)
        s+=i;
}
```

（优化后的 C 语言程序）

```
unsigned int s;
unsigned int n=100;
void func(void)
{
    s=n*(n+1)>>1;
}
```

（扩展入汇编语言代码；优化前）

```
_func:
      MOV.L       #_s:32,ER1
      SUB.W       R0,R0
      MOV.W       R0,@ER1
      MOV.W       #1:16,E0
      BRA         L7:8
L6:   MOV.W       @ER1,R0
      INC.W       #1,R0
      MOV.W       R0,@ER1
      INC.W       #1,E0
L7:   MOV.W       @_n:32,R0
      CMP.W       R0,E0
      BLS         L6:8
      RTS
```

（扩展入汇编语言代码；优化后）

```
_func:
      MOV.W       @_n:32,R1
      MOV.W       R1,R0
      INC.W       #1,R0
      MULXU.W     R1,ER0
      SHLR.W      R0
      MOV.W       R0,@_s:32
      RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	38	32	38	34	38
之后	24	20	24	20	30

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	36	36	30
之后	24	24	20

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	1322	1118	2644	2438	2450
之后	20	17	54	48	144

	H8SX		
CPU 类型	MAX	ADV	NML
之前	916	916	815
之后	14	14	13

6.2.6 使用本地变量

大小	O	速度	O	堆栈大小	O

重点

通过将可用作本地变量的临时变量、循环计数器等声明为本地变量，可增进 ROM 的效率和执行速度。

同样地，在对本地变量执行操作前，可通过将多个算术表达式的普遍外部变量分配给本地变量来提高效率。

描述

由于本地变量（与外部变量不同）多数为分配给寄存器而设计，使用本地变量将生成一个不包含存储器与寄存器之间的数据转移的目标。

在变量不因功能中断和其他原因而更改值的情形下，这些变量必须在执行算术操作前分配给本地变量。 基于上述原因，这也将增进 ROM 的效率和执行速度。

实例

将变量 *a* 增加到变量 *b*、*c* 和 *d*；将结果存储在变量 *b*、*c* 和 *d* 内。

（优化前的 C 语言程序）

```
unsigned char a,b,c,d;
void func(void)
{
    b+=a;
    c+=a;
    d+=a;
}
```

（优化后的 C 语言程序）

```
unsigned char a,b,c,d;
void func(void)
{
    unsigned char wk;
    wk=a;
    b+=wk;
    c+=wk;
    d+=wk;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        STM.L       (ER2-ER3),@-SP
        MOV.L       #_a:32,ER3
        MOV.B       @ER3,R0L
        MOV.L       #_b:32,ER1
        MOV.B       @ER1,R2L
        ADD.B       R0L,R2L
        MOV.B       R2L,@ER1
        MOV.B       @ER3,R0L
        MOV.L       #_c:32,ER1
        MOV.B       @ER1,R2L
        ADD.B       R0L,R2L
        MOV.B       R2L,@ER1
        MOV.B       @ER3,R3L
        MOV.L       #_d:32,ER0
        MOV.B       @ER0,R1L
        ADD.B       R3L,R1L
        MOV.B       R1L,@ER0
        LDM.L       @SP+,(ER2-ER3)
        RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
        MOV.B       @_a:32,R1H
        MOV.L       #_b:32,ER0
        MOV.B       @ER0,R1L
        ADD.B       R1H,R1L
        MOV.B       R1L,@ER0
        MOV.L       #_c:32,ER0
        MOV.B       @ER0,R1L
        ADD.B       R1H,R1L
        MOV.B       R1L,@ER0
        MOV.L       #_d:32,ER0
        MOV.B       @ER0,R1L
        ADD.B       R1H,R1L
        MOV.B       R1L,@ER0
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	50	36	60	50	50
之后	50	36	48	40	40

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	32	32	24
之后	32	32	24

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	36	28	156	64	64
之后	36	28	56	48	48

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	20	20	18
之后	20	20	18

说明与备注

此技术对 3.0 之前版本的编译程序有效。

编译程序 4.0 版本或以上版本的改进使变量被分配到寄存器，因此，使用本地变量可让编译程序扩展相同的汇编语言代码。

汇编此段中的扩展代码、目标大小和执行速度，将编码编译程序 3.0 版本（除了 H8SX）的编译结果。

在某些情况下，被分配了外部变量的本地变量，未被分配给寄存器。 检查目标列表以决定哪个本地变量是被分配到寄存器的。

6.2.7 将 f 分配给浮点类型常数

大小	O	速度	O	堆栈大小	O

重点

在浮点算术操作涉及包含在浮点类型的可分配值范围内（7.0064923216240862e-46f 到 3.4028235677973364e+38f）的常数的情形下，在数字值之后分配字母“f”以消除可能转换为*复式 (double)* 类型的冗余类型转换。

描述

浮点常数通常被视为*复式*类型常数。如果直接使用，这类常数将以*复式*类型计算，使得需要扩展操作。如果这个常数是数值在范围内（7.0064923216240862e-46f 到 3.4028235677973364e+38f）的对数常数，字母“f”应被连接到常数的尾端，以使它被视为*浮点*类型常数。这将有效缩减所生成的指令数目，同时增进 ROM 的效率、RAM 的效率，以及执行速度。

实例

将变量 *b* 和一个常数的总和分配给变量 *a*。

（优化前的 C 语言程序）

```
float a,b;
void func(void)
{
    a=b+1.0;
}
```

（优化后的 C 语言程序）

```
float a,b;
void func(void)
{
    a=b+1.0f;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        PUSH.L      ER2
        SUB.W       #16,R7
        MOV.L       @_b:32,ER1
        MOV.L       SP,ER0
        ADD.W       #8,R0
        JSR         @$FTOD$3:24
        MOV.L       ER0,ER1
        MOV.L       #L5,ER2
        MOV.L       SP,ER0
        JSR         @$ADDD$3:24
        JSR         @$DTOF$3:24
        MOV.L       ER0,@_a:32
        ADD.W       #16,R7
        POP.L       ER2
        RTS
L5:   .DATA.L   H'3FF00000,H'00000000
_a:   .RES.L      1
_b:   .RES.L      1
```

（扩展入汇编语言代码：优化后）

```
_func:
        MOV.L       @_b:32,ER0
        MOV.L       #1065353216,ER1
        JSR         @$ADDF$3:24
        MOV.L       ER0,@_a:32
        RTS
_a:     .RES.L    1
_b:     .RES.L    1
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	76	66	70	60	62
之后	28	24	28	24	26

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	78	72	66
之后	30	28	24

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	351	337	798	770	1076
之后	124	119	260	250	352

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	259	260	261
之后	96	97	103

6.2.8 指定移位操作中的常数

大小	O	速度	O	堆栈大小	O

重点

对于移位操作，如果移位计数是一个变量，编译程序将会调用运行时例程以处理操作。 如果移位计数是一个常数，编译程序将不会调用可有效增进执行速度的运行时例程。

描述

如果解决了常数，编译程序将可直接予以处理。

实例

以 8 位平移变量*数据*。

（优化前的 C 语言程序）

```
int data;
int sht=8;
void func(void)
{
    data=data<<sht;
}
```

（优化后的 C 语言程序）

```
#define SHT 8
int data;
void func(void)
{
    data=data<<SHT;
}
```

（扩展入汇编语言代码；优化前）

```
_func:
            MOV.L       #_data,ER0
            MOV.W       @_sht:32,R1
            JSR         @$DSLI$3:24
            RTS
_sht:
            .DATA.W     H'0008
_data:
            .RES.W      1
```

（扩展入汇编语言代码；优化后）

```
_func:
        MOV.B       @_data+1:32,R0H
        SUB.B       R0L,R0L
        MOV.W       R0,@_data:32
        RTS
_data:
        .RES.W      1
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	26	20	20	16	16
之后	16	12	16	12	12

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	26	26	20
之后	16	16	12

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	44	38	166	136	140
之后	14	11	28	22	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	14	14	12
之后	11	11	10

6.2.9 使用移位操作

大小	O	速度	O	堆栈大小	O

重点

在进行乘法和加法运算时，尽可能使用移位操作。

描述

复合赋值运算符（+=、-=、&=、|= …等等）和移位运算符的设计目的，是为了充分利用所使用 CPU 的性能特性。 在正确使用时，这些运算符可缩减代码大小并同时增进大小和速度。尤其是在将变量乘以常数时，应使用 <<（左移运算符）。

实例

分配三次*数据*值给变量 *a*。

（优化前的 C 语言程序）

```
int data,a;
void main()
{
    a=data+data+data;
}
```

（优化后的 C 语言程序）

```
int data,a;
void main()
{
    a=(data<<1)+data;
}
```

（扩展入汇编语言代码；优化前）

```
_main:
    PUSH.L    ER6
    MOV.L     #_data:32,ER6
    MOV.W     @ER6,R0
    MOV.W     R0,R1
    ADD.W     R1,R0
    ADD.W     R1,R0
    MOV.W     R0,@_a:32
    POP.L     ER6
    RTS
```

（扩展入汇编语言代码；优化后）

```
_main:
    MOV.W     @_data:32,R0
    SHLL.W    R0
    MOV.W     @_data:32,R1
    ADD.W     R1,R0
    MOV.W     R0,@_a:32
    RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	30	22	30	22	22
之后	24	18	24	18	18

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	20	20	16
之后	20	20	16

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	17	13	34	26	26
之后	14	11	28	22	22

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	12	12	11
之后	13	13	12

说明与备注

此技术适用于 3.0 之前版本的编译程序。

编译程序 4.0 版本或以上版本的改进，使乘法和加法运算中的移位操作可被执行，所以，编译程序扩展相同的汇编语言代码。

汇编本段中的扩展代码、目标大小和执行速度，将编码编译程序 3.0 版本（除了 H8SX）的编译结果。

6.2.10 统一连续 ADD 指令

大小	O	速度	O	堆栈大小	Δ

重点

加法应该连续编码以确保一致化和缩减代码大小。

描述

在遇到连续加法代码时，编译程序将执行一致的优化。 要充分利用此优化，必须尽可能地将加法运算连续编码。

实例

相加变量 *a* 的值。

（优化前的 C 语言程序）

```
int a,b;
void main()
{
    a+=10;
    b=10;
    a+=20;
}
```

（优化后的 C 语言程序）

```
int a,b;
void main()
{
    b=10;
    a+=10;
    a+=20;
}
```

（扩展入汇编语言代码；优化前）

```
_main:
     MOV.W       @_a:32,E0
     ADD.W       #10,E0
     MOV.W       #10,R0
     MOV.W       R0,@_b:32
     ADD.W       #20,E0
     MOV.W       E0,@_a:32
     RTS
```

（扩展入汇编语言代码；优化后）

```
_main:
     MOV.W       #10,R0
     MOV.W       R0,@_b:32
     MOV.W       @_a:32,E0
     ADD.W       R0,E0
     ADD.W       #20,E0
     MOV.W       E0,@_a:32
     RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	28	22	32	26	26
之后	28	22	30	24	24

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	18	18	14
之后	18	18	14

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	21	17	46	38	38
之后	21	17	44	36	36

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	13	13	12
之后	13	13	12

6.3 循环处理

6.3.1 选择循环计数器

大小	O	速度	O	堆栈大小	Δ

重点

通过使用减量计数器及将终止条件和零相比，可增进 ROM 的效率和执行速度。

描述

在 H8S 及 H8/300 系列微型计算机内执行数据转移指令（MOV 指令）的过程中，条件码寄存器中的 N 和 Z 标志都会更改。这将消除紧随数据转移指令之后对比较指令的需要，从而增进了 ROM 的效率和执行速度。

实例

将数组 *a* 的所有元素复制到数组 *b*。

（优化前的 C 语言程序）

```
unsigned char a[10],b[10];
         int  i;
void func(void)
{
    for(i=0; i<10; i++)
        b[i]=a[i];
}
```

（优化后的 C 语言程序）

```
unsigned char a[10],b[10];
         int  i;
void func(void)
{
    for(i=9; i>=0; i--)
        b[i]=a[i];
}
```

（扩展入汇编语言代码；优化前）

```
_func:
       STM.L       (ER4-ER5),@-SP
       MOV.L       #_i,ER5
       MOV.W       #1,R0
       MOV.W       R0,@ER5
       BRA         L8:8
L6:  MOV.W       R0,R1
       EXTS.L      ER1
       MOV.L       ER1,ER4
       MOV.B       @(_a:32,ER4),R0L
       MOV.B       R0L,@(_b:32,ER4)
       INC.W       #1,R1
       MOV.W       R1,@ER5
L8:  MOV.W       @ER5,R0
       CMP.W       #10,R0
       BLT         L6:8
       LDM.L       @SP+,(ER4-ER5)
       RTS
```

（扩展入汇编语言代码；优化后）

```
_func:
       STM.L       (ER4-ER5),@-SP
       MOV.L       #_i,ER5
       MOV.W       #9,R0
       MOV.W       R0,@ER5
       BRA         L8:8
L6:  MOV.W       R0,R1
       EXTS.L      ER1
       MOV.L       ER1,ER4
       MOV.B       @(_a:32,ER4),R0L
       MOV.B       R0L,@(_b:32,ER4)
       DEC.W       #1,R1
       MOV.W       R1,@ER5
L8:  MOV.W       @ER5,R0
       BGE         L6:8
       LDM.L       @SP+,(ER4-ER5)
       RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	46	30	54	38	38
之后	48	34	52	36	36

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	30	30	24
之后	36	36	32

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	163	121	624	366	386
之后	164	132	582	324	324

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	125	107	98
之后	106	118	108

6.3.2 选择重复控制语句

大小	O	速度	O	堆栈大小	Δ

重点

通过对至少执行一次的循环语句使用 *do-while* 语句，可增进 ROM 的效率和执行速度。

描述

如果循环语句被执行至少一次，应该使用 *do-while* 语句来将它编码以缩减循环计数决定的一次操作，从而增进 ROM 的效率和执行速度。

实例

将数组 *p2* 的内容复制到数组 *p1*。

（优化前的 C 语言程序）

```
unsigned char a[10],len=10;
unsigned char p1[10],p2[10];
void func(void)
{
    char i;
    for (i=len; i>0; i--)
        p1[i-1]=p2[i-1];
}
```

（优化后的 C 语言程序）

```
unsigned char a[10],len=10;
unsigned char p1[10],p2[10];
void func(void)
{
    char i=len;
    do{
        p1[i-1]=p2[i-1];
    } while(--i);
}
```

（扩展入汇编语言代码；优化前）

```
_func:
          PUSH.L   ER5
          MOV.B @_len:32,R1L
          BRA      L9:8
L8:       EXTS.W   R1
          EXTS.L   ER1
          MOV.L ER1,ER5
          MOV.B @(_p2-1:32,ER5),R0L
          MOV.B R0L,@(_p1-1:32,ER5)
          DEC.B R1L
L9:       MOV.B R1L,R1L
          BGT   L8:8
          POP.L ER5
          RTS
```

（扩展入汇编语言代码；优化后）

```
_func:
      PUSH.LER5
      MOV.B @_len:32,R1L
L9:
      EXTS.WR1
      EXTS.LER1
      MOV.L ER1,ER5
      MOV.B @(_p2-1:32,ER5),R0L
      MOV.B R0L,@(_p1-1:32,ER5)
      DEC.B R1L
      BNE   L9:8
      POP.L ER5
      RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	47	29	47	39	33
之后	43	25	43	35	29

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	35	35	29
之后	31	31	25

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	210	142	388	324	296
之后	195	127	358	294	266

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	134	134	133
之后	117	117	116

6.3.3 将不变量表达式从循环的内部移到外部

大小	O	速度	O	堆栈大小	Δ

重点

通过在循环外部定义出现在循环内部的不变量表达式，可增进执行速度。

描述

如果出现在循环内部的不等在循环外部被定义，不等只在循环的开头被评估，这将缩减在循环中执行的指令数目。 结果执行速度获得增进。

实例

使用变量 *b* 和 *c* 的总和来初始化数组 *a*。

（优化前的 C 语言程序）

```
unsigned char a[10],b,c;
         int  i;
void func(void)
{
    for (i=9; i>=0; i--)
        a[i]=b+c;
}
```

（优化后的 C 语言程序）

```
unsigned char a[10],b,c;
         int  i;
void func(void)
{
    unsigned char tmp;
    tmp=b+c;
    for (i=9; i>=0; i--)
        a[i]=tmp;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        PUSH.L      ER5
        MOV.L       #_i,ER5
        MOV.W       #9,R0
        MOV.W       R0,@ER5
        BRA         L9:8
L7:     MOV.W       R0,R1
        MOV.B       @_b:32,R0L
        MOV.B       @_c:32,R0H
        ADD.B       R0H,R0L
        EXTS.L      ER1
        MOV.B       R0L,@(_a:32,ER1)
        DEC.W       #1,R1
        MOV.W       R1,@ER5
L9:     MOV.W       @ER5,R0
        BGT         L7:8
        POP.L       ER5
        RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
        PUSH.W      R4
        MOV.L       #_i,ER1
        MOV.B       @_b:32,R4L
        MOV.B       @_c:32,R0L
        ADD.B       R0L,R4L
        MOV.W       #9,R0
        BRA         L10:8
L8:     MOV.W       @ER1,R0
        EXTS.L      ER0
        MOV.B       R4L,@(_a:32,ER0)
        DEC.W       #1,R0
L10:    MOV.W       R0,@ER1
        BGT         L8:8
        POP.W       R4
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	50	40	58	42	42
之后	50	40	50	38	38

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	46	46	38
之后	46	46	38

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	119	109	516	404	400
之后	119	109	322	254	254

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	101	95	83
之后	101	95	83

6.3.4 合并循环条件

大小	O	速度	O	堆栈大小	O

重点

在循环条件相同或相似的情形下，可通过将它们合并来增进 ROM 的效率和执行速度。

描述

此技术将缩减循环决定语句的目标大小，从而有效增进执行速度。

实例

以 0 来初始化数组 *a*，并以 1 来初始化数组 *b*。

（优化前的 C 语言程序）

```
int a[10],b[10];
void f(void)
{
    int i,j;
    for (i=0; i<10; i++)
        a[i]=0;
    for (j=0; j<10; j++)
        b[j]=1;
}
```

（优化后的 C 语言程序）

```
int a[10],b[10];
void f(void)
{
    int i;
    for (i=0; i<10; i++){
        a[i]=0;
        b[i]=1;
    }
}
```

（扩展入汇编语言代码；优化前）

```
_f:
        PUSH.L      ER6
        SUB.W       R6,R6
        SUB.W       R1,R1
L9:     EXTS.L      ER6
        MOV.L       ER6,ER0
        SHLL.L      ER0
        MOV.W       R1,@(_a:32,ER0)
        INC.W       #1,R6
        CMP.W       #10,R6
        BLT         L9:8
        SUB.W       R6,R6
        MOV.W       #1,R1
L10:    EXTS.L      ER6
        MOV.L       ER6,ER0
        SHLL.L      ER0
        MOV.W       R1,@(_b:32,ER0)
        INC.W       #1,R6
        CMP.W       #10,R6
        BLT         L10:8
        POP.L       ER6
        RTS
```

（扩展入汇编语言代码；优化后）

```
_f:
        PUSH.L      ER6
        SUB.W       R6,R6
        SUB.W       R1,R1
L7:     EXTS.L      ER6
        MOV.L       ER6,ER0
        SHLL.L      ER0
        MOV.W       R1,@(_a:32,ER0)
        INC.W       #1,R6
        CMP.W       #10,R6
        BLT         L7:8
        EXTS.L      ER6
        SHLL.L      ER6
        MOV.W       #1,R0
        MOV.W       R0,@(_b:32,ER6)
        POP.L       ER6
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	58	48	62	44	50
之后	46	32	46	32	34

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	38	38	34
之后	28	28	24

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	245	197	558	418	468
之后	188	134	432	350	342

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	175	177	168
之后	125	117	117

6.4 指针

6.4.1 使用指针变量

大小	O	速度	O	堆栈大小	O

重点

如相同变量（外部变量）被屡次参考或必须存取数组元素，ROM 的效率和执行速度可通过使用指针变量来增进。

描述

使用指针变量会生成采用有效率的寻址模式（@Rn、@Rn+、@-Rn）的代码。

实例

将数组 *data2* 的元素复制到数组 *data1*。

（优化前的 C 语言程序）

```
void func(int data1[],int data2[])
{
    int i;

    for (i=0; i<10; i++)
        data1[i]=data2[i];
}
```

（优化后的 C 语言程序）

```
void func(int *data1,int *data2)
{
    int i;

    for (i=0; i<10; i++){
        *data1=*data2;
        data1++; data2++;
    }
}
```

（扩展入汇编语言代码：优化前）

```
_func:
            PUSH.L      ER3
            STM.L       (ER4-ER6),@-SP
            MOV.L       ER0,ER4
            MOV.L       ER1,ER3
            SUB.W       R6,R6
L6:
            EXTS.L      ER6
            MOV.L       ER6,ER5
            SHLL.L      ER5
            MOV.L       ER4,ER0
            ADD.L       ER5,ER0
            MOV.L       ER3,ER1
            ADD.L       ER5,ER1
            MOV.W       @ER1,R1
            MOV.W       R1,@ER0
            INC.W       #1,R6
            CMP.W       #10:16,R6
            BLT         L6:8
            LDM.L       @SP+,(ER4-ER6)
            POP.L       ER3
            RTS
```

（扩展入汇编语言代码：优化后）

```
__func:
            PUSH.L      ER5
            MOV.L       ER0,ER5
            MOV.W       #10:16,E0
L7:
            MOV.W       @ER1,R0
            MOV.W       R0,@ER5
            INC.L       #2,ER5
            INC.L       #2,ER1
            DEC.W       #1,E0
            BNE         L7:8
            POP.L       ER5
            RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	46	38	40	38	40
之后	24	22	28	26	30

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	42	42	34
之后	18	18	18

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	171	152	482	336	428
之后	107	102	216	208	238

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	156	164	155
之后	79	81	80

6.5 数据结构

6.5.1 确保数据兼容性

大小	O	速度	O	堆栈大小	–

重点

数据项目以所声明的顺序分配。通过有效指定数据项目声明的顺序以消除虚设存储区的生成，可增进使用 ROM 和 RAM 的效率。

描述

如果大于或等于 2 字节的变量，在必须保持双数存储器按址时，从奇数存储器地址被分配，编译程序将创建 1 字节虚设区域。要避免这个问题，相同大小的变量必须在单一群组中被声明，以减少为数据对齐而创建的虚设数据区域。

此情况不只适用于外部变量，同时也适用于本地变量、结构和公用的成员，以及函数参数。

实例

分配总数 8 字节的数据。

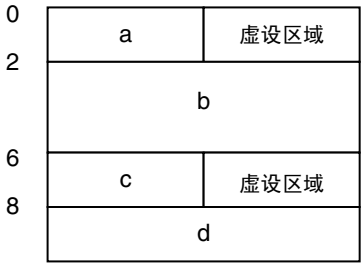
（优化前的 C 语言程序）

```
char  a;
long  b;
char  c;
short d;
```

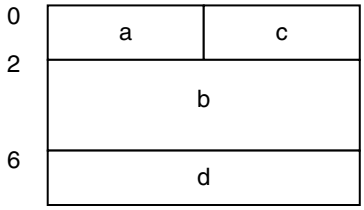
（优化后的 C 语言程序）

```
char  a;
char  c;
long  b;
short d;
```

（数据赋值，优化前）



（数据赋值，优化后）



目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	10	10	10	10	10
之后	8	8	8	8	8

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	8	8	8
之后	8	8	8

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	–	–	–	–	–
之后	–	–	–	–	–

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	–	–	–
之后	–	–	–

说明与备注

以上除了在 H8SX 中，皆为 3.0 版本操作的结果。

由于**对齐**在编译程序 4.0 版本或以上版本中为默认设定，边界对齐将自动进行以缩减空的区域。所以，此项改进没有起到特别的作用。

6.5.2 数据初始化的技术

大小	O	速度	O	堆栈大小	Δ

重点

要缩减程序大小，任何需要初始化的变量必须在声明时被初始化。

描述

在声明时被初始化的数据将率先在初始化数据区域（D 段）中被分配，然后在程序执行时被复制到 RAM。初始值的赋值只在开始执行程序时执行一次。

相反情况下，任何在声明时没有被初始化的数据将会在未初始化数据区域（B 段）中被分配，而它所需要的存储器是数据被分配到初始化数据区的一半。

另外，第二个方法以通过赋值语句设定程序中的初始值，使程序区域的大小（P 段）增加。

如果存在需要初始值的多个变量，为获得更好的效率，这些变量应在声明时被初始化。

实例

初始化变量 *a*。

（优化前的 C 语言程序）

```
int a;
void main(void)
{
    a=1;
}
```

（优化后的 C 语言程序）

```
int a=1;
void main(void)
{
}
```

（扩展入汇编语言代码：优化前）

```
_main:
            MOV.W       #1:16,R0
            MOV.W       R0,@_a:32
            RTS
            .SECTION    B,DATA,ALIGN=2
_a:
            .RES.W      1
```

（扩展入汇编语言代码：优化后）

```
_main:
            RTS
            .SECTION    D,DATA,ALIGN=2
_a:
            .DATA.W     H'0001
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	12	10	12	10	10
之后	4	4	4	4	4

	H8SX		
CPU 类型	MAX	ADV	NML
之前	8	8	6
之后	4	4	4

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	11	9	22	18	18
之后	5	4	10	8	8

	H8SX		
CPU 类型	MAX	ADV	NML
之前	8	8	7
之后	6	6	6

6.5.3 统一数组元素的初始化

大小	O	速度	O	堆栈大小	Δ

重点

在一些数组元素必须被初始化的情况下，通过将它们分组到结构以使它们可以在单一操作中被初始化，将可增进 ROM 的效率。

描述

通过在群组中初始化数据，所需的转移指令执行次数可被缩减至一次。

实例

使用相应值对化数组 *a*、*b* 和 *c* 进行初始化。

（优化前的 C 语言程序）

```
void f(void)
{
    unsigned char a[]={0,1,2,3};
    unsigned char b[]="abcdefg";
    unsigned char c[]="ABCDEFG";
}
```

（优化后的 C 语言程序）

```
void f(void)
{
    struct x{
        unsigned char a[4];
        unsigned char b[8];
        unsigned char c[7];
    } A
    ={0,1,2,3,"abcdefg","ABCDEFG"};
}
```

（扩展入汇编语言代码：优化前）

```
_f:
        PUSH.L      ER2
        SUB.W       #20,R7
        MOV.L       #L4,ER0
        MOV.L       SP,ER1
        ADD.W       #16,R1
        SUB.L       ER2,ER2
        MOV.B       #4,R2L
        JSR         @$MVN$3:24
        MOV.L       #L6,ER0
        MOV.L       SP,ER1
        ADD.W       #8,R1
        SUB.L       ER2,ER2
        MOV.B       #8,R2L
        JSR         @$MVN$3:24
        MOV.L       #L8,ER0
        MOV.L       SP,ER1
        SUB.L       ER2,ER2
        MOV.B       #8,R2L
        JSR         @$MVN$3:24
        ADD.W       #20,R7
        POP.L       ER2
        RTS
L4:         .DATA.B     H'00,H'01,H'02,H'03
L6:         .SDATAZ     "abcdefg"
L8:         .SDATAZ     "ABCDEFG"
```

（扩展入汇编语言代码：优化后）

```
_f:
            PUSH.L      ER2
            SUB.W       #20,R7
            MOV.L       #L4,ER0
            MOV.L       SP,ER1
            SUB.L       ER2,ER2
            MOV.B       #19,R2L
            JSR         @$MVN$3:24
            ADD.W       #20,R7
            POP.L       ER2
            RTS
L4:         .DATA.B   H'00,H'01,H'02,H'03
            .SDATAZ     "abcdefg"
            .SDATA      "ABCDEFG"
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	120	106	122	106	110
之后	81	69	81	75	79

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	104	104	96
之后	79	77	69

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	294	256	690	572	632
之后	162	145	488	324	376

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	51	48	41
之后	90	89	79

说明与备注

H8SX 可通过转移指令而非运行时函数转移数据。 因此，之前的执行速度会比之后的快。

6.5.4 将参数传递为结构地址

大小	O	速度	O	堆栈大小	Δ

重点

未被分配到寄存器的参数必须使用结构的地址来传递以缩减程序大小。

描述

所使用的参数数目和大小必须适当地调整以便分配到寄存器。要获取有关如何将参数传递到寄存器的描述，请参考适当的用户手册。

在需要大的参数和使用大量参数的情况下，必须在将参数传递到目标函数之前，将它们在结构中分组以缩减程序大小。 如果参数被声明为结构的成员，且结构的起始地址被当作参数传递到目标函数，接收函数将可依据接收地址来存取成员。

实例

将长类型数据 *a*、*b*、*c* 和 *d* 传递到函数 *func*。

（优化前的 C 语言程序）

```
void sub(long,long,long,long);
long a,b,c,d;

void func(void)
{
    sub(a,b,c,d);
}
```

（优化后的 C 语言程序）

```
void sub(struct ctag *);
struct ctag{
    long a;
    long b;
    long c;
    long d;
}x;

void func(void)
{
    sub(&x);
}
```

（扩展入汇编语言代码：优化前）

```
_func:
          MOV.L       @_d:32,ER0
          PUSH.L      ER0
          MOV.L       @_c:32,ER0
          PUSH.L      ER0
          MOV.L       @_b:32,ER1
          MOV.L       @_a:32,ER0
          JSR         @_sub:24
          ADDS.L      #4,SP
          ADDS.L      #4,SP
          RTS
_a:       .RES.L      1
_b:       .RES.L      1
_c:       .RES.L      1
_d:       .RES.L      1
```

（扩展入汇编语言代码：优化后）

```
_func:
          MOV.L       #_x:32,ER0
          JMP         @_sub:24
_x:
          .RES.W      8
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	58	50	52	44	62
之后	14	12	10	10	10

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	50	48	40
之后	16	14	12

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	52	45	102	88	126
之后	18	14	22	18	18

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	30	28	28
之后	14	14	14

6.5.5 将结构分配给寄存器

大小	O	速度	O	堆栈大小	O

重点

在本地变量被当作结构使用时，必须声明成员以便可以将变量直接分配到寄存器。

描述

由于结构也可以被分配到寄存器，所以大小效率和处理速度都可以通过适当地分配结构成员来增进。

实例

将结构数据传递到函数 *func*。

（优化前的 C 语言程序）

```
struct ST {
    char  a;
    short b;
    char  c;
}pst;
void main()
{
    struct ST s;
    s.a=pst.a+10;
    s.b=s.a+s.c;
    func(s);
}
```

（优化后的 C 语言程序）

```
struct ST {
    short b;
    char  a;
    char  c;
}pst;
void main()
{
    struct ST s;
    s.a=pst.a+10;
    s.b=s.a+s.c;
    func(s);
}
```

（扩展入汇编语言代码；优化前）

```
_main:
        STM.L       (ER2-ER3),@-SP
        SUBS.L      #4,SP
        SUBS.L      #2,SP
        MOV.L       SP,ER3
        MOV.B       #10,R0L
        MOV.B       R0L,@_pst:32
        MOV.B       R0L,@ER3
        EXTS.W      R0
        MOV.B       @(4:16,ER3),R1L
        EXTS.W      R1
        ADD.W       R1,R0
        MOV.W       R0,@(2:16,ER3)
        MOV.L       ER3,ER0
        SUBS.L      #4,SP
        SUBS.L      #2,SP
        MOV.L       SP,ER1
        SUB.L       ER2,ER2
        MOV.B       #6,R2L
        JSR         @$MVN$3:24
        JSR         @_func:24
        ADDS.L      #4,SP
        ADDS.L      #4,SP
        ADDS.L      #4,SP
        LDM.L       @SP+,(ER2-ER3)
        RTS
```

（扩展入汇编语言代码；优化后）

```
_main:
        PUSH.L      ER6
        MOV.B       #10,R0L
        MOV.B       R0L,@_pst+2:32
        MOV.B       R0L,R6H
        EXTS.W      R0
        MOV.B       R6L,R1L
        EXTS.W      R1
        ADD.W       R1,R0
        MOV.W       R0,E6
        PUSH.L      ER6
        JSR         @_func:24
        ADDS.L      #4,SP
        POP.L       ER6
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	66	60	64	64	80
之后	44	46	42	40	72

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	62	60	64
之后	42	40	38

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	126	110	416	252	286
之后	42	40	84	76	260

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	40	40	36
之后	31	29	32

6.6 函数

6.6.1 增进函数被定义的程序位置

大小	O	速度	O	堆栈大小	Δ

重点

通过在同一文件中定义任何经常在模块中调用的函数，可能可以增进 ROM 的效率和执行速度。

描述

在转移目标地址范围处于 –128 到 127 字节的情况下，H8S 或 H8/300 系列微型计算机将使用 PC 相关寻址模式 (BSR)。与由外部参考函数所声明的绝对寻址模式 (JSR) 相比，此模式可以增进 ROM 的效率和执行速度。

实例

从函数 *func* 和 *func1* 调用函数 *func2*。

（优化前的 C 语言程序）

```
extern int func2(void);
int ret;
void func(void)
{
  int i;
  i=func2();
  ret = i;
}
void func1(void)
{
  int i;
  i=func2();
  ret = i;
}
```

（优化后的 C 语言程序）

```
int ret;
int func2(void)
{
  return 0;
}
void func(void)
{
  int i;
  i=func2();
  ret = i;
}
void func1(void)
{
  int i;
  i=func2();
}
```

（扩展入汇编语言代码：优化前）

```
_func:
          JSR         @_func2:24
          MOV.W       R0,@_ret:32
          RTS
_func1:
          JSR         @_func2:24
          MOV.W       R0,@_ret:32
          RTS
```

（扩展入汇编语言代码：优化后）

```
_func2:
          SUB.W       R0,R0
          RTS
_func:
          BSR         _func2:8
          MOV.W       R0,@_ret:32
          RTS
_func1:
          BSR         _func2:8
          MOV.W       R0,@_ret:32
          RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	28	24	28	24	24
之后	24	20	24	20	20

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	32	28	24
之后	24	24	20

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	20	16	40	32	32
之后	19	15	38	30	30

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	16	16	15
之后	16	16	16

6.6.2 宏调用

大小	O	速度	O	堆栈大小	Δ

重点

通过将经常被调用的函数定义为宏，以增进大小效率和处理速度。

描述

在相同处理例程被定义为宏时，它们将会在被调用的位置进行内联扩展。 这将消除代码的生成和增进效率。

实例

调用函数 *abs*。

（优化前的 C 语言程序）

```
extern int a,b,c;
int abs(x)
int x;
{    return x>=0?x:-x;  }
void f(void)
{
    a=abs(b);
    b=abs(c);
}
```

（优化后的 C 语言程序）

```
#define abs(x) ((x)>=0?(x):-(x))
extern int a,b,c;
void f(void)
{
    a=abs(b);
    b=abs(c);
}
```

（扩展入汇编语言代码；优化前）

```
_abs:
            PUSH.W      R6
            MOV.W       R0,R6
            BLT         L9:8
            MOV.W       R6,R1
            BRA         L10:8
L9:         MOV.W       R6,R1
            NEG.W       R1
L10:        MOV.W       R1,R0
            POP.W       R6
            RTS
_f:         MOV.W       @_b:32,R0
            BSR         _abs:8
            MOV.W       R0,@_a:32
            MOV.W       @_c:32,R0
            BSR         _abs:8
            MOV.W       R0,@_b:32
            RTS
```

（扩展入汇编语言代码；优化后）

```
_f:
        PUSH.W      R6
        MOV.W       @_b:32,R6
        BLT         L7:8
        MOV.W       R6,R0
        BRA         L8:8
L7:     MOV.W       R6,R0
        NEG.W       R0
L8:     MOV.W       R0,@_a:32
        MOV.W       @_c:32,R6
        BLT         L9:8
        MOV.W       R6,R0
        BRA         L10:8
L9:     MOV.W       R6,R0
        NEG.W       R0
L10:    MOV.W       R0,@_b:32
        POP.W       R6
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	38	30	46	38	42
之后	32	26	50	42	46

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	38	38	30
之后	34	34	26

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	45	36	106	88	112
之后	24	20	74	64	64

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	36	36	34
之后	20	20	17

6.6.3 声明原型

大小	O	速度	O	堆栈大小	Δ

重点

具有 *字符*类型或*无符号字符*类型参数的函数，必须在被调用之前进行原型声明以消除冗余类型转换代码的输出。

描述

如果调用没有声明原型，具有 *字符*类型或*无符号字符*类型参数的函数将转换为*整数 (int)* 类型，这将生成冗余符号扩展指令和零扩展指令。

此外，参数可能无法正确传递。

实例

调用具有*字符*类型或*无符号字符*类型参数的函数 *sub1*。

（优化前的 C 语言程序）

```
char a;
unsigned char b;
void func(void)
{
    sub1(a,b);
}
```

（优化后的 C 语言程序）

```
void sub1(char, unsigned char);
char a;
unsigned char b;
void func(void)
{
    sub1(a,b);
}
```

（扩展入汇编语言代码：优化前）

```
_func:
          MOV.B       @_b:32,R0L
          EXTU.W      R0
          MOV.B       @_a:32,R1L
          EXTS.W      R1
          MOV.W       R0,E0
          MOV.W       R1,R0
          JMP         @_sub1:24
          RTS
```

（扩展入汇编语言代码：优化后）

```
_func:
          MOV.B       @_b:32,R0H
          MOV.B       @_a:32,R0L
          JMP         @_sub1:24
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	22	18	24	20	18
之后	18	14	16	20	12

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	24	22	18
之后	20	18	14

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	19	16	40	34	32
之后	23	18	32	26	26

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	15	15	14
之后	19	17	17

6.6.4 尾递归的优化

大小	O	速度	O	堆栈大小	O

重点

如果函数进行了一个函数调用，观察函数调用是否可以移动到调用源函数的尾端。 这可同时增进 ROM 的效率和执行速度。

描述

尾递归优化会在满足以下所有条件后执行：

- 调用源函数不会在堆栈上放置它的参数或返回值地址。
- 该函数调用之后即是 RTS 指令。

实例

调用函数 *sub* 并更新外部变量的值。

（优化前的 C 语言程序）

```
void g(void);
int a;
void main(void)
{
    if (a==0)     a++;
    else{
        g();
        a+=2;
    }
}
```

（优化后的 C 语言程序）

```
void g(void);
int a;
void main(void)
{
    if (a==0)     a++;
    else{
        a+=2;
        g();
    }
}
```

（扩展入汇编语言代码：优化前）

```
_main:
       PUSH.L      ER6
       MOV.L       #_a,ER6
       MOV.W       @ER6,R0
       BNE         L6:8
       INC.W       #1,R0
       BRA         L8:8
L6:    JSR         @_g:24
       MOV.W       @ER6,R0
       INC.W       #2,R0
L8:    MOV.W       R0,@ER6
       POP.L       ER6
       RTS
```

（扩展入汇编语言代码：优化后）

```
_main:
       MOV.L       #_a64,ER1
       MOV.W       #1,R0
       INC.W       #2,R0
       MOV.W       R0,@ER1
       JMP         @_g:24
       RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	38	32	38	32	32
之后	30	32	30	28	28

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	36	34	30
之后	30	28	34

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	37	29	74	58	58
之后	20	27	40	36	36

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	22	25	20
之后	15	15	23

说明与备注

以上的程序假定变量 *a* 并未在函数 *g* 内被参考。

6.6.5 增进参数传递的方式

大小	O	速度	O	堆栈大小	O

重点

要缩减代码大小，必须调整参数列出时的顺序以使参数之间没有间隙。

描述

传过寄存器的参数会按照被声明的顺序分配给寄存器 ER0 和 ER1（或在 H300 CPU 的情形下是 R0 和 R1）。因此，参数声明的顺序必须被调整，以便减少参数之间的任何间隙，进而缩减代码大小。

实例

调用函数 *func*。

（优化前的 C 语言程序）

```
long rtn;
void func(char,short,char);
void main()
{
    short a;
    char  b,c;

    func(b,a,c);
}
void func(char x,short y,char z)
{
    rtn=x*y+z;
}
```

（优化后的 C 语言程序）

```
long rtn;
void func(char,char,short);
void main()
{
    short a;
    char  b,c;

    func(b,c,a);
}
void func(char x,char y,short z)
{
    rtn=x*y+z;
}
```

（扩展入汇编语言代码：优化前）

```
_main:
    SUBS.L    #4,SP
    SUBS.L    #2,SP
    MOV.B     @(5:16,SP),R0H
    MOV.W     @(2:16,SP),E0
    MOV.B     @SP,R0L
    BSR       _func:8
    ADDS.L    #2,SP
    ADDS.L    #4,SP
    RTS
_func:
    PUSH.L      ER6
    MOV.B       R0H,R6H
    EXTS.W      R0
    MOV.W       R0,R1
    MULXU.W     E0,ER1
    MOV.B       R6H,R6L
    EXTS.W      R6
    ADD.W       R6,R1
    EXTS.L      ER1
    MOV.L       ER1,@_rtn:32
    POP.L       ER6
    RTS
```

（扩展入汇编语言代码：优化后）

```
_main:
    SUBS.L  #4,SP
    MOV.W   @(2:16,SP),E0
    MOV.B   @(1:16,SP),R0H
    MOV.B   @SP,R0L
    BSR     _func:8
    ADDS.L  #4,SP
    RTS
_func:
    MOV.B   R0L,R1L
    MULXS.B R0H,R1
    ADD.W   E0,R1
    EXTS.L  ER1
    MOV.L   ER1,@_rtn:32
    RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	26	24	56	54	60
之后	22	20	38	36	48

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	26	26	24
之后	22	22	20

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	29	25	120	112	228
之后	26	22	82	74	174

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	25	21	21
之后	21	19	19

说明与备注

要获取有关如何分配参数的描述，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.3.3 节，参数赋值的实例 (Examples of Parameter Assignment)。

注意当传递参数的寄存器数目被一个选项更改时，接收参数的寄存器数目也将更改。

6.7 转移指令

6.7.1 将切换语句重写为表

大小	O	速度	O	堆栈大小	Δ

重点

如果 case 语句所执行的与 *switch* 关联的处理任务是相似的，*switch* 语句必须使用一个表来编码以缩减目标大小。

描述

使用一个表来重写 *switch* 语句可有效缩减程序大小，虽然数据的大小将会增加。然而，若 *case* 语句的值范围很广，以表的形式来重写 *switch* 语句就会增加整个程序的大小。

实例

根据函数 *a* 的值转移到一个函数。

（优化前的 C 语言程序）

```
extern void f1(void);
extern void f2(void);
extern void f3(void);
extern void f4(void);
extern void f5(void);
extern int a;
void sub(void)
{
    switch(a){
    case 0:f1();break;
    case 1:f2();break;
    case 2:f3();break;
    case 3:f4();break;
    case 4:f5();break;
    }
}
```

（优化后的 C 语言程序）

```
extern void f1(void);
extern void f2(void);
extern void f3(void);
extern void f4(void);
extern void f5(void);
extern int a;
void sub(void)
{
    static int (*key[5])()=
        {f1,f2,f3,f4,f5};

    (*key[a])();
}
```

（扩展入汇编语言代码；优化前）

```
_sub:    MOV.W      @_a:32,R0
         MOV.B      R0H,R0H
         BNE        L15:8
         CMP.B      #0:8,R0L
         BEQ        L10:8
         CMP.B      #1:8,R0L
         BEQ        L11:8
         CMP.B      #2:8,R0L
         BEQ        L12:8
         CMP.B      #3:8,R0L
         BEQ        L13:8
         CMP.B      #4:8,R0L
         BEQ        L14:8
         RTS
L10:     JMP        @_f1:24
L11:     JMP        @_f2:24
L12:     JMP        @_f3:24
L13:     JMP        @_f4:24
L14:     JSR        @_f5:24
L15:     RTS
```

（扩展入汇编语言代码；优化后）

```
_sub: MOV.W      @_a:32,R0
      EXTS.L     ER0
      SHLL.L     #2,ER0
      MOV.L      @(L9:32,ER0),ER0
      JSR        @ER0
      RTS
L9:   .DATA.L    _f1,_f2,_f3,_f4,_f5
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	78	64	66	62	76
之后	56	36	56	34	34

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	94	78	58
之后	50	50	36

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	31	28	74	54	68
之后	27	18	42	26	26

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	24	32	24
之后	19	20	18

6.7.2 编写 Case 语句跳转至相同标签的程序

大小	O	速度	O	堆栈大小	Δ

重点

包含相同表达式的 *case* 语句必须组合在一起，以减少转移指令的数目并缩减目标大小。

描述

在 *switch* 语句使用 *if-then* 扩展方法的情况下，转移指令的数目越小，代码大小也越小，而程序效率则更高。

实例

根据 *c* 的值分配值给 *II*。

（优化前的 C 语言程序）

```
long ll;
void func(void)
{
    char c;
    switch(c){
    case 0: ll=0; break;
    case 1: ll=0; break;
    case 2: ll=1; break;
    case 3: ll=1; break;
    case 4: ll=2; break;
    }
}
```

（优化后的 C 语言程序）

```
long ll;
void func(void)
{
    char c;
    switch(c){
    case 0:
    case 1: ll=0; break;
    case 2:
    case 3: ll=1; break;
    case 4: ll=2; break;
    }
}
```

（扩展入汇编语言代码：优化前）

```
_func:
            SUBS.L      #2,SP
            MOV.L       #_ll:32,ER1
            MOV.B       @(1:16,SP),R0L
            BEQ         L6:8
            CMP.B       #1:8,R0L
            BEQ         L7:8
            CMP.B       #2:8,R0L
            BEQ         L8:8
            CMP.B       #3:8,R0L
            BEQ         L9:8
            CMP.B       #4:8,R0L
            BEQ         L10:8
            BRA         L11:8
L6:
L7:         SUB.L       ER0,ER0
            BRA         L15:8
L8:         SUB.L       ER0,ER0
            MOV.B       #1:8,R0L
            BRA         L15:8
L9:         SUB.L       ER0,ER0
            MOV.B       #1:8,R0L
            BRA         L15:8
L10:        SUB.L       ER0,ER0
            MOV.B       #2:8,R0L
L15:        MOV.L       ER0,@ER1
L11:        ADDS.L      #2,SP
            RTS
_ll:        .RES.L      1
```

（扩展入汇编语言代码：优化后）

```
_func:
            SUBS.L      #2,SP
            MOV.L       #_ll:32,ER1
            MOV.B       @(1:16,SP),R0L
            BEQ         L6:8
            CMP.B       #1:8,R0L
            BEQ         L7:8
            CMP.B       #2:8,R0L
            BEQ         L8:8
            CMP.B       #3:8,R0L
            BEQ         L9:8
            CMP.B       #4:8,R0L
            BEQ         L10:8
            BRA         L11:8
L6:
L7:         SUB.L       ER0,ER0
            BRA         L13:8
L8:
L9:         SUB.L       ER0,ER0
            MOV.B       #1:8,R0L
            BRA         L13:8
L10:        SUB.L       ER0,ER0
            MOV.B       #2:8,R0L
L13:        MOV.L       ER0,@ER1
L11:        ADDS.L      #2,SP
            RTS
_ll:        .RES.L      1
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	49	45	63	59	69
之后	45	43	57	53	61

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	55	55	47
之后	51	51	45

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	**ADV**	**NML**	**ADV**	**NML**	**NML**
之前	33	30	82	72	68
之后	20	18	82	72	68

	H8SX		
CPU 类型	**MAX**	**ADV**	**NML**
之前	24	24	23
之后	13	14	12

说明

以上的性能衡量是以 *c* = 4 的情形为准。

此技术适用于 3.0 之前版本的编译程序。

编译程序 4.0 版本或以上版本的改进可将 case 语句的跳转目标组合在一起，因此，编译程序扩展相同的汇编语言代码（除了 H8SX）。

汇编此段中的扩展代码、目标大小和执行速度，将编码编译程序 3.0 版本（除了 H8SX）的编译结果。

据一般规则来说，*switch* 语句中经常执行的 *case* 值必须先被测试以增进执行速度。 用户被鼓励在执行程序时尝试此技术。

6.7.3 转移到直接在指定语句下编码的函数

大小	O	速度	O	堆栈大小	O

重点

如果函数调用出现在函数的尾端，必须将调用目标函数直接放置在函数调用下面。

描述

如果尾递归优化正在进行，必须将调用目标函数直接放置在函数调用下面，以充分利用此优化来产生删除函数调用代码的效果。

由于删除了函数调用代码，程序的大小将会缩减而处理速度将会增加。

实例

从函数 *main* 调用函数 *func*。

（优化前的 C 语言程序）

```
int a;
void func();
void func()
{
    a++;
}
void main()
{
    a=0;
    func();
}
```

（优化后的 C 语言程序）

```
int a;
void func();
void main()
{
    a=0;
    func();
}
void func()
{
    a++;
}
```

（扩展入汇编语言代码：优化前）

```
_func:
        MOV.L   #_a,ER0
        MOV.W   @ER0,R1
        INC.W   #1,R1
        MOV.W   R1,@ER0
        RTS
_main:
        SUB.W   R0,R0
        MOV.W   R0,@_a:32
        BRA     _func:8
```

（扩展入汇编语言代码：优化后）

```
_main:
        SUB.W   R0,R0
        MOV.W   R0,@_a:32
_func:
        MOV.L   #_a:32,ER0
        MOV.W   @ER0,R1
        INC.W   #1,R1
        MOV.W   R1,@ER0
        RTS
```

目标大小表 [字节]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	26	20	24	20	20
之后	24	18	22	18	18

	H8SX		
CPU 类型	MAX	ADV	NML
之前	18	18	14
之后	16	16	12

执行速度表 [周期]

	H8S/2600, H8S/2000		H8/300H		H8/300
CPU 类型	ADV	NML	ADV	NML	NML
之前	21	17	40	34	34
之后	19	15	36	30	30

	H8SX		
CPU 类型	MAX	ADV	NML
之前	14	14	13
之后	12	12	11

H8S, H8/300 系列 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	节
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	制作堆栈信息文件	7.3.1
19		启动 Call Walker	7.3.2
20		文件打开和 Call Walker 窗口	7.3.3
21		编辑堆栈信息文件	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		其他函数	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) ...**

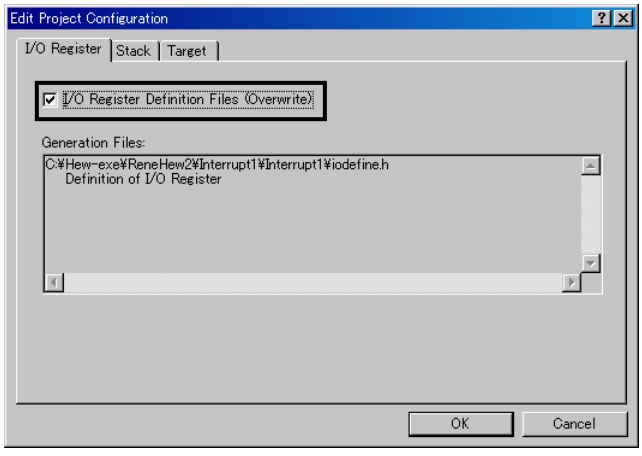
• 可被再生成的文件包括：

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。

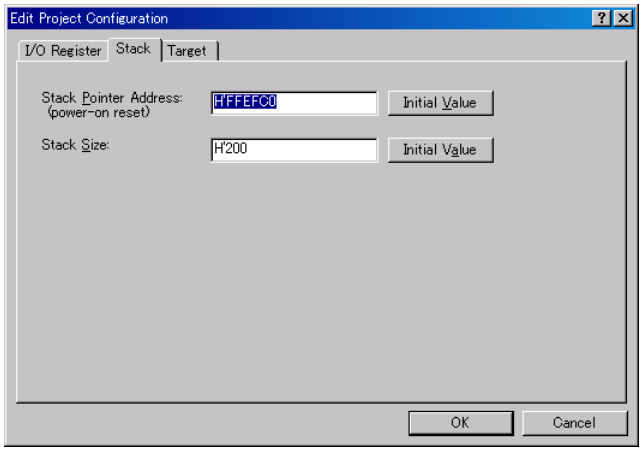
若您不小心修改了 iodefine.h，您可以再生成该文件，并覆盖已修改的文件。



• 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被 HEW 2.0 或以上版本支持。

7.1.2 命令描述文件的输出

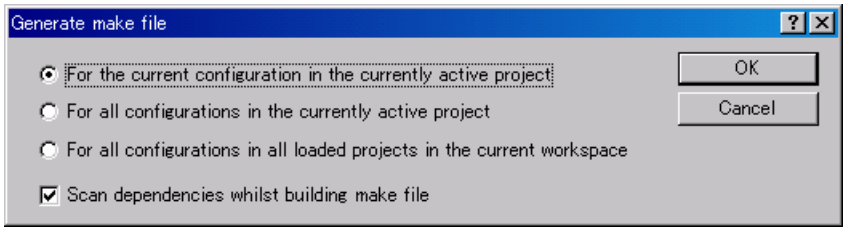
• 描述：

HEW 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被执行。

• 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：

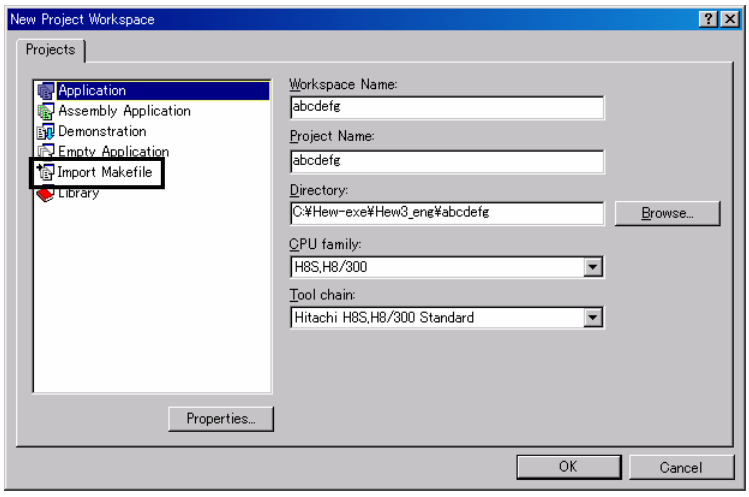
HEW 允许输入由 HEW 生成或在 UNIX 环境中使用的命令描述文件。

从命令描述文件，您可自动获取工程的**文件结构**。

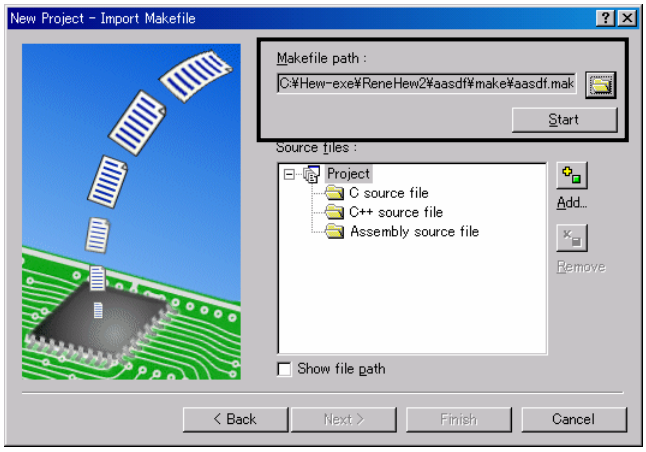
（不过，您不能获取选项设定或类似的规格。） 这方便了从命令行至 HEW 的转移。

• 命令描述文件的输入方法：

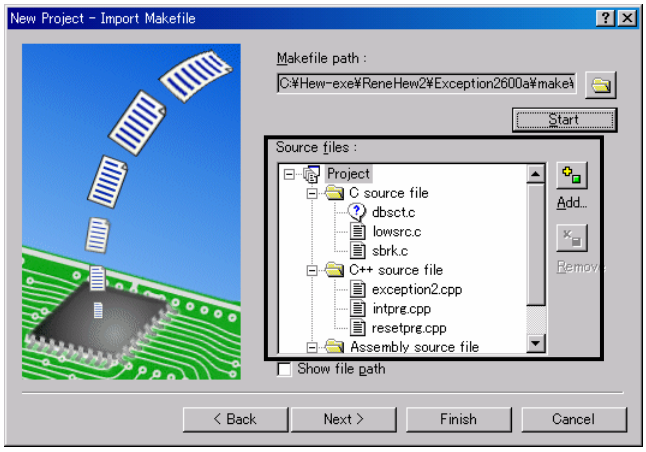
1. 当创建新的工作空间时，在 [新的工程工作空间 (New Project Workspace)] 对话框内，从工程类型选项选取 [导入命令描述文件 (Import Makefile)]。



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

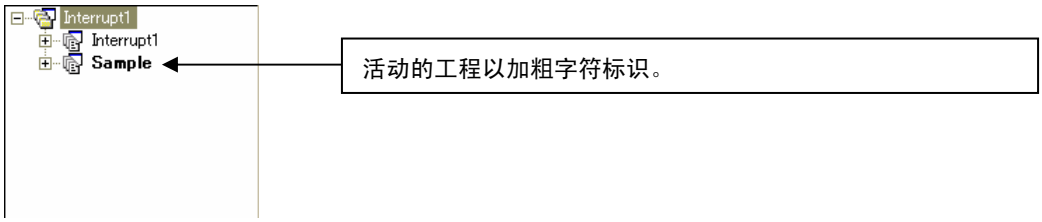
• 描述：

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。

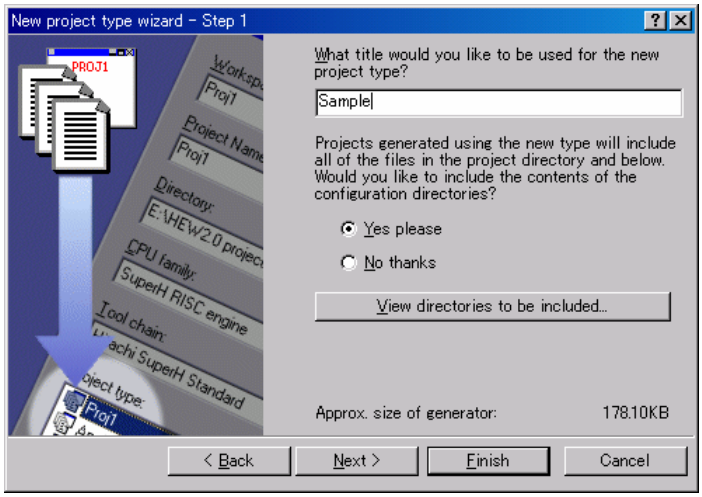
可从模板中获取的信息包括工程文件结构、创建选项、调试程序设定，及其他和工程有关的信息。

• 工程类型的存储方法：

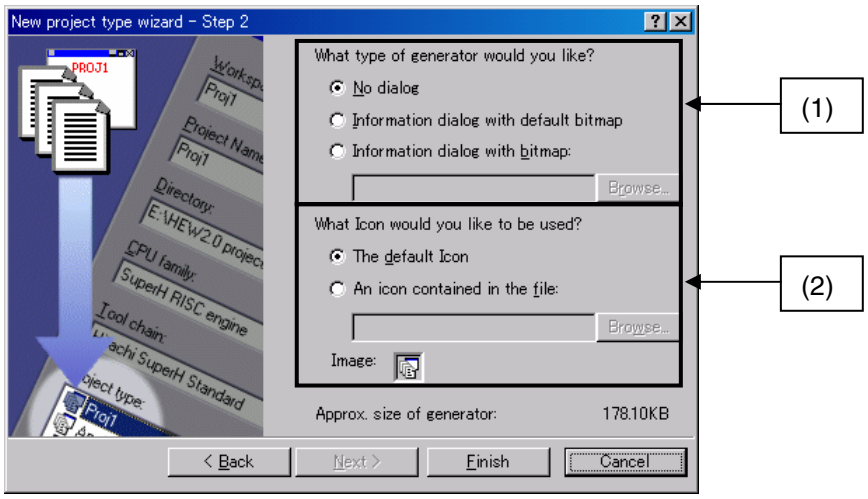
1. 启动您要用以存储工程信息的工程，因为活动的工程将在工作空间打开时接收工程信息。 要启动工程，通过选择 [工程 (Project) -> 设定当前工程 (Set Current Project)] 来选取工程。



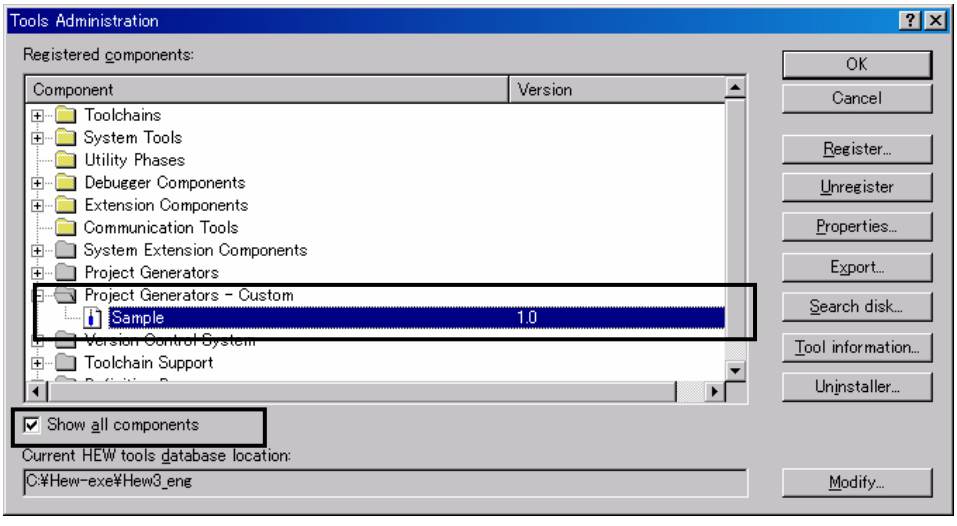
2. 通过选择 [工程 (Project) -> 创建工程类型 (Create Project Type) …] 来打开下列的工程类型向导，为您将用作模板的工程类型指定名称，同时指定是否要将含有后创建可执行文件及其他资源的配置目录，包含在模板内。
   您可通过单击 [完成 (Finish)] 按钮，在此退出工程类型向导。



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。

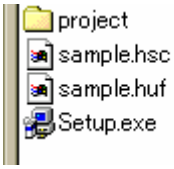


5. 下列对话框将打开。选取用以存储定制工程生成程序模板的目录。该目录必须是空的。
   工程类型的存储过程至此完成。

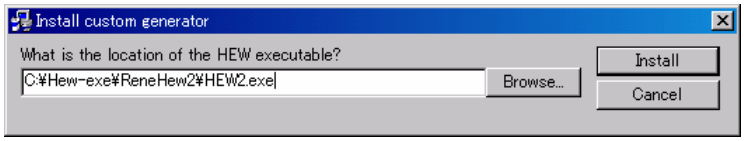
• 安装定制工程生成程序：

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）

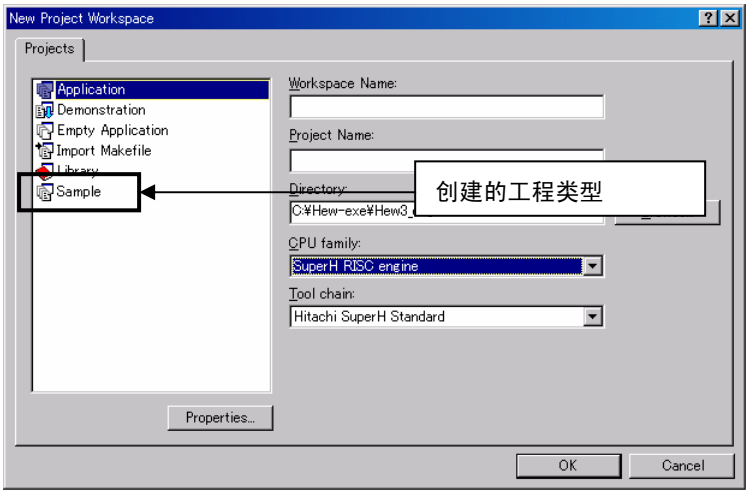


3. 环境已创建完成。

• 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)]。已安装的工程类型被添加到 [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

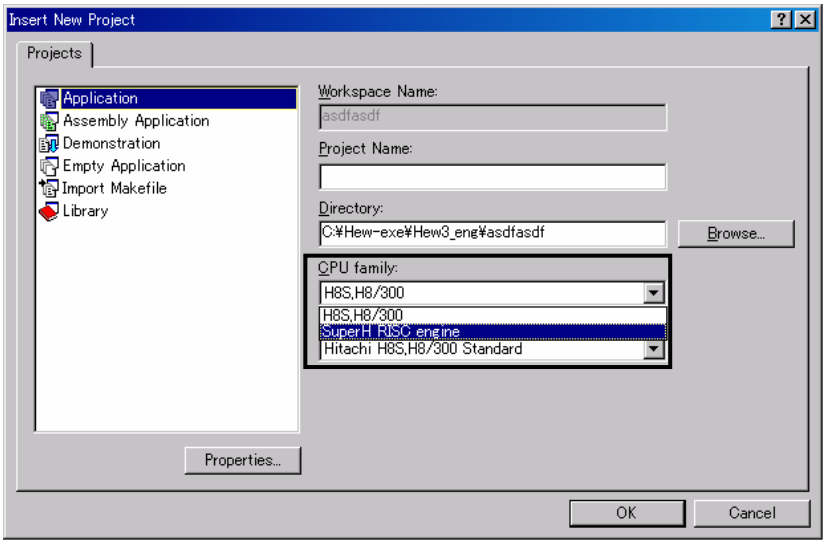
• 描述：

当在工作空间中插入新工程时，您可以插入另一类型的 CPU。这将允许 SH 和 H8 的工程在单一工作空间内被管理。

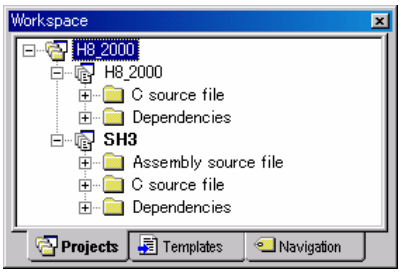
• 插入不同 CPU 家族的实例：

1. 在 SH (H8) 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。您可以在工作空间内放置当前 CPU 类型以外的不同 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：

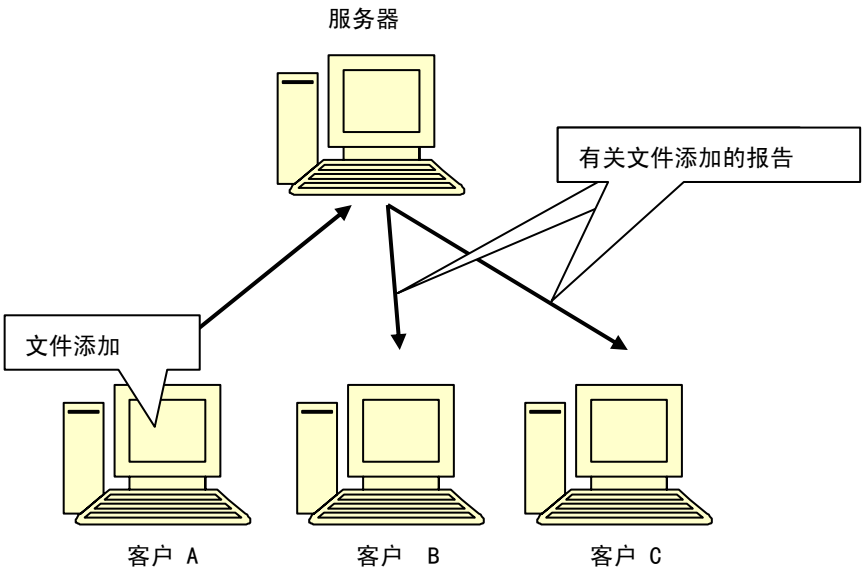
HEW 允许将工作空间和工程，通过网络被不同用户所共享。

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。

此系统将一架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。

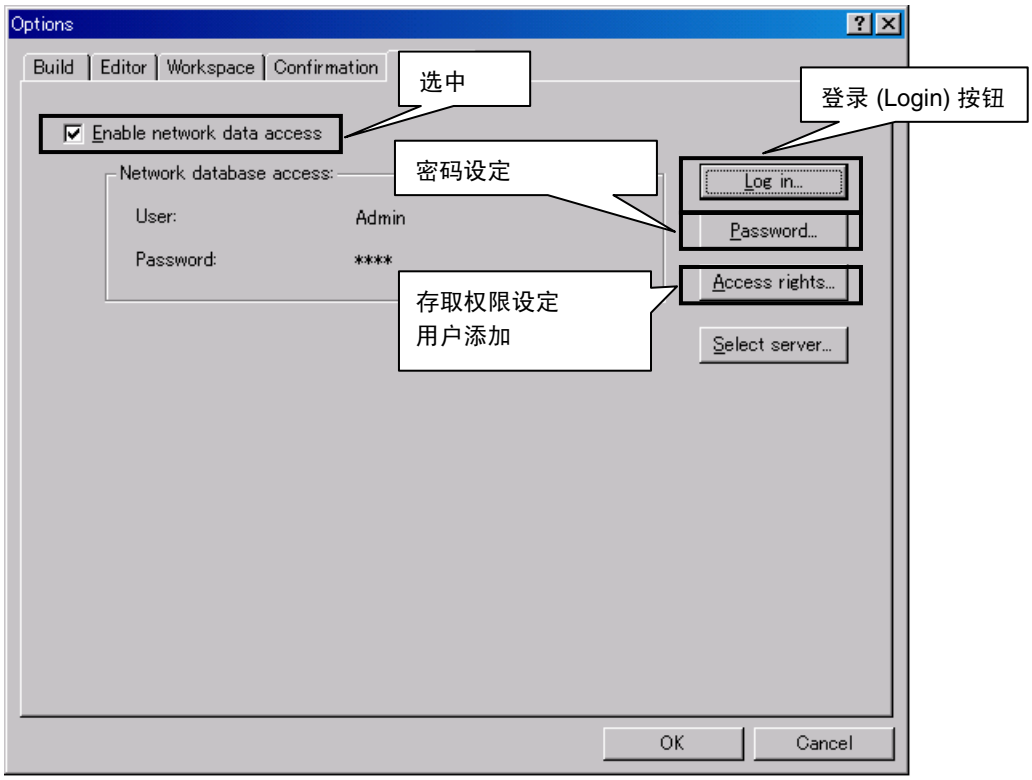
另外，用户可被授予特定工程或文件的存取权限。



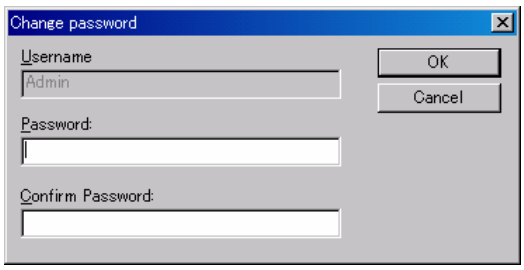
• 网络存取设置：

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. 添加了一位管理员。由于管理员一开始没有密码，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



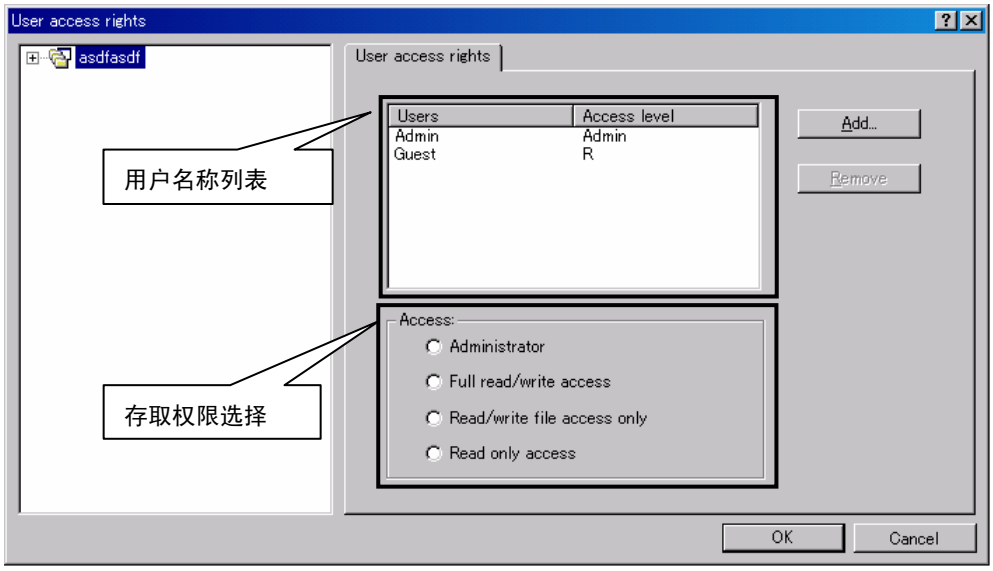
[更改密码 (Change password)] 对话框



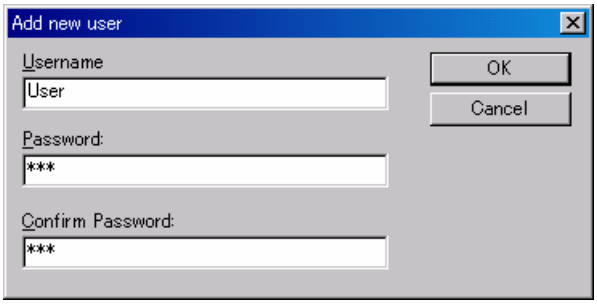
• 添加新用户：

默认情形下，添加了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. 输入新用户名称与密码（密码指定是强制性的）。



• 选取服务器机器

选取将被用作服务器的机器。 若您希望以自己的机器作为服务器，您不需采取任何行动。

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。

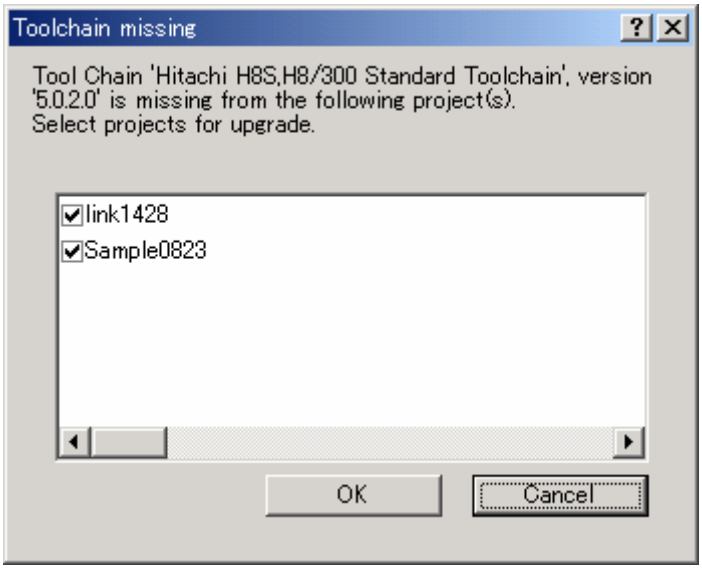
7.1.7 从 HEW 的旧版本转换

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:H8C 3.0C）中创建的工作空间在新版本（如 HEW3.0:H8C 6.0）中打开，下列对话框将会出现。

(1) 检查要升级的工程

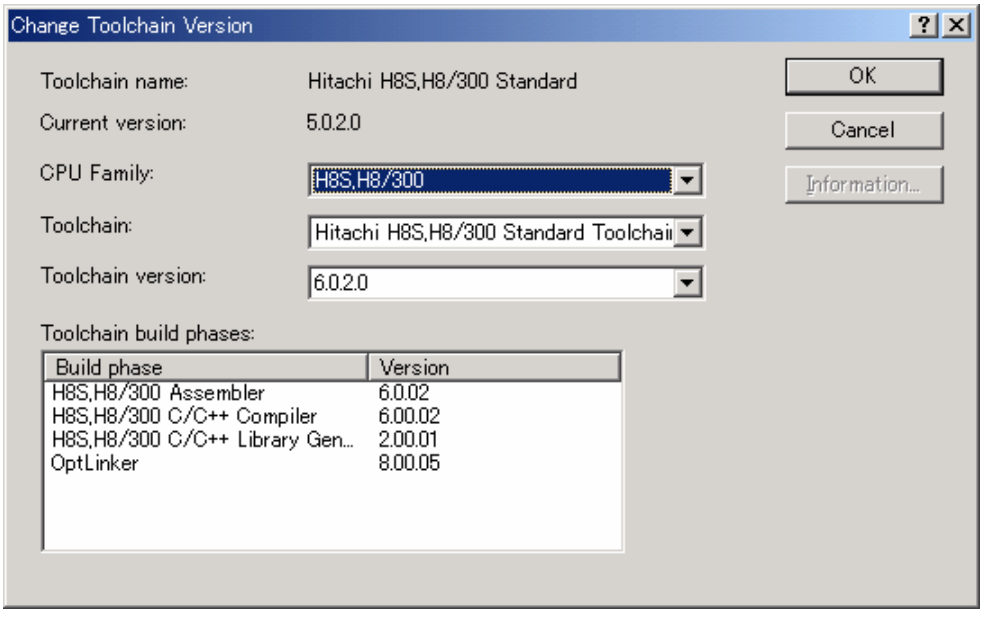
检查要升级的工程的名称。



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。

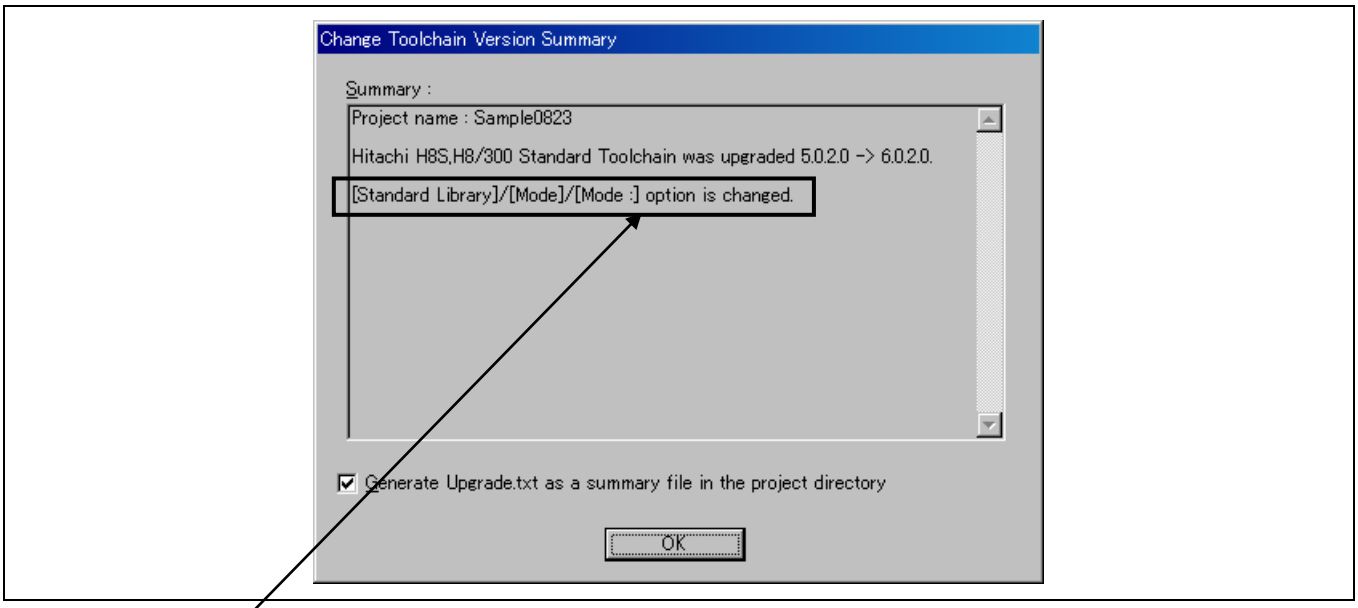


“更改工具链版本” (Change Toolchain Version) 对话框

(3) 确认信息

C/C++ 编译程序 4.0 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，“标准程序库生成程序”（Standard Library Generator）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。

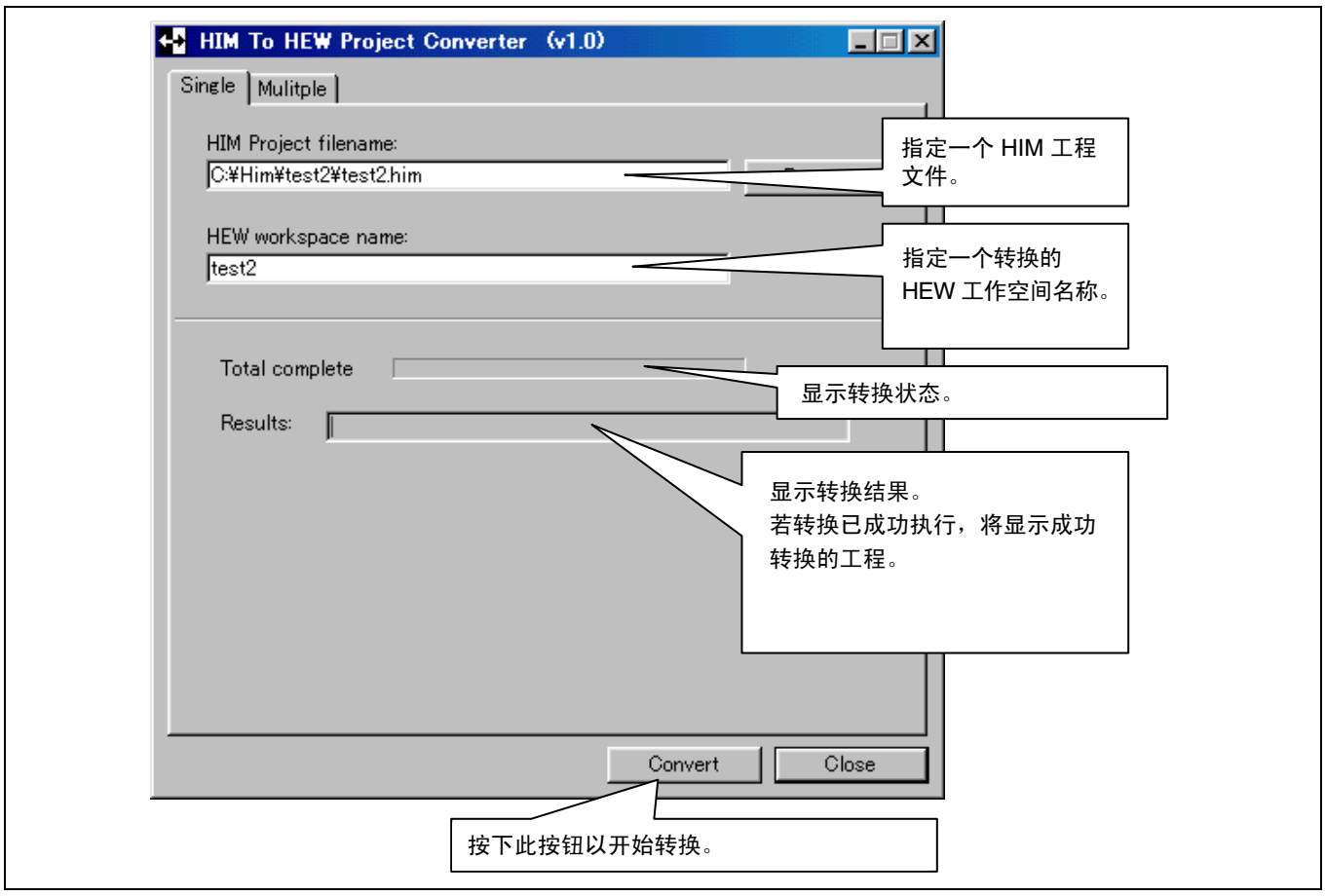
在 Windows® [开始 (Start) 菜单] 上的 [程序(P) (Programs(P))] 中，从 [瑞萨高性能嵌入式工作区 (Renesas High-performance Embedded Workshop)] 选取 [Him 到 Hew 的工程转换器 (Him To Hew Project Converter)]。

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。

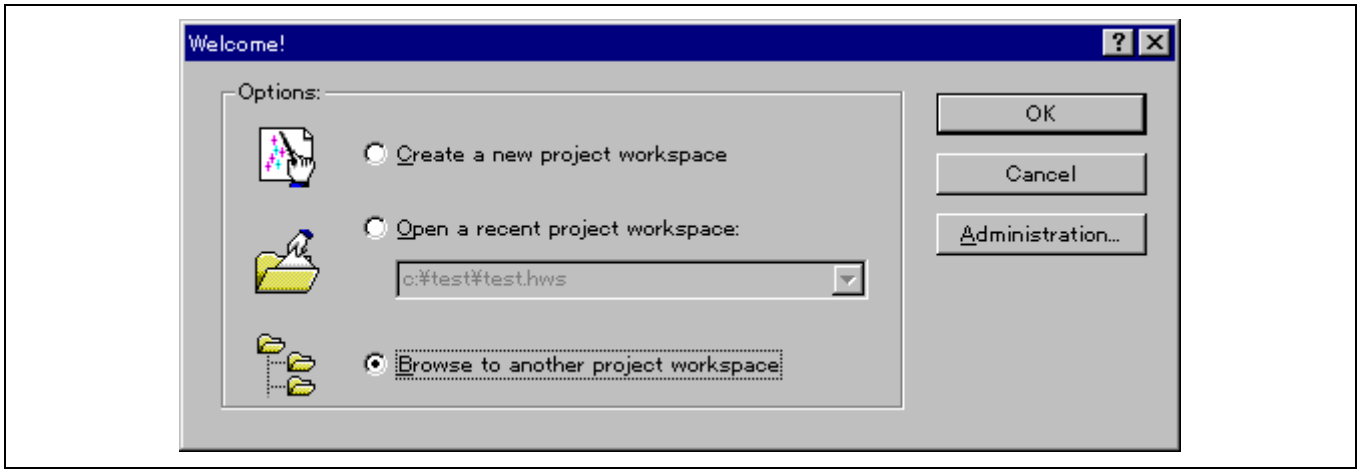
若将多个 HIM 工程转换为 HEW 工程，并将它们成批注册在 HEW 工作空间内时，则请选择多个 (Multiple) 标签。

(1) 单一 (Single) 标签

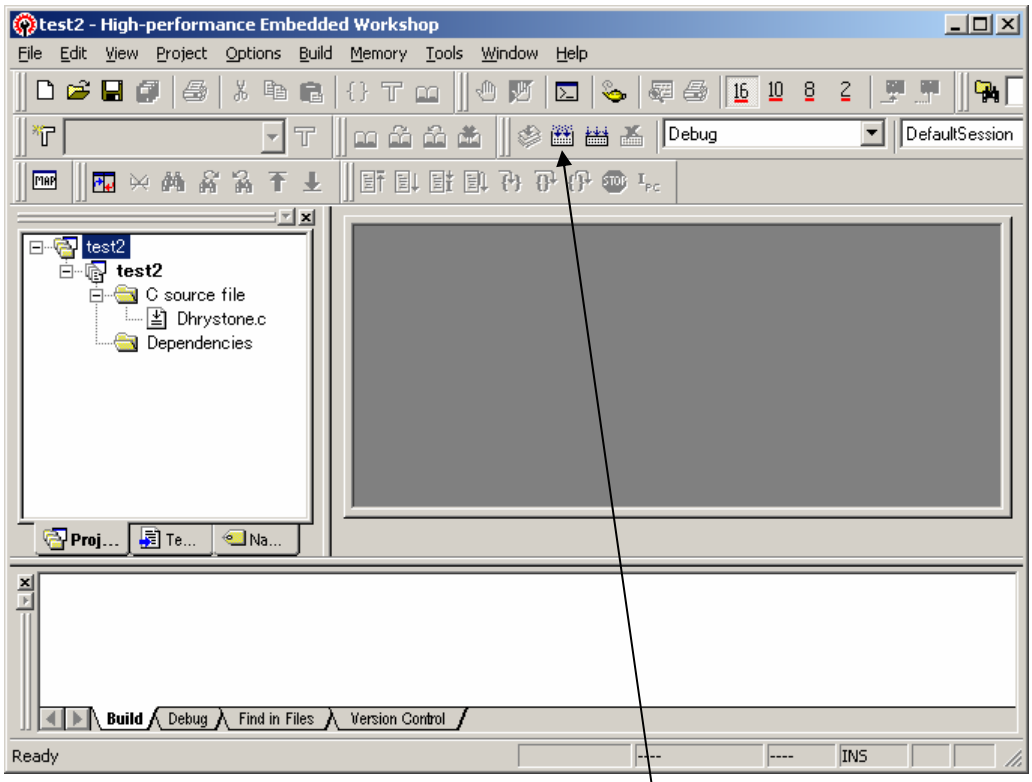


在下一个步骤中，启动 HEW。

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。



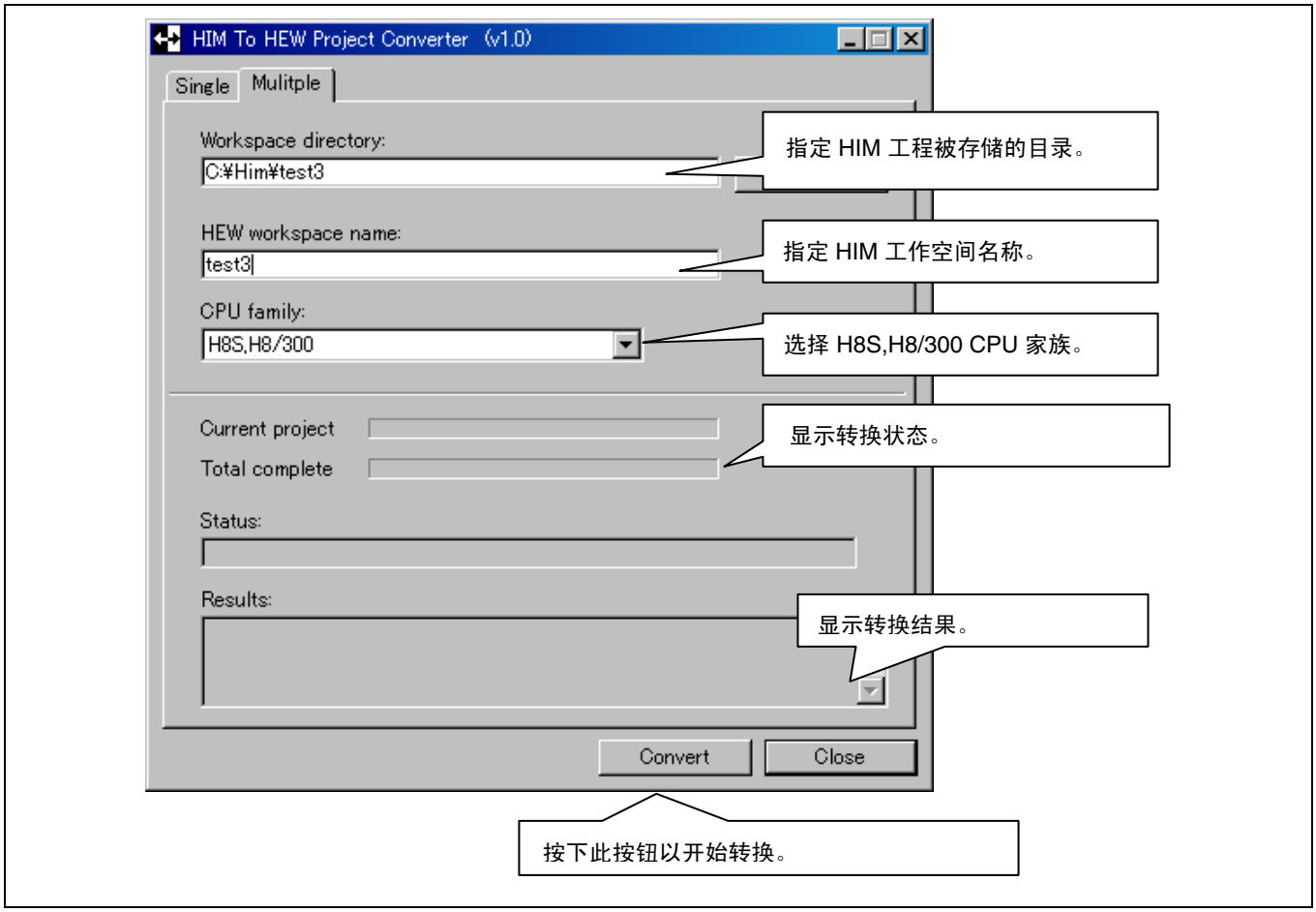
HEW 工程将如以下所示般打开：



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，此工具也可以将生成的文件更新至已修复错误的版本。

• 如何获取**设备更新程序 (DeviceUpdater)**

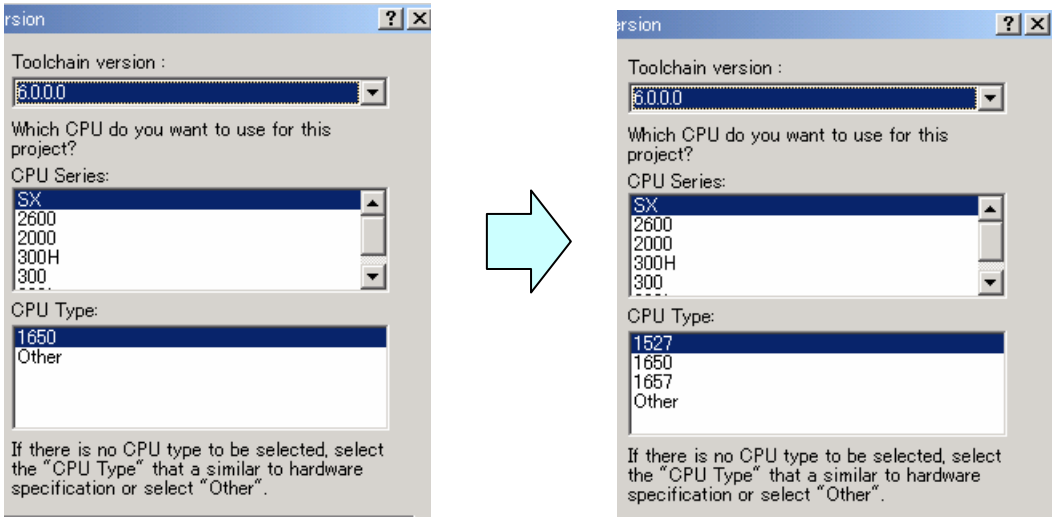
从 Renesas Technology Corp 的下列 URL 下载。

也请参考此页的“注意”。

http://www.renesas.com/

• **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：

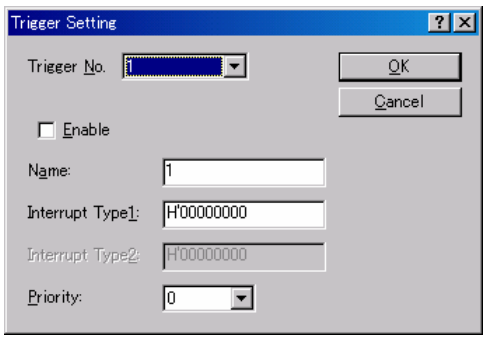
当单击模拟特定中断导因的伪中断按钮时，将造成人为的伪中断。

每个按钮，都指定了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，并选择 [设定 (Setting) …]。将显示 [触发器设定 (Trigger Setting)] 对话框。
   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)]

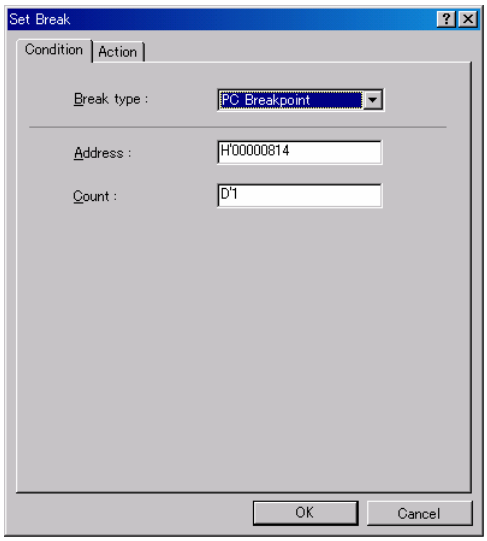
注意： 在 HEW 3.0 或以上版本内，前往 [断点 (Breakpoints)] 视图，然后单击 [软件事件 (Software Event)] 标签。

• 文件输入的设定实例：

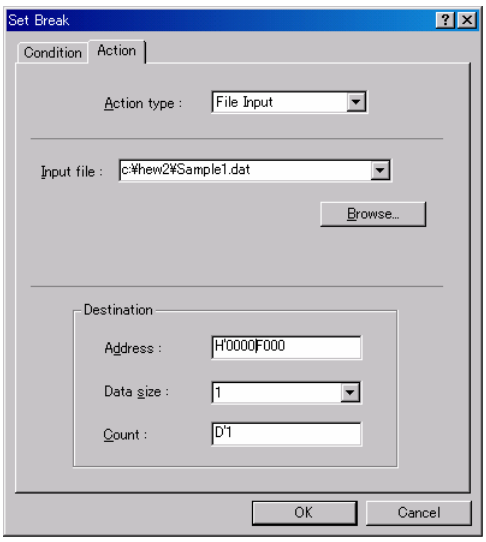
在 [断点 (Breakpoints)] 视图上右击，然后选择 [设定 (Setting) …]，以打开下列 [设定中断 (Set Break)] 对话框。如下所示，PC 断点将被使用，以使 PC 在到达下列地址时，满足中断条件。其他断点类型的设定方法大致相同。

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)



([操作 (Action)] 标签)



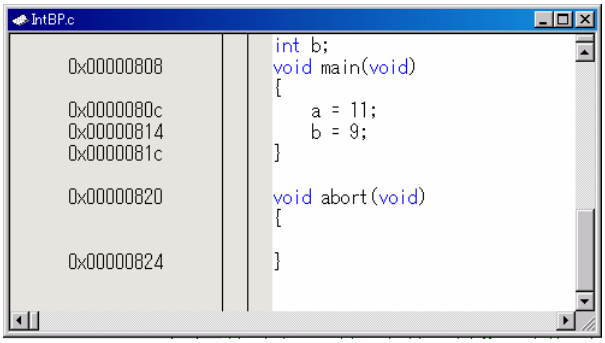
• 文件输入的操作实例：

让我们看看下列实际的操作实例：

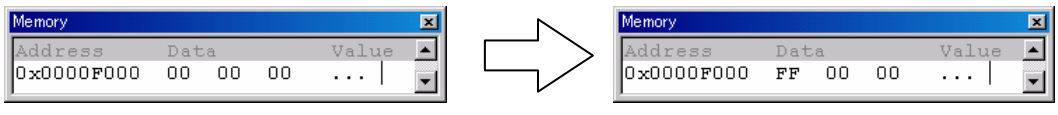
由于以上设定的结果，断点是在 [H'00000814]，同时输入文件包含 [H'FF]。

使用 Go 命令或类似的方法来运行程序。

(源代码段)



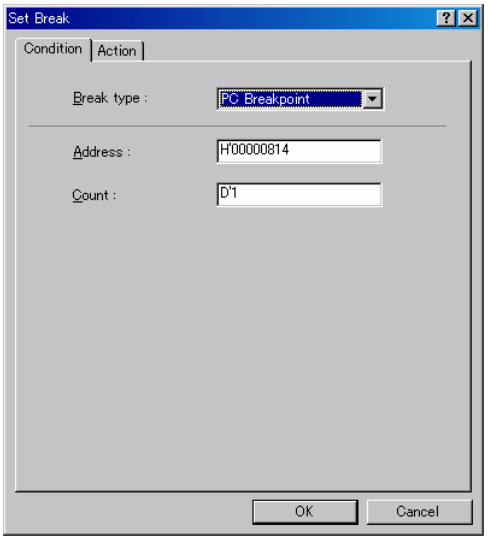
您将看到，当 PC 到达 [H'00000814] 时，由于中断条件被满足，因此地址 H'F000 的存储内容更改。



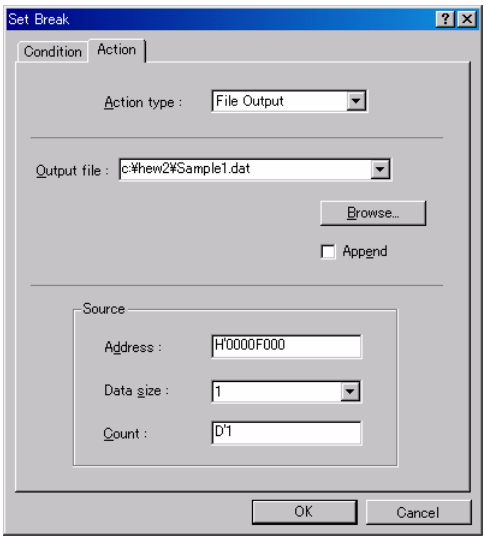
• **文件输出**的设定实例：

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)



([操作 (Action)] 标签)



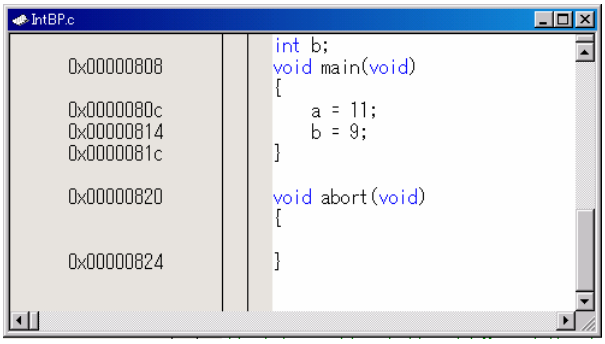
• 文件输出的操作实例：

让我们看看下列实际的操作实例：

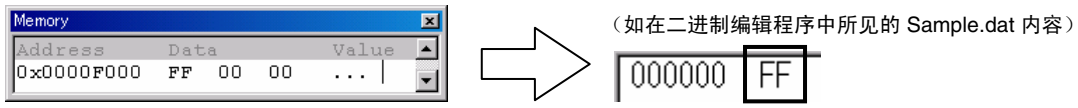
由于以上设定的结果，断点是在 [H’00000814]，同时地址 H’F000 的内容是 [H’FF]。

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。

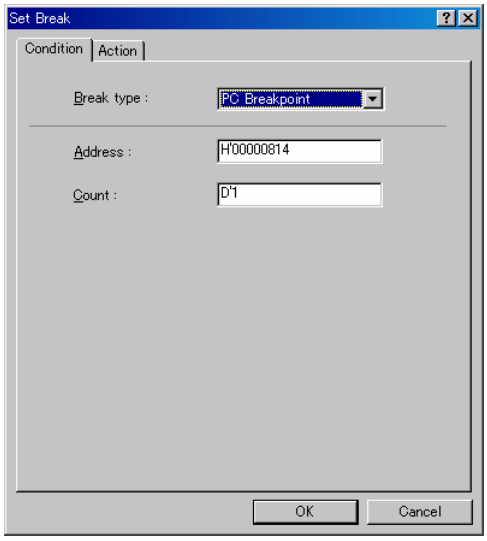


• **中断**设定实例：

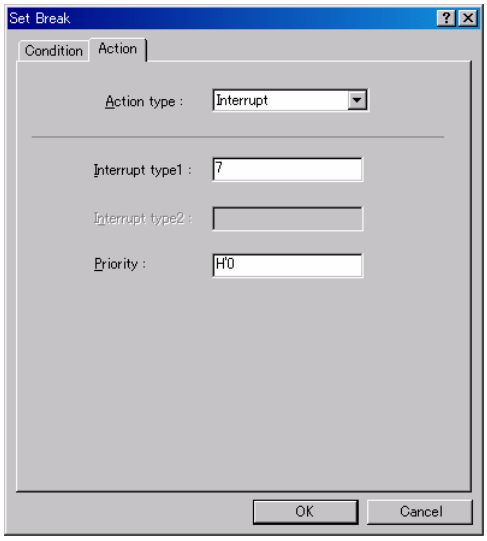
在 [设定中断 (Set Break)] 对话框中设定文件输出的方法，与设定文件输入的方法雷同。 如下所示，使用 PC 断点，以使中断条件在 PC 到达下列地址时获得满足。 其他断点类型的设定方法大致相同。

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)



([操作 (Action)] 标签)



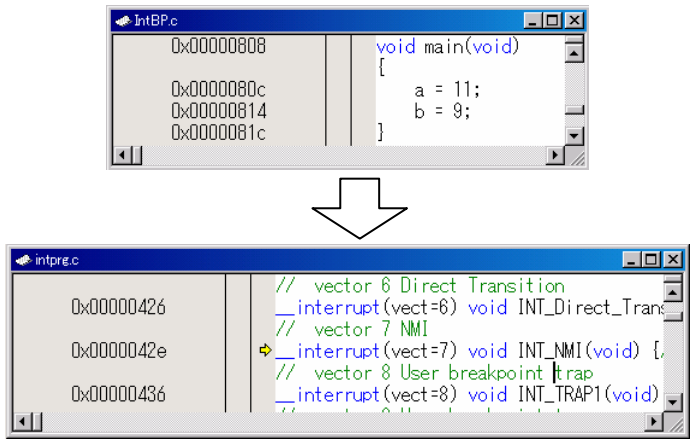
• **中断**操作实例：

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。

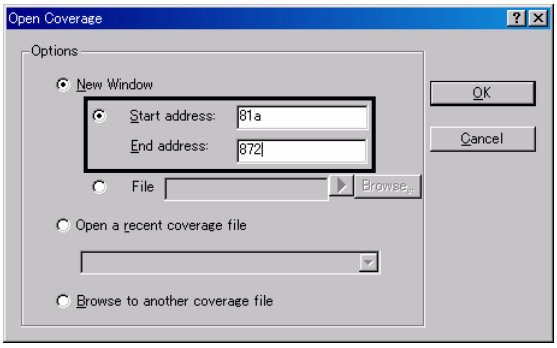
• 如何开启 [打开覆盖 (Open Coverage)] 对话框：

[[视图 (View) -> 代码 (Code) -> 覆盖 (Coverage) …]

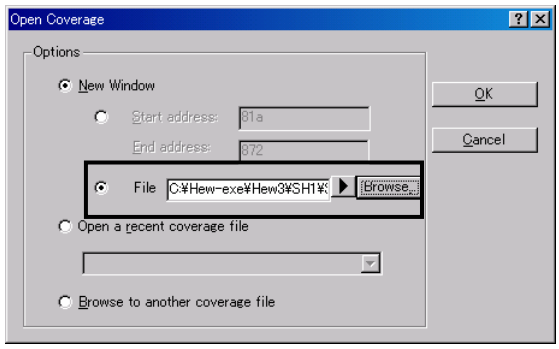
• 如何收集新的覆盖信息：

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。

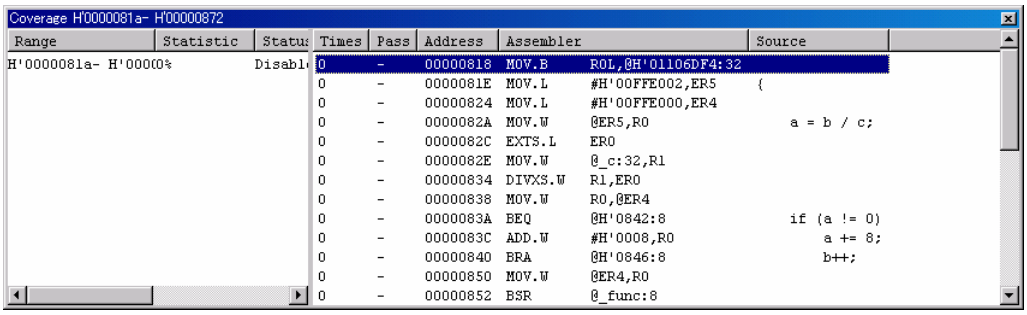
(指定地址)



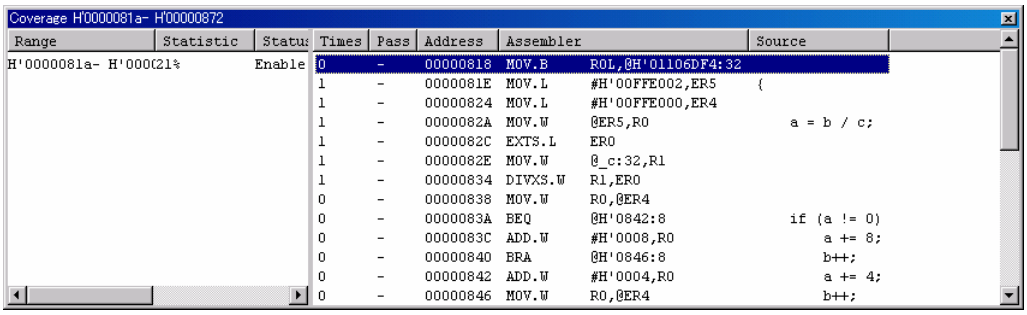
(指定文件名称) * 受 HEW 3.0 或以上版本支持



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。

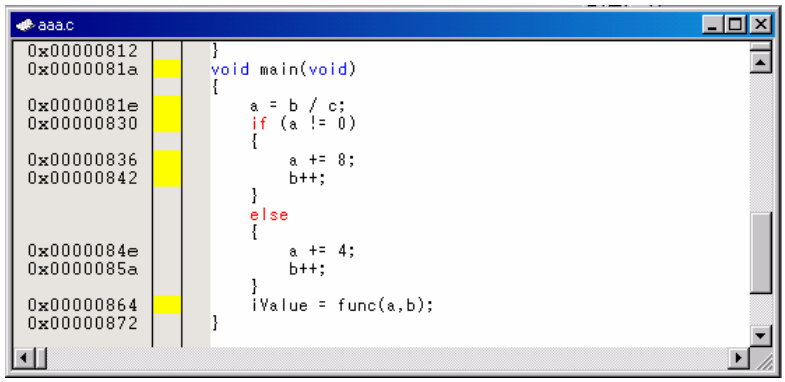


3. 让我们来运行程序。在覆盖视图右边，留意到 [时间 (Times)] 列更改为 1 的行。这表示与此行对应的地址上的语句已被执行。
在视图左边，显示了地址范围内的 C0 覆盖值。



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：

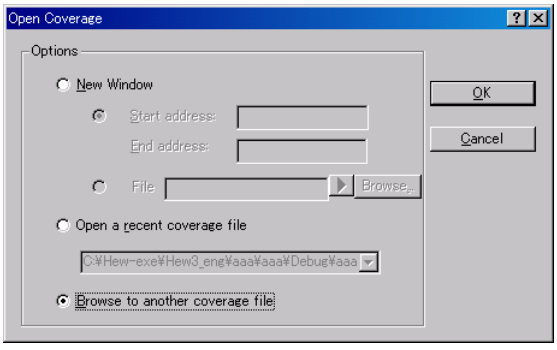
要保存覆盖信息，在覆盖视图右边单击鼠标右键，然后输入具备 *.cov 的扩展名的文件名称。

• 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。

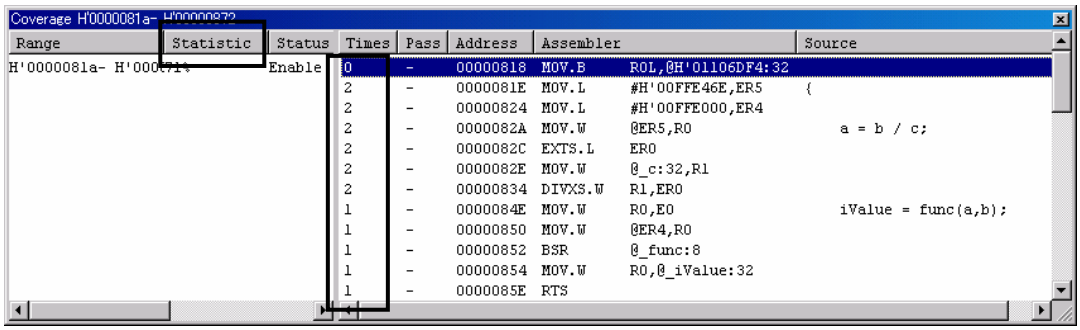
您可能要在重复收集覆盖的步骤时，增加覆盖的比率，并在不同的测试条件下执行。

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 Renesas Technology Corp URL 上的“文件可操作的模拟程序与调试程序低层界面例程指导”(Guideline for File Operatable Low-Level Interface Routines for Simulator and Debugger) 页下载文件。

http://www.renesas.com/

• 如何创建环境：

(1) 如何创建环境：
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)],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。

   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):

（样品程序代码）

```
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：

（stdout 的内容）

```
***** 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++ )
     {
        fp->_bufptr = NULL;
        fp->_bufcnt = 0;
        fp->_buflen = 0;
        fp->_bufbase = NULL;
        fp->_ioflag1 = 0;
        fp->_ioflag2 = 0;
        fp->_iofd = 0;
     }



   if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdin",  "r", 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;
}
```

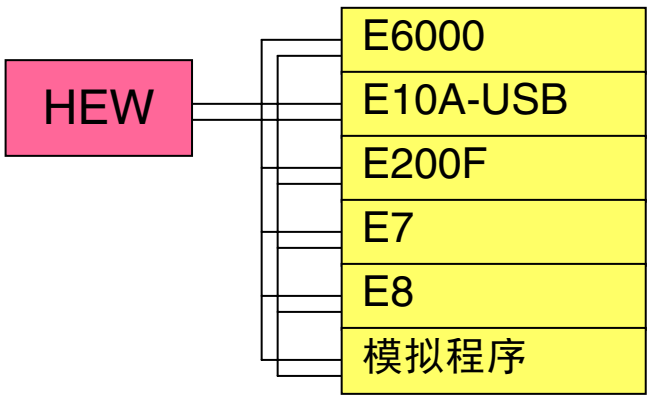
7.2.5 调试程序目标同步

• 描述：

HEW 允许您在单个 HEW 示例上，调试多个目标。

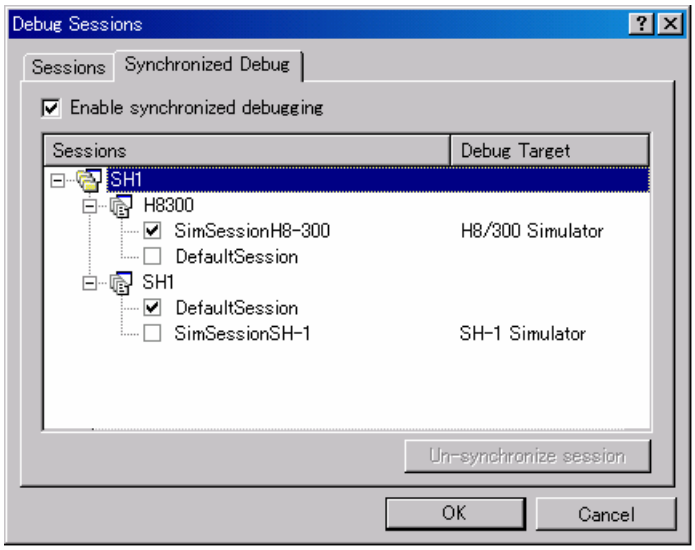
这意味着您可以同时调试多个目标，并使它们相互同步。

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。

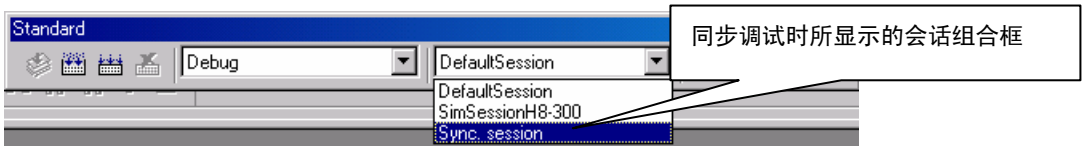


• 如何同步化调试程序目标：

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：

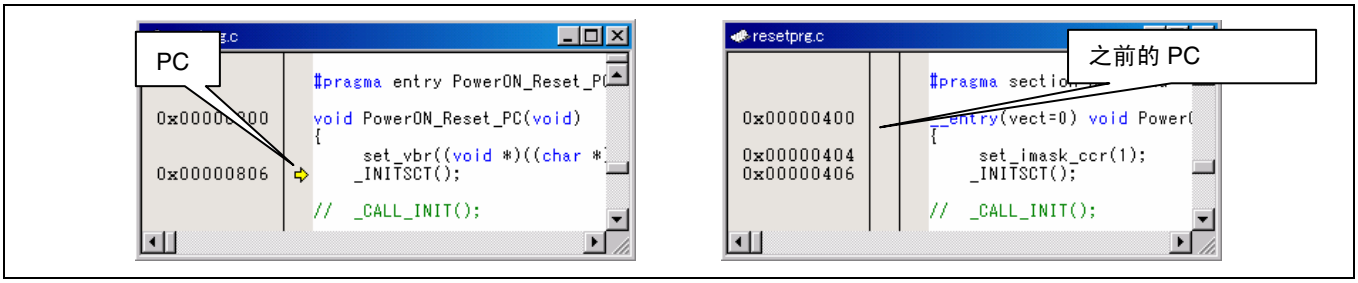
用户操作	目标调试程序会话 1	目标调试程序会话 2
在其中一个会话期间 [运行 (Run)]	"Run"	"Run"
在其中一个会话期间 [逐步执行 (Step)]	"Step"	"Step"
在其中一个会话期间按下 ESC	"Stop"	"Stop"
-	因为断点或用户程序错误而“Stop”	Stop（和按下 ESC 时相同）
-	Stop（和按下 ESC 时相同）	因为断点或用户程序错误而“Stop”
在其中一个会话期间 [复位 CPU (CPU reset)]	"CPU reset"	"CPU reset"

• 同步化调试实例

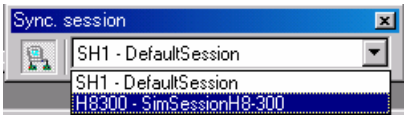
下面提供一个执行逐步命令的实例。

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

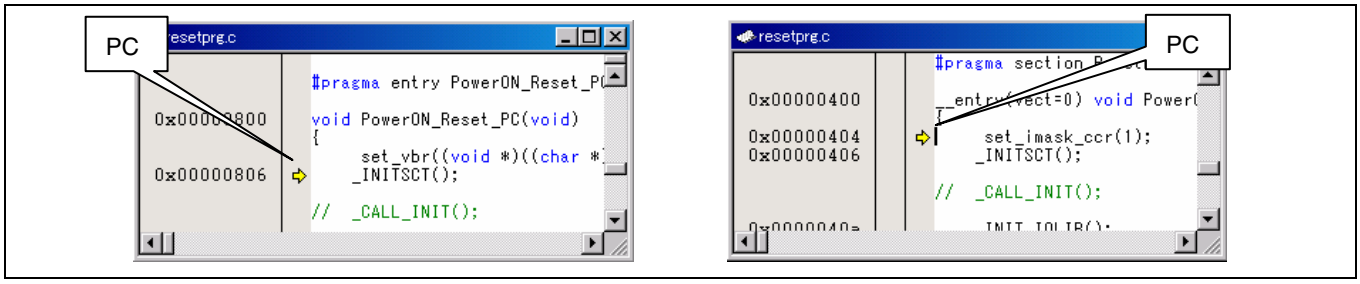


2. 使用 [同步会话 (Sync. session)] 工具栏更改会话。



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。

• 受支持的定时器控制寄存器

下表中“受支持的”列，O 标示该寄存器受支持，而 Δ 标示只有与 [描述 (Description)] 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
H8SX	TPU0	TSTR	Δ
		TCR	Δ
		TIER	O
		TSR	O
		TCNT	O
		TGRA	O
		TGRB	O
		TGRC	O
		TGRD	O

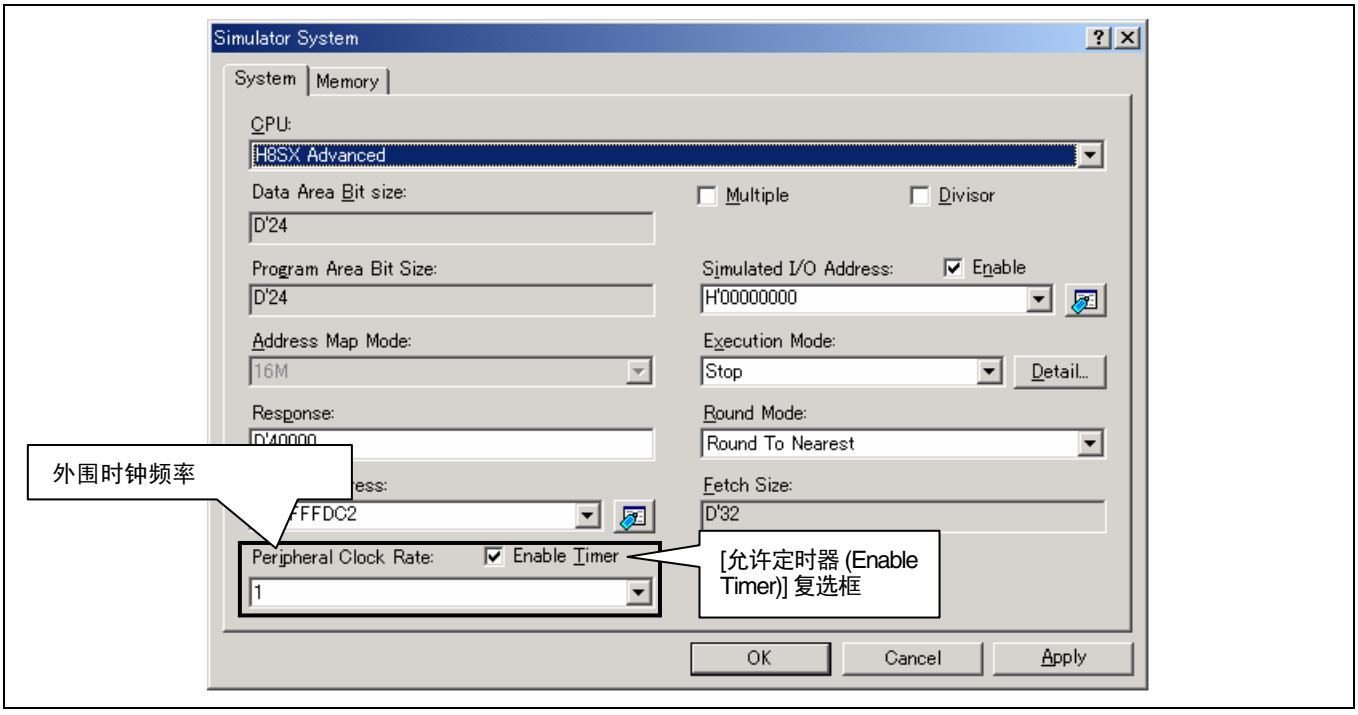
• 受支持的中断优先级设定寄存器

下表中“受支持的”列，O 标示该寄存器受支持，而 Δ 标示只有与 [描述 (Description)] 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
H8SX	IPRF	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。

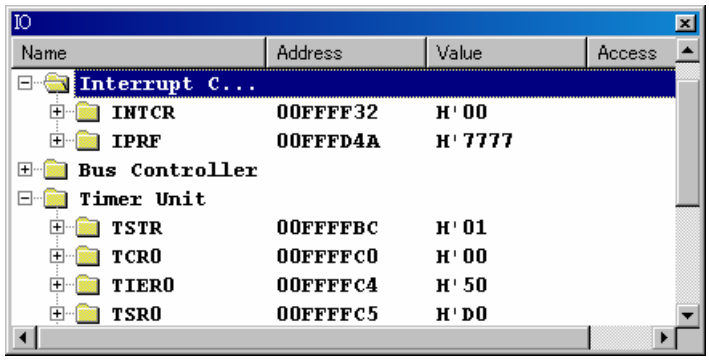
```
// TPU0 start
TPU.TSTR.BIT.CST0 = 1;
// TPU0 Overflow interrupt enable
TPU0.TSR.BIT.TCFV = 1;
TPU0.TIER.BIT.TCIEV = 1;
while(1);
```

允许定时器 ITU0。

注意： 在将值设定到定时器寄存器前，确认可在模拟程序系统对话框之存储器标签上存取定时器寄存器。若存取不被允许，您将无法设定值到寄存器，同时也不能使用定时器。

• 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 3.1 或以上版本支持。

这仅在 H8SX 中有效。

7.2.7 定时器的使用实例

• 描述：

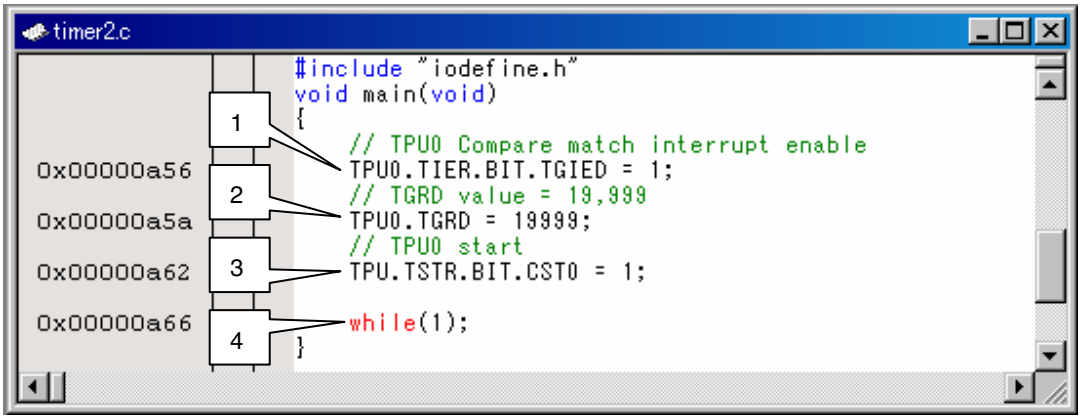
本小节使用 H8SX/1650 (H8SX) 中的 TPU 作为例子，概述如何使用比较匹配和循环控制程序中断。

• 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。



[中断生成程序的说明]

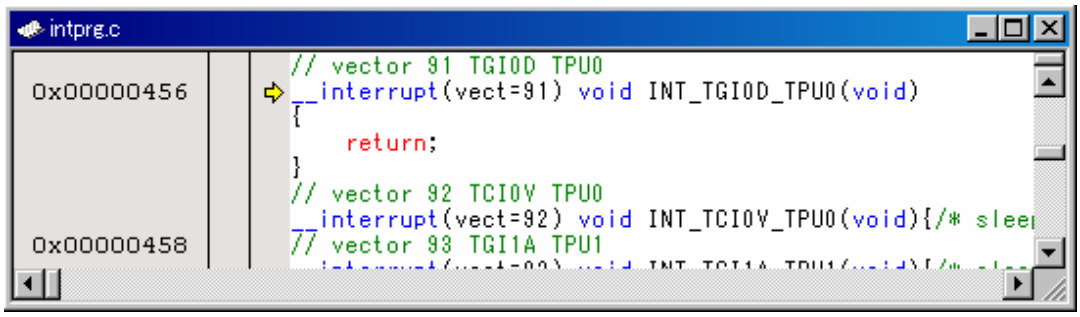
1. 当 TIER（启用定时器中断的寄存器）中的 TGIED（TGR 中断启用 D）位变成 1 时，中断将被启用。
2. 设定 TGRD 的值。
3. 启动 TPU0 定时器。
4. 等待直到 TCNT0 与 TGRD 的值匹配（等待比较匹配）。

• 程序执行：

等待直到 TCNT0（定时器计数器 0）和 TGRD（定时器一般寄存器 D）在名为“中断生成程序的说明”段落中的步骤 4 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。

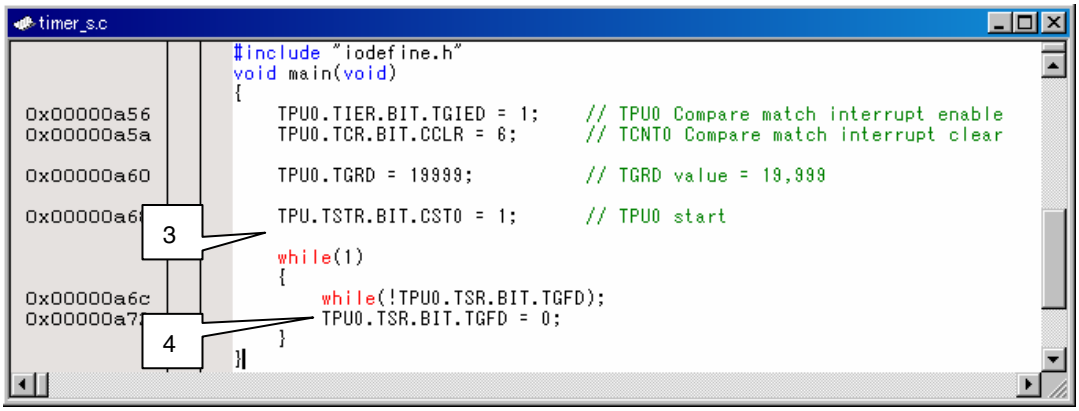


• 包含用于循环处理程序的代码之样品程序

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 IPRF（中断优先级寄存器）中的中断优先级。



1. 当 TIER（启用定时器中断的寄存器）中的 TGIED（TGR 中断启用 D）位变成 1 时，中断将被启用。
2. 设定 TGRD 的值。
3. 启动 TPU0 定时器。
4. 清除比较匹配标志。

• 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。

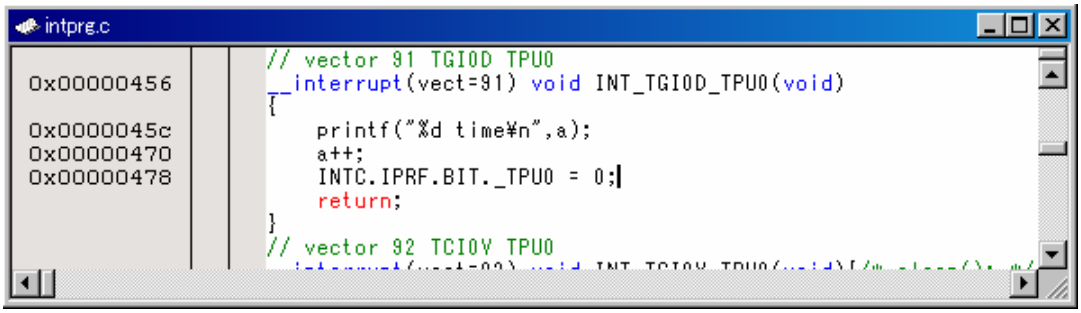
中断处理可以在这种方式中完成。

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，PC 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 PC。



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：

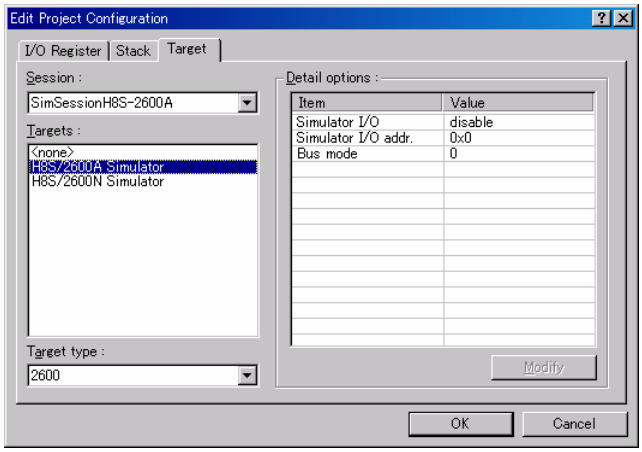
HEW 菜单：**工程 (Project) > 编辑工程配置 (Edit Project Configuration) ...**

• 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

Call Walker（堆栈分析工具）通过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)，来显示堆栈数。

对于无法在堆栈信息文件中输出的汇编程序的堆栈数，该信息可使用编辑函数来添加或修改。

另外，整个系统的堆栈数可被计算。

被编辑的堆栈数上的信息可作为调用信息文件 (*.cal) 被保存和读取。

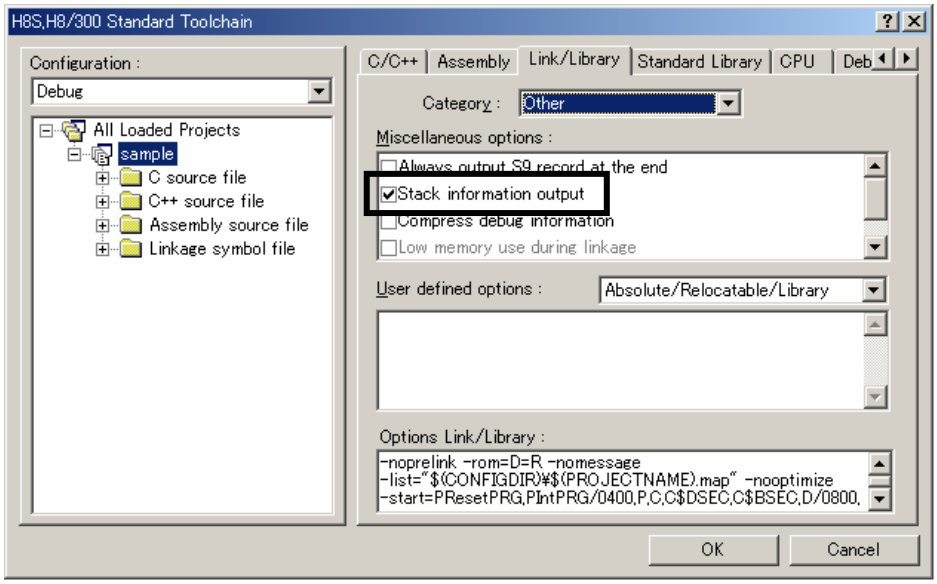
同时，一些调用信息文件可被合并。

7.3.1 制作堆栈信息文件

遵循以下步骤，您可以制作堆栈信息文件或配置文件信息文件。

• 制作堆栈信息文件 (*.sni)

您可使用优化连接编辑程序的下列选项，来制作堆栈信息文件。



指定方法

对话框菜单： **连接/程序库 (Link/Library) 标签类别： [其他 (Other)] 堆栈信息输出 (Stack information output)**

命令行： *STACk*

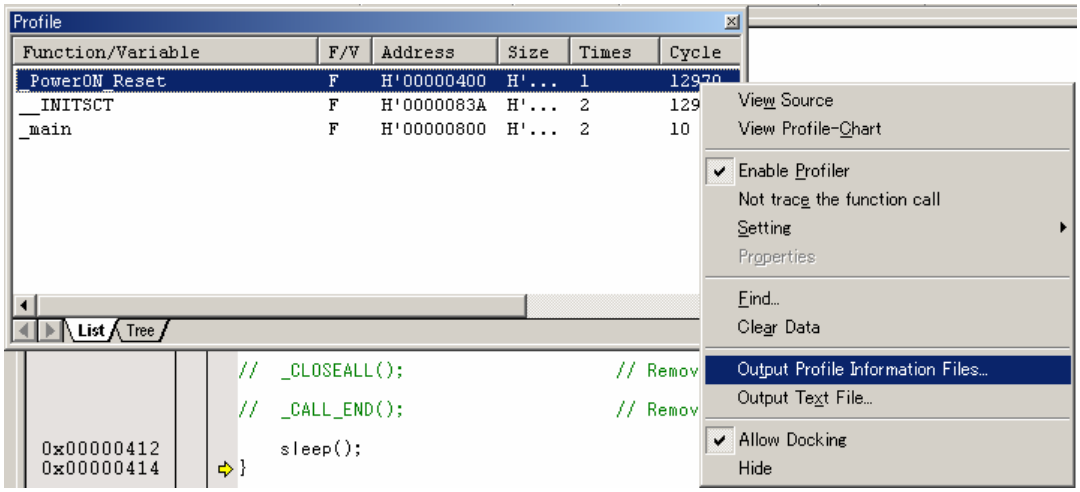
• 制作配置文件信息文件 (*.pro)

使用下列 [配置文件 (Profile)] 函数执行用户程序。

在执行后，在 [配置文件 (Profile)] 窗口上单击鼠标右键，然后选择 [输出配置文件信息文件 (Output Profile Information Files) …]，以便制作配置文件信息文件 (*.pro)。

要获取有关制作配置文件信息文件的更多信息，请参考“H8S，H8/300 系列高性能嵌入式工作区 3 用户手册 (H8S, H8/300 Series High-performance Embedded Workshop 3 User’s Manual)”4.12 节，查看函数调用历史记录 (Viewing the Function Call History)。

选择 [视图 (View) -> 性能 (Performance) -> 配置文件 (Profile)] 以打开 [配置文件 (Profile)] 窗口。



7.3.2 启动 Call Walker

使用下列步骤以启动 Call Walker。

• 从开始 (Start) 菜单启动

单击 [程序 (Program) -> 瑞萨高性能嵌入式工作区 (Renesas High-performance Embedded Workshop) -> Call Walker]

• 从 HEW 启动。

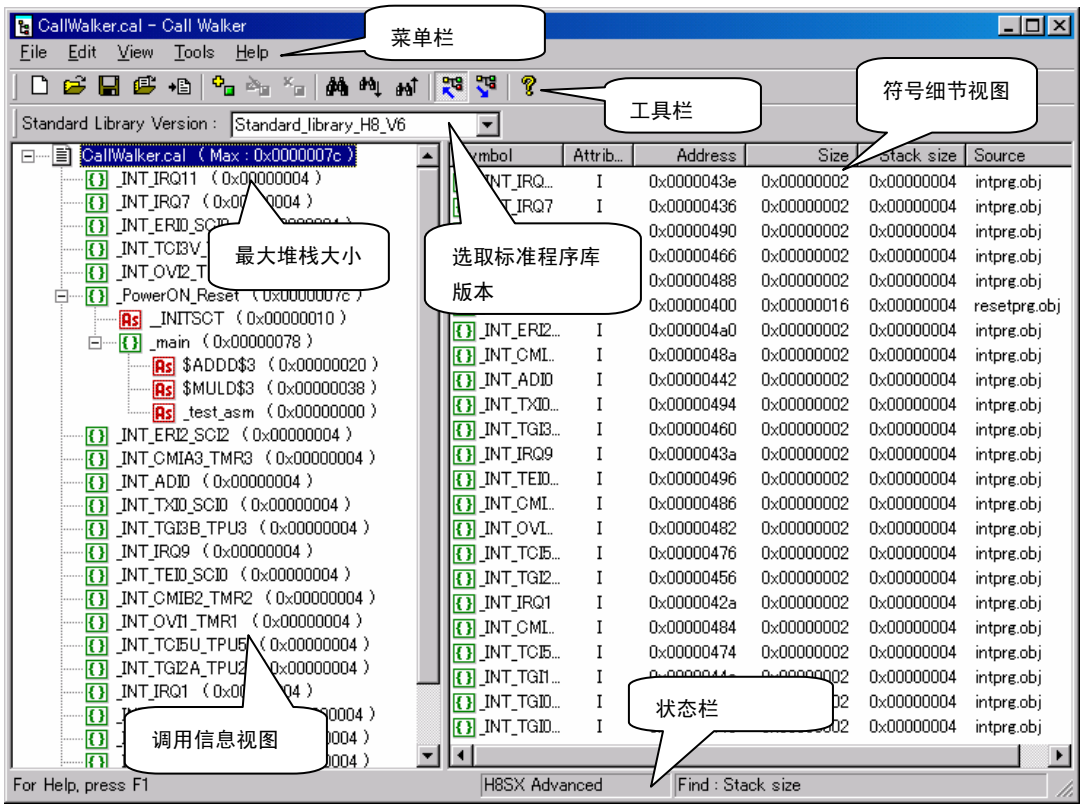
单击 [工具 (Tools) -> Call Walker]

7.3.3 文件打开和 Call Walker 窗口

在启动 Call Walker 后，选择 [文件 (File) -> 导入堆栈文件 (Import Stack File) …] 以打开堆栈信息文件 (*.sni)，或配置文件信息文件 (*.pro)。

选择 [文件 (File) -> 打开 (Open) …] 以打开现有的调用信息文件 (*.cal)。

之后，将显示下列窗口。



注意：

除了标准程序库之外的汇编函数的堆栈数被显示为零。

参考 7.3.4 节，编辑堆栈信息文件，以设定堆栈数量。

• 调用信息视图

符号间的连接级结构被显示。

堆栈所使用的数目被显示在每个符号的左边。

(1) 符号显示

符号归类（类别）符号以图标显示在每个符号的左边。

符号归类（类别）符号如下：

 : 编辑文件

 : 汇编程序标签

 : C/C++ 函数

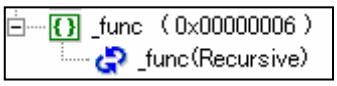
 : 递归调用函数或流通函数

(a) 递归调用函数

下面显示，当在函数内调用相同函数时的情形。

实例：

```
void func(int x)
{
    x++;
    if(x != OFF)
        func(x);

    if(x == MAX)
        return;
}
```

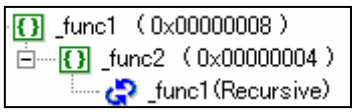


(b) 流通函数

下面显示，当间接调用相同函数时的情形。

实例：

```
void func1(int a)
{
    func2(10);
}
void func2(int b)
{
    func1(9);
}
```



: RTOS 函数（实例： ITRON 符号）

: 参考源为未知的函数。 （被未知所参考）

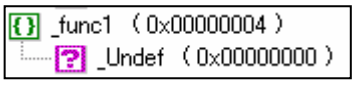
在下例中，函数 (**func1**) 调用函数 (**Undef**)。 当函数 (**Undef**) 不被定义时，此图标显示在函数 (**Undef**)。

未定义的函数调用在连接时出现错误，但连接选项 **change_message** 可将错误修改为警告。 装入模块文件在警告时产生，于是堆栈信息文件也产生。

要获取有关 **change_message** 的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”4.2.7 节，其他选项，Change_message (Other Options, Change_message)。

实例：

```
void func1(void)
{
    Undef();
}
```

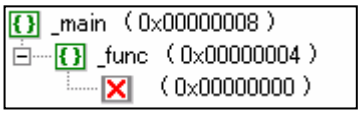


: 地址参考未被分解的函数（地址未分解）。

下面显示，当函数被表调用时的情形。

实例：

```
static int (*key[3])()=
                {nop, stop, play};
void func(int x)
{
    (*key[a])();
}
```



: 被省略的符号

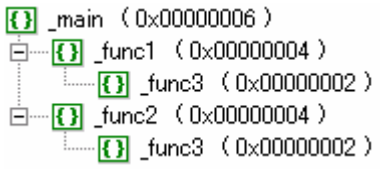
由于此工具显示所有连接层，若大小是庞大的，则所显示的数目将非常大。

因此，只有第一层被显示，其他部分被省略符号所省略，以缩减显示的数目。

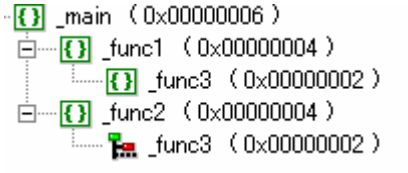
选择 [视图 (View) -> 显示全部符号/显示简单符号 (Show All Symbols/Show Simple Symbols)] 以切换此显示格式。

实例：

显示全部符号



显示简单符号



• 符号细节视图

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

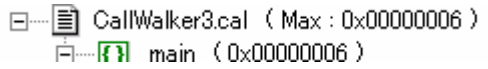
对于每个符号，将显示地址、属性，及堆栈所使用的数目。

在单击符号后，单击鼠标右键以执行各项编辑命令。

• 状态栏

将显示函数信息、CPU 类型，及当前堆栈信息文件的其他信息。

• 最大堆栈大小



将显示当前堆栈信息文件的堆栈所使用的静态最大数目。

• 选取标准程序库版本

可选取当前堆栈信息文件的标准程序库版本。

使用此信息，将显示标准程序库中汇编函数的堆栈数目。

若只安装了一个 HEW 封装，则无需选取此项。

7.3.4 编辑堆栈信息文件

当在符号细节视图（画面上的右边框架）中选取了符号后，在编辑 (Edit) 菜单内选择添加 (Add) …、修改 (Modify) …、删除 (Delete) …，以添加、更改、删除符号。

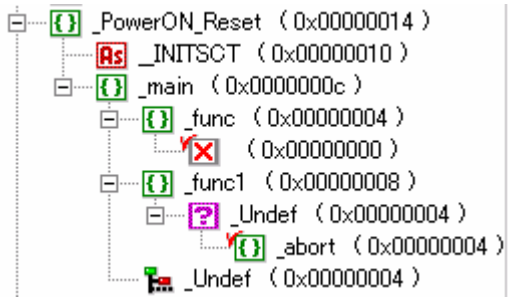
在符号细节视图单击鼠标右键，以执行相同的编辑命令。

此工具可测量堆栈所使用的静态最大数目。

在多个中断的情形下，用户应编辑文件信息，以测量堆栈所使用的动态最大数目。

在调用信息视图（画面上的左边框架）中拖放符号，以将之移动。

当符号被移动或编辑后，一个选中标记将在调用信息视图（画面上的左边框架）中的符号旁边显示如下：



编辑命令将在下一节中说明。

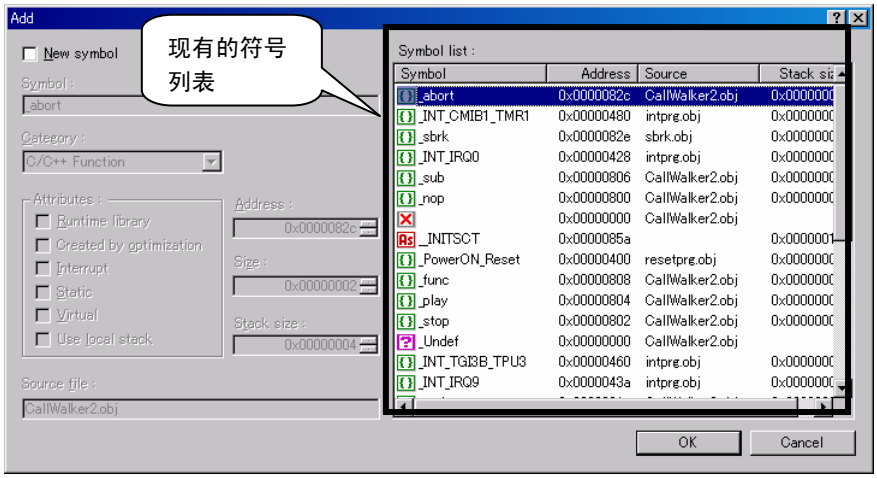
• 添加 (Add) 命令

(1) 添加现有的符号

单击 [添加 (Add) …] 以显示下列对话框。

右边框架是当前文件的现有符号列表。

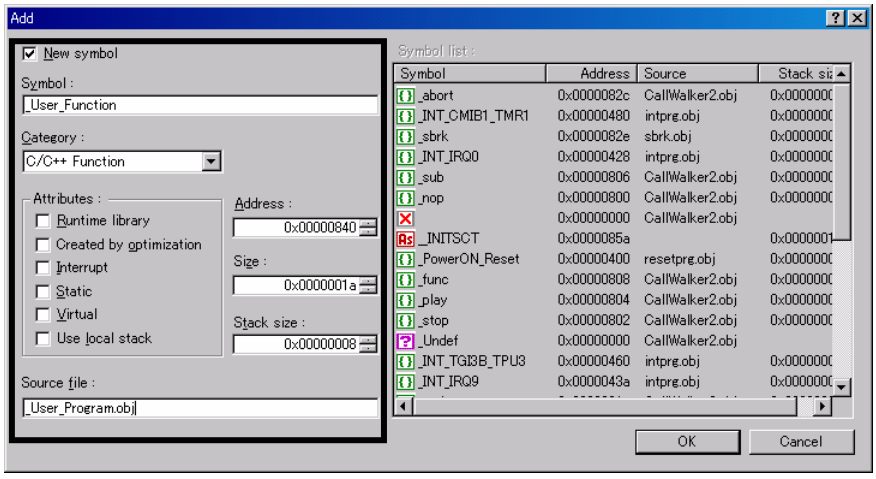
在此列表中选择一个符号，然后单击 [确定 (OK)] 按钮，以添加现有的符号。



(2) 添加新的符号

选中下列 [新符号 (New symbol)] 复选框，以添加新的符号。

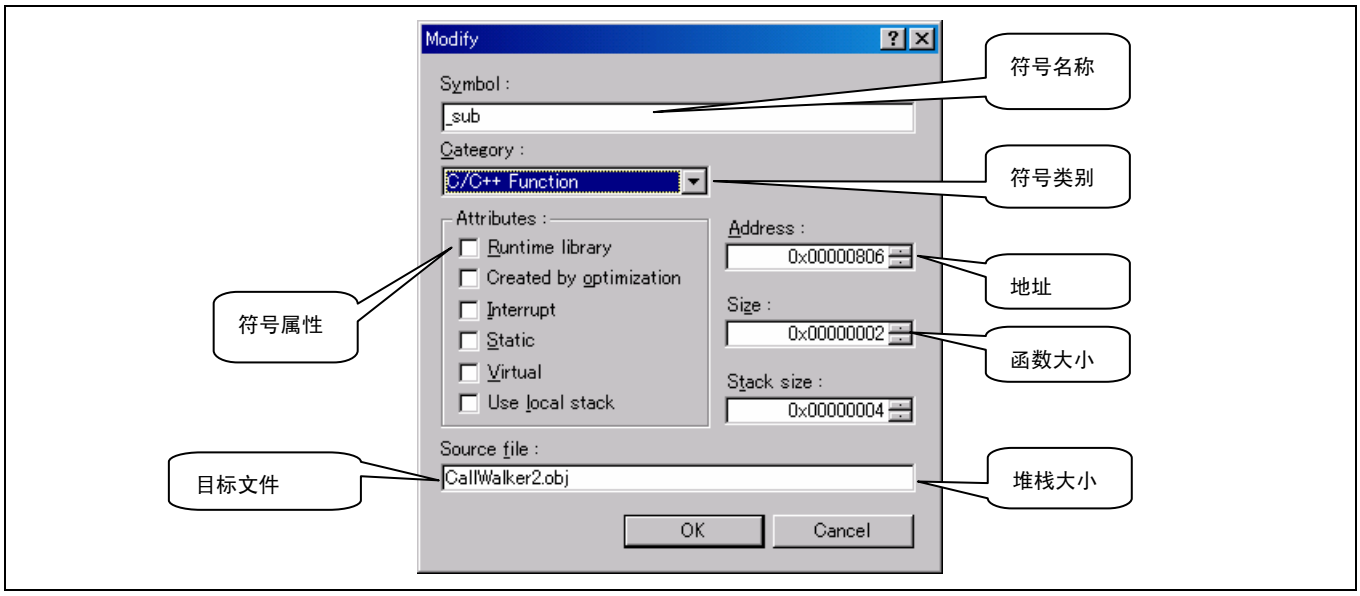
可在此指定符号名称、类别、属性、地址、堆栈大小。



• 修改 (Modify) 命令

选择要更改信息的符号，然后单击 [修改 (Modify) …]，以显示下列对话框。

可在此修改某些种类的信息。



• 删除 (Delete) 命令

选择其信息不被用以测量堆栈使用数目的符号，然后单击 [删除 (Delete) …]，以删除符号。

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.STACKΔ <符号> = <堆栈值>

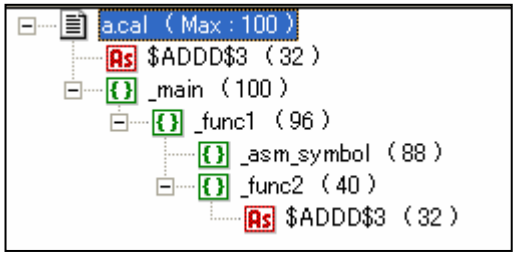
•汇编程序的实例

```
            .CPU        H8SXA:24
            .EXPORT     _asm_symbol
            .SECTION    P,CODE,ALIGN=2
_asm_symbol:
            .STACK      _asm_symbol=88
             :
            RTS
            .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 H8S，H8/300 系列汇编程序版本 6.01 或以上版本支持。

7.3.6 合并堆栈信息

保存的或编辑堆栈信息文件，可与其他堆栈信息文件合并。

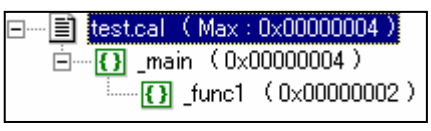
通过使用此函数，被编辑的堆栈信息将无法被再创建的堆栈信息所覆盖。

• 合并实例

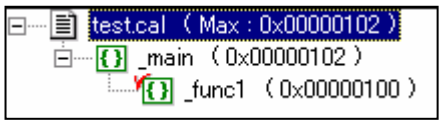
(1) **test.c**

```
void main(void)
{
    func1();
}
```

(2) 在 Call Walker 中打开堆栈信息文件



(3) 将 **func1** 的堆栈大小更改为 100

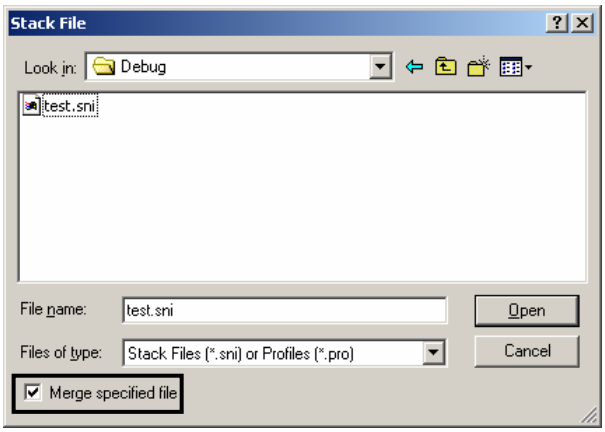


(4) 更改 **test.c** 并再创建（添加 **func2** 调用）

```
void main(void)
{
    func1();
    func2();
}
```

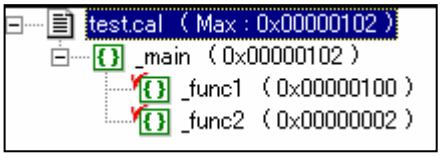
(5) 在打开 **test.sni**, 的同时，打开 **test.cal**

选中下列 [合并指定的文件 (Merge specified file)] 复选框，然后单击 [打开 (Open)] 按钮。

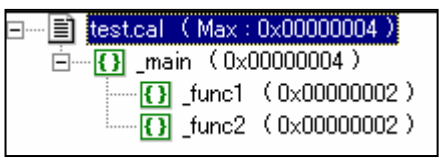


(6) 在那之后，堆栈信息将被合并。

在 (3) 更改的 **func1** 的堆栈大小被使用，同时添加了 **func2** 的信息。



当 [合并指定的文件 (Merge specified file)] 复选框未在 (5) 被选中时，在 (3) 更改的 **func1** 的堆栈大小将被再创建的值所覆盖，即和更改之前相同。



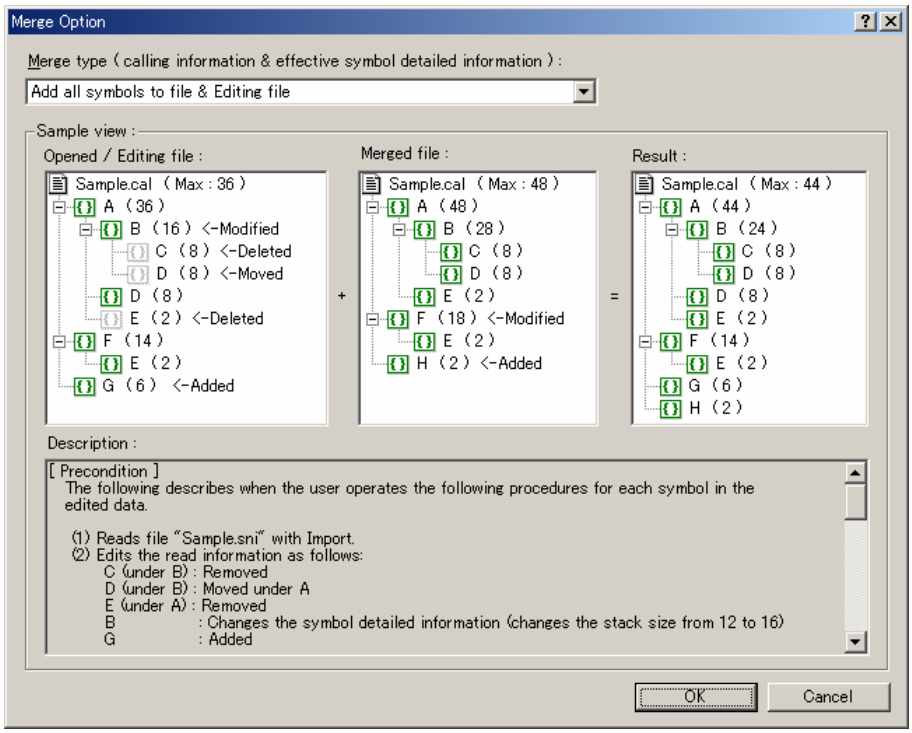
• 合并选项

合并方法可通过五种可能方式被修改。

要获取与此有关的更多信息，请参考此对话框中的**描述**。

[指定方法]

工具 (Tools) 菜单 -> **合并选项 (Merge Option) …**



• 说明

此合并函数在 Call Walker 1.3 或以上版本中有效。

7.3.7 其他函数

• 实时 OS 符号

指定下列，以显示如调用信息视图 RTOS（画面上的左边框架）中的实时 OS 符号。

[指定方法]

工具 (Tools) 菜单 -> **实时 OS 选项 (Realtime OS Option) …**

•输出列表

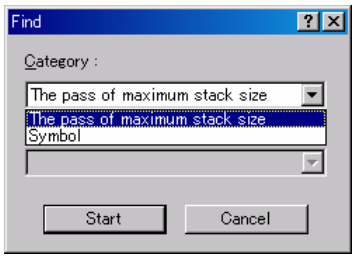
堆栈信息被输出到文本文件。

[指定方法]

文件 (File) 菜单 -> **输出列表 (Output List) …**

• 查找

调用信息视图中的下列对话框，提供两种搜索方式。



(1) 搜索最大堆栈大小的传递

(2) 搜索符号名称

[指定方法]

编辑 (Edit) 菜单 -> **查找 (Find) …**
编辑 (Edit) 菜单 -> **查找下一个 (Find Next) …**（搜索下一个）
编辑 (Edit) 菜单 -> **查找上一个 (Find Previous) …**（搜索上一个）

• 在调用信息视图中设定显示格式

通过下列命令，提供堆栈使用数目的两种显示格式。

(1) 显示所需的堆栈
堆栈大小从下端符号添加到上端。

(2) 显示所用的堆栈
堆栈大小从上端符号添加到下端。

[指定方法]

视图 (View) 菜单 -> **显示所需的堆栈 (Show Required Stack)** 或**显示所用的堆栈 (Show Used Stack)**

H8S, H8/300 系列 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		如何实施*新建 (new)* 和*删除 (delete)*	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。

（在函数内的类目标声明）

```
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。

当不使用 _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)

当使用 **_CALL_INIT** 时 **--> 10**

当不使用 **_CALL_INIT** 时 **--> 0**

```
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 如何实施新建 (new) 和删除 (delete)

• 重点：

要使用*新建 (new)*，请实施低层函数。

• 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，实施低层界面例程 (*sbrk*) 以指定将被分配的堆存储器大小。

• 实施方法：

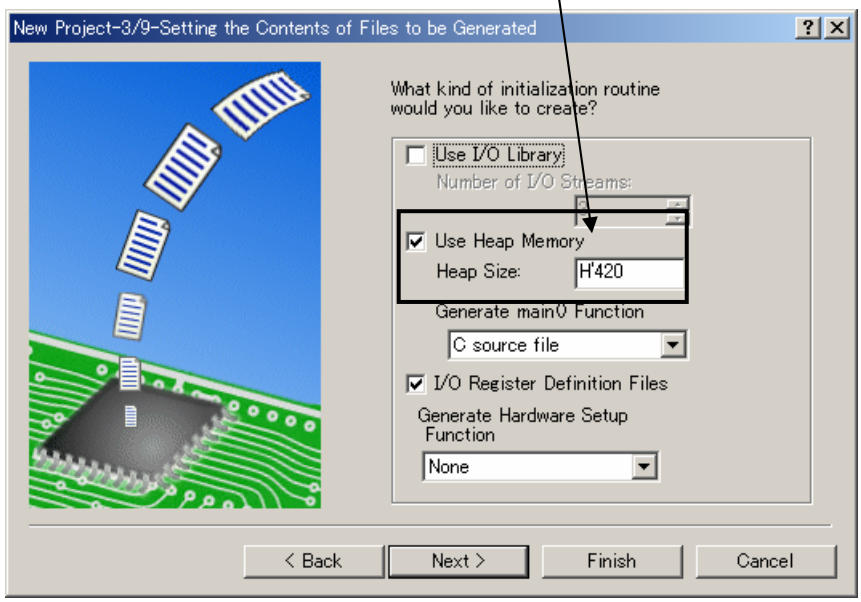
要使用 HEW，确保在创建工作空间时选中了[使用堆存储器 (Use Heap Memory)]。

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，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，然后将其实施到工程中。



```
(sbrk.c)

#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=   /* 指定定义的堆区域          */
                             /* 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 ;                   /* 区域赋值        */
   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 (static)* 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：

在主要函数内创建五个 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。

[发生 L2310 的错误]

当使用了静态成员变量时，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”””) 的消息。

[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::a;`

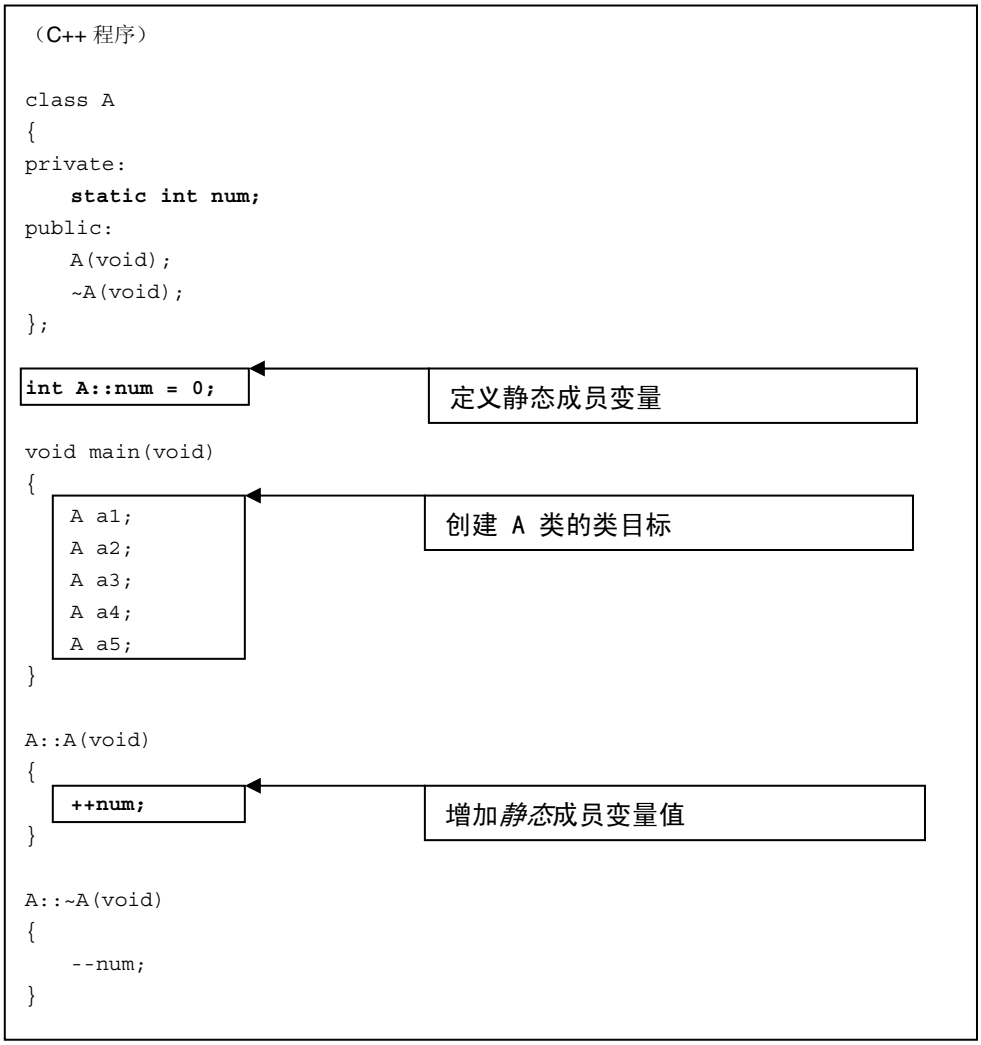
[无法分配初始值]

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 实施支持选项，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，则此解决方法不需被执行。



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：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，一项错误消息将被输出。

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，一项错误消息将被输出。

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 (Prelinker control)**
然后，选取自动 (Auto) 或运行 (Run) 预连接程序。

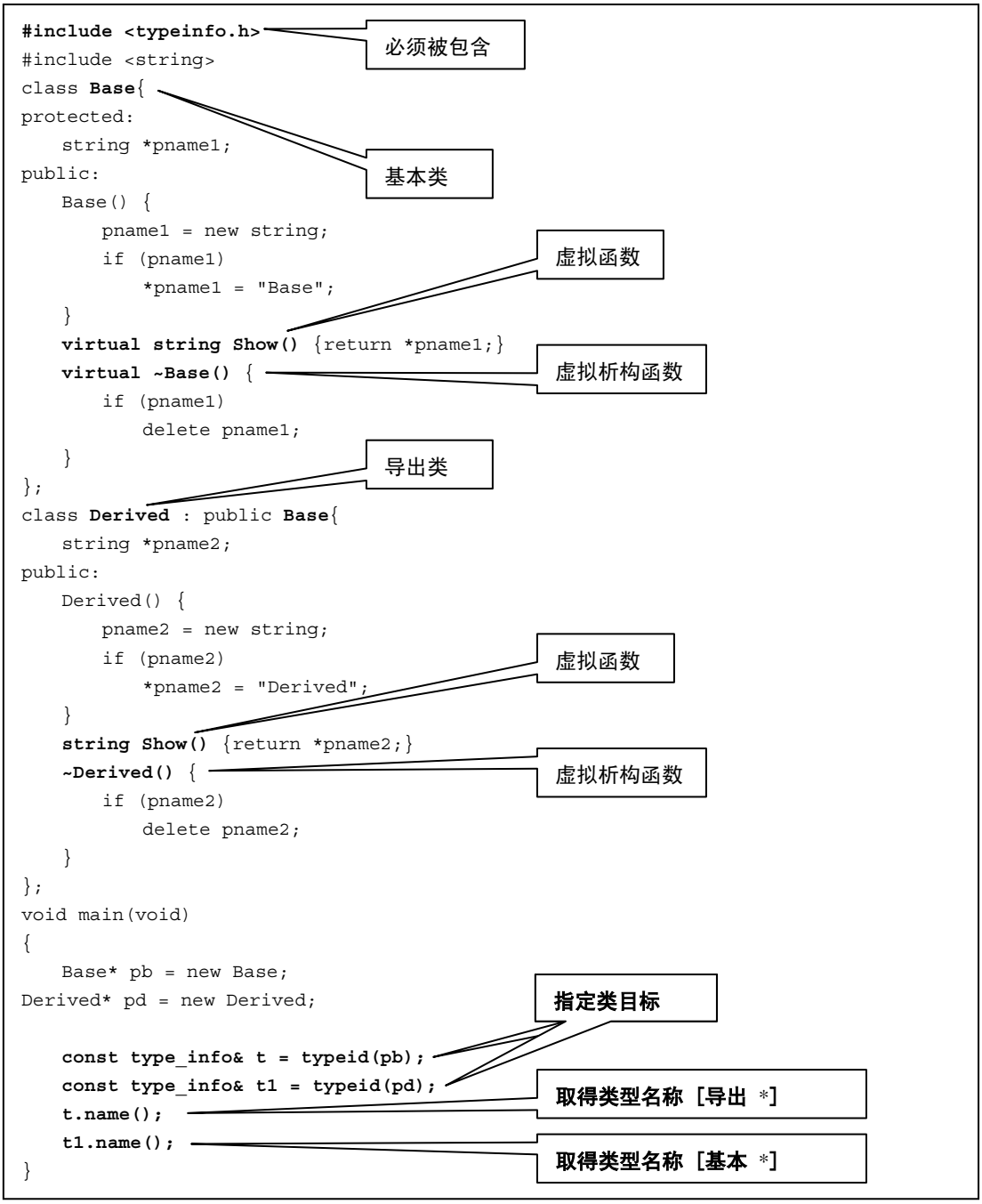
命令行： *Do not specify noprelink (default).*

• 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，以在程序执行时比较类型或取得类。

要使用 *type_info* 类，使用 typeid 运算符来指定具备虚拟函数的类目标。




```
#include <typeinfo.h>
#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)
{
    Base* pb = new Base;
Derived* pd = new Derived;

    const type_info& t = typeid(pb);
    const type_info& t1 = typeid(pd);
    t.name();
    t1.name();
}
```

• 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;
}
```

基本类

虚拟函数

虚拟析构函数

导出类

虚拟函数

虚拟析构函数

在运行时转型至基本 *

取得类名称基本

8.3.3 异常处理函数

• 描述：

不像 C，C++ 具有被称为异常的处理错误的机制。

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 (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	◎	O	O	8.4.3
4	内联扩展	O	Δ	O	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	◎	O	O	8.4.7
8	参考类型	◎	O	O	8.4.8
9	静态函数	◎	O	O	8.4.9
10	静态成员变量	◎	O	O	8.4.10
11	匿名的 *联合 (union)*	◎	O	O	8.4.11
12	虚拟函数	◎	Δ	Δ	8.4.12

◎： 同 C
O： 使用时需谨慎
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：

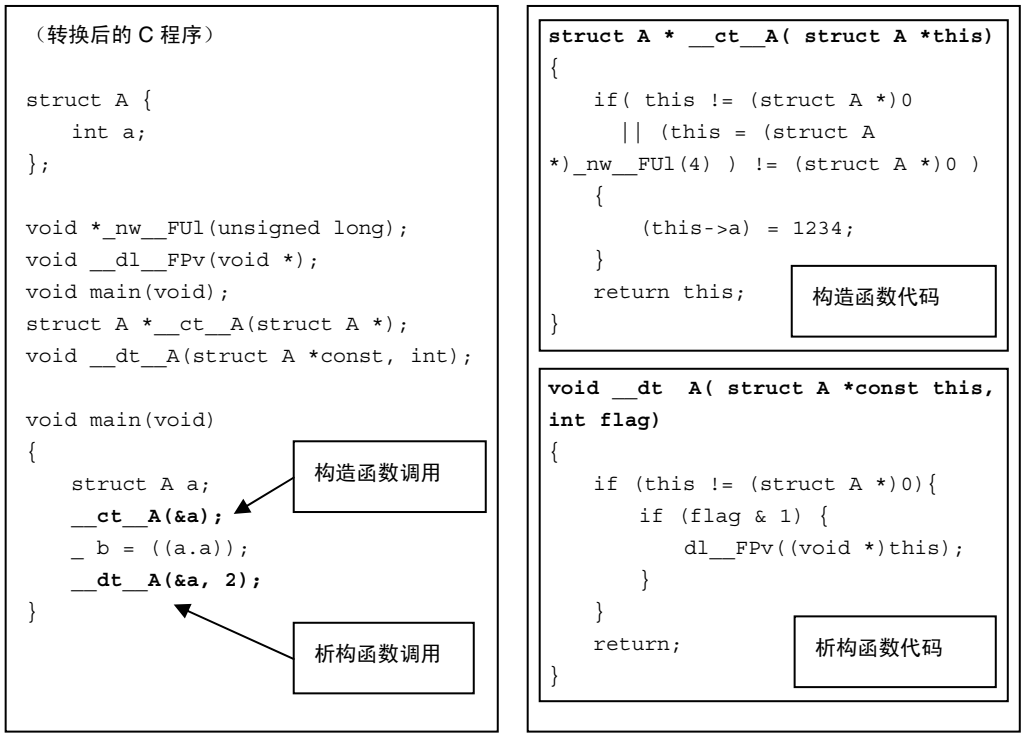
使用构造函数，以自动初始化类目标。 然而，请谨慎使用，因为它也将影响目标大小和处理速度，如下所示：

• 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)
{
}
```



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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.2.2 节，执行环境设定 (Execution Environment Settings)。）

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）

```
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 = ((__34_4_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	O	速度	O

• 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：

下面显示当在函数 *sub* 的声明中指定 0 为默认参数值时，对函数 *sub* 进行调用的实例。

如下面所示，若调用函数 *sub* 时，默认参数值是可接受的，则参数不需被指定。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。

（C++ 程序）

在函数声明中指定 0 为参数值

```
void main(void);
int sub(int, int = 0);

void main(void)
{
    int ret1;
    int ret2;
    ret1 = sub(1,2);
    ret2 = sub(3);
}

int sub(int a, int b /* =0 */ )
{
    return a + b;
}
```

未指定第二个参数

（转换后的 C 程序）

```
void main(void);
int sub(int, int);

void main(void)
{
    int ret1;
    int ret2;
    ret1 = sub(1, 2);
    ret2 = sub(3, 0);
}

int sub( int a,  int b)
{
    return a + b;
}
```

被转换为默认参数值

8.4.4 内联扩展

开发与维护	O	大小缩减	Δ	速度	O

• 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* 在类成员函数代码中被存取。

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	O	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	O	速度	O

• 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	O	速度	O

• 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	O	速度	O

• 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 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

（转换后的 C 程序）

```
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	O	速度	O

• 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*主要 (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 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

创建 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	O	速度	O

• 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。

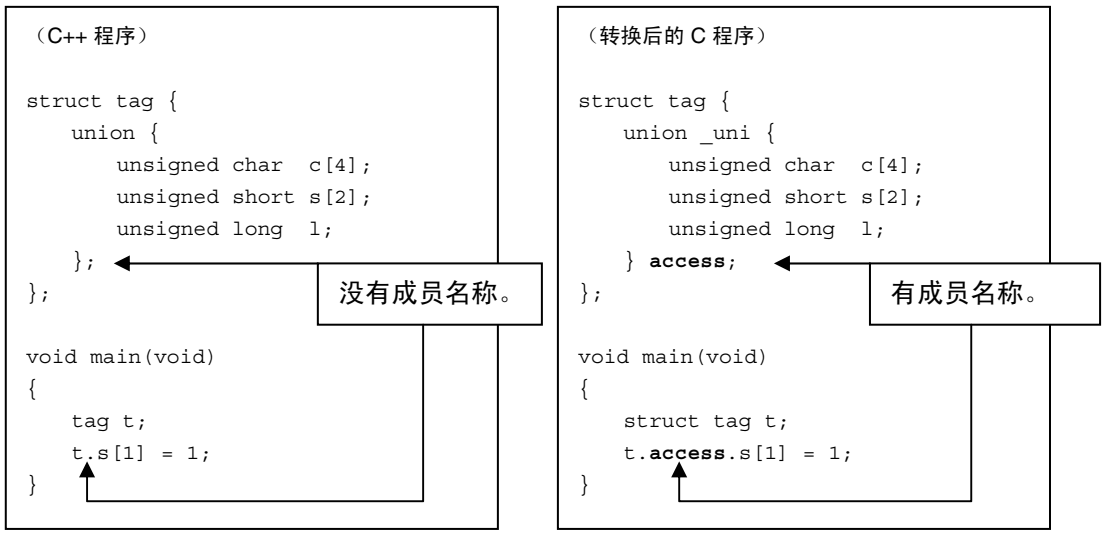
这将增进开发效率。另外，它不会影响大小和处理速度。

• 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。

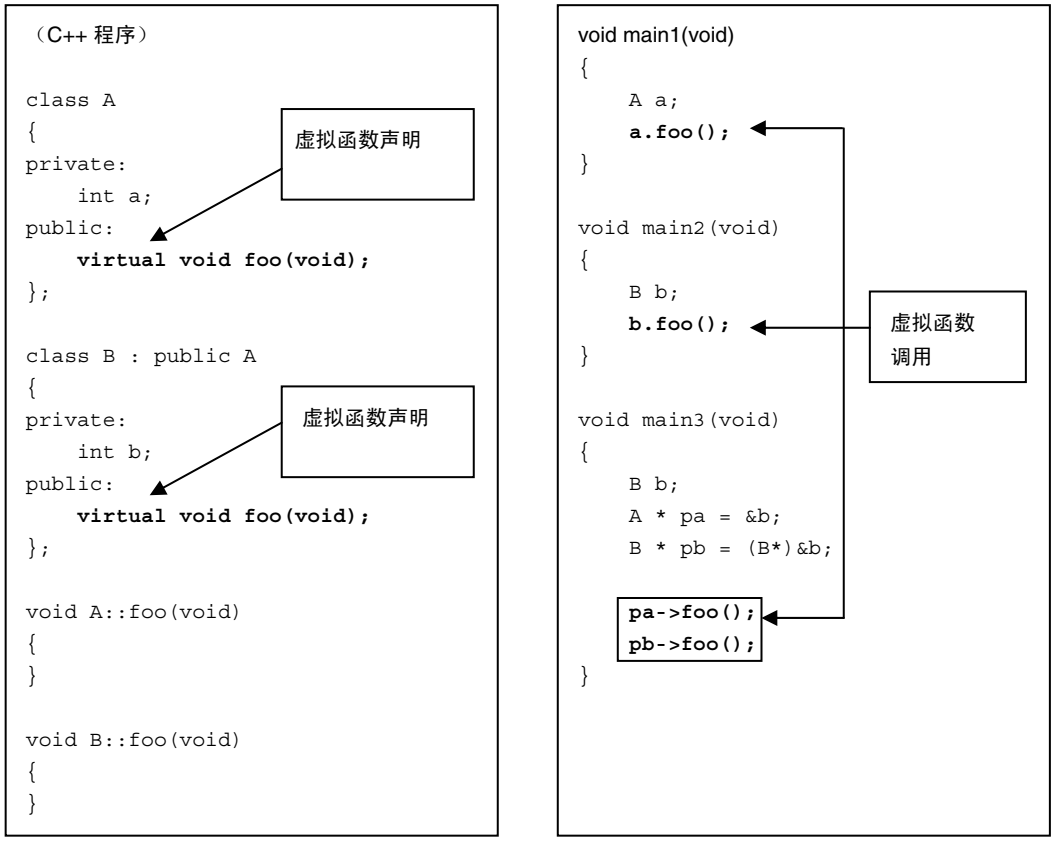
• 使用的实例：

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。



• 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 ); // A::foo() 的调用
    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 _a;
    struct A *_pa;
    struct B *_pb;

    (*((struct A*)(&_b))).__vptr = __vtbl__1A;
    (*((struct A*)(&_b))).__vptr = __vtbl__1B;

    _pa = (struct A *)&_b;
    _pb = &_b;

    _tmp = _pa->__vptr + 1;
    (  (void (*)(struct A *const))  _tmp->f  )  (  (struct A *)_pa + tmp->b);
    // 调用至 B::foo()

    _tmp = _pb->__b_A.__vptr + 1;
    (  (void (*)(struct B *const))  _tmp->f  )  (  (struct B *)_pb + tmp->b);
    // 调用至 B::foo()

    return;
}
```

H8S, H8/300 系列 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		使用短的绝对寻址模式	9.4.4
16		优化寄存器保存/恢复代码	9.4.5
17		统一公用代码	9.4.6
18		使用间接寻址模式	9.4.7
19		优化转移指令	9.4.8
20		缩短寻址模式	9.4.9
21		禁止部分优化	9.4.10
22		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/程序库标签类别：(Link/Library Tab Category:) [输入 (Input)] 显示有关项目：(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。

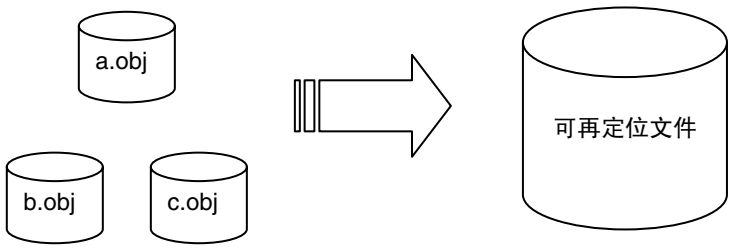
(2) 可再定位文件

可再定位（地址未分解）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。

因此在使用可再定位文件时，必须注意上述事项，以免因连接了不必要的文件而增加 ROM 大小。

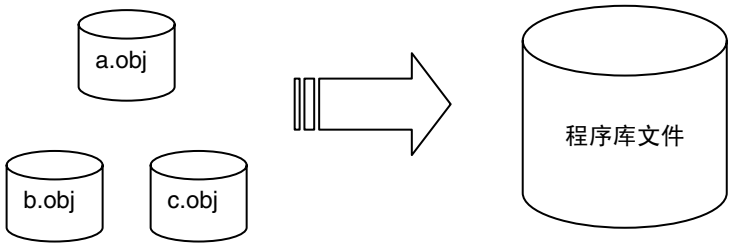


(3) 程序库文件

可再定位（地址未分解）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。



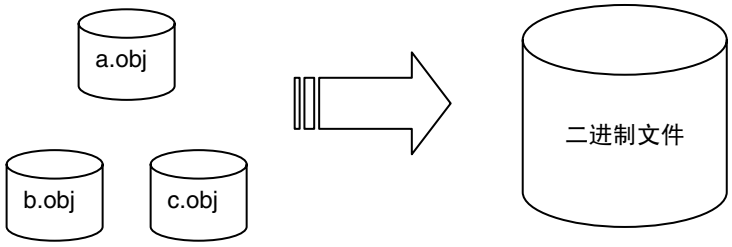
(4) 二进制文件

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。

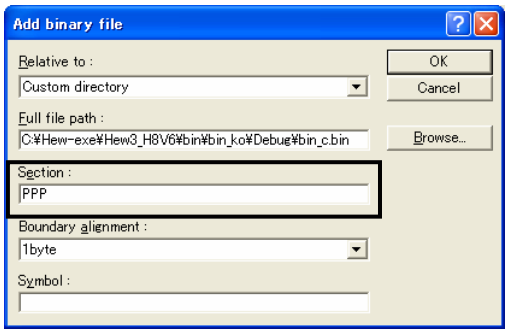


[指定方法 1]

必须指定段名称。

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [输入 (Input)] 显示有关项目：(Show entries for:) 二进制文件 (Binary files)**

命令行： ***binary=bin_c.bin(PPP)***



[指定方法 2]

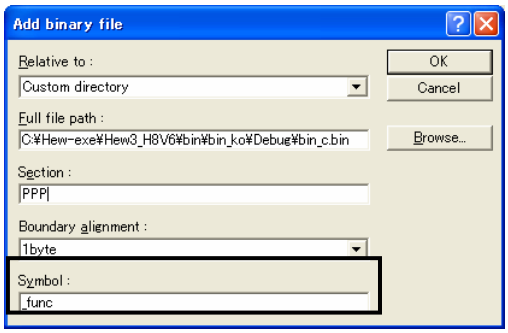
符号可以在二进制文件的开头处定义。

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/程序库标签类别：(Link/Library Tab Category:) [输入 (Input)] 显示有关项目：(Show entries for:) 二进制文件 (Binary files)**

命令行： ***binary=bin_c.bin(PPP,_func)***



[指定方法 3]

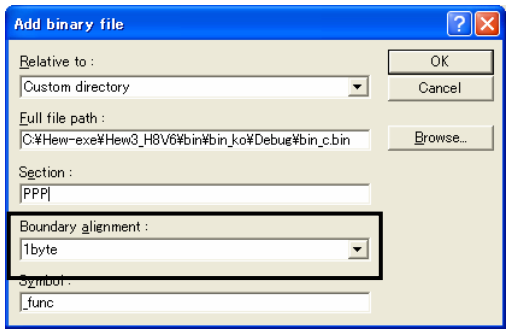
输入二进制文件时，可以指定边界对齐值。

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/程序库标签类别：(Link/Library Tab Category:) [输入 (Input)] 显示有关项目：(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）


Add binary file
Relative to :
Custom directory
OK
Cancel
Full file path :
C:¥Hew-exe¥Hew3_H8V6¥bin¥bin_ko¥Debug¥bin_c.bin
Browse...
Section :
PPP
Boundary alignment :
1byte
Symbol :
_func

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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] 输出文件的类型：(Type of output file:)**

命令行： *orm{ 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分解地址的文件。

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、HEX 或二进制。

(2) 可再定位文件

可再定位（地址未分解）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 (Other)] 用户定义的选项：(User defined options:)**

[提取选项 (Extract Options)]

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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”19.1.2 节，HEX 文件格式 (HEX File Format)。

(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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”19.1.1 节，S 类型文件格式 (S-Type File Format)。

(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。

(1) 符号地址 **-ADDR**

(2) 大小 **-SIZE**

(3) 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/程序库标签类别：(Link/Library Tab Category:) [列表 (List)] 内容：(Contents :) 显示符号 (Show symbol)**

命令行： *list [=<文件名>]*
show symbol

```
<*.map 文件>

*** 选项 ***
        :
*** 错误信息 ***
        :
*** 映像列表 ***
        :
*** 符号列表 ***


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

*** 删除符号 ***
        :
*** 可使用 Abs8 存取的变量 ***
        :
*** 可使用 Abs16 存取的变量 ***
        :
*** 函数调用 ***
        :
```

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。

(1) 符号参考计数 -COUNTS

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [列表 (List)] 内容：(Contents :) 显示参考 (Show reference)**

命令行： *list [=<文件名>]*
Show reference

<*.map 文件>
*** 选项 ***
:
*** 错误信息 ***
:
*** 映像列表 ***
:
***** 符号列表 *****

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

*** 删除符号 ***
:
*** 可使用 Abs8 存取的变量 ***
:
*** 可使用 Abs16 存取的变量 ***
:
*** 函数调用 ***

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/程序库标签类别：(Link/Library Tab Category:) [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>
*** 交叉参考列表 ***

```
编号      单元名称          全局符号          位置             外部信息
(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，顶层的数位将使用 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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] 显示有关项目：(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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] 显示有关项目：(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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] 显示有关项目：(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 [H'FF] 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 记录将添加到现有数据范围中的未使用区。

如果未指定此选项，非现有数据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，添加到非现有数据范围中的区域。

<未指定要填入未使用区的值>

```
S00E000062696E2020202020206D6F74C8
S1070000000040DF4
S107001400000041AC6
S107001C0000041CBC
S10700200000041EB6
S1070024000004200B0
S107002800000422AA
S107002C00000424A4
S10700400000042688E
S107004400000042888
S107004800000042A82
S107004C0000042C7C
S10700500000042E76
S107005400000043070
```

…

```
S11308901F9045EC7A00000008C67A01000008D2D7
S11308A040180100D0401006D0501006D0640064D
S11308B06C4A68EA0B061FD445F61F9045E4012DF4
S10908C06D766D725470A8
S10F08C8000008DA000008DE00FFE42A4D
S10B08D200FFE00000FFE42A2E
S10708DA00FFE00A2D
S10F08DE7900000A6BA00000200C54708C
S9030400F8
```

<指定要填入未使用区 [H'FF] 的值>

```
S00E000062696E2020202020206D6F74C8
S1070000000040DF4
S1130004FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8
S107001400000041AC6
S1070018FFFFFFFE4
S107001C0000041CBC
S10700200000041EB6
S1070024000004200B0
S107002800000422AA
S107002C00000424A4
S1130030FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCC
S10700400000042688E
S107004400000042888
S107004800000042A82
```

…

```
S113FF8AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF73
S113FF9AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF63
S113FFAAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF53
S113FFBAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S113FFCAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF33
S113FFDAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S113FFEAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF13
S109FFFAFFFFFFFFFFF03
S9030400F8
```

• **二进制文件的实例**

如下例所示，现有数据范围中的未使用区从 0x00 更改为 0xFF。

如果未指定此选项，非现有数据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，添加到非现有数据范围中的区域。

<未指定要填入未使用区的值>

```
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 [H'FF] 的值>

```
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 记录将添加到现有数据范围中的未使用区。

如果未指定此选项，非现有数据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，添加到非现有数据范围中的区域。

<未指定要填入未使用区的值>

```
:040000000000400F8
:040014000000041ACA
:04001C000000041CC0
:040020000000041EBA
:040024000000420B4
:04002800000000422AE
:04002C000000424A8
:040040000000042692
:040044000000042880
:040048000000042A86
:04004C000000042C80
```

…

```
:0C08C600000008DA000008DE00FFE42A51
:0808D20000FFE00000FFE42A32
:0408DA0000FFE00A31
:0C08DE007900000A6BA00000200C547090
:00000001FF
:040000030000400F5
```

<指定要填入未使用区 [H'FF] 的值>

```
:040000000000400F8
:10000400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
:040014000000041ACA
:04001800FFFFFFFFE8
:04001C000000041CC0
:040020000000041EBA
:040024000000420B4
:04002800000000422AE
:04002C000000424A8
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:040040000000042692
```

…

```
:FFFCF500FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:FFFDF400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:FFFEF300FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
:0EFFF200FFFFFFFFFFFFFFFFFFFFFFFFFFFF0F
:00000001FF
:040000030000400F5
```

• 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

通过指定此选项，终止代码将会永远是 S9。
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。这是因为如果项目地址超过 0x10000，终止代码将会是 S7 或 S8。

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”19.1.1 节，S 类型文件格式 (S-Type File Format)。

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/程序库标签类别：(Link/Library Tab Category:) [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/程序库标签类别：(Link/Library Tab Category:) [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

<存储=高>

111 秒

<存储=低>

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* 也必须指定。

注意： * 连接/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] [显示有关项目：(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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] [显示有关项目：(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”（未使用的符号“文件”-“符号”）

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/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] [显示有关项目：(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>

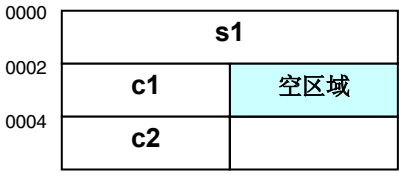
在不指定 **data_stuff** 时，边界对齐的一个字节空区域将在 **file1.c** 和 **file2.c** 之间生成，因为边界对齐的 H8 CPU 指定的值是 2。

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>

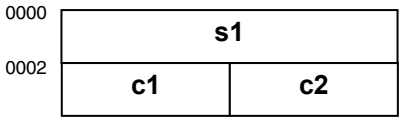
指定 **data_stuff** 时，如此例所示，如果下一个要连接的顶端数据的大小是一个字节，将不会生成边界对齐的空区域。

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/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 优化项目(Optimize items)**

命令行： *optimize=<子选项>*
<子选项> 在 9.4.2 节到 9.4.9 节中描述。

下列补遗信息的指定在编译/汇编时是必要的，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。

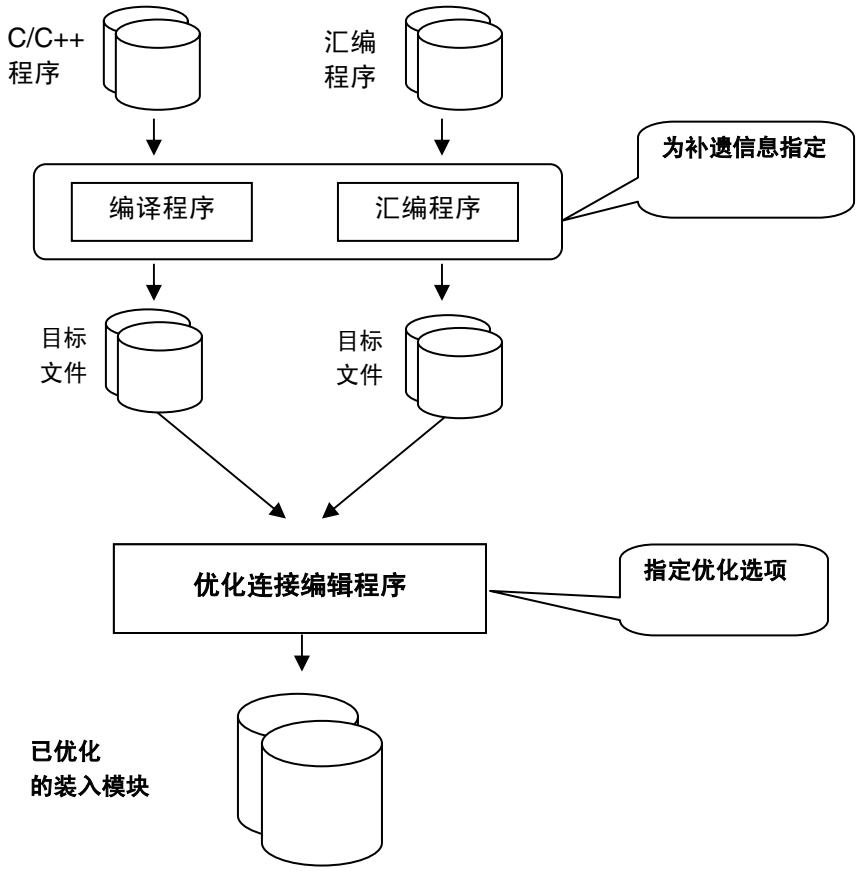
• 补遗信息的指定方法

对话框菜单： **C/C++ 标签类别：(C/C++ Tab Category:) [优化 (Optimize)] 为模块间优化生成文件 (Generate file for inter-module optimization)**

对话框菜单： **汇编标签类别：(Assembly Tab Category:) [目标 (Object)] 为模块间优化生成文件 (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**

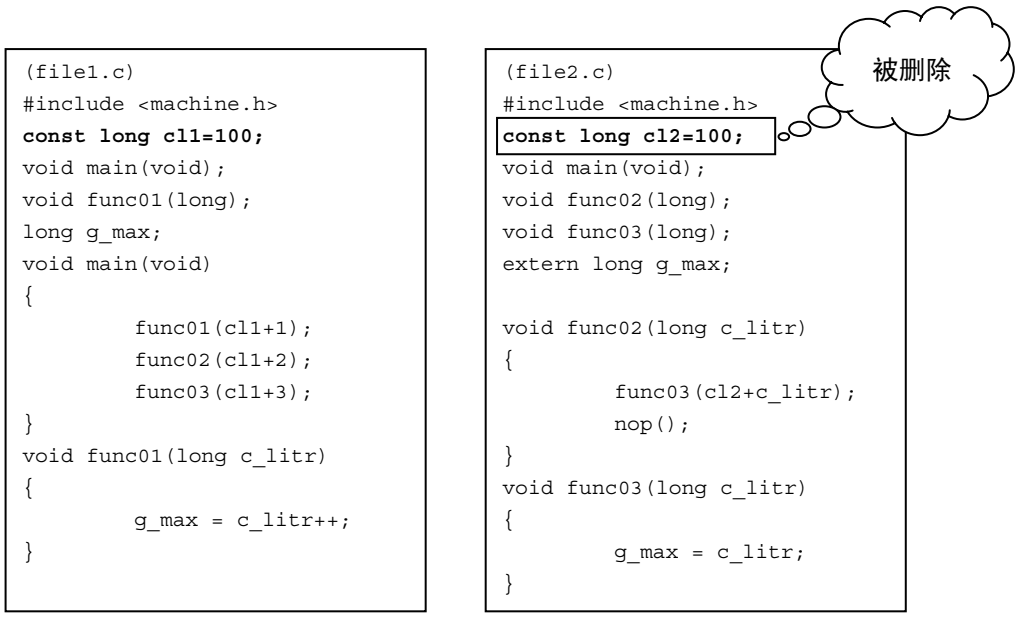
对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 优化项目(Optimize items) 统一字符串 (Unify strings)**

命令行： *optimize=string_unify*

• **相同值常数的实例**

const long 变量称为“**cl1** 、**cl2**”，具备统一为一个常数的相同常数值。

这将缩小 ROM 大小的 4 字节。



• 说明

此选项仅对由 C/C++ 编译程序生成的目标文件有效。 由汇编程序生成的目标文件将不会被优化。

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，必须指定一个项目函数。 没有指定项目函数，此优化将不会执行。

这是因为 CPU 会从向量表跳转至项目函数，而项目函数的优化或地址位于项目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 (Optimize)] 优化项目(Optimize items) 删除死码 (Eliminate dead code)**

命令行： *optimize=symbol_delete*

• 项目函数的指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [输入 (Input)] 使用入口点 (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。

```
(file1.c)
void main(void);
extern void func01(long);
extern void func02(long);
char g_c1;
long g_max1,g_max2;
void main(void)
{
    g_max1 = 0x7FFFF;
         func01(g_max1 % 3);
         func02(g_max1 / 3);
}
```

被删除

```
(file2.c)
void func01(long);
void func02(long);
void func03(long);
extern long g_max1;
void func01(long l1)
{
         g_max1 = l1 % 4;
}
void func02(long l1)
{
         g_max1 = (l1 << 1);
}
void func03(long l1)
{
         g_max1 += (l1 * (l1 / 2));
}
```

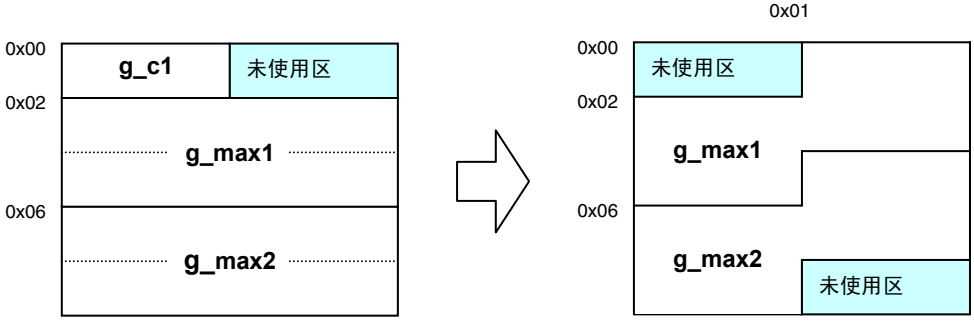
被删除

char 类型变量 **g_c1** 从未被参考，但不会删除。

这是因为 H8 是 2 字节边界对齐，如果删除 **g_c1**，下一个变量的地址将不是 2 的倍数。

旧地址符号的存取因为 CPU 的指定（除了 H8SX）而出现地址错误。

[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 0x01 存取。

• 说明

此选项仅对由 C/C++ 编译程序生成的目标文件有效。 由汇编程序生成的目标文件将不会被优化。

9.4.4 使用短的绝对寻址模式

大小	O	速度	O

• 描述

若在 8 位或 16 位绝对寻址模式中可存取的区域有空间，被频繁存取的变量将被分配，同时变量的存取代码通过此规格获得优化。

优化连接编辑程序会自动将这些变量分配到自动生成的段。

编译程序具有类似功能，但优化连接编辑程序能自动分配段。

由于短绝对寻址区域因 CPU 类型而有所不同，ROM 或 RAM 的地址应由 cpu 选项所指定。

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 优化项目(Optimize items) 使用短寻址 (Use short addressing)**

命令行： *optimize=variable_access*

• cpu 选项的指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [验证 (Verify)]**

命令行： *cpu=<存储器类型>=<地址范围> 或*
： *cpu=<cpu 信息文件名称>*
： *<存储器类型> : {ROm | RAm | XROm | XRAm | YROm | YRAm }*
： *<地址范围>:<起始地址> - <终止地址>*

• 这项优化的实例

char 类型变量 **g_c1** 被分配到 **ABS8B_OPT1** 段，同时 **short** 类型变量 **g_s1**、**g_s2** 被分配到 **ABS16B_OPT1** 段。

因此这些变量的存取代码在大小效率与执行速度上获得增进。

```
#include <machine.h>
void init(void);
void main(void);
short func01(short);
char g_c1;
short g_s1,g_s2;

void init(void)
{
        main();
        sleep();
}
```

```
void main(void)
{
    g_s1 = 7;
    g_s2 = 8;
    g_c1 = 10;
g_s1 = func01(g_s1+g_s2+g_c1);
nop();
}
short func01(short p_s1)
{
short wk = ++p_s1;
return wk;
}
```

• **已优化之存取代码的实例**

在下列 H8S/2600 高级模式的例子中，

ROM 大小： 40 字节到 30 字节

执行速度： 41 周期到 36 周期

（选项未指定）

```
_main:
MOV.W       #7,R0
MOV.W       R0,@_g_s1:32
MOV.B       #8,R0L
MOV.W       R0,@_g_s2:32
MOV.B       #10,R0L
MOV.B       R0L,@_g_c1:32
MOV.B       #25,R0L
BSR         _func01:8
MOV.W       R0,@_g_s1:32
NOP
RTS
```

（选项已指定）

```
_main:
MOV.W       #7,R0
MOV.W       R0,@_g_s1:16
MOV.B       #8,R0L
MOV.W       R0,@_g_s2:16
MOV.B       #10,R0L
MOV.B       R0L,@_g_c1:8
MOV.B       #25,R0L
BSR         _func01:8
MOV.W       R0,@_g_s1:16
NOP
RTS
```

• **说明**

(1) 有关短绝对寻址区域的详细资料，请参考 5.4.11 节，使用 8 位绝对地址区，及 5.4.12 节，使用 16 位绝对地址区。
(2) 此选项对由 C/C++ 编译程序或汇编程序生成的目标文件有效。

9.4.5 优化寄存器保存/恢复代码

大小	O	速度	O

• **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 (Optimize)] 优化项目(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 **main** 调用函数 **func01**，而 **func01** 调用 **func02**。

```
(file1.c)
void main();
extern void func01(long *,long *,long *,long *);
long g_l1,g_l2,g_l3,g_l4;
void main()
{
          g_l1 = 1;
          g_l2 = 2;
          g_l3 = 3;
          g_l4 = 4;
     func01(&g_l1,&g_l2,&g_l3,&g_l4);
}
```

```
(file2.c)
extern long g_l1,g_l2,g_l3,g_l4;
extern void func02(long *,long *,long *,long *);
void func01(long *l_p1,long *l_p2,long *l_p3,long *l_p4)
{
     g_l2 = 2;
          g_l2 += *l_p1;
     func02(&g_l1,&g_l2,&g_l3,&g_l4);
}
```

```
(file3.c)
extern long g_l1,g_l2,g_l3,g_l4;
void func02(long *l_p1,long *l_p2,long *l_p3,long *l_p4)
{
     g_l1++;
          *l_p1 = g_l1;
}
```

• **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。

在下列 H8S/2600 高级模式的例子中，

ROM 大小： 202 字节到 198 字节

执行速度： 172 周期到 166 周期

（优化前）
保存/恢复 ER2-ER3（**2 个寄存器**）

```
_main:
STM.L       (ER2-ER3),@-SP
SUB.L       ER0,ER0
MOV.B       #1,R0L
MOV.L       ER0,@_g_l1:32
SUB.L       ER1,ER1
   :
MOV.L       #_g_l3,ER2
PUSH.L      ER2
MOV.L       #_g_l2,ER1
MOV.L       #_g_l1,ER0
JSR         @_func01:24
ADDS.L      #4,SP
ADDS.L      #4,SP
LDM.L       @SP+,(ER2-ER3)
RTS
```

（优化后）
保存/恢复 ER2-ER4（**3 个寄存器**）

```
_main:
            STM     (ER2-ER3),@-SP
            PUSH.L  ER4
            SUB.L   ER0,ER0
            MOV.B   #1:8,R0L
            MOV.L   ER0,@_g_l1:32
            SUB.L   ER1,ER1
   :
            MOV.L   #h'00f00004:32,ER1
            MOV.L   #h'00f00000:32,ER0
            BSR     _func01:8
ADDS    #4,SP
            ADDS    #4,SP
            POP.L   ER4
            LDM     @SP+,(ER2-ER3)
            RTS
```

保存/恢复 ER2-ER3（**2 个寄存器**）

```
_func01:
STM.L       (ER2-ER3),@-SP
MOV.L       #_g_l2,ER1
   :
MOV.L       #_g_l1,ER0
JSR         @_func02:24
ADDS.L      #4,SP
ADDS.L      #4,SP
LDM.L       @SP+,(ER2-ER3)
RTS
```

保存/恢复 ER2（**1 个寄存器**）

```
_func01:
            PUSH.L  ER2
            MOV.L   #_g_l2,ER1
   :
            MOV.L   #_g_l1,ER0
            BSR     _func02:8
            ADDS    #4,SP
            ADDS    #4,SP
            POP.L   ER2
            RTS
```

保存/恢复 ER2（**1 个寄存器**）

```
_func02:
PUSH.L      ER2
MOV.L       #_g_l1,ER1
MOV.L       @ER1,ER2
INC.L       #1,ER2
MOV.L       ER2,@ER1
MOV.L       ER2,@ER0
POP.L       ER2
RTS
```

无保存/恢复（**0 个寄存器**）

```
_func02:
            MOV.L   #_g_l1,ER1
            MOV.L   @ER1,ER2
            INC.L   #1,ER2
            MOV.L   ER2,@ER1
            MOV.L   ER2,@ER0
            RTS
```

• 说明

此选项仅对由 C/C++ 编译程序生成的目标文件有效。 由汇编程序生成的目标文件将不会被优化。

9.4.6 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 (Optimize)] 优化项目(Optimize items) 删除相同的代码 (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 (Optimize)] 删除的大小 (Eliminated size)**

命令行： *samesize=<大小>*

• 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);
void func00(void);
long g_l1,g_l2,g_l3,g_l4,g_l5;
void main(void)
{
        func00();
        func01();
}
void func00(void)
{
        g_l1 = 1;
        g_l2 = 3;
        g_l3 = 5;
        g_l4 = 7;
        g_l5 = 9;
}
```

```
(file2.c)
void func01(void);
extern long g_l1,g_l2,g_l3,g_l4,g_l5;
void func01(void)
{
        g_l1 = 1;
        g_l2 = 3;
        g_l3 = 5;
        g_l4 = 7;
        g_l5 = 9;
}
```

• 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。

在下列 H8S/2600 高级模式的例子中，

ROM 大小： 114 字节到 66 字节
执行速度： 91 周期到 108 周期

（优化前）

```
(file1.c)
_main:
    BSR     _func00:8
    JMP     @_func01:24
_func00:
    SUB.L    ER0,ER0
    MOV.B   #1,R0L
    MOV.L   ER0,@_g_l1:32
    MOV.B   #3,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.B   #5,R0L
    MOV.L   ER0,@_g_l3:32
    MOV.B   #7,R0L
    MOV.L   ER0,@_g_l4:32
    MOV.B   #9,R0L
    MOV.L   ER0,@_g_l5:32
    RTS
```

公用代码

```
(file2.c)
_func01:
    SUB.L    ER0,ER0
    MOV.B   #1,R0L
    MOV.L   ER0,@_g_l1:32
    MOV.B   #3,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.B   #5,R0L
    MOV.L   ER0,@_g_l3:32
    MOV.B   #7,R0L
    MOV.L   ER0,@_g_l4:32
    MOV.B   #9,R0L
    MOV.L   ER0,@_g_l5:32
    RTS
```

（优化后）

```
(file1.c)
_main:
        BSR     _func00:8
        BRA     _func01:8
_func00:
        BSR     _com_opt1:8
        RTS
```

新函数

```
_com_opt1:
        SUB.L   ER0,ER0
        MOV.B   #1:8,R0L
        MOV.L   ER0,@_g_l1:32
        MOV.B   #3:8,R0L
        MOV.L   ER0,@_g_l2:32
        MOV.B   #5:8,R0L
        MOV.L   ER0,@_g_l3:32
        MOV.B   #7:8,R0L
        MOV.L   ER0,@_g_l4:32
        MOV.B   #9:8,R0L
        MOV.L   ER0,@_g_l5:32
        RTS
```

```
(file2.c)
_func01:
        BSR     _com_opt1:8
        RTS
```

• 说明

此选项仅对由 C/C++ 编译程序生成的目标文件有效。 由汇编程序生成的目标文件将不会被优化。

9.4.7 使用间接寻址模式

大小	O	速度	-

• 描述

若间接存储器存取空间具有空间区域，频繁存取的函数地址会被分配到 **INDIRECT_OPT** 段，该段则被自动分配到间接存储器存取空间。

当函数以间接存储格式来存取时，大小效率将被提高。

由于向量表也使用此区域，因此必须谨慎。

ROM 地址应由 **cpu** 选项指定。

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 优化项目(Optimize items) 使用间接调用/跳转 (Use indirect call/jump)**

命令行： *optimize=function_call*

• cpu 选项的指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [验证 (Verify)]**

命令行： *cpu=<存储器类型>=<地址范围> 或*
： *cpu=<cpu 信息文件名称>*
： *<存储器类型> : {ROm | RAm | XROm | XRAm | YROm | YRAm }*
： *<地址范围>:<起始地址> - <终止地址>*

• **实例： C 源程序**

函数 **main** 调用函数 **func01**、**func02**、**func03**。函数 **func01** 在这里被频繁调用。

```
(file1.c)
extern long func01(void);
extern long func02(void);
extern long func03(void);
void main(void);
long g_l1,g_l2,g_l3,g_l4,g_l5;
void main(void)
{
        g_l1 = 100;
        g_l1 = func01();
        g_l2 = 1000;
        g_l2 = func02();
        g_l3 = func03();
        g_l1 = func01();
        g_l1 = func01();
}
```

```
(file2.c)
long func01(void);
long func02(void);
long func03(void);
extern long g_l1,g_l2,g_l3,g_l4,g_l5;
long func01(void)
{
        return g_l1 *= 100;
}
long func02(void)
{
        return g_l2 /= 100;
}
long func03(void)
{
        return g_l2 %= 4;
}
```

• **实例： 代码**

优化前和优化后的代码实例如下所示。

被频繁调用的函数 func01 以间接存储格式加以存取。

在下列 H8S/2600 高级模式的例子中，

ROM 大小： 288 字节到 274 字节

执行速度： 491 周期到 485 周期

（优化前）

```
_main
          PUSH.L      ER6
          MOV.L       #_g_l1,ER6
          SUB.L       ER0,ER0
          MOV.B       #100,R0L
          MOV.L       ER0,@ER6
          JSR         @_func01:24
          MOV.L       ER0,@ER6
          MOV.L       #1000,ER0
          MOV.L       ER0,@_g_l2:32
          JSR         @_func02:24
          MOV.L       ER0,@_g_l2:32
          JSR         @_func03:24
          MOV.L       ER0,@_g_l3:32
          JSR         @_func01:24
          MOV.L       ER0,@ER6
          JSR         @_func01:24
          MOV.L       ER0,@ER6
          POP.L       ER6
          RTS
```

```
_func01:
          MOV.L       @_g_l1:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #100,R1L
          JSR         @$MULL$3:24
          MOV.L       ER0,@_g_l1:32
          RTS
_func02:
          MOV.L       @_g_l2:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #100,R1L
          JSR         @$DIVL$3:24
          MOV.L       ER0,@_g_l2:32
          RTS
_func03:
          MOV.L       @_g_l2:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #4,R1L
          JSR         @$DIVL$3:24
          MOV.L       ER1,@_g_l2:32
          MOV.L       ER1,ER0
          RTS
```

（优化后）

```
_main:
          PUSH.L      ER6
          MOV.L       #_g_l1,ER6
          SUB.L       ER0,ER0
          MOV.B       #100,R0L
          MOV.L       ER0,@ER6
          JSR         @@_$ind_opt1:8
          MOV.L       ER0,@ER6
          MOV.L       #1000,ER0
          MOV.L       ER0,@_g_l2:32
          BSR         _func02:8
          MOV.L       ER0,@_g_l2:32
          BSR         _func03:8
          MOV.L       ER0,@_g_l3:32
          JSR         @@_$ind_opt1:8
          MOV.L       ER0,@ER6
          JSR         @@_$ind_opt1:8
          MOV.L       ER0,@ER6
          POP.L       ER6
          RTS
```

间接存储器调用

```
_func01:
          MOV.L       @_g_l1:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #100:8,R1L
          BSR         $MULL$3:8
          MOV.L       ER0,@_g_l1:32
          RTS
_func02:
          MOV.L       @_g_l2:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #100,R1L
          BSR         $DIVL$3:8
          MOV.L       ER0,@_g_l2:32
          RTS
_func03:
          MOV.L       @_g_l2:32,ER0
          SUB.L       ER1,ER1
          MOV.B       #4,R1L
          BSR         $DIVL$3:8
          MOV.L       ER1,@_g_l2:32
          MOV.L       ER1,ER0
          RTS
```

• 说明

(1) 有关间接存储器存取空间的详细资料，请参考 5.4.13 节，使用间接存储格式，及 5.4.14 节，使用扩展的间接存储格式。

(2) 此选项对由 C/C++ 编译程序或汇编程序生成的目标文件有效。

9.4.8 优化转移指令

大小	O	速度	O

• 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* 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： * 可通过 PC 相对寻址模式存取的地址范围： –126 到 128 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 (Optimize)] 优化项目(Optimize items) 优化转移 (Optimize branches)**

命令行： *optimize=branch*

• 实例： C 源程序

函数 **main** 调用其他文件中的函数 **func01**。

```
(file1.c)
#include <machine.h>
extern 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);
}
```

```
(file2.c)
long func01(long,long);
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。

其他文件中的函数 **func01** 通过 **BSR** 调用。

在下列 H8S/2600 高级模式的例子中，

ROM 大小： 52 字节到 50 字节

执行速度： 46 周期到 45 周期

（优化前）

```
_main:
        PUSH.L  ER6
        MOV.L   #_g_l1,ER6
        SUB.L   ER0,ER0
        MOV.B   #100,R0L
        MOV.L   ER0,@ER6
        MOV.B   #-56,R0L
        MOV.L   ER0,@_g_l2:32
        MOV.L   ER0,ER1
        MOV.L   @ER6,ER0
        JSR  @_func01:24
        MOV.L   ER0,@ER6
        POP.L    ER6
        RTS
```

```
_func01
        ADD.L   ER1,ER0
        RTS
```

（优化后）

```
_main:
        PUSH.L  ER6
        MOV.L   #_g_l1,ER6
        SUB.L   ER0,ER0
        MOV.B   #100:8,R0L
        MOV.L   ER0,@ER6
        MOV.B   #56,R0L
        MOV.L   ER0,@_g_l2:32
        MOV.L   ER0,ER1
        MOV.L   @ER6,ER0
        BSR  _func01:8
        MOV.L   ER0,@ER6
        POP.l   ER6
        RTS
```

```
_func01:
        ADD.L   ER1,ER0
        RTS
```

• **说明**

此选项对由 C/C++ 编译程序或汇编程序生成的目标文件有效。

9.4.9 缩短寻址模式

大小	O	速度	-

• **描述**

当位移或立即值的代码大小可被缩减时，优化连接编辑程序将以较小的指令来替换现有指令。

由于对每个文件执行编译，参考变量的指令地址与变量定义地址之间的距离不详。

由于指令和变量地址在连接时确定，因此它们之间的距离可被计算，从而使这项优化能够执行。

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 优化项目(Optimize items) 使用 short disp/imm (Use short disp/imm)**

命令行： *optimize=short_format*

• 实例： C 源程序

下列例子显示数组的替换，同时变量地址被存储到变量。

```
(file1.c)
short str1[4];
short str2[4];
void main(void);
void func01(short);
void func02(void);
char g_c1;
unsigned long g_l1;
void main(void)
{
int i;
for (i = 0;i < 4;i++)
    {
         str1[i] = i + 1;
         str2[i] = i * 2;
    }
func01(i - 1);
func02();
}
void func01(short s1)
{
    str1[s1] = s1;
    str2[s1] = s1+4;
}
void func02(void)
{
    g_l1 = (unsigned long)&g_c1;
}
```

• 实例： 代码

优化前和优化后的代码实例如下所示。

32 位存取分别被更改成 16/8 位存取。

在下列 H8SX 高级模式的例子中，

ROM 大小： 80 字节到 68 字节

执行速度： 96 周期到 96 周期

（优化前）

```
_main:
          SUB.W   R1,R1
L36:
          MOV.W   R1,R0
          INC.W    #1,R0
          MOV.W   R0,@(_str1:32,R1.W)
          MOV.W   R1,R0
          SHLL.W  R0
          MOV.W   R0,@(_str2:32,R1.W)
          INC.W   #1,R1
          CMP.    #4:3,R1
          BLT     L36:8
          DEC.W   #1,R1
          MOV.W   R1,R0
          BSR     _func01:8
          BSR     _func02:8
          RTS
_func01:
          MOV.W    R0,@(_str1:32,R0.W)
          MOV.W    R0,E0
          ADD.W    #4:3,E0
          MOV.W   E0,@(_str2:32,R0.W)
          RTS
_func02:
          MOV.L  #_g_c1:32,@_g_l1:32
          RTS
```

（优化后）

```
_main:
          SUB.W   R1,R1
L36:
          MOV.W   R1,R0
          INC.W   #1,R0
          MOV.W   R0,@(h'0044:16,R1.W)
          MOV.W   R1,R0
          SHLL.W  R0
          MOV.W   R0,@(h'004c:16,R1.W)
          INC.W   #1,R1
          CMP.W   #4:3,R1
          BLT     L36
          DEC.W   #1,R1
          MOV.W   R1,R0
          BSR     _func01:8
          BSR     _func02:8
          RTS
_func01:
          MOV.W   R0,@(_str1:16,R0.W)
          MOV.W   R0,E0
          ADD.W   #4:3,E0
          MOV.W   E0,@(_str2:16,R0.W)
          RTS
_func02:
          MOV.L   #_g_c1:8,@_g_l1:32
          RTS
```

• 说明

(1) 此选项仅在 CPU 为 H8SXN、H8SXM、H8SXA 或 H8SXX 时有效。
(2) 此选项对由 C/C++ 编译程序或汇编程序生成的目标文件有效。

9.4.10 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 (Optimize)] 禁止项目 (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 (Optimize)] 禁止项目 (Forbid item)**
相同代码的删除 (Elimination of same code)

命令行： *samecode_forbid=<函数名称>*

• 禁止短绝对地址区的分配

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 禁止项目 (Forbid item)**
将短寻址使用到 (Use of short addressing to)

命令行： *variable_forbid=<符号名称>*

• 禁止间接地址调用

• 指定方法

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [优化 (Optimize)] 禁止项目 (Forbid item)**
将间接调用/跳转使用到 (Use of indirect call/jump to)

命令行： *function_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 (Optimize)] 禁止项目 (Forbid item)**
存储器分配在 (Memory allocation in)

命令行： *absolute_forbid=<地址>[+大小]*

9.4.11 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

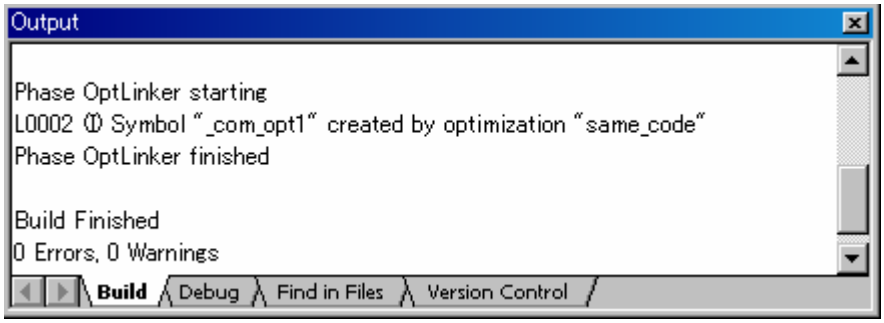
使用 HEW 时，可以通过不在以下对话框中核选，来输出优化结果。

对话框菜单： **连接/程序库标签类别：(Link/Library Tab Category:) [输出 (Output)] 显示有关项目：(Show entries for:)**
压制的信息级别消息 (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 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 (List)] 内容：(Contents :) 符号 (Symbol)**

命令行： *list [=<文件名>]*
show symbol

H8S, H8/300 系列 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。

实例

```
switch(x) {
    :
defalt:        拼错
    err = 1;
    break;
}
```

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。

实例

```
switch(x) {
    :
defalt:        拼错
    err = 1;
    break;
}
```

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零和一之间，并且小于左操作数的位宽度。 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。

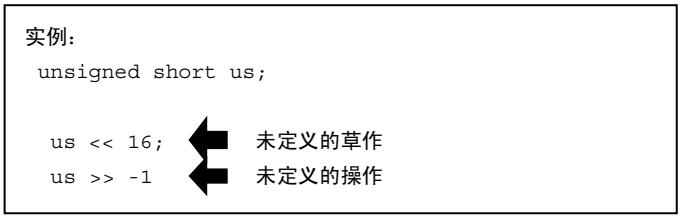


图 10.3 规则 38

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。

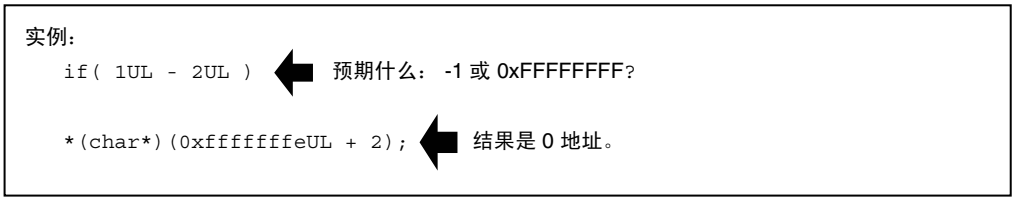


图 10.4 规则 51

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。

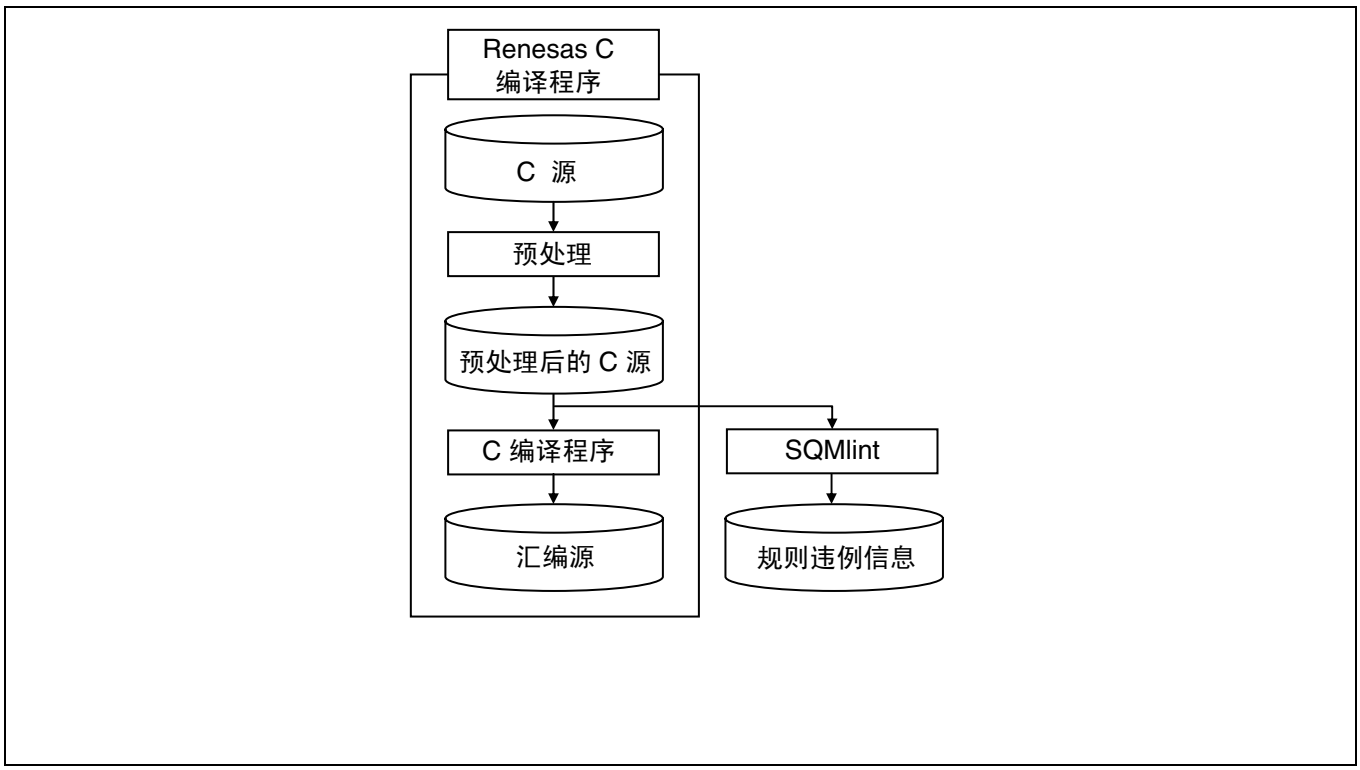


图 10.5 SQMlint 的定位

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	O	26	X	51	O*	76	O	101	O	126	O
2	X	27	X	52	X	77	O	102	O	127	O
3	X	28	O	53	O	78	O	103	O		
4	X	29	O	54	O*	79	O	104	O		
5	O	30	X	55	O	80	O	105	O		
6	X	31	O	56	O	81	X	106	O*		
7	X	32	O	57	O	82	O	107	X		
8	O	33	O	58	O	83	O	108	O		
9	X	34	O	59	O	84	O	109	X		
10	X	35	O	60	O	85	O	110	O		
11	X	36	O	61	O	86	X	111	O		
12	O	37	O	62	O	87	X	112	O		
13	O	38	O	63	O	88	X	113	O		
14	O	39	O	64	O	89	X	114	X		
15	X	40	O	65	O	90	X	115	O		
16	X	41	X	66	X	91	X	116	X		
17	O*	42	O	67	X	92	X	117	X		
18	O	43	O	68	O	93	X	118	O		
19	O	44	O	69	O	94	X	119	O		
20	O	45	O	70	O*	95	X	120	X		
21	O*	46	O*	71	O	96	X	121	O		
22	O*	47	X	72	O*	97	X	122	O		
23	X	48	O	73	O	98	X	123	O		
24	O	49	O	74	O	99	O	124	O		
25	X	50	O	75	O	100	X	125	O*		

O： 可测试的 X: 不可测试的 *: 测试具有限制

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。

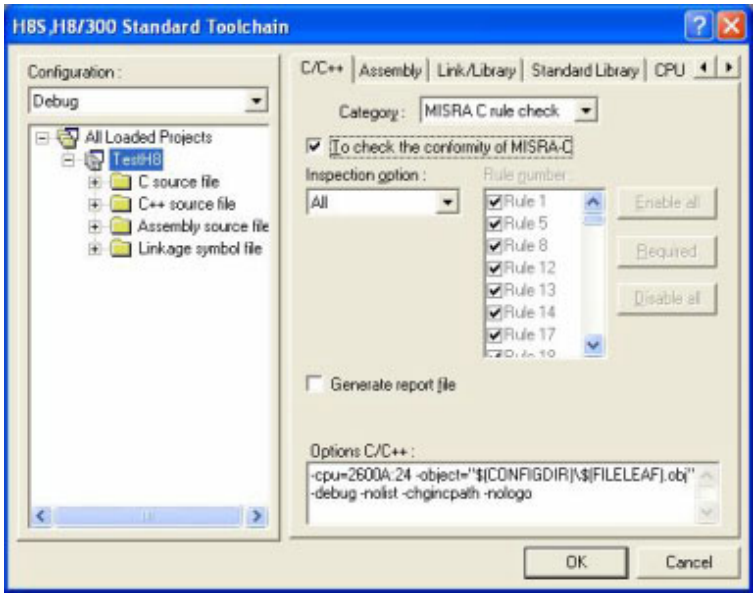


图 10.6 HEW 选项窗口

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){
4 : LABEL:
    [MISRA(55) Complain] 标签 ('LABEL') 不应该使用
5 :
6 : goto LABEL;
    [MISRA(56) Complain]不应该使用 'goto' 语句
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。

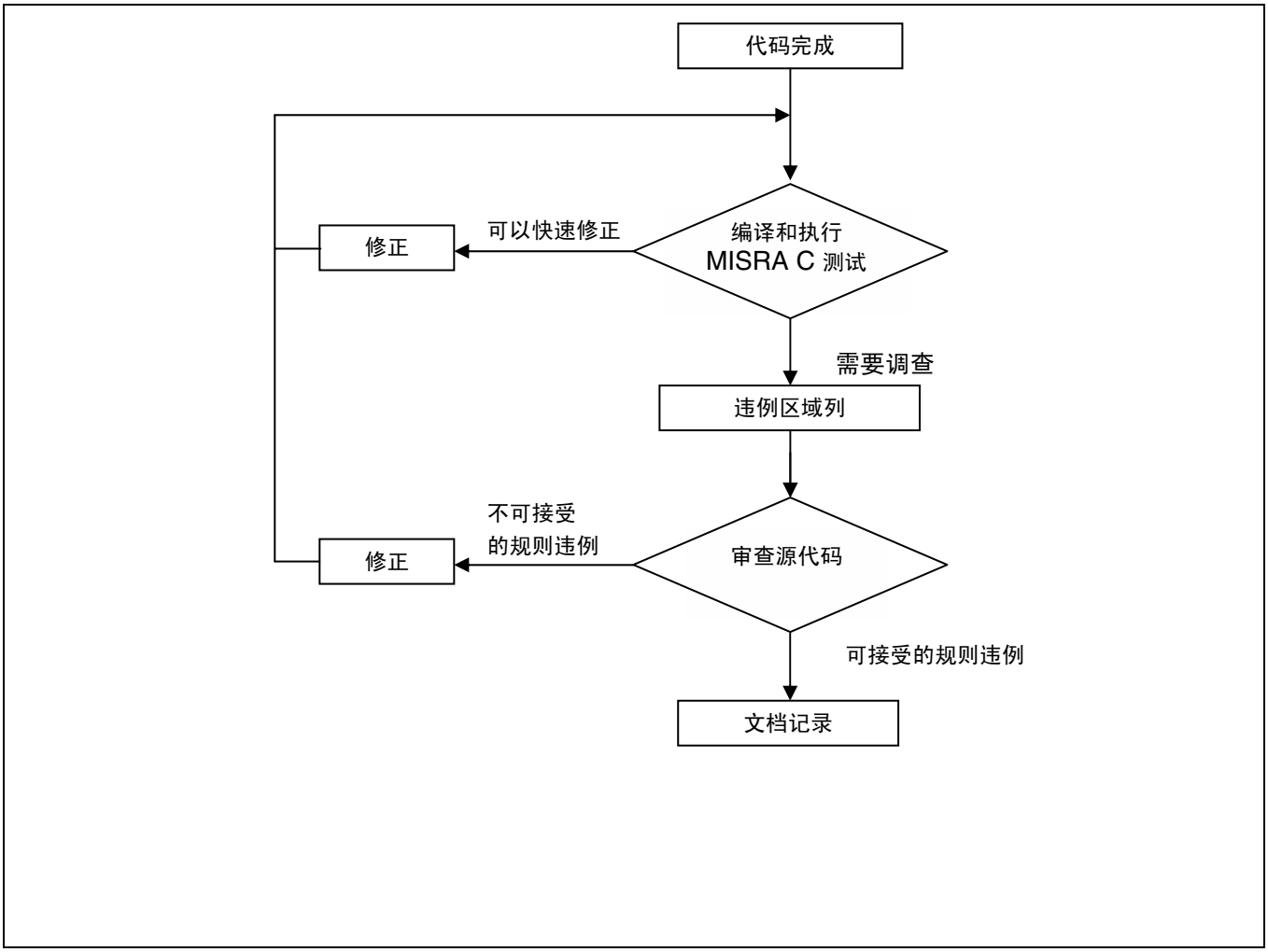


图 10.8 使用 SQMlint 的开发程序

- 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：

- H8C/C++ 编译程序封装 6.01 版本 00 和以上版本

H8S, H8/300 系列 C/C++编译程序应用笔记

问题解答

第 11 节　问题解答

本节提供用户常见问题的解答。

编号	工具名称	描述	参考的节
1	C/C++ 编译程序	如何更改字符串赋值目标	11.1.1
2		标识 1 位数据失败	11.1.2
3		从 DOS 画面启动	11.1.3
4		运行时例程指定和执行速度	11.1.4
5		H8 家族目标兼容性	11.1.5
6		有关宿主机和 OS 的问题	11.1.6
7		C 源代码级调试失败	11.1.7
8		内联扩展中所显示的警告消息	11.1.8
9		“函数未被优化”的输出	11.1.9
10		如何指定包含文件	11.1.10
11		使用日文字体的程序编码	11.1.11
12		交叉汇编程序的“操作数中的非法值”的输出	11.1.12
13		因优化所删除的大量代码	11.1.13
14		如何在调试期间查看局部变量的值	11.1.14
15		关于优化选项	11.1.15
16		传递函数参数失败	11.1.16
17		只写寄存器中的位操作失败	11.1.17
18		连接汇编语言程序的注意事项	11.1.18
19		如何检查可能导致不正确操作的编码	11.1.19
20		注解编码	11.1.20
21		如何为每个文件指定选项	11.1.21
22		当汇编程序被嵌入时，如何创建程序	11.1.22
23		连接时语法错误的输出	11.1.23
24		C++ 语言规格	11.1.24
25		如何在预处理程序扩展后查看源程序	11.1.25
26		如何输出 MACH 或 MACL 寄存器的保存/恢复代码	11.1.26
27		程序在 ICE 上正确运行，但在实际芯片上安装后运行失败	11.1.27
28		如何使用为 SH 微型计算机开发的 C 语言程序	11.1.28
29		如何修改全局选项	11.1.29
30		导致无穷循环的优化	11.1.30
31		位字段的读/写指令	11.1.31
32		长时间运行程序时发生的一般无效指令异常	11.1.32
33		整数乘法失败	11.1.33

11.1 C/C++ 编译程序

11.1.1 如何更改字符串赋值目标

问题

我该如何修改被分配字符串和数据的段的属性？

解答

虽然字符串通常被分配到常数区，它们也可通过以下操作分配到初始化区域：

(1) 使用选项修改

可使用以下选项将字符串分配到 D 段。

[指定方法]

对话框菜单： **C/C++** 标签**类别：**(**C/C++** Tab **Category**:) **[目标 (Object)]**，更改**将字符串数据存储在 (Store string data in)：到数据段 (to Data section)**

命令行： *string=data*

(2) 限制字符串的存储区域如下：

```
char *str1="ABC";
char str2[4]="ABC";
```

字符串 ABC 存储到 C 段

字符串 ABC 存储到 D 段

其成果如下：

```
        .SECTION    D,DATA,ALIGN=2
_str1:
        .DATA.L     L2
_str2:
        .SDATAZ     "ABC"
        .SECTION    C,DATA,ALIGN=2
L2:
        .SDATAZ     "ABC"
```

(3) 分配到常数区的数据通过易失性规格分配到初始化区域。

实例：

```
const int a=1;
```

指定了易失性选项。

（未指定）

```
    .SECTION    C,DATA,ALIGN=2
_a:
    .DATA.W     H'0001
```

（已指定）

```
    .SECTION    D,DATA,ALIGN=2
_a:
    .DATA.W     H'0001
```

[指定方法]

对话框菜单： **C/C++** 标签**类别：**(**C/C++** Tab **Category**:) **[其他 (Other)]，**
避免将外部符号视为易失来优化 (Avoid optimizing external symbols treating them as volatile)

命令行： *volatile*

11.1.2 标识 1 位数据失败

问题

在将 1 位数据与“1”相比时，转移操作有时会失败，为什么？

解答

确保数据未被声明为带符号的变量（int、short、char）。

若 1 位数据在位字段中被声明为带符号的变量，1 位数据本身会被解释为该符号。

因此，仅有“0”和“-1”的值可被表示。

要表示“0”和“1”，该数据必须声明为无符号。

（将始终提供错误结果的实例）

```
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;
}
```

（提供正确结果的实例）

```
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;
}
```

11.1.3 从 DOS 画面启动

问题

我该如何在 PC 版本中从 DOS 画面使用命令启动 H8S，H8/300C/C++ 编译程序系统？

解答

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： 若可用的工具是 C:\Hew2\Tools\Hitachi\H8\5_0_1\bin

```
c:\> PATH=%PATH%;C:\Hew2\Tools\Hitachi\H8\5_0_1\bin (RET)
```

这应被添加到现有路径 (PATH)。

(2) 设定 CH38

这将指明编译程序所使用的系统包含文件的位置。

实例： 若系统包含文件位于 C:\Hew2\Tools\Hitachi\H8\5_0_1\include

```
c:\> set CH38=C:\Hew2E\Tools\Hitachi\H8\5_0_1\include (RET)
```

(3) 设定 CH38TMP

设定编译程序生成文件的中间文件目录。

实例： 若中间文件目录是 C:\temp,

```
c:\> set CH38TMP=C:\temp
```

若未指定，中间文件将在当前目录中创建。 通常，并不需要此项指定；然而，指定有时是必要的，如当前目录中的磁盘空间不足时。

(4) 设定 H38CPU

指定 CPU/操作模式。

实例： 要指定 CPU/操作模式 2600a:24,

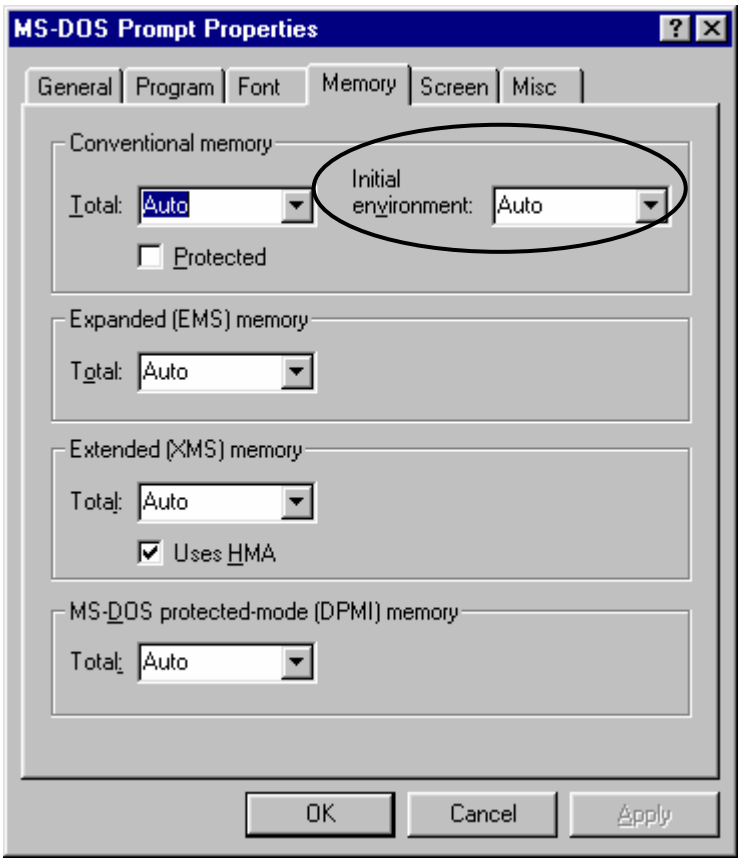
c:\> set H38CPU=2600a:24

此项指定也可在编译程序选项中指定。 若此项指定与编译程序选项不同，编译程序选项将获得优先。

说明

若在使用此环境指定来启动编译程序时显示“环境变量的区域不足 (insufficient area for environment variables)”的消息，请如下修改设定值：

打开“MS-DOS 提示属性 (MS-DOS Prompt Properties)”。



增加 [常规存储器 (Conventional memory)] 环境变量的初始分配大小。 建议使用 1024 或以上的值。

执行此更改后，重新打开 DOS 提示。

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。 用于创建程序库的选项是默认指定选项：

运行时例程速度列表 (1)

编号	类型	函数名称	300	300HN	300HA	2000N	2,000A	H8sxn	H8sxa	H8sxx
1	加	$ADDD$3	1002	746	480	206	208	175	175	175
2		$ADDF$3	426	216	174	102	104	87	87	87
3		$ADDL$3	76	-	-	-	-	-	-	-
4	减	$SUBD$3	1212	618	626	268	272	240	226	226
5		$SUBF$3	448	224	228	106	108	91	91	91
6		$SUBL$3	76	-	-	-	-	-	-	-
7	乘	$MULD$3	1886	984	992	606	610	539	539	539
8		$MULF$3	702	388	392	220	222	192	192	192
9		$MULI$3	102	-	-	-	-	-	-	-
10		$MULL$3	304	130	134	95	88	-	-	-
11		$MULXSB$3	60	-	-	-	-	-	-	-
12		$MULXSW$3	168	-	-	-	-	-	-	-
13		$MULXUW$3	148	-	-	-	-	-	-	-
14		$CMLI$3	142	-	-	-	-	-	-	-
15	除	$DIVC$3	82	-	-	-	-	-	-	-
16		$DIVD$3	7304	2544	356	1236	1238	1248	1248	1248
17		$DIVF$3	1688	1176	1180	551	553	649	649	649
18		$DIVI$3	262	-	-	-	-	-	-	-
19		$DIVL$3	1068	154	162	95	99	91	91	91
20		$DIVUI$3	208	-	-	-	-	-	-	-
21		$DIVUL$3	1038	100	108	68	70	91	91	91
22		$DIVUX$3	936	-	-	-	-	-	-	-
23		$DIVXSB$3	80	-	-	-	-	-	-	-
24		$DIVXSW$3	188	-	-	-	-	-	-	-
25		$DIVXUW$3	158	-	-	-	-	-	-	-
26		$CDVC$3	132	-	-	-	-	-	-	-
27		$CDVI$3	310	-	-	-	-	-	-	-
28		$CDVUI$3	258	-	-	-	-	-	-	-
29	余数	$MODL$3	254	-	-	-	-	-	-	-
30		$MODUL$3	224	-	-	-	-	-	-	-
31		$CMDC$3	132	-	-	-	-	-	-	-
32		$CMDI$3	310	-	-	-	-	-	-	-
33		$CMDUI$3	256	-	-	-	-	-	-	-
34	增量后	$POID$3	1164	624	542	278	283	-	-	-
35		$POIF$3	476	-	-	-	-	-	-	-
36		$POIL$3	102	-	-	-	-	-	-	-

运行时例程速度列表 (2)

编号	类型	函数名称	300	300HN	300HA	2000N	2,000A	H8sxn	H8sxa	H8sxx
37	减量后	$PODD$3	1114	604	618	268	273	-	-	-
38		$PODF$3	490	-	-	-	-	-	-	-
39		$PODL$3	98	-	-	-	-	-	-	-
40	增量前	$PRID$3	1112	572	498	254	267	229	228	228
41		$PRIF$3	448	314	292	123	127	101	99	99
42		$PRIL$3	56	-	-	-	-	-	-	-
43	减量前	$PRDD$3	1066	556	578	246	259	216	212	212
44		$PRDF$3	466	326	342	131	135	108	106	106
45		$PRDL$3	56	-	-	-	-	-	-	-
46	逻辑运算	$ANDL$3	78	-	-	-	-	-	-	-
47		$NEGD$3	74	76	80	38	40	20	20	20
48		$NEGF$3	50	-	-	-	-	-	-	-
49		$NEGL$3	76	-	-	-	-	-	-	-
50		$ORL$3	78	-	-	-	-	-	-	-
51		$XORL$3	78	-	-	-	-	-	-	-
52	块转移	$MV4$3	48	-	-	-	-	-	-	-
53		$MV8$3	72	72	76	36	38	17	17	17
54		$MVN$3	170	296	328	138	146	64	71	71
55		$mv3mm $	-	-	-	30	32	-	-	-
56		$mv3mr$	-	-	-	28	30	-	-	-
57		$mv3rm$	-	-	-	17	19	-	-	-
58		$mv4mm$	-	-	-	36	38	-	-	-
59		$mv4mr$	-	-	-	31	33	-	-	-
60		$mv4rm$	-	-	-	20	22	-	-	-
61	设定位字段	$BFINC$3	102	96	100	47	49	-	-	-
62		$BFINCR$3	94	88	92	43	45	-	-	-
63		$BFINI$3	256	180	184	71	73	35	35	35
64		$BFINIR$3	248	156	160	67	69	31	31	31
65		$BFINL$3	820	346	350	135	137	45	45	45
66		$BFINLR$3	-	330	334	127	129	39	39	39
67	参考位字段	$BFSC$3	78	78	82	38	40	-	-	-
68		$BFSI$3	196	168	172	67	69	34	34	34
69		$BFSL$3	578	270	270	122	124	37	37	37
70		$BFUC$3	68	68	72	33	35	-	-	-
71		$BFUI$3	168	144	148	55	57	-	-	-
72		$BFUL$3	546	236	240	105	107	-	-	-
73	比较	$CMPD$3	230	226	218	101	97	66	62	62
74		$CMPF$3	178	90	94	45	47	36	36	36
75		$CMPL$3	94	-	-	-	-	-	-	-
76		$EQD$3	254	250	246	113	111	87	73	73

运行时例程速度列表 (3)

编号	类型	函数名称	300	300HN	300HA	2000N	2,000A	H8sxn	H8sxa	H8sxx
77	比较	$EQF$3	202	114	122	57	61	49	47	47
78		$GED$3	264	250	256	118	116	91	77	77
79		$GEF$3	202	114	122	57	61	49	47	47
80		$GTD$3	262	250	254	117	115	90	76	76
81		$GTF$3	202	114	122	57	61	49	47	47
82		$LED$3	264	250	266	123	121	93	79	79
83		$LEF$3	212	114	122	57	61	49	47	47
84		$LTD$3	264	250	266	123	121	93	79	79
85		$LTF$3	212	115	122	57	61	49	47	47
86		$NED$3	250	252	248	114	112	78	75	75
87		$NEF$3	204	116	124	58	62	47	47	47
88	转换	$CTOL$3	60	-	-	-	-	-	-	-
89		$DTOF$3	316	238	242	110	112	87	87	87
90		$DTOI$3	508	-	-	-	-	-	-	-
91		$DTOL$3	464	290	294	100	102	105	105	105
92		$FTOD$3	178	144	148	62	64	56	56	56
93		$FTOI$3	608	-	-	-	-	-	-	-
94		$FTOL$3	564	338	342	150	152	188	188	188
95		$ITOD$3	176	152	156	74	76	82	84	84
96		$ITOF$3	164	124	128	62	64	80	80	80
97		$ITOL$3	44	-	-	-	-	-	-	-
98		$LTOD$3	366	244	256	126	128	150	150	150
99		$LTOF$3	334	224	236	116	118	151	151	151
100		$ULTOD$3	180	84	124	54	56	55	51	51
101		$ULTOF$3	150	22	104	50	52	47	47	47
102		$UTOD$3	114	62	94	43	45	38	36	36
103		$UTOF$3	80	22	52	21	23	25	25	25
104	左移	$DSLC$3	70	70	84	31	37	-	-	-
105		$DSLI$3	82	78	92	35	41	-	-	-
106		$DSLL$3	-	98	112	45	51	-	-	-
107		$SLC$3	-	-	-	23	25	-	-	-
108		$SLI$3	62	-	-	26	28	-	-	-
109		$SLL$3	118	-	-	29	31	-	-	-
110	右移	$DSRC$3	70	70	84	31	37	-	-	-
111		$DSRI$3	88	78	92	35	41	-	-	-
112		$DSRL$3	-	98	112	45	51	-	-	-
113		$DSRUC$3	70	70	84	31	37	-	-	-
114		$DSRUI$3	88	78	92	35	39	-	-	-
115		$DSRUL$3	-	98	112	45	51	-	-	-
116		$SRC$3	-	-	-	18	25	23	18	19

运行时例程速度列表 (4)

编号	类型	函数名称	300	300HN	300HA	2000N	2,000A	H8sxn	H8sxa	H8sxx
117	右移	$SRI$3	68	-	-	28	28	17	17	17
118		$SRL$3	110	-	-	29	31	18	18	18
119		$SRUC$3	-	-	-	23	25	-	-	-
120		$SRUI$3	68	-	-	26	28	-	-	-
121		$SRUL$3	110	-	-	29	31	-	-	-
122	寄存器保存/恢复	fp_regld3	52	70	80	-	-	-	-	-
123		$fp_rgld3$3	46	60	70	-	-	-	-	-
124		fp_regsv3	52	70	80	-	-	-	-	-
125		$fp_rgsv3$3	46	60	70	-	-	-	-	-
126		sp_regld3	58	80	90	-	-	-	-	-
127		$sp_rgld3$3	52	70	90	-	-	-	-	-
128		sp_regsv3	58	80	90	-	-	-	-	-
129		$sp_rgsv3$3	52	70	90	-	-	-	-	-
130		$spregld2$3	50	66	70	-	-	-	-	-
131		$sprgld23$3	40	56	60	-	-	-	-	-
132	其他	$SWI$3	124	-	-	-	-	-	-	-

说明

测量从进入运行时例程直到退出计算。

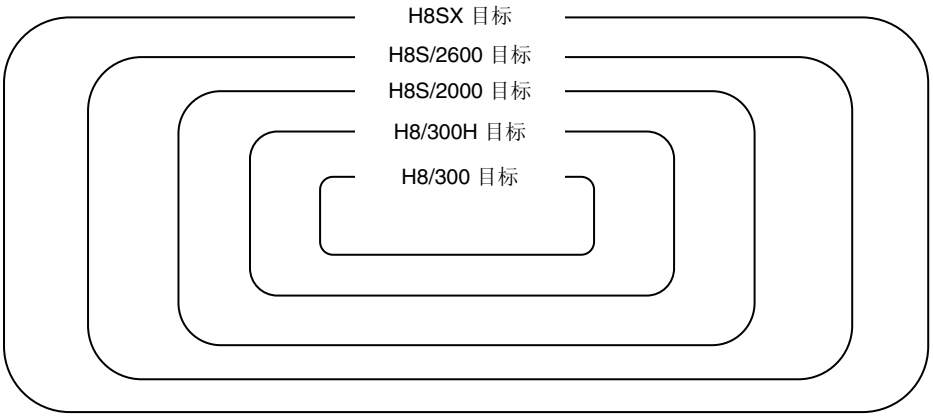
11.1.5 H8 家族目标兼容性

问题

对使用编译选项“-cpu=300”（或 300h、2000、2600、h8sx）编译的目标进行连接会出现问题吗？

解答

基本上，只要 H8 CPU 具备向上兼容性，H8/300 目标和 H8S/2000 目标就可以进行连接然后在 H8S/2000 上执行。这意味着可以继续使用之前的资源而无须修改。



目标兼容性

11.1.6 有关宿主机和 OS 的问题

问题

我该如何识别编译程序所运作的宿主机和 OS 版本？

解答

操作环境显示如下：

H8S，H8/300 C/C++ 编译程序封装

宿主机	OS	磁盘空间
IBM-PC/AT 系列	Windows98/Me/2000/XP/NT 4.0	大约 120 MB
HP9000	HP-UX 10.2	大约 30 MB
Sun SPARC	Japanese Solaris2.5 或以上	大约 30 MB

也提供在线手册。

在线手册的操作环境如下：

- 安装 Pentium® 处理器的个人计算机
- Microsoft Windows®98、Windows®/ME 或 Microsoft WindowsNT®4.0、Microsoft Windows®2000、Windows®XP
- 倍速或以上的 CD-ROM 驱动器
- 可用磁盘空间：大约 15MB

在线手册可在 Windows®98、Windows®ME、Windows NT®4.0、Windows®2000 或 Windows®XP 下参考。

Pentium® 是（美国）Intel Corporation 的注册商标。
Windows® 和 Microsoft® 是 Microsoft Corporation 在美国和其他国家/地区的注册商标。

11.1.7 C 源代码级调试失败

问题

在编译时指定了调试信息的输出，C 源代码级的调试却无法执行。 为什么？

解答

检查以下项目：

(1) 是否指定在编译时为每一个模块间优化步骤输出调试信息？

输出调试信息的目标输出格式和方式因调试程序而异。下表列出可用调试程序以及输出目标和调试信息之间的关系的实例：

可用的调试程序	目标格式	调试信息输出	调试信息输出格式
第三方 ELF/DWARF 2 支持调试程序 (3rd party ELF/DWARF 2 support debugger)	ELF/DWARF2	debug	在装入模块中
第三方 ELF/DWARF 支持调试程序 (3rd party ELF/DWARF support debugger)	ELF	debug	在装入模块中
Hitachi 集成管理员（版本 4 或以上）+E7000 (Hitachi Integration Manager (Ver.4 or higher) +E7000)	SYSROFPLUS	sdebug	调试信息文件
Hitachi 集成管理员（版本 3 或以上）+E7000 (Hitachi Integration Manager (Ver.3 or higher) +E7000)	SYSROF	debug	调试信息文件
Hitachi 调试界面（版本 2 或以上）+E6000 (Hitachi Debugging Interface (Ver.2 or higher)	SYSROF	debug	在装入模块中
Hitachi 调试界面（版本 3 或以上）+E6000 (Hitachi Debugging Interface (Ver.3 or higher)	ELF	sdebug	调试信息文件

注意： 若程序以 C++ 语言编写，目标应以 ELF 格式输出。

(2) 包含用于编译的源程序的目录是否被指定更改？

调试信息与源程序的目录位置信息存储在一起。 因此，若要编译的源程序的所在目录更改，源程序将无法被修改。 一些调试程序支持允许用户指定源程序目录的功能。

当再定位目录时，也确保移动 dwfinf 目录。

调试时可能需要 dwfinf 目录，因为它包含模块间优化的加载信息文件。

(3) 您在调试的文件是否将 C 源文件以汇编语言输出？

若是如此，同时在编译和汇编时指定调试信息的输出。

然后，就可在 C 源代码级中执行逐步执行和外部变量参考。

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。

说明

要获取有关在 C 源代码级中进行调试的详细信息，请参考“H8S，H8/300 系列模拟程序/调试程序用户手册 (H8S, H8/300 Series Simulator/Debugger User's Manual)”。

11.1.8 内联扩展中所显示的警告消息

问题

在函数的内联扩展时，警告消息“#pragma inline 中的函数 <“函数名称”> 未被扩展 (Function <"function name"> in #pragma inline is not expanded)”将显示。 为什么？

解答

此警告消息并不影响程序的执行。

然而，在以下的情形中，内联扩展将不被执行：

- 函数在 #pragma inline 规格之前被定义。
- 函数具有变量参数。
- 参数地址在函数中被参考。
- 函数调用通过要扩展的函数地址进行。
- 第二个或以后的条件/逻辑运算符。

```
#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。
    {

    }
}
```

在 #pragma inline 中指定的函数和在函数说明符 inline（C++ 语言）中指定的函数，将被内联扩展到它们被调用的位置。

11.1.9 “函数未被优化” 的输出

问题

显示“函数未被优化 (Function not optimized)”的警告消息。之前，使用相同编译选项和在相同系统环境下，对相同的程序进行编译并未产生问题。 为什么？

解答

此警告消息并不影响程序的执行。

此消息可能因为下列任何一项原因而输出：

(1) 超出了编译程序的限制。

因为编译程序在进行优化处理时生成新的内部变量，所以可能超出限制。在此情形下，分隔出现问题的函数。

(2) 存储器不足

若在进行优化处理时存储器不足，H8S，H8/300 C/C++ 编译程序将停止在表达式或以上的单元中执行优化，并在继续编译的同时输出此警告消息。在此情形下，所实现的优化级别与未使用优化选项无异。若要防止出现此警告消息，请在 C/C++ 程序中重写大型函数，以将它们分隔。此外，增加编译程序可用的存储器数量。

11.1.10 如何指定包含文件

问题

(1) 我该如何在其他目录中指定包含文件？

(2) 我该如何对现有的文件提供包含规格？

解答

这可以通过编译程序函数来指定。

下面提供有关描述：

(1) 已备有在指定的目录中指定包含文件的编译程序选项。

[指定方法]

对话框菜单： **C/C++ 标签类别：(C/C++ Tab Category:) [源 (Source)] 显示有关项目：(Show entries for:)，包含文件目录 (Include file directories)**

命令行： *include*

(2) 此选项将文件指定为包含文件，即使该文件并未包含在源文件中。

[指定方法]

对话框菜单： **C/C++ 标签类别：(C/C++ Tab Category:) [源 (Source)] 显示有关项目：(Show entries for:)，预包含文件 (Preinclude files)**

命令行： *preinclude*

11.1.11 使用日文字体的程序编码

问题

可以在程序中用日文来编码字符串和注解吗？

解答

是，您可以用日文编码。 但是，日文环境会依据宿主机而异。 下面列出不同宿主机的日文环境：

宿主机	日文代码
PC	移位 JIS 代码
HP9000	移位 JIS 代码
SPARC	EUC 代码

例如，当在 SPARC 机上创建的文件在 PC 上编译时，必须使用一个选项来修改编译程序识别日文的方式。

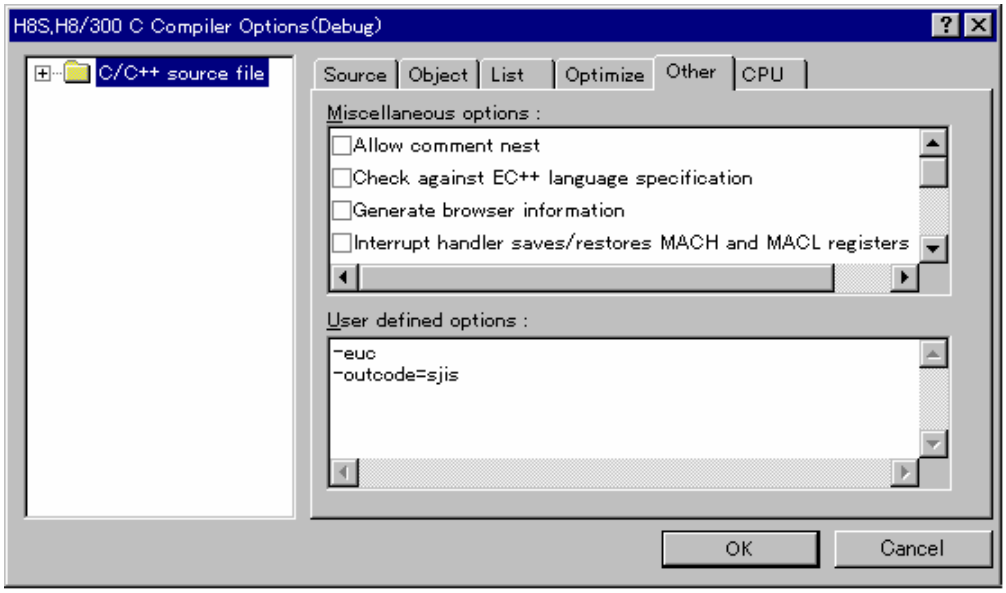
以下命令选项被提供：

命令选项	描述
sjis	选择移位 JIS 代码。
euc	选择 EUC 代码。
latin1	选择 Latin 1 代码。

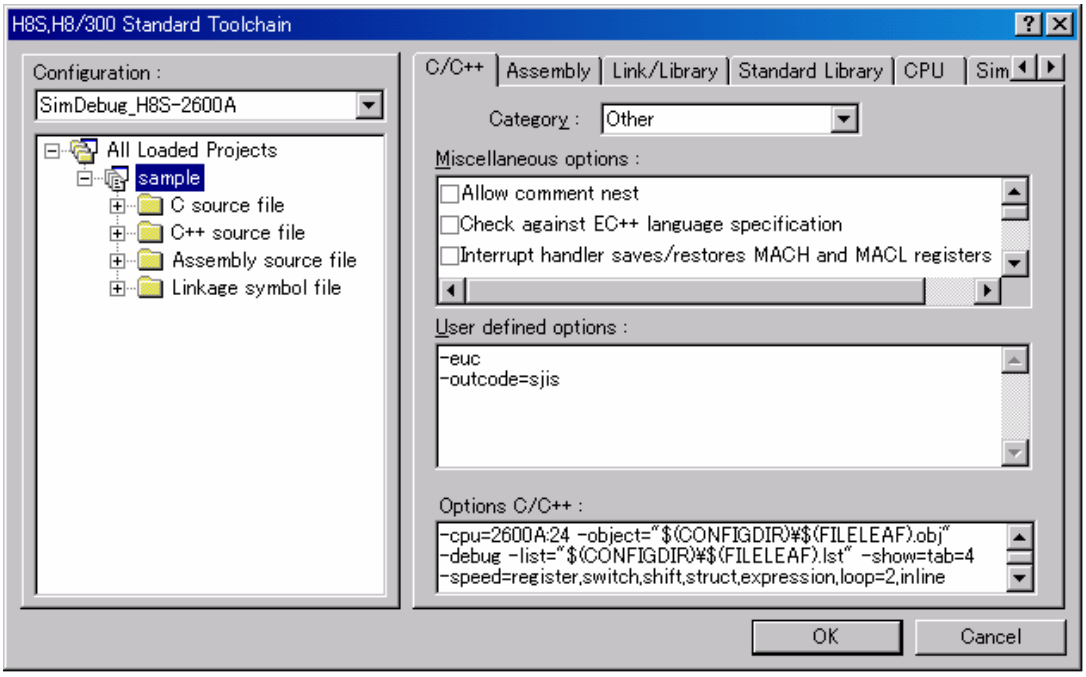
您可以在输出代码 (outcode) 选项中指定要输出到目标程序的日文代码。 outcode=sjis 选项以移位 JIS 代码输出日文代码。 同样的，outcode=euc 选项以 EUC 代码输出日文代码。

在 HEW 中，当为日文环境指定目标程序的输出代码时，在“其他 (Other)”标签上的“用户定义的选项 (User defined options)”中编码。您可以像在命令行中进行指定般来指定这些选项。

<HEW1.2>



<HEW 2.0 或以上版本>



11.1.12 交叉汇编程序的“操作数中的非法值”的输出

问题

当由编译程序的汇编源输出的文件，被交叉汇编程序进行汇编时，将显示“操作数中的非法值 (Illegal value in operand)”的消息。为什么？

解答

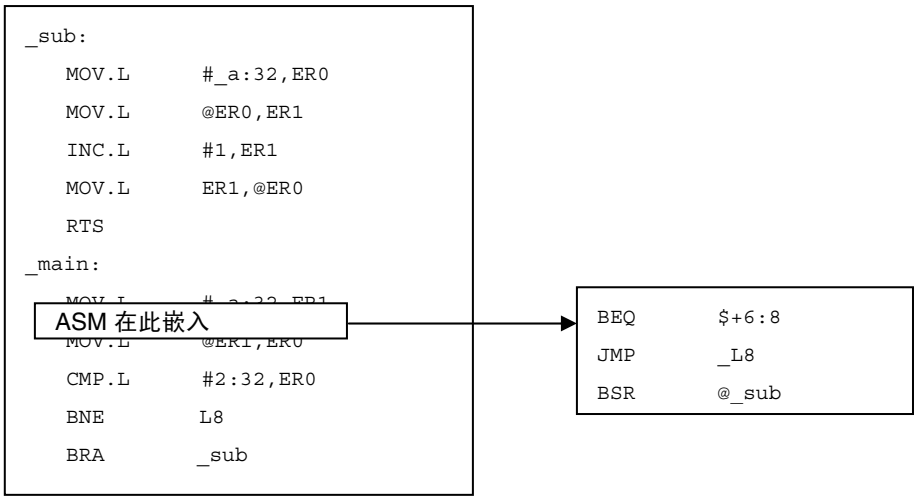
确保汇编嵌入不是以 #pragma asm 和 #pragma endasm 或 #pragma inline_asm 执行。

在此情形下，包含汇编程序固有代码的转移宽度将在 16 位位移中输出，由于实际转移宽度超出了范围，因此输出此消息。若要解决这项问题，可使用 JMP 指令修改编译程序的汇编语言程序输出，以符合转移范围。

实例

要修改程序如下：

<汇编扩展的结果>



11.1.13 因优化所删除的大量代码

问题

在进行编译后大量的代码被删除了。为什么？

解答

以下是可能导致此问题的原因：

(1) 删除局部变量的替换

若替换局部变量的值不被参考，替换操作本身将被优化所删除。

```
  void func(void)
_func:
         PUSH.L     ER6
   {
      int res1,res2,res3;
      res1=data1*data2;
         MOV.W      @_data1:32,R0
         MOV.W      @_data2:32,E0
         MULXU.W     E0,ER0
         MOV.W      R0,R6
      res2=data2*data3;
      res3=data3*data1;
         MOV.W      @_data1:32,R1
```

← 由于 res2 此下不被参考，
表达式本身会被删除。

← 第二个参数中的编码错误。
编码 res1 -> res2 以避免删除。

由于局部变量直到函数的结束部分均有效，一般上，在函数中替换值时，不会发生局部变量未被参考的情形。所以，此问题是由上述实例所示的编码错误所导致。

(2) 优化外部变量的替换

以下外部变量的替换表达式类型被优化了，且只反映了最后的算术表达式结果：

```
    int glb;
    void main()
_main:
    {
        glb=0;
        glb=1;
           MOV.W    #1:16,R0
           MOV.W    R0,@_glb:32
    }
           RTS
```

← 未生成代码 glb=0

若易失性在外部变量类型的声明中指定，代码 glb=0 将会生成。通过编译程序指定易失性选项，易失性规格将被应用到整个文件的外部变量。

11.1.14 如何在调试期间查看局部变量的值

问题

无法查看局部变量值。

我尝试以调试程序参考局部变量，但却无法参考该值或不正确的值被返回。 为什么？

解答

以下是可能导致此问题的原因：

(1) 编译时的常数操作

在编译时决定值的变量会在编译时被操作，而非在运行时被操作，然后变量本身可能会消失。

```
int x;
void func(void)
{
    int a;
    a=3;
    x=x+a;
}
```

在此情形下，x=x+3 在编译时被设定为“a”。 如果变量“a”未在别处使用，则无需将“a”视为变量。因此，它将被当作调试信息删除。

```
void func(int a,int b)
{
    int tmp;
    int len;

    tmp=a*a+b*b;
    len=sq(tmp);
}
```

设定了 len=sq(a*a+b*b)，且变量 tmp 被删除。

此问题可能是由于这些情况所造成，但是，实际的程序执行将不受影响。

(2) 删除未被参考的变量

```
int data1,data2,data3;
void func(void)
{
    int res1,res2,res3;

    res1=data1*data2;
    res2=data2*data3;
    res3=data3*data1;
    sub(res1,res1,res3);
}
```

由于局部变量直到函数的结束部分均有效，一般上，在函数中替换值时，不会发生局部变量未被参考的情形。 所以，此问题是由上述实例所示的编码错误所导致。

11.1.15 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。

11.1.16 传递函数参数失败

问题

函数的参数未被正确传递。为什么？

解答

在参数类型未声明为原型时，将相同类型指定到调用源函数及调用目标函数，以正确传递参数。

（不保证结果的规格）

```
void f(x)
char x;
{
    x+=10;
}
void main(void)
{
    char x;
    f(x);
}
```

（正确的规格）

```
void f(char x)
{
    x+=10;
}
void main(void)
{
    char x;
    f(x);
}
```

此问题可通过使用编译时的 **C/C++** 标签**类别：(C/C++tab Category:) [源 (Source)] 消息 (Message)** 上的**显示信息级消息（消息选项） (Display information level message (message option))** 来检查。 每则信息消息的输出可通过此项指定来选定。可使用 (I)0200 No prototype 函数来检查函数是否被声明为原型。

说明

在上述的“不保证结果的规格”实例中，不包含函数 f 参数的原型声明。 在此情形下，当参数 x 从 main 函数被调用时，将被转换为 int 类型。 在参数的原型声明未提供类型声明时，将产生下列的类型转换：

- 字符类型和无符号字符类型参数将会被转换为 int 类型。
- 浮动类型参数将被转换为复式类型。
- 未有其他类型被转换。

11.1.17 只写寄存器中的位操作失败

问题

只写寄存器中的位操作并未产生预期结果。 应该怎么做？

解答

编译程序生成用于 BSET、BCLR、BNOT、BST 和 BIST 的位操作指令。这些指令以字节单元读取数据，并在执行位操作后将它们以字节单元写回。另一方面，在读取只写寄存器时，CPU 将提取未定义的数据，而不管寄存器内容。结果，只写寄存器中的位操作指令可能会更改除了操作位以外的位值。

计数器衡量

避免直接在只写寄存器中执行位操作。

在将值替换为 1 字节数据后执行任何操作。下面显示一个实例：

（包含文件 (300x.h)）

```
struct  S_p4ddr {
  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;
};
union SS {
  unsigned char Schar;
  struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

（C 语言程序）

```
#include "300x.h"
unsigned char DDR;
//指定要备份到
//只写寄存器的数据
void sub(void)
{

  DDR &=~P0;
  P4DDR.Schar=DDR;
}
```

说明

有各种不同的只写寄存器，如 I/O 端口寄存器或外围设备寄存器。 在开发程序时，通过参考每一种产品随附的硬件手册，确认所操作的是适当的只写寄存器。

11.1.18 连接汇编语言程序的注意事项

问题

(1) 当从 C 语言程序调用了汇编语言程序的子例程时，我应该如何处理汇编语言程序？
(2) 当从汇编语言程序调用了 C 语言程序的子例程时，我应该如何处理汇编语言程序？

解答

(1) 当从 C 语言程序调用了汇编语言程序的子例程，且下面列出的寄存器被使用时，在函数入口/出口点保存/恢复寄存器：

CPU 系列	参数传递寄存器的数量： 2	参数传递寄存器的数量： 3
H8SX、 H8S/2600、H8S/2000 H8/300H	指定的优化： ER2 到 ER6	指定的优化： ER3 到 ER6
	未指定的优化： ER2 到 ER5	未指定的优化： ER3 到 ER5
H8/300	指定的优化： R2 至 R6	指定的优化： R3 至 R6
	未指定的优化： R2 至 R5	未指定的优化： R3 至 R5

(2) 当从汇编语言程序调用了 C 语言程序的子例程时，下列寄存器值在子例程被调用之前和之后的 C 语言程序上不被保证。若在汇编语言程序中使用了寄存器，请在调用 C 语言程序前将它保存：

CPU 系列	参数传递寄存器的数量： 2	参数传递寄存器的数量： 3
H8SX、H8S/2600、 H8S/2000 H8/300H	ER0、ER1	ER0、ER1、ER2
H8/300	R0、R1	R0、R1、R2

说明

要获取有关汇编语言程序连接的详细描述，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户指南 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)”9.3 节，连接 C/C++ 程序和汇编程序 (Linking C/C++ Programs and Assembly Programs)。

11.1.19 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 6.1 或以上版本配合使用。

```
实例）
  ch38 Δ -message Δ test.c (RET)

（C 语言程序）

/*  /* COMMENT */       →0001 : 注解中的“/*”字符串
int ;                   →0002 : 不含声明符的声明
int tmp;
void func(int);
void main(void)
{
  long a;
  tmp=a;                →0011 : 未定义局部变量的参考
  func(a+1);            →0006 : 函数参数表达式转换为原型声明中所指定的参数类型
  sub();                →0200 : 没有用于调用目标函数的原型声明
}
```

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。 此选项对从 0001 到 0307 的错误编号有效。要获取有关错误编号的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户指南 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)”第 12 节，编译程序的错误消息 (Compiler Error Messages)。

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。

11.1.20 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。编译程序 4.0 版本中的注解嵌套级是无限的，而在编译程序 3.0 版本中，可用于注解的嵌套级则多达 255。

[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 (Allow comment nest)**

命令行： *comment*

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;        //  /* comment  */        ←// 之后的代码被识别为注解

    def=1;        /* comment
    ghi=2;        // comment        */      ←以 /*  */ 括起来的代码被识别为注解
}
```

11.1.21 如何为每个文件指定选项

问题

我该如何在 HEW 系统上修改工程中每个文件的选项？

解答

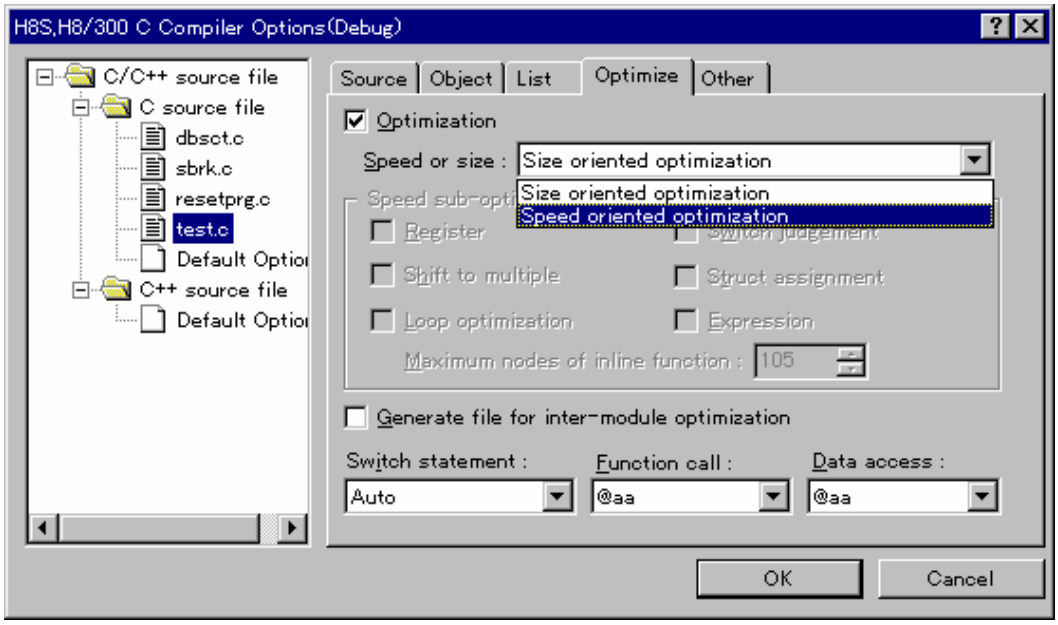
HEW 系统的支持函数可通过编译程序或汇编程序在每一个文件中个别修改和指定选项。

当使用编译程序选项来指定时，展开选项画面左侧的 C/C++ 源文件目录。然后，单击特定文件以设定所需的选项。

若在文件夹单元中指定了选项，该选项将对指定目录中的所有文件有效。

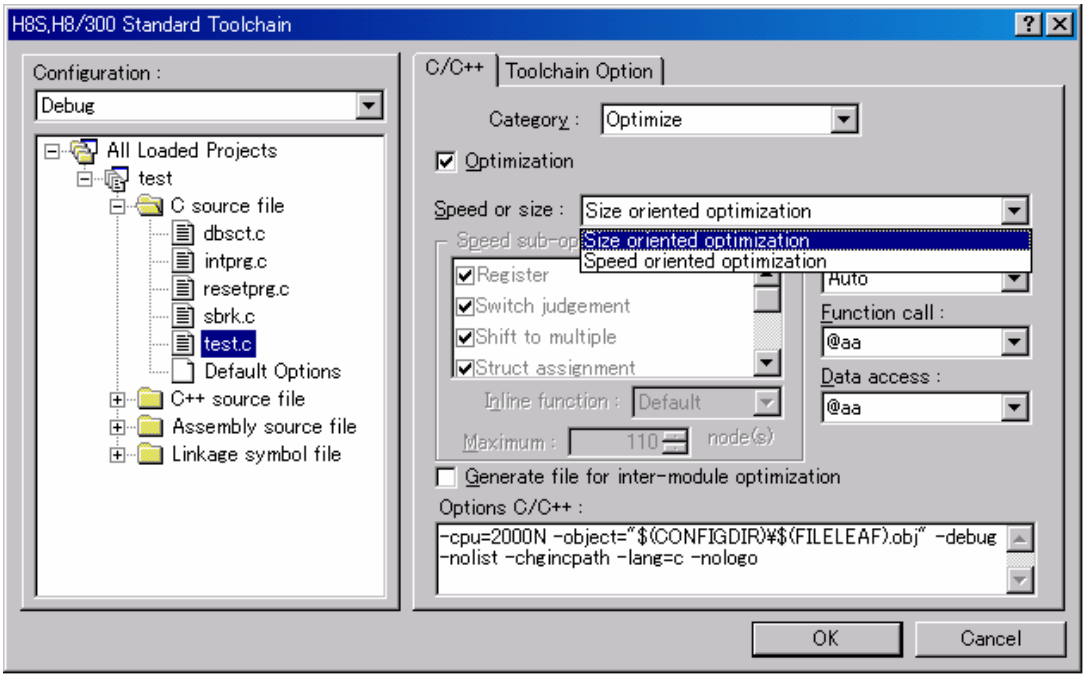
在下面的实例中，速度效率选项仅指定到工程中的 test.c 文件。

<HEW1.2>



从画面左侧选取 test.c，并从优化 (Optimize) 标签上的速度或大小 (Speed or size)：，选取面向速度的优化 (Speed oriented optimization) 。

<HEW 2.0 或以上版本>



从画面左侧选取 test.c，并从 **C/C++** 标签**类别：(C/C++** Tab **Category:) [优化 (Optimize)] 速度或大小 (Speed or size)：**， 选取**面向速度的优化 (Speed oriented optimization)** 。

11.1.22 如何在汇编程序被嵌入时创建程序

问题

在编译时使用 #pragma asm 和 #pragma endasm 或 #pragma inline_asm 执行汇编程序固有代码，将造成警告消息的输出。

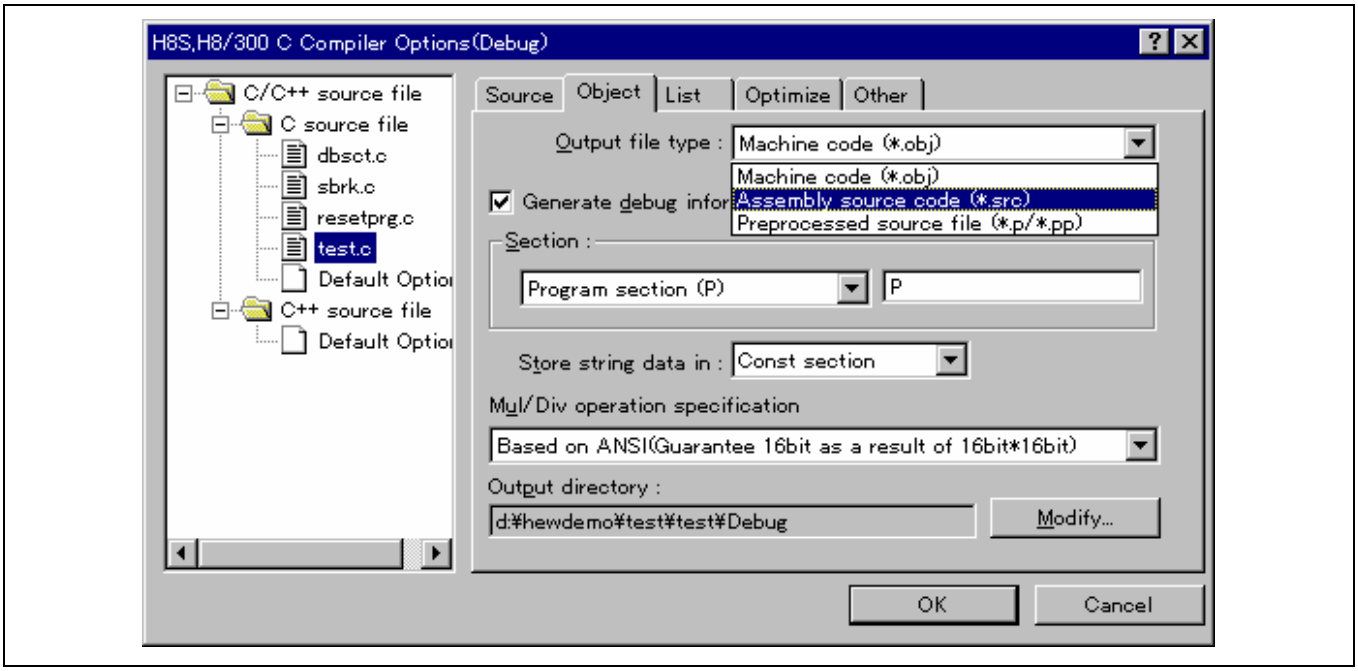
解答

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，请参考 11.1.21 节，如何为每个文件指定选项中所描述的步骤，将包含汇编程序嵌入的文件指定到汇编输出。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：

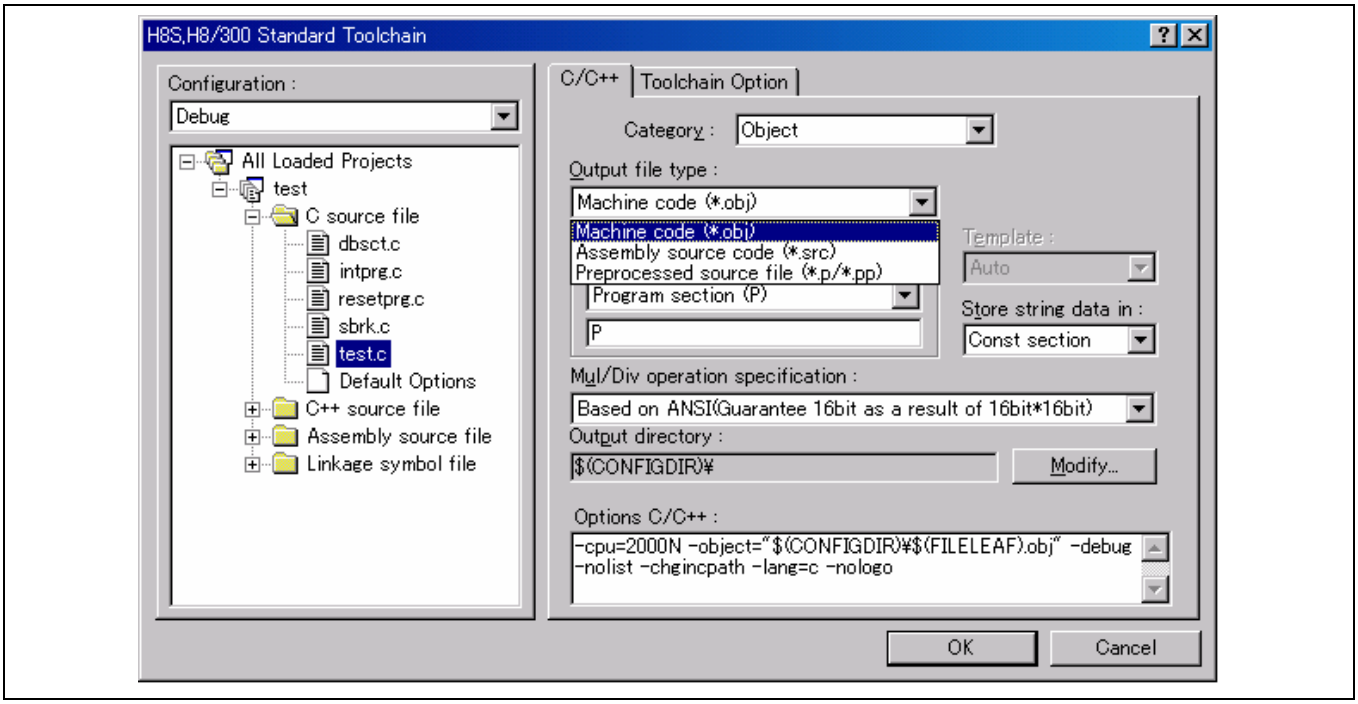
<HEW1.2>



从目标 (Object) 标签上的输出文件类型 (Output file type)：，选取汇编源代码 (*.src) (Assembly source code (*.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。

<HEW 2.0 或以上版本>



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) (Assembly source code (*.src))** 。

文件以此规格正常创建。

11.1.23 连接时语法错误的输出

问题

HEW1.2 的模块间优化器上显示错误消息“202 语法错误 (202 SYNTAX ERROR)”。为什么？

解答

文件名或工程名是否包含日文字符、减号或空格字符？

日文字符、减号或空格字符无法在编译程序、汇编程序、模块间优化器、库管理程序或 S 类型转换器中为文件名指定。例如，若工程名包含日文字符，在以模块间优化器选项来指定输出目标时会出现语法错误。

说明

在 HEW2.0 或以上版本中，可成功创建程序而不会显示错误消息，即使文件名或工程名包含日文字符、减号或空格字符。但是，应尽可能不使用日文字符、减号或空格字符。

11.1.24 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

H8S，H8/300 C/C++ 编译程序的下列函数支持 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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”10.3.2 节，C++ 类程序库 (C++ Class Libraries)。

(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“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.2.2(5) 节，C/C++ 程序库函数初始设定 (_INILIB) (C/C++ library function initial settings(_INILIB))。

11.1.25 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) (Preprocessed source file (*.p/*.pp))**

命令行： *preprocessor*

11.1.26 如何输出 MACH 或 MACL 寄存器的保存/恢复代码

问题

我该如何输出 MAC 寄存器的保存/恢复代码？

解答

可在编译程序选项中指定 MAC 寄存器的保存/恢复代码的输出。

当在中断函数中使用 MAC 寄存器（通过使用内建函数 mac 或 macl）或在中断函数中进行函数调用时，此项指定使 MAC 寄存器的值始终被保证。

当 MAC 寄存器在中断函数中使用时，即使未指定此选项，MAC 寄存器的保存/恢复代码将会输出。

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:) 中断处理器保存/恢复 MACH 及 MACL 寄存器，若使用 (Interrupt handler saves/restores MACH and MACL registers if used)**

命令行： *macsave*

实例

要从中断函数调用函数 sub：

（CC++ 程序）

```
extern void sub(void);
#pragma interrupt func
void func(void)
{
    sub();
}
```

（编译的汇编扩展代码）

不使用选项

```
_func:
       STM.L    (ER0-ER1),@-SP



       JSR      @_sub:24



       LDM.L    @SP+,(ER0-ER1)
       RTE
```

使用选项

```
_func:
       STM.L    (ER0-ER1),@-SP
       STMAC.L  MACL,ER1
       PUSH.L   ER1
       STMAC.L  MACH,ER1
       PUSH.L   ER1
       JSR      @_sub:24
       POP.L    ER1
       LDMAC.L  ER1,MACH
       POP.L    ER1
       LDMAC.L  ER1,MACL
       LDM.L    @SP+,(ER0-ER1)
       RTE
```

11.1.27 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 3.3 节，段地址运算符。

11.1.28 如何使用为 SH 微型计算机开发的 C 语言程序

问题

在 H8S，H8/300 微型计算机上使用为 SH 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 2 字节数据

在 SH 上，int 类型数据被视为 4 字节数据，但在 H8S，H8/300 系列上，它们被视为 2 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。
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，H8S，H8/300 系列无法处理任何嵌入 SH 系列汇编源的代码。

说明

如果您要在 H8 开发环境中使用于 M16C 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 M16C 开发环境中创建的所有 C 源文件，顺利转换到 H8 开发环境。

“转换助手”(Translation Helper) 可以从“Renesas 开发环境”(Renesas Development Environment) 网站免费下载。

11.1.29 如何修改全局选项

问题

当参数传递寄存器的数量被更改时，模块间优化器将生成错误。 造成这个问题的原因是什么？

解答

指定参数传递寄存器数量的编译程序选项是一个全局选项，它在工程中的指定必须保持一致。

因此，若仅修改编译程序选项，将导致错误。

共有两个全局选项，一个指定参数传递寄存器的数量，另外一个则指定 CPU 类型。

全局选项可以修改如下：

实例： 若要更改参数传递寄存器的数量：

(1) 修改编译程序选项

[指定方法]

对话框菜单： **CPU** 标签，**将参数的数量从 2（默认）更改至 3 (Change number of parameter from 2(default) to 3)**

命令行： *regparam=3*

这将更改所有相应 C/C++ 文件的选项指定。

(2) 修改汇编程序文件

所使用的寄存器会在 C/C++ 程序和汇编程序文件连接时更改。

这将需要修改汇编程序文件。

当参数传递寄存器的数量设定为三时，所使用的寄存器将更改如下：

对于 H8/300 CPU： R0, R1 → R0, R1, R2
其他 CPU： ER0, ER1 → ER0, ER1, ER2

要获取界面的详细资料，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.3.2(3) 节，有关寄存器的规则 (Rules concerning registers)。

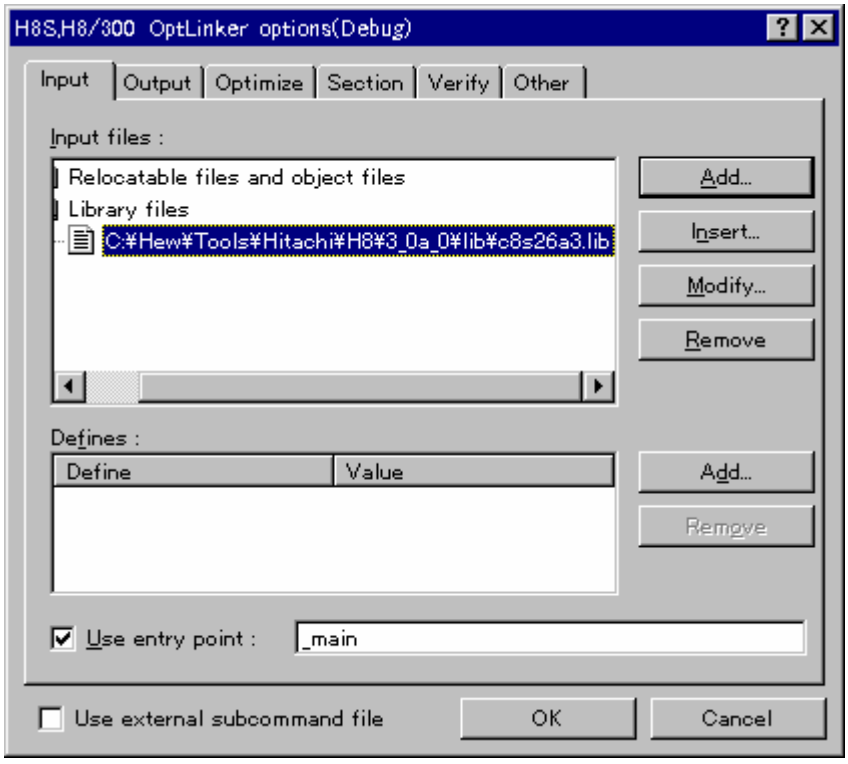
所描述的界面也适用于嵌入 C/C++ 程序的汇编语言代码。

(3) 更改要连接的标准程序库/EC++ 类程序库

使用模块间优化器来更改要连接的程序库。

在 HEW1.2 的修改显示如下：

若之前连接的程序库是 c8s26a.lib，将它更改为 c8s26a3.lib：



要获取有关全局选项和相关程序库的详细信息，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序，及优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”中的表 1.1，标准程序库和编译选项之间的关系 (The Relationship between Standard Libraries and Compile Options)。

在 HEW2.0 或以上版本中，全局选项在编译程序、汇编程序和程序库生成工具中被共同设定，且它们在优化连接编辑程序中属于非强制设定。

（若旧版本的程序库在**连接/程序库**标签**类别：(Link/Library** Tab **Category:) [输入 (Input)] 程序库文件 (Library files)** 上设定，以上所示的修改将是必要的。）

11.1.30 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

无穷优化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

```
int i1;
void f( int *d)
{
int a;
    do
    {
       a=*d;
    }while(a!=0);
    i1 = a;
}
```

具备优化的汇编程序源

```
_f:                                          ; 函数:  f
          .STACK      _f=4
          MOV.W       @ER0,R1
L25:
          MOV.W       R1,R1          ; 不从存储器读取
          BNE         L25:8
          MOV.W       R1,@_i1:32
          RTS
```

被修改的 C 源

```
int i1;
void f( volatile int *d)
{
int a;
    do
    {
       a=*d;
    }while(a!=0);
    i1 = a;
}
```

具备优化之被修改的汇编程序源

```
_f:                                          ; 函数:  f
          .STACK      _f=4
          MOV.L       ER0,ER1
L25:
          MOV.W       @ER1,R0        ; 从存储器读取
          BNE         L25:8
          MOV.W       R0,@_i1:32
          RTS
```

11.1.31 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。因此，存取可以通过意外的指令执行。指定 __evenaccess 以使用为成员变量设定的类型执行存取。

要防止存取方法以及使用编译程序的多次存取的改变，请明确地为您要防止这类改变的变量指定 __evenaccess。

没有 __evenaccess 的 C 源

```
struct bit reg;

void main()
{
    reg.b6=1;
}
```

没有 __evenaccess 的汇编程序源

```
_main:                                  ; 函数:  main
          .STACK      _main=4
          BSET.B      #1,@_reg:32
          RTS
```

具有 __evenaccess 的 C 源

```
__evenaccess struct bit reg;

void main()
{
    reg.b6=1;
}
```

具有 __evenaccess 的汇编程序源

```
_main:                                  ; 函数:  main
          .STACK      _main=4
          MOV.W       @_reg:32,R0
          BSET.B      #1,R0H
          MOV.W       R0,@_reg:32
          RTS
```

说明

有关 **__evenaccess** 关键字的详细资料，请参考 3.5.3 节，偶数字节存取规格功能。

11.1.32 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。

11.1.33 整数乘法失败

问题

当整数乘法的结果被分配到 **long** 类型全局变量时，其结果不是预定的值。

这里所举的例子是 [20 * 2000] 的结果。但 [15 * 2000] 却产生了正确的结果。超出 **short** 范围的整数乘法无法产生正确的结果，即使结果被分配到 **long** 类型的变量。为什么？

<实例>

```
long l_max;
    :
  l_max = 20 * 2000;
```

解答

编译程序将常数表达式中所描述的整数识别为 **int** 类型（2 字节），即使结果被分配到 **long** 类型变量。

因此，[20 * 2000] 的相乘结果为 0x9C40，并因为符号扩展而作为 0xFFFF9C40 被分配到 **long** 类型变量。

[15 * 2000] 是 0x7530，并因为不存在符号扩展而以其预定值 0x00007530 被分配。

要获取预定的乘法结果，常数表达式应加上“L”的后缀，以便编译程序能够将常数识别为 **long** 类型。

<实例>

```
long l_max;
    :
  l_max = 20L * 2000L;  // 具有“L”后缀的常数，至少一个后缀已足够
```

11.2 优化连接编辑程序

11.2.1 “未定义的外部符号”的输出

问题

在未使用 XXX 符号时，模块间优化器输出了“未定义的外部符号 (XXX) (Undefined external symbol(XXX))”的消息。为什么？

解答

(1) 是否连接编译程序提供的标准程序库或 EC++ 类程序库？

标准程序库包括 C 程序库函数和运行时例程（执行 C 语言程序所需的操作例程）。

即使用户程序不包括 C 程序库函数，编译程序生成的目标程序有时可能需要标准程序库所提供的函数。在此情形下，模块间优化器的程序库选项必须用来指定标准程序库。

对于 HEW1.2，要指定的标准程序库/EC++ 类程序库必须根据使用的 CPU 类型、使用的优化方法和要传递参数的寄存器数量来选定如下。

		标准 C 程序库		EC++ 类程序库	
CPU/操作模式	**参数传递寄存器的数量：**	**大小优先级**	**速度优先级**	**大小优先级**	**速度优先级**
H8/300	2	c38reg.lib	c38regs.lib	ec2reg.lib	ec2regs.lib
	3	c38reg3.lib	c38regs3.lib	ec2reg3.lib	ec2regs3.lib
H8/300H NRM	2	c38hn.lib	c38hns.lib	ec2hn.lib	ec2hns.lib
	3	c38hn3.lib	c38hns3.lib	ec2hn3.lib	ec2hns3.lib
H8/300H ADV	2	c38ha.lib	c38has.lib	ec2ha.lib	ec2has.lib
	3	c38ha3.lib	c38has3.lib	ec2ha3.lib	ec2has3.lib
H8S NRM	2	c8s26n.lib	c8s26ns.lib	ec226n.lib	ec226ns.lib
	3	c8s26n3.lib	c8s26ns3.lib	ec226n3.lib	ec226ns3.lib
H8S ADV	2	c8s26a.lib	c8s26as.lib	ec226a.lib	ec226as.lib
	3	c8s26a3.lib	c8s26as3.lib	ec226a3.lib	ec226as3.lib

图例：
NRM： 普通模式；ADV： 高级模式

大小效率和速度效率的指定不必根据编译程序选项。但是，CPU 类型和参数传递寄存器的数量必须符合编译程序选项的规格。

对于 HEW2.0 或以上版本，通过选取标准程序库标签类别：(Standard Library tag Category:)，在 [模式 (Mode)] 中，选取创建程序库文件（选项更改）(Build a library file (Option Changed))，来检查已创建的标准程序库。

(2) 由于指定了 I/O 或存储器管理程序库，可能会造成问题。为了指定 C 程序库函数 stdio.h 或 stdlib.h 中声明的函数，必须有低层界面例程。

在创建低层界面例程时，请参考“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户指南 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)”9.2.2 节，执行环境设定 (Execution Environment Settings)。

也请参考包含在样品程序中的低层界面例程的实例。

以下的低层界面例程可用：

名称	函数
open	打开文件。
close	关闭文件。
read	从文件读取。
write	写至文件。
lseek	设定文件读/写位置。
sbrk	分配存储器区域。

11.2.2 “再定位大小溢出”的输出

问题

在连接 HEW2.0 或以上版本的优化连接编辑程序和 HEW1.2 的模块间优化器期间，显示“再定位大小溢出 (Relocation size overflow)”的消息。 我该怎么做？

解答

首先，检查连接映像。

- $ABS8 和 $ABS16 段是否处于可使用 8 位绝对地址和 16 位绝对地址来存取的 CPU 范围内？
- $INDIRECT 段是否包括在 CPU 从 0 到 FF 的范围内？

若使用选项或 #pragma 运算符来指定分配到 8 位绝对地址、16 位绝对地址，或间接存储地址的数据，未被分配到正确地址，此警告消息将会显示。

要获取有关范围的详细资料，请参考个别的编程手册。检查范围，若段被分配在范围以外，必须适当地将它调整。

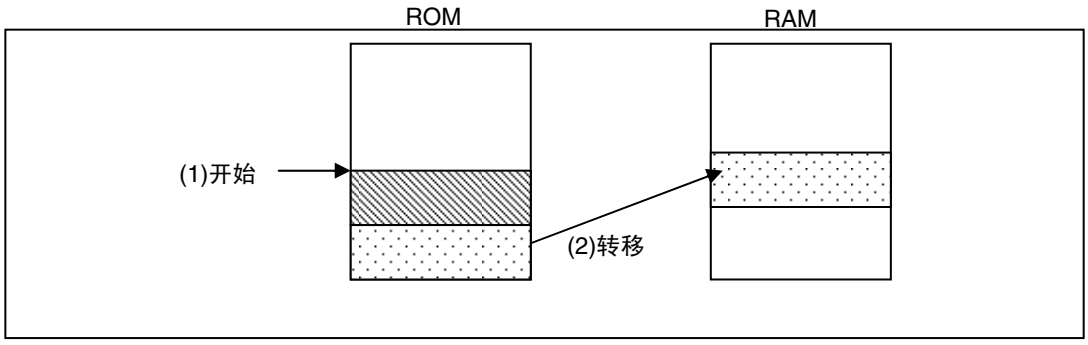
确认汇编编码被正确提供到要嵌入汇编代码的程序位置内。

此消息有时会在尝试将汇编代码嵌入 C/C++ 程序，而转移宽度不适当时显示。

11.2.3 如何在 RAM 中运行程序

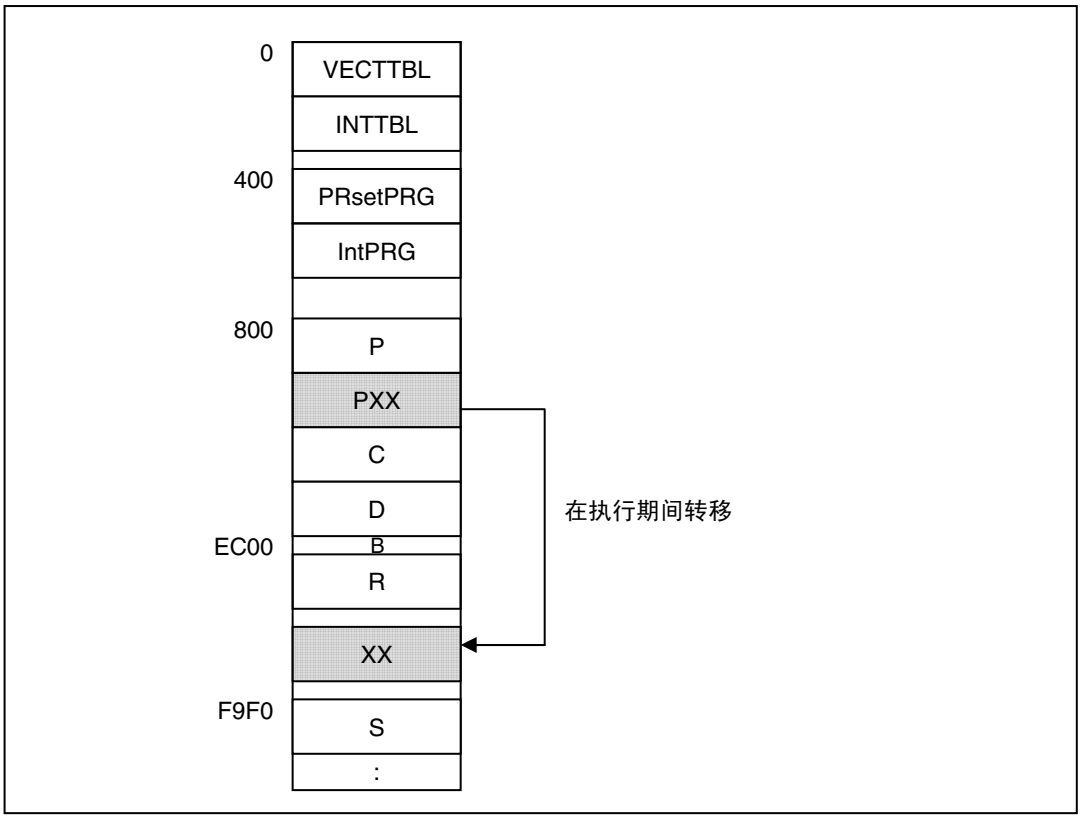
问题

我该如何在快速 RAM 中分配程序？



解答

您可以使用 HEW2.0 或以上版本的优化连接编辑程序和 HEW1.2 的模块间优化器的 ROM 支持功能，在运行时将要执行的程序部分复制到固定地址（在连接时决定），以在 RAM 中执行该程序。



此程序可安装在第 2 节中创建的工程内，如下所示：

首先，在启动时指定将要进行转移，以便在 RAM 中执行的程序段的地址。 此处理将添加到一个现有的文件。

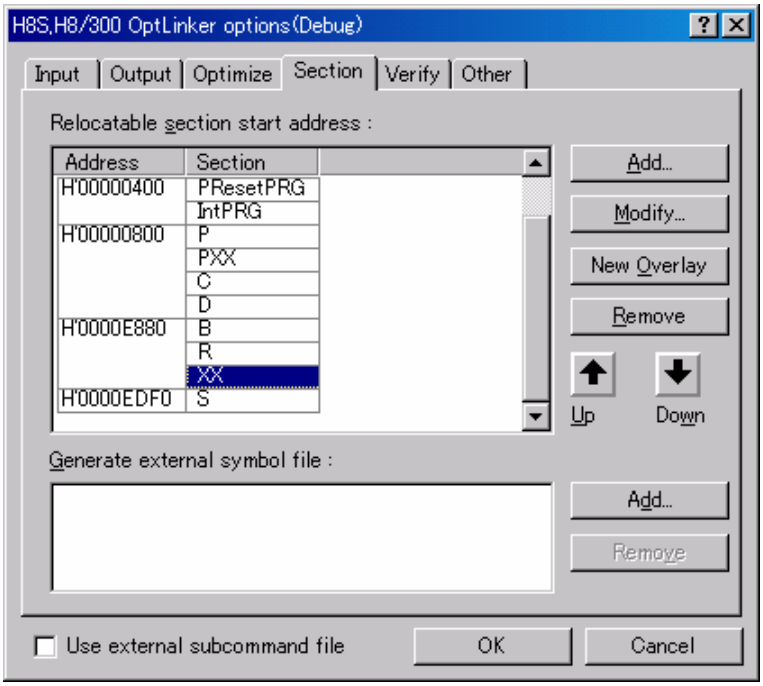
```
#pragma section $DSEC
static const struct {
   char *rom_s;        /* ROM 内已初始化数据段的起始地址 */
   char *rom_e;        /* ROM 内已初始化数据段的终止地址  */
   char *ram_s;        /* RAM 内已初始化数据段的起始地址 */
}DTBL[]= {
   {__sectop("D"), __secend("D"), __sectop("R")},
// {__sectop("$ABS8D"), __secend("$ABS8D"), __sectop("$ABS8R")},
// {__sectop("$ABS16D") , __secend("$ABS16D") , __sectop("$ABS16R") },
   {__sectop("PXX"),__secend("PXX"),__sectop("XX")}
};

#pragma section $BSEC
static const struct {
   char *b_s;          /* 未初始化数据段的起始地址 */
   char *b_e;          /* 未初始化数据段的终止地址 */
}BTBL[]= {
   {__sectop("B"), __secend("B")},
// {__sectop("$ABS8B"), __secend("$ABS8B")},
// {__sectop("$ABS16B"), __secend("$ABS16B")}
};
```

↑设定 PXX 和 XX 段的地址。

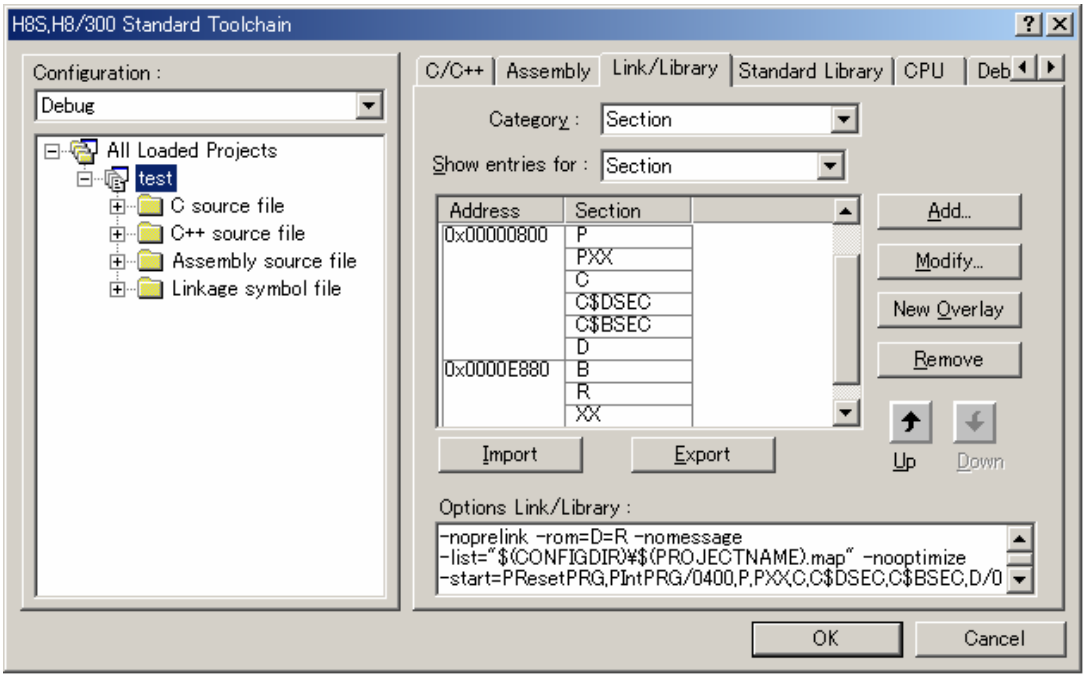
使用此项指定，PXX 段将会在启动时复制到 XX 段。

接着，使用优化连接编辑程序和模块间优化工具来指定目标段 XX 的起始地址。

<HEW1.2>



<HEW 2.0 或以上版本>

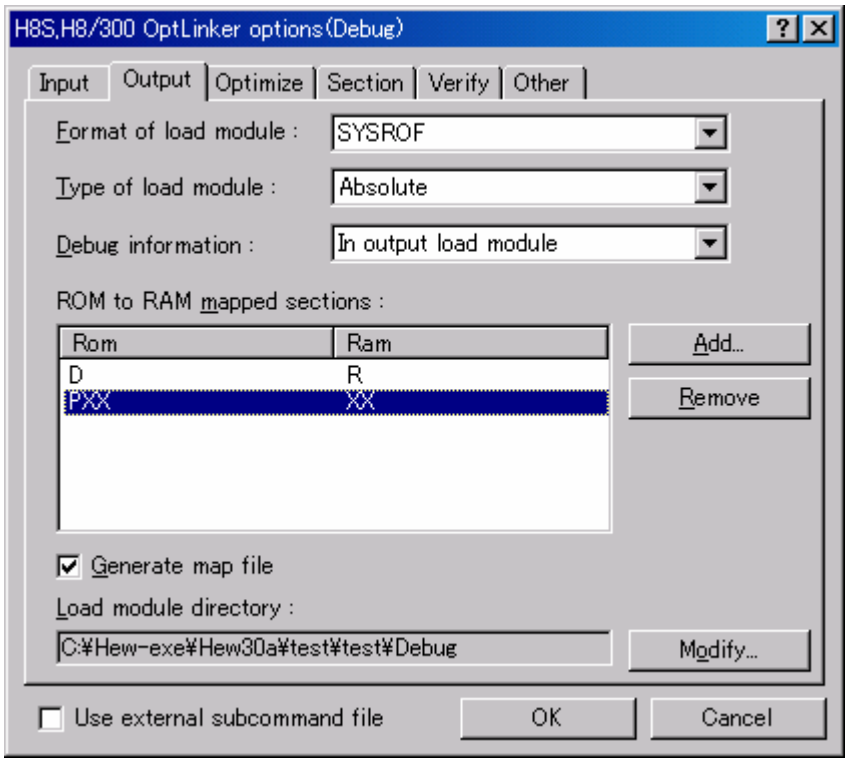


之后，使用 ROM 实施支持选项来分配源段 PXX 在 RAM 中所占有的相同区域。在指定了 ROM 后，此区域的大小将等于 PXX 段的大小。

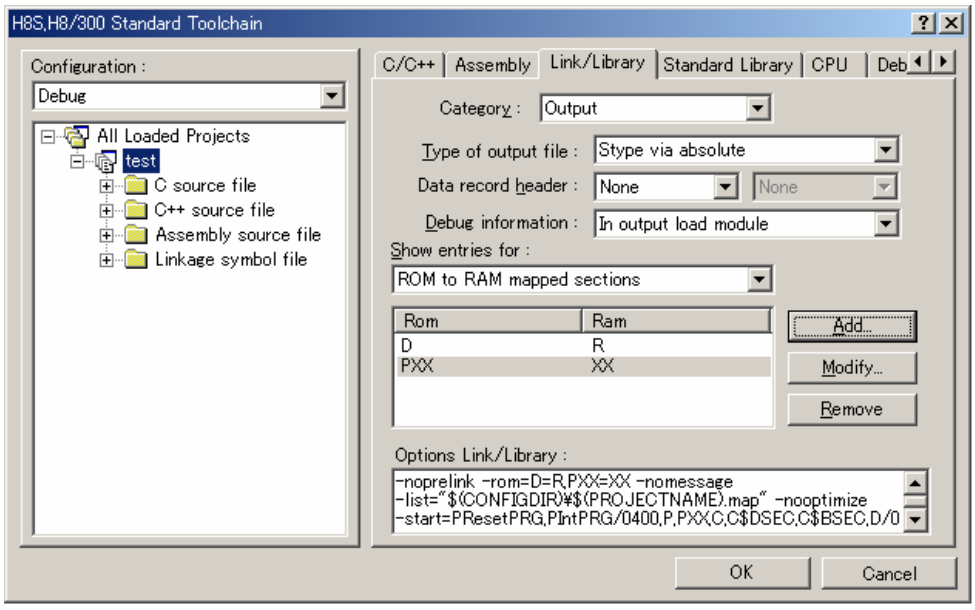
优化连接编辑程序和模块间优化器的此项操作可在子命令中编码如下：

```
            :
start VECTTBL,INTTBL(0),PRsetPRG,IntPRG(0400),P,C,D(0800),B,R,XX(0EC00),S…
            :
```

<HEW1.2>



<HEW 2.0 或以上版本>



在子命令中，使用 rom 如下指定：

```
          :
rom(D,R)
rom(PXX,XX)
          :
```

使用包含此代码的子命令文件，启动优化连接编辑程序和模块间优化器：

<HEW1.2>

% optlnk38 –sub=test.sub(RET) （模块间优化器）

<HEW 2.0 或以上版本>

% optlnk –sub=test.sub(RET) （优化连接编辑程序）

说明与备注

在执行以上处理时，HEW1.2 模块间优化器可能会显示警告消息（1300 段属性在 ROM 选项/子命令 (XX) 中不相符）(1300 SECTION ATTRIBUTE MISMATCH IN ROM OPTION/SUBCOMMAND(XX))。

此消息的显示是由于使用了 __sectop 和 __secend 运算符来指定程序段。 在此情形下，可将它忽略。

HEW2.0 或以上版本的改进一般上已不再导致此消息的显示。 但在下列情况下，警告消息（L1323 (W) 段属性不匹配：“FXX”(L1323 (W) Section attribute mismatch :“FXX”)） 可能会像 HEW1.2 般显示。若是这样，您可以将它忽略。

(1) 程序段 (P) 由 C/C++ 编译程序的段选项更改至其他名称
(2) (1) 的段被指定为源段

11.2.4 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。

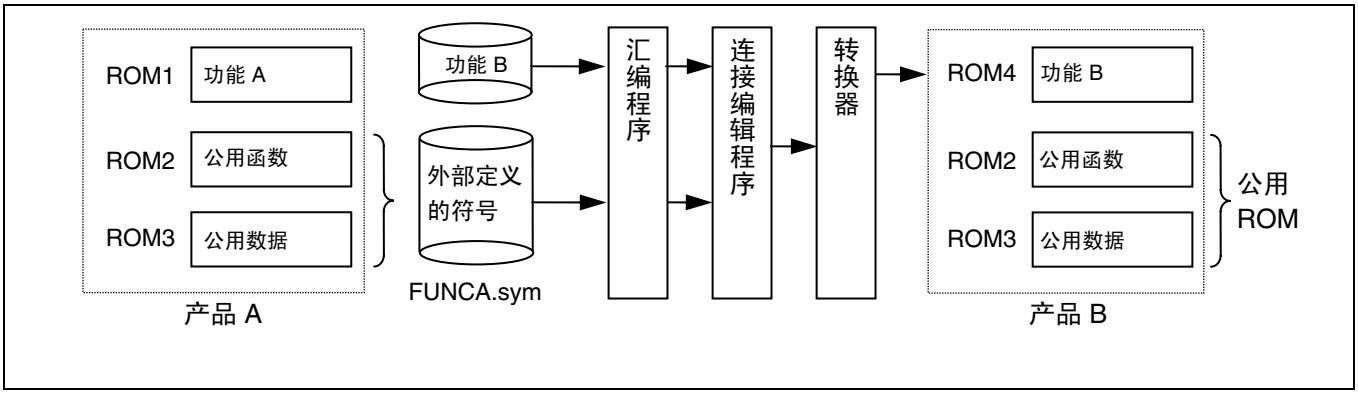
解答

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：

产品 A 的功能 A 被修改为功能 B，以开发产品 B 的说明实例。使用此功能，通过分解共享 ROM 中的符号地址，即可使用公用 ROM。



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 SERIES LINKAGE EDITOR GENERATED FILE   1997.10.10
;fsymbol = sct2, sct3

;SECTION NAME = sct1
.export sym1
sym1:   .equ      h'00FF0080
.export sym2
sym2:   .equ      h'00FF0100
;SECTION NAME = sct2
.export sym3
sym3:   .equ      h'00FF0180
     .end
```

指定汇编和重新连接的实例

asm38ΔROM4
asm38ΔFUNCA.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 的情形下分解。

注意： 使用此步骤时，功能 A 中的符号不能从公用函数参考。

11.2.5 如何实施覆盖

问题

我该如何将不同时存在的段分配给相同的地址？

解答

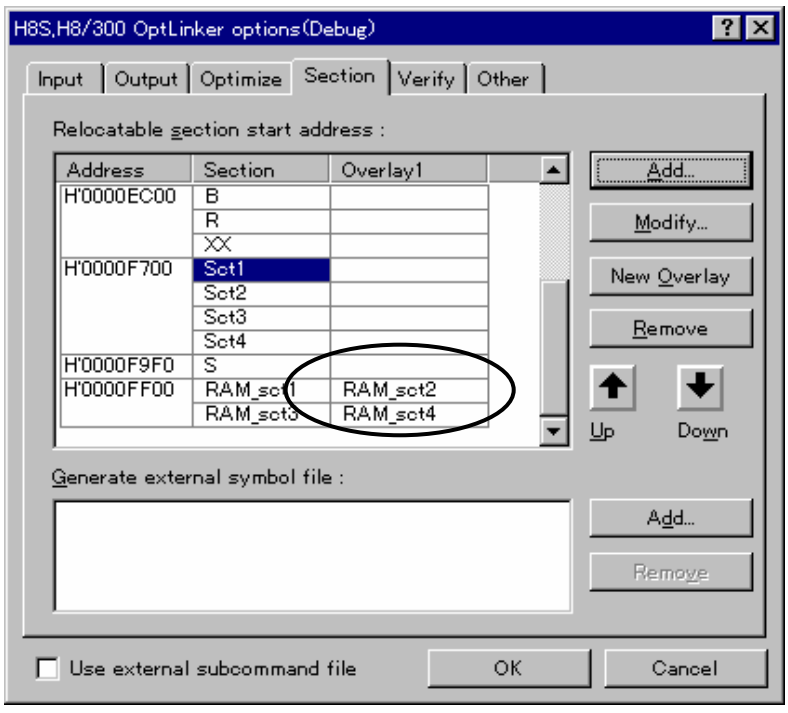
此类赋值可使用 HEW2.0 或以上版本的优化连接编辑程序和 HEW1.2 的模块间优化器选项来指定。

指定方法

<HEW1.2>

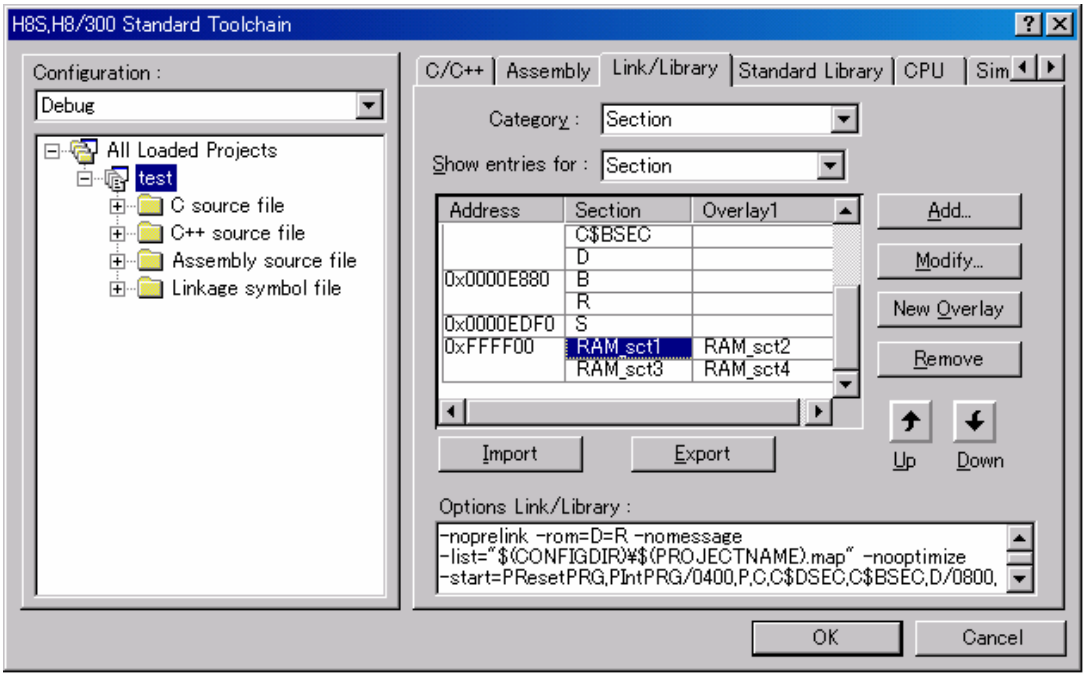
将光标移至：**段 (Section)** 标签，**可再定位的段起始地址 (Relocatable section start address)**：，选取段，以指定“新的覆盖 (New Overlay)”选项。

下面显示指定画面：



<HEW 2.0 或以上版本>

将光标移至： **连接/程序库**标签类别： **(Link/Library** tab Category:**) [段 (Section)]**，选取段，以指定“新的覆盖 (New Overlay)”选项。



若是子命令，则使用开始 (start) 选项：

```
        :
start RAM_Sct1,RAM_Sct3:RAM_Sct2,RAM_Sct4(0FF00)
        :
```

说明与备注

若要在程序的段上指定覆盖，该段将需要被转移。

请参考 11.2.3 节，如何在 RAM 中执行程序。

11.2.6 如何指定未定义的符号错误的输出

问题

我该如何指定错误消息的输出，以便在连接中找到未定义的符号时，禁止输出装入模块？

解答

未定义的符号可使用 HEW1.2 的模块间优化器的选项来检查。

当指定了此选项时，若包含未定义的符号将导致错误消息的显示，同时装入模块的输出将被禁止。

如果没有这项规格，警告消息将在装入模块生成时显示。

指定方法

对话框菜单： **连接/程序库**标签**类别：(Link/Library** Tab **Category:) [其他 (Other)] 杂项：(Miscellaneous options:) 检查未定义的符号 (Check for undefined symbols)**

子命令： *udfcheck*

说明

在 HEW2.0 或以上版本中，未定义的符号始终由优化连接编辑程序进行检查，若发现包含了未定义的符号，错误消息将显示，且装入模块的输出被禁止。

11.2.7 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。

11.2.8 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：0x0-0xFFFF 的一个区域输出到 optlnk test.abs -form=stype -output=test1. mot=0-FFFF test2.mot=10000-1FFFF、test1.mot，0x10000-0x1FFFF 的一个区域输出到 test2.mot。

11.2.9 优化连接编辑程序的输出文件格式

问题

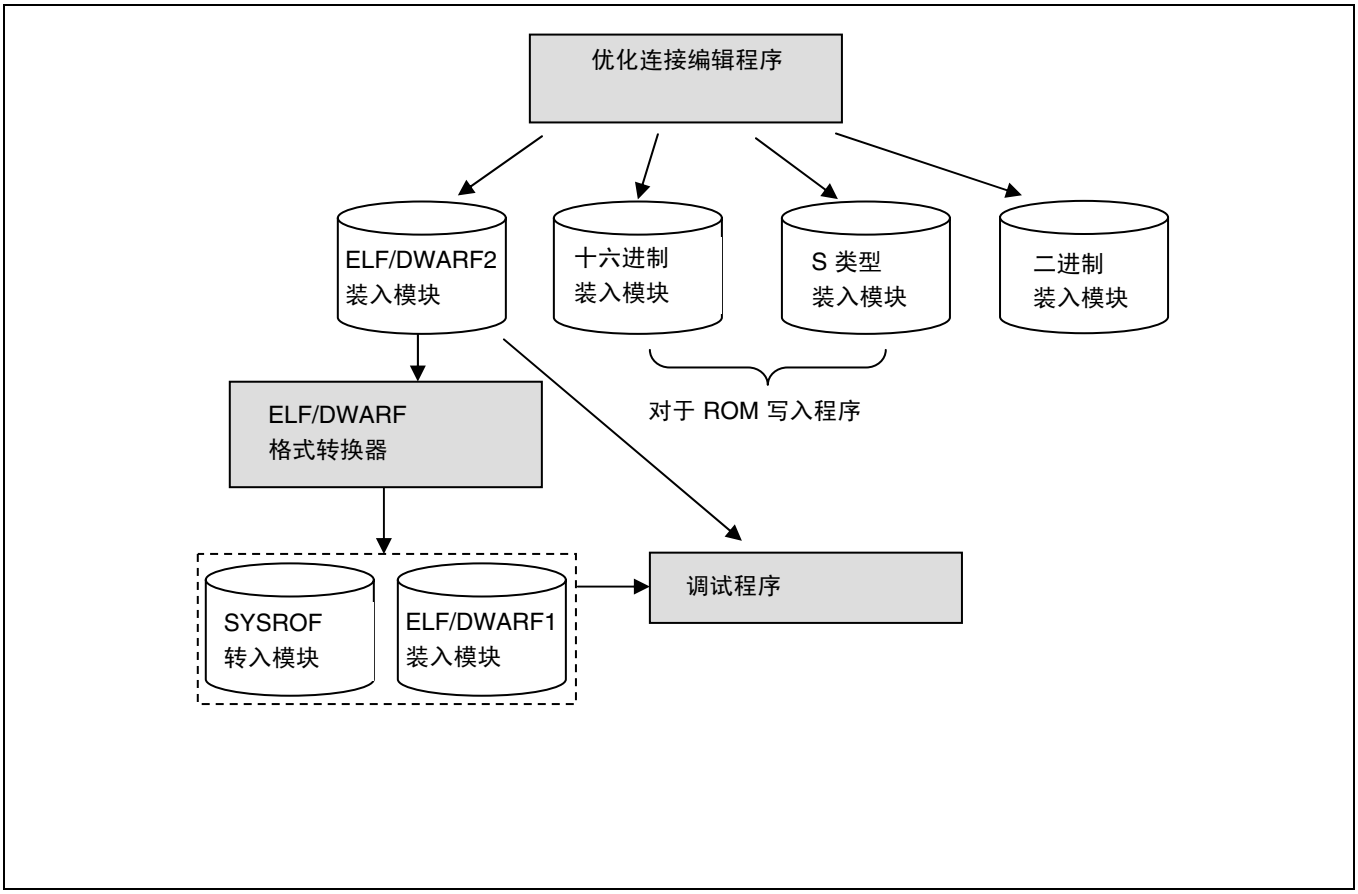
请告诉我 ROM 写入程序可使用的装入模块文件格式。

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 S 类型格式将它输出。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 4.0 或以上版本在调试时以 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。



优化连接编辑程序输出装入模块

11.2.10 如何计算程序大小 (ROM, RAM)

问题

我如何能准确计算 ROM、RAM 的大小？

解答

可使用优化连接编辑程序所输出的列表文件来计算。

指定方法

对话框菜单： **连接/程序库**标签**类别：(Link/Library** Tab **Category:) [列表 (List)] 生成列表文件 (Generate list file)**

命令行： *list=<文件名>*

计算方法

当指定了此选项后，可输出下列列表文件 (*.map)。

在此例中，代码段是 PResetPRG、PIntPRG、P、C$DSEC、C$BSEC 和 D，因此 ROM 大小为 0x00000146。

由 B、R 和 S 获得的 RAM 大小为 0x00000628。

（列表文件的例子）

*** 映像列表 ***

SECTION	START	END	SIZE	ALIGN
PResetPRG				
	00000400	00000415	16	2
PIntPRG				
	00000416	0000048f	7a	2
P				
	00000800	0000089d	9e	2
C$DSEC				
	0000089e	000008a9	c	2
C$BSEC				
	000008aa	000008b1	8	2
D				
	000008b2	000008b5	4	2
B				
	00ffe000	00ffe423	424	2
R				
	00ffe424	00ffe427	4	2
S				
	00ffedc0	00ffefbf	200	2

11.2.11 “段对齐不符”的输出

问题

当段地址运算符如下所示在二进制文件输入中参考二进制文件的段名称时，显示了“段对齐不符”(Section alignment mismatch)的 L1322 警告消息。 我该如何防止这个问题？

[选项指定]
binary=project.bin(BIN_SECTION)

[C/C++ 程序]
```
void main(void)
{
    unsigned char *s_ptr;
    s_ptr = __sectop("BIN_SECTION");

    dumy(s_ptr);
}
```

解答

当指定了段地址运算符 (__sectop, __second) 时，编译程序会如下所示在由编译程序生成的代码中，为所指定的段生成大小为 0，而边界对齐值为 2 的段。

在此情况下，由于二进制段实体在二进制段输入中的边界对齐值是 1，于是相同名称的不同边界对齐便造成了 L1322 警告消息的显示。

此警告消息并不影响程序的执行。

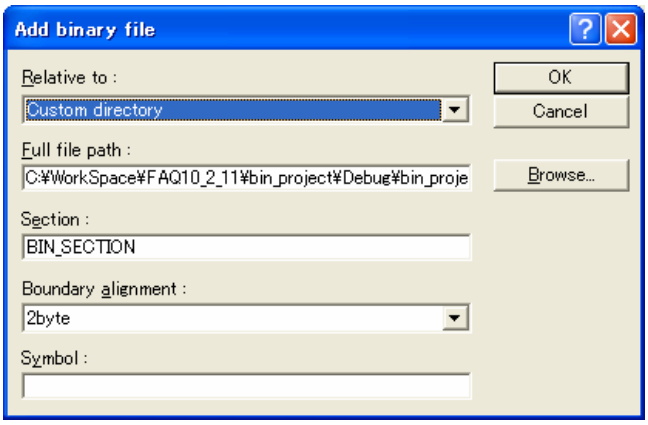
为防止此警告消息，通过优化连接编辑程序在二进制文件输入中指定边界对齐值。

[代码： __sectop used]
```
_main:                          ; function: main
        .STACK     _main=4
        MOV.L      #STARTOF BIN_SECTION,ER0
        BRA        _dumy:8
        .SECTION   BIN_SECTION,DATA,ALIGN=2 ← 段： 大小为 0，边界对齐值为 2
        .END
```

指定实例

对话框菜单： **连接/程序库**标签**类别：(Link/Library** Tab **Category:) [输入 (Input)] 显示有关项目：(Show entries for:) 二进制文件 (Binary files)**

命令行： binary=project.bin(BIN_SECTION:2)



说明

边界对齐值在二进制文件输入中的这项指定对优化连接编辑程序 9.0 或以上版本有效。

有关详情，请参考 9.1.1(4) 节，二进制文件。

11.3 程序库生成程序

11.3.1 可重入的和标准程序库

问题

可以在使用标准程序库时创建可重入的目标程序吗？

解答

在使用设定或参考外部变量的程序库函数时，目标程序不再是可重入的。下表列出可用的可重入程序库，其中符号（ 表示设定变量 _errno 的函数。 若这些函数未在程序中参考变量 _errno，可重入将可用。

可重入程序库的列表

可重入的列　O：可重入 X：不可重入 Δ：设定 _errno。

编号	标准包含文件	编号	函数	可重入	说明
1	stddef.h	1	offsetof	O	宏
2	assert.h	2	assert	×	宏
3	ctype.h	3	isalnum	O	
		4	isalpha	O	
		5	iscntrl	O	
		6	isdigit	O	
		7	isgraph	O	
		8	islower	O	
		9	isprint	O	
		10	ispunct	O	
		11	isspace	O	
		12	isupper	O	
		13	isxdigit	O	
		14	tolower	O	
		15	toupper	O	
4	math.h	16	acos	Δ	浮点
		17	asin	Δ	同上
		18	atan	Δ	同上
		19	atan2	Δ	同上
		20	cos	Δ	同上
		21	sin	Δ	同上
		22	tan	Δ	同上
		23	cosh	Δ	同上
		24	sinh	Δ	同上
		25	tanh	Δ	同上
		26	exp	Δ	同上
		27	frexp	Δ	同上
		28	ldexp	Δ	同上
		29	log	Δ	同上
		30	log10	(	同上
		31	modf	(	浮点
		32	pow	(	同上
		33	sqrt	(	同上
		34	ceil	(	同上
		35	fabs	(	同上
		36	floor	Δ	同上
		37	fmod	Δ	同上

编号	标准包含文件	编号	函数	可重入	说明
5	mathf.h	38	acosf	Δ	浮点
		39	asinf	Δ	同上
		40	atanf	Δ	同上
		41	atan2f	Δ	同上
		42	cosf	Δ	同上
		43	sinf	Δ	同上
		44	tanf	Δ	同上
		45	coshf	Δ	同上
		46	sinhf	Δ	同上
		47	tanhf	Δ	同上
		48	expf	Δ	同上
		49	frexpf	Δ	同上
		50	ldexpf	Δ	同上
		51	logf	Δ	同上
		52	log10f	Δ	同上
		53	modff	Δ	同上
		54	powf	Δ	同上
		55	sqrtf	Δ	同上
		56	ceilf	Δ	同上
		57	fabsf	Δ	同上
		58	floorf	Δ	同上
		59	fmodf	Δ	同上
6	setjmp.h	60	setjmp	O	
		61	longjmp	O	
7	stdarg.h	62	va_start	O	宏
		63	va_arg	O	宏
		64	va_end	O	宏
8	stdio.h	65	fclose	×	
		66	fflush	×	
		67	fopen	×	
		68	freopen	×	
		69	setbuf	×	
		70	setvbuf	×	
		71	fprintf	×	
		72	fscanf	×	
		73	printf	(	
		74	scanf	(	
		75	sprintf	(	
		76	sscanf	(	
		77	vfprintf	(	
		78	vprintf	(	

编号	标准包含文件	编号	函数	可重入	说明
8	stdio.h	79	vsprintf	Δ	
		80	fgetc	×	
		81	fgets	×	
		82	fputc	×	
		83	fputs	×	
		84	getc	(	
		85	getchar	(	
		86	gets	(	
		87	putc	(	
		88	putchar	(	
		89	puts	(	
		90	ungetc	(	
		91	fread	(	
		92	fwrite	(	
		93	fseek	(	
		94	ftell	(	
		95	rewind	(	
		96	clearerr	(	
		97	feof	(	
		98	ferror	(	
		99	perror	(	
9	stdlib.h	100	atof	(	非 ANSI
		101	atoi	(	同上
		102	atol	(	同上
		103	strtod	(	
		104	strtol	(	
		105	rand	(	浮点
		106	srand	(	
		107	calloc	(	
		108	free	(	
		109	malloc	(	
		110	realloc	(	
		111	bsearch	O	
		112	qsort	O	递归函数
		113	abs	O	
		114	div	(	
		115	labs	O	
		116	ldiv	(	
10	string.h	117	memcpy	O	
		118	strcpy	O	
		119	strncpy	O	

编号	标准包含文件	编号	函数	可重入	说明
10	string.h	120	strcat	O	
		121	strncat	O	
		122	memcmp	O	
		123	strcmp	O	
		124	strncmp	O	
		125	memchr	O	
		126	strchr	O	
		127	strcspn	O	
		128	strpbrx	O	
		129	strrchr	O	
		130	strspn	O	
		131	strstr	O	
		132	strtok	×	
		133	memset	O	
		134	strerror	O	
		135	strlen	O	
		136	memmove	O	

11.3.2 要在标准程序库文件中使用可重入程序库函数

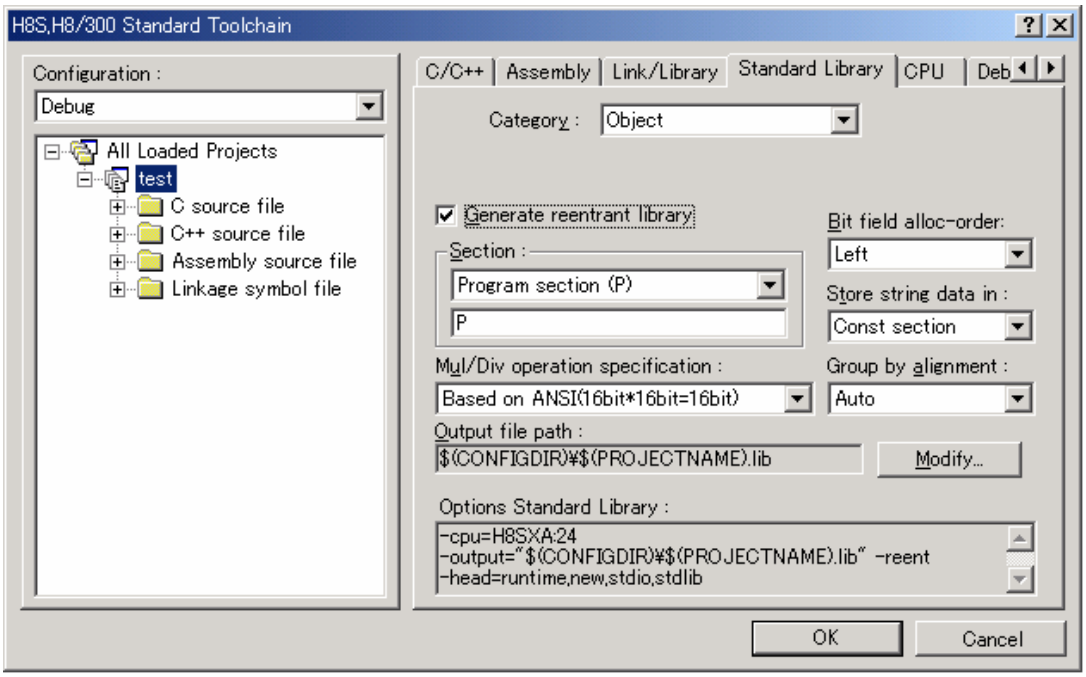
问题

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 H8C 6.0 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 lbg38 -reent 选项。
- HEW 中的设定在下面显示。



标准程序库对话框

11.3.3 没有标准程序库文件（H8C V4 或以上版本）

问题

H8C V3 中有数种类型的标准程序库。

但 H8C V4 或以上版本却没有标准程序库文件。

解答

从 H8C V4 开始，标准程序库的指定有所更改，同时选项变得能够被指定。这使用户能通过选项调谐标准程序库。

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 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")]。

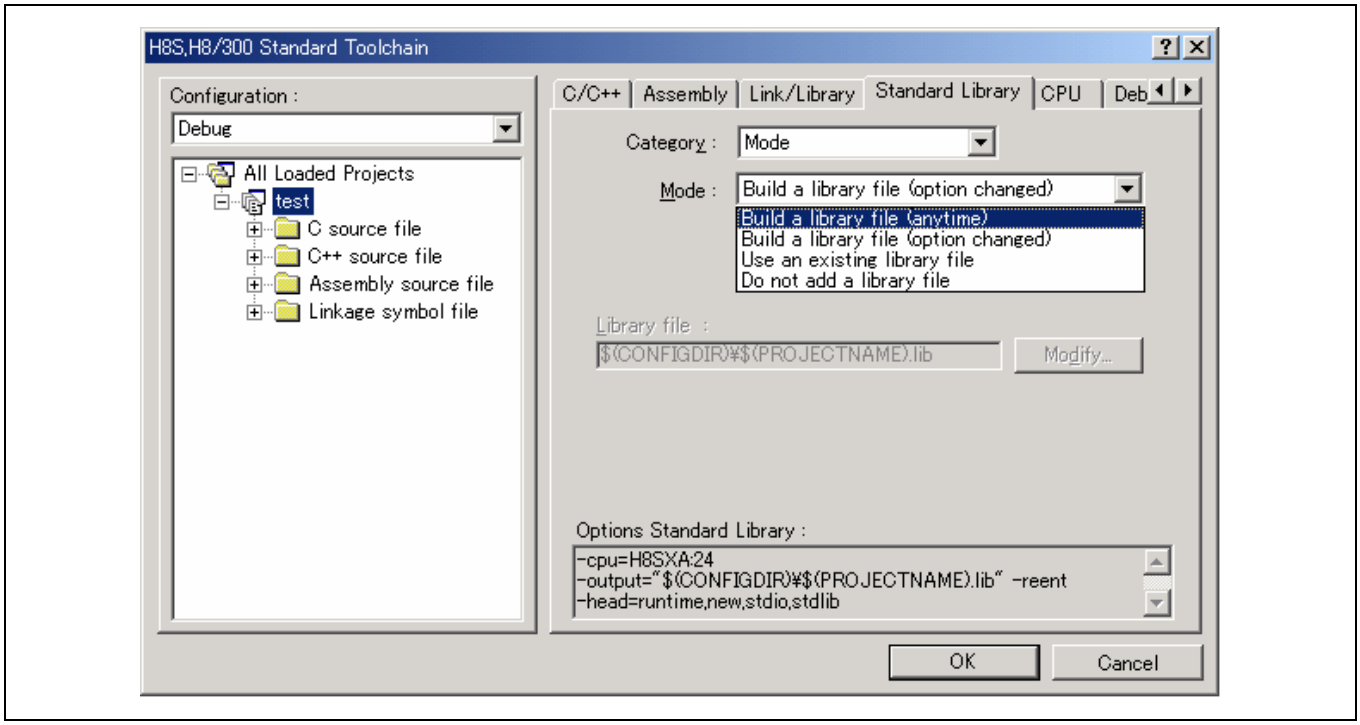
解答

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。

若您在 HEW/[选项 (OPTIONS)]/[H8S，H8 标准工具链… (H8S,H8 Standard Toolchain…)] 的 [标准程序库 (Standard Library)] 模式中选取了“使用现有的程序库文件” (Use an existing library file)，该警告将不会发出。 当您在 HEW 中选择“全部建”(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)]/[H8S，H8 标准工具链… (H8S,H8 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“使用现有的程序库文件”(Use an existing library file)，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。



标准程序库对话框

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并不足够，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 如何缩减 I/O 程序库的 ROM 大小

问题

我该如何为标准包含文件缩减 I/O 程序库的 ROM 大小？

解答

当指定了 no_float.h 包含文件时，不包含浮点转移处理的简单 I/O 函数可被使用。

此项指定可用于以下函数：

fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf

添加 no_float.h 选项以将文件指定为标准 I/O 文件 stdio.h 前的包含文件。

实例：

```
#include <no_float.h>

#include <stdio.h>
void main(void)
{
    printf("HELLO¥n");
}
```

← **宏声明**

对于使用现有标准 I/O 程序库的文件，则使用预包含选项。

使用简单 I/O 函数时，ROM 大小将在执行文件的 I/O 操作时缩减。

然而，若此选项和浮点 (%f, %e, %E, %g,%E) 规格一同指定，运行时的执行将不被保证。

11.3.7 如何编辑程序库文件

问题

我如何能编辑程序库文件，以重新使用现有的程序库文件？

解答

可通过在优化连接编辑程序中指定选项来进行编辑。 这些选项的使用显示如下。

H 系列库管理程序界面可从 GUI 启动优化连接编辑程序。

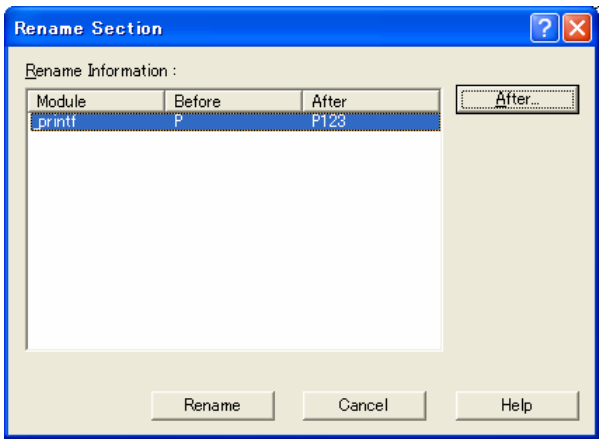
如何启动 H 系列库管理程序界面

在 HEW 中选择 [工具 (Tools) ->Hitachi H 系列库管理程序界面 (Hitachi H Series Librarian Interface)]，以启动 H 系列库管理程序界面。

(A) 在程序库中修改模块的段名称

程序库中所指定模块的段名称可被修改，以在任何地址查找段。

(1) 打开程序库，选择模块以分配到任何地址。
(2) 通过 [操作 (Action) -> 重命名段… (Rename Section…)] 显示下列对话框，并使用**之后 (After)** 按钮修改段名称。



命令行： optlnk –lib=<程序库文件名> -rename=<程序库中的模块名称>(P=P123)

(B) 替换程序库中的模块/添加模块到程序库

程序库中的模块可被替换。新的模块可被添加到程序库。

(1) 打开程序库，选择 [操作 (Action) -> 添加/替换… (Add/Replace…)]。
(2) 使用相同名称打开模块进行替换。使用新名称打开模块进行添加。

命令行： optlnk –lib=<程序库文件名> - replace=<程序库中的模块名称>

(C) 删除程序库中的模块

程序库中的模块可被删除。

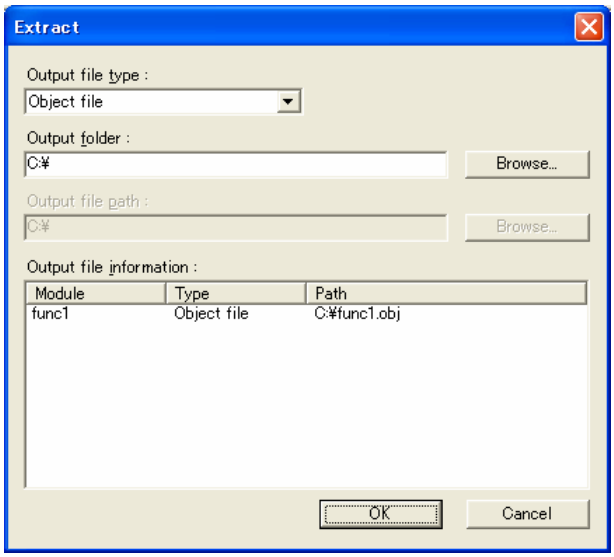
(1) 打开程序库，选择要删除的模块。（可复选）
(2) 通过 [操作 (Action) -> 删除… (Delete…)] 显示**删除 (Delete)** 对话框，然后按下**删除 (Delete)** 按钮。

命令行： optlnk –lib=<程序库文件名> - delete=<程序库中的模块名称>

(D) 提取程序库中的模块

程序库中的模块可被提取。

(1) 打开程序库，选择要提取的模块。（可复选）
(2) 通过 [操作 (Action) -> 提取… (Extract…)] 显示下列对话框，指定**输出文件夹 (Output folder)**，然后按下确定 (OK) 按钮。
(3) 所指定的模块将被输出到指定的**输出文件夹 (Output folder)**。（下例中为 C:\）



命令行： optlnk –lib=<程序库文件名> - extract=<程序库中的模块名称> -form=<输出文件类型>

在这里，输出文件类型是此例目标。

11.4 HEW

11.4.1 显示对话框菜单失败

问题

HEW 的工具选项 (Tools Options) 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。

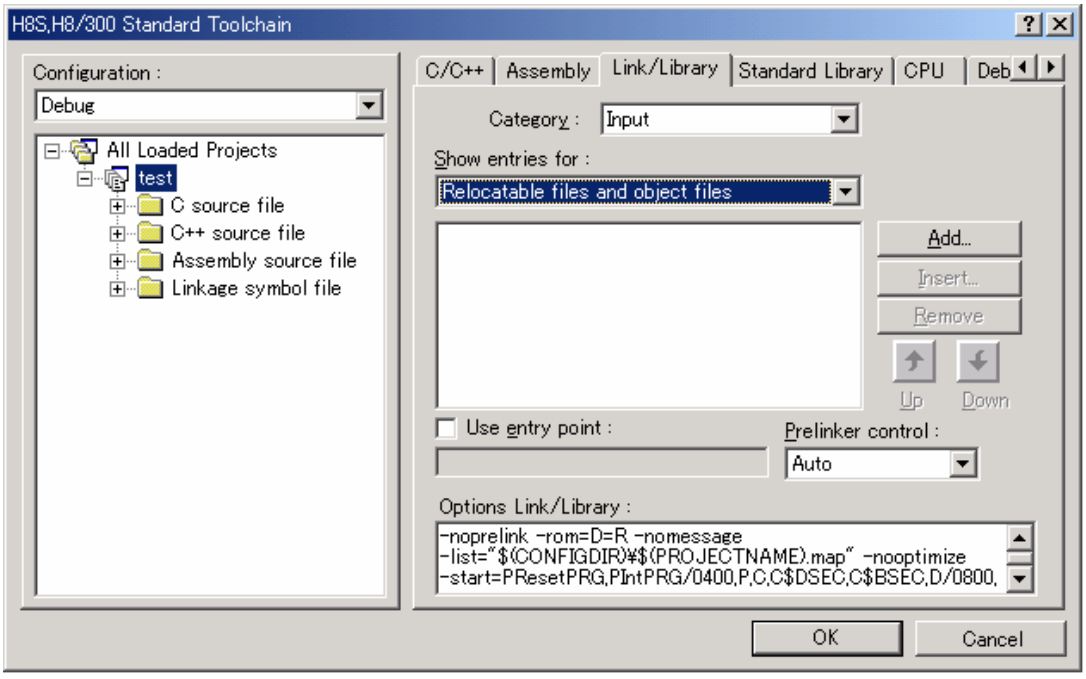
11.4.2 目标文件的连接顺序

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“H8S，H8 标准工具链…”(H8S, H8 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。

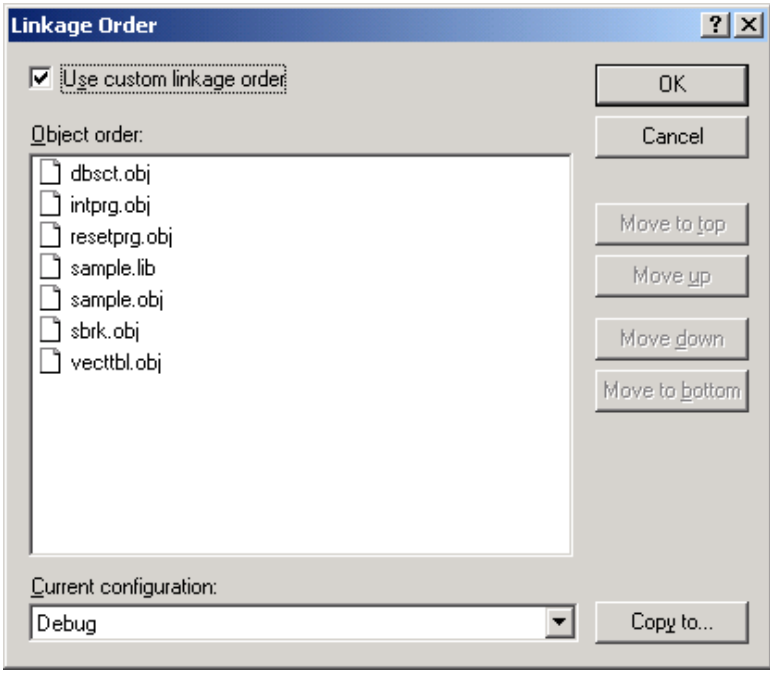


连接/程序库对话框

H8C 6.00 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。



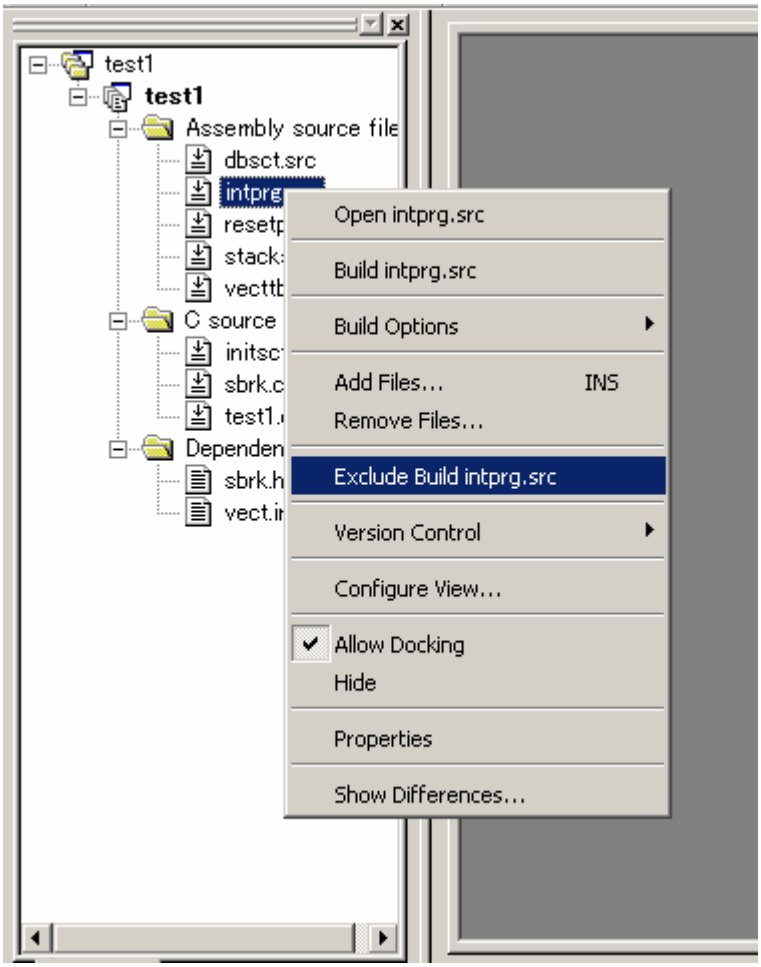
11.4.3 排除工程文件

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>)]。



排除创建菜单

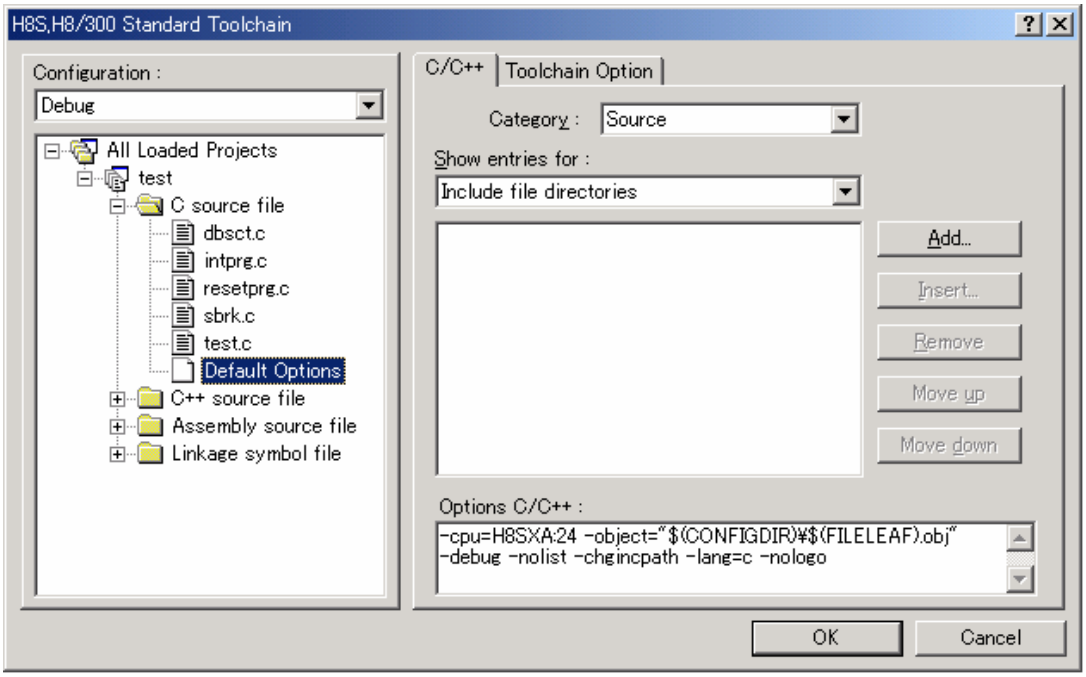
11.4.4 为工程文件指定默认选项

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“H8S，H8 标准工具链”(H8S, H8 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。“默认选项”图标将显示在文件夹中。请选择一个图标，在“选项”(option) 对话框的右侧指定选项，然后单击“确定”（OK）。此选项可以在首先将文件组中的文件添加到工程时应用。



默认选项

11.4.5 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。

11.4.6 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。

11.4.7 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

11.4.8 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。

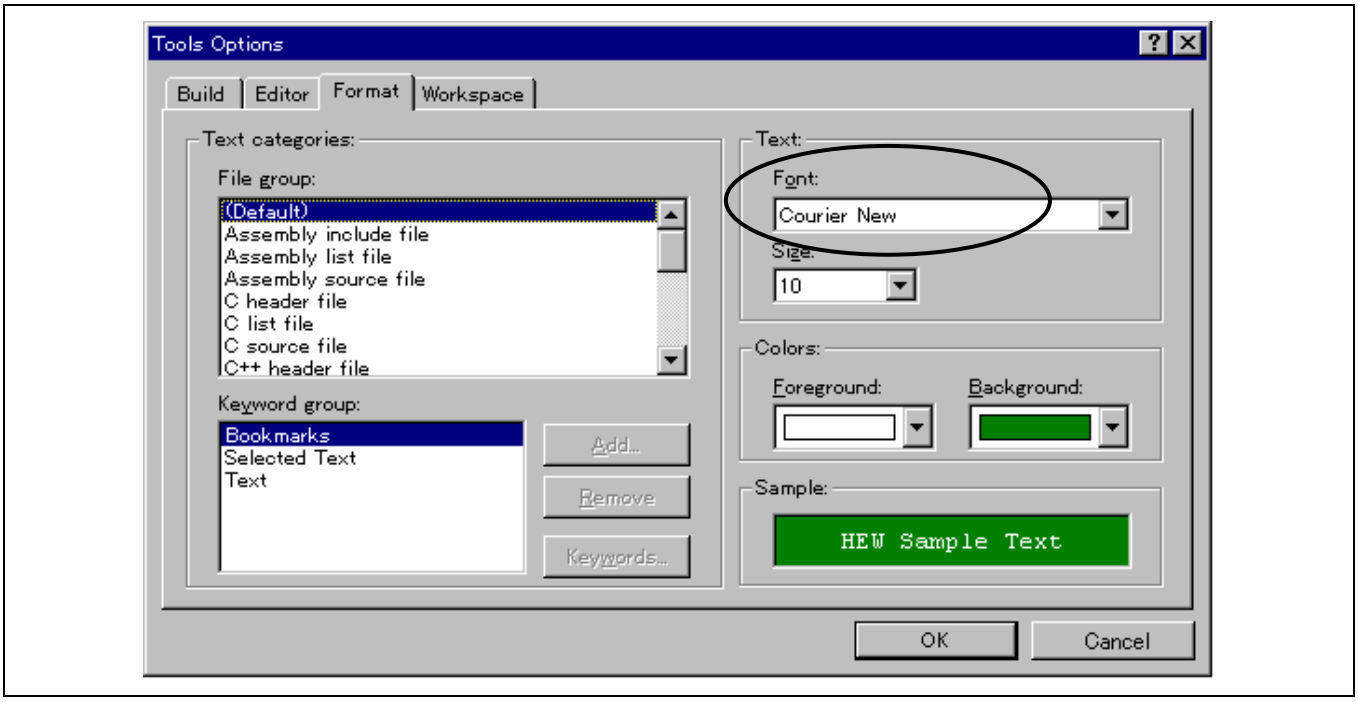
解答

使用 HEW 编辑程序编码日文时，指定日文字体如下：

使用工具 (Tools)->选项 (Options) 中的**格式 (Format)** 标签的**文本 (Text)** 列内的**字体 (Font)**:

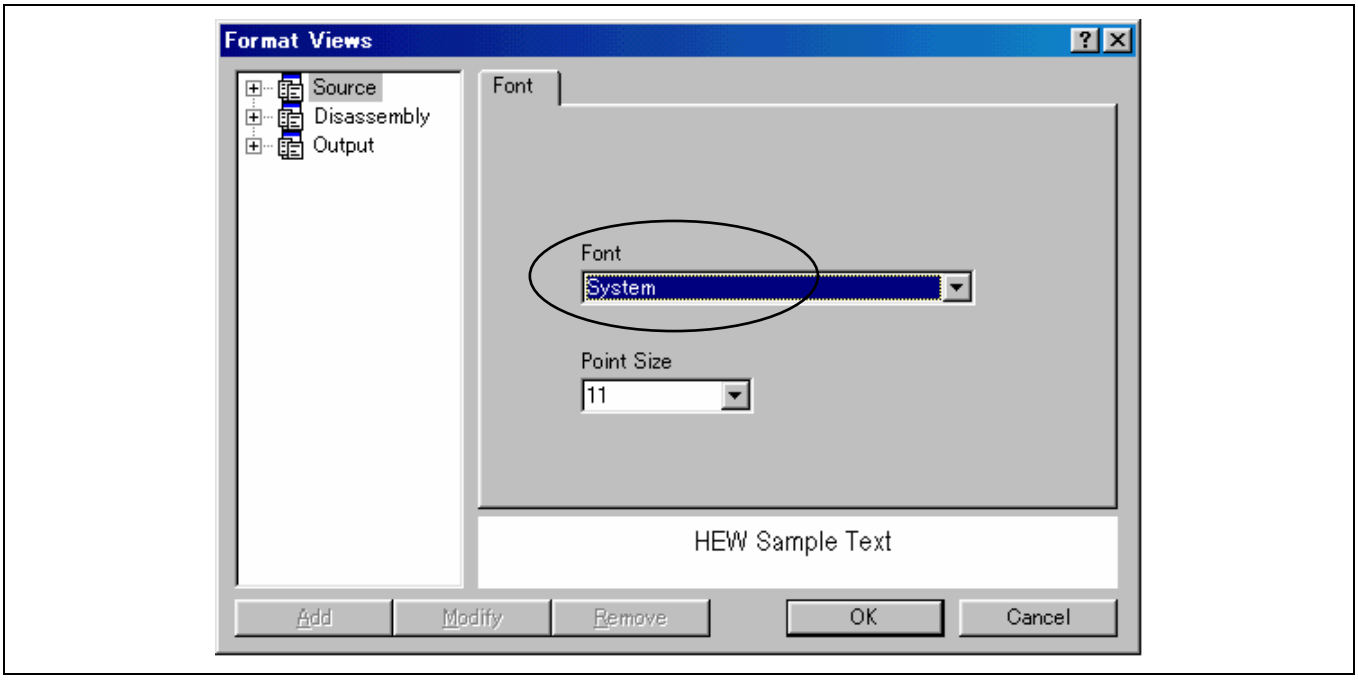
<HEW1.2>

使用工具 (Tools)->选项 (Options) 中的**格式 (Format)** 标签的**文本 (Text)** 列内的**字体 (Font)**：



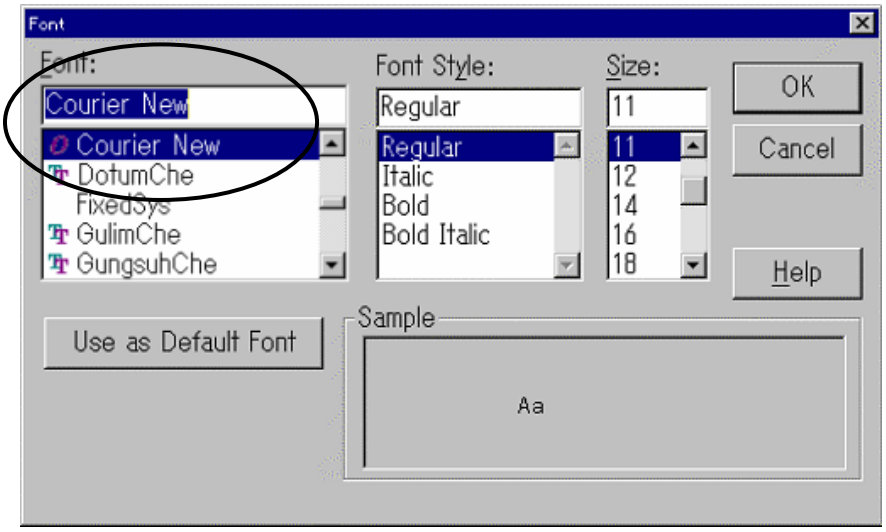
<HEW 2.0 或以上版本>

使用工具 (Tools)-> 格式视图 (Format Views) 中的**字体 (Font)** 标签上的**字体 (Font)**：



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)…]



11.4.9 如何将程序从 HIM 转换到 HEW

问题

我该如何在 HEW 上使用 HIM（Hitachi 集成管理员）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。

11.4.10 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。

[H8C 3.0 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。

（注意）

若您有 H8C 3.0A 版本编译程序封装，则可使用“HIM 到 HEW 的工程转换器”(HIM to HEW Project Converter)。

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 H8C 3.0A 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。

[H8C 3.0A 版本]

< 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。

[H8C 4 版本]

< 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。

[H8C 5 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。

H8S, H8/300 系列 C/C++编译程序应用笔记

浮点算术操作性能列表

附录 A 浮点算术操作性能列表

A. 浮点操作性能

A.1 单精度浮点操作性能

A.1.1 单精度浮点操作性能（H8/300，H8/300H，H8S/2600）

					H8/300H		H8S/2000,H8S/2600	
编号	函数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
1	acos	0.4	-	30,850	23,316	25,636	8,495	8,852
		1.57075	-	3,830	3,022	3,484	930	999
		0.6	-	30,864	23,226	25,546	8,450	8,806
		-0.4	-	30,918	23,302	25,622	8,496	8,853
2	asin	0.4	-	29,840	22,390	24,576	8,158	8,494
		1.57075	-	2,904	2,194	2,522	642	690
		0.6	-	29,850	22,310	24,496	8,118	8,453
		-0.4	-	29,926	22,402	24,588	8,172	8,508
3	atan	0.11	-	13,166	9,948	11,010	3,581	3,767
		0.27	-	18,122	14,302	15,718	5,269	5,502
		0.547	-	17,964	14,128	15,544	5,179	5,412
		0.777	-	18,890	14,820	16,270	5,436	5,672
		0.975	-	17,924	14,210	15,582	5,277	5,502
		54.45	-	21,834	17,744	19,390	6,659	6,922
		154.233	-	21,952	17,840	19,486	6,707	6,970
		-54.45	-	21,920	17,754	19,400	6,672	6,935
		-0.975	-	18,010	14,220	15,592	5,290	5,515
		-0.777	-	18,976	14,830	16,280	5,449	5,685
4	atan2	0.3	0.7	20,898	16,758	18,494	6,182	6,441
		0.2	0.1	24,736	20,204	22,214	7,523	7,820
		0.1	0.9	16,156	12,648	14,030	4,619	4,831
5	cos	0.523333333	-	11,124	8,148	8,780	3,114	3,262
		1.046666667	-	13,090	9,610	10,506	3,588	3,757
		1.9625	-	12,420	9,024	9,842	3,404	3,562
		2.7475	-	12,074	8,932	9,642	3,398	3,557
		3.5325	-	11,332	8,284	8,916	3,183	3,331
		4.3175	-	13,184	9,748	10,644	3,656	3,825
		5.1025	-	12,462	9,114	9,932	3,448	3,606
		5.8875	-	12,050	8,960	9,670	3,411	3,570
		-0.52333333	-	11,210	8,158	8,790	3,127	3,275
		-1.04666667	-	13,176	9,620	10,516	3,601	3,770

					H8/300H		H8S/2000,H8S/2600	
编号	函数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
5	cos	-1.9625	-	12,506	9,034	9,852	3,417	3,575
		-2.7475	-	12,160	8,942	9,652	3,411	3,570
		-3.5325	-	11,418	8,294	8,926	3,196	3,344
		-4.3175	-	13,270	9,758	10,654	3,669	3,838
		-5.1025	-	12,548	9,124	9,942	3,461	3,619
		-5.8875	-	12,136	8,970	9,680	3,424	3,583
6	sin	0.523333333	-	12,170	8,838	9,656	3,314	3,472
		1.046666667	-	11,942	8,872	9,582	3,373	3,532
		1.9625	-	11,196	8,202	8,834	3,147	3,295
		2.7475	-	13,240	9,764	10,660	3,671	3,840
		3.5325	-	12,482	9,060	9,878	3,428	3,586
		4.3175	-	12,120	8,954	9,664	3,415	3,574
		5.1025	-	11,382	8,312	8,944	3,203	3,351
		5.8875	-	13,204	9,776	10,672	3,674	3,843
		-0.52333333	-	12,348	8,876	9,694	3,342	3,500
		-1.04666667	-	12,120	8,910	9,620	3,401	3,560
		-1.9625	-	11,374	8,240	8,872	3,175	3,323
		-2.7475	-	13,418	9,802	10,698	3,699	3,868
		-3.5325	-	12,660	9,098	9,916	3,456	3,614
		-4.3175	-	12,298	8,992	9,702	3,443	3,602
		-5.1025	-	11,560	8,350	8,982	3,231	3,379
		-5.8875	-	13,382	9,814	10,710	3,702	3,871
7	tan	0.3925	-	16,682	12,494	13,374	4,768	4,997
		1.1775	-	17,522	13,240	14,198	5,055	5,276
		1.9625	-	16,908	12,634	13,514	4,863	5,074
		2.7475	-	17,696	13,344	14,302	5,111	5,332
8	cosh	0.33	-	44,886	33,624	35,796	13,237	13,735
		0.78	-	46,018	34,462	36,646	13,354	13,864
		-0.33	-	44,904	33,636	35,808	13,243	13,741
		-0.78	-	46,036	34,474	36,658	13,360	13,870
9	sinh	0.33	-	12,538	9,004	9,660	3,375	3,520
		0.98	-	47,040	35,310	37,568	13,689	14,209
		-0.33	-	12,538	9,004	9,660	3,375	3,520
		-0.98	-	47,058	35,322	37,580	13,695	14,215
10	tanh	0.0033+00	-	9,772	7,102	7,710	2,553	2,672
11	exp	0.33	-	21,860	16,184	17,180	6,471	6,713
		0.98	-	22,588	16,740	17,742	6,598	6,846
		-0.33	-	21,980	16,212	17,208	6,485	6,727
		-0.98	-	22,684	16,732	17,734	6,594	6,842

					H8/300H		H8S/2000,H8S/2600	
编号	函数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
12	frexp	0.3	-	186	102	118	54	60
		400	-	186	102	118	54	60
13	ldexp	0.3	30	1,382	964	1,068	316	337
		0.1	100	1,382	964	1,068	316	337
14	log	1.2	-	18,766	14,272	15,410	5,353	5,575
		2.5	-	18,882	14,476	15,614	5,455	5,677
		0.999	-	19,376	14,996	16,134	5,715	5,937
		0.3	-	19,016	14,604	15,742	5,519	5,741
15	log10	1.2	-	20,138	15,260	16,532	5,686	5,929
		2.5	-	20,254	15,464	16,736	5,788	6,031
		0.999	-	20,764	16,008	17,280	6,060	6,303
		0.3	-	20,372	15,572	16,844	5,482	6,085
16	modf	256.3	-	3,518	2,890	3,388	914	975
		0.032	-	3,342	2,760	3,252	850	908
		10000.2345	-	3,608	2,962	3,460	950	1,011
17	pow	2.3	4.2	43,236	33,010	35,340	12,577	13,074
		45.2	-5	43,642	33,412	35,742	12,789	13,286
		-4.56	-3	47,134	36,326	39,066	13,678	14,231
		-85.55	476	45,988	35,406	38,064	13,360	13,904
18	sqrt	2	-	4,918	1,878	1,980	829	852
		3	-	4,966	1,910	2,012	845	868
		0.1	-	4,906	1,890	1,992	835	858
19	ceil	0.3	-	2,998	2,452	2,790	749	801
		-0.6	-	1,806	1,314	1,502	393	426
20	fabs	5	-	126	38	40	24	27
		-5	-	126	38	40	24	27
21	floor	0.3	-	1,806	1,314	1,502	393	426
		-0.6	-	2,998	2,446	2,784	746	798
22	fmod	11.1	3.2	1,964	1,498	1,654	533	564
		500.55	0.4	2,436	1,858	2,014	713	744
		1.05E+06	9.54E-07	4,178	3,186	3,342	1,377	1,408

A.1.2 单精度浮点操作性能（H8SX）

				H8SX		
编号	函数	参数 1	参数 2	NRM	ADV	MAX
1	acos	0.4	-	6,108	6,403	6,397
		1.57075	-	495	438	438
		0.6	-	6,079	6,373	6,368
		-0.4	-	6,100	6,396	6,390
2	asin	0.4	-	5,880	6,219	6,220
		1.57075	-	340	309	309
		0.6	-	5,885	6,195	6,196
		-0.4	-	5,884	6,225	6,226
3	atan	0.11	-	2,167	2,550	2,549
		0.27	-	3,542	4,012	4,011
		0.547	-	3,449	3,919	3,918
		0.777	-	3,643	4,122	4,121
		0.975	-	3,595	4,055	4,054
		54.45	-	4,769	5,308	5,307
		154.233	-	4,828	5,368	5,367
		-54.45	-	4,773	5,314	5,313
		-0.975	-	3,599	4,061	4,060
		-0.777	-	3,647	4,128	4,127
4	atan2	0.3	0.7	4,370	4,811	4,806
		0.2	0.1	5,570	6,088	6,085
		0.1	0.9	3,146	3,497	3,493
5	cos	0.523333333	-	2,039	2,347	2,349
		1.046666667	-	2,229	2,658	2,660
		1.9625	-	2,208	2,543	2,547
		2.7475	-	2,222	2,552	2,556
		3.5325	-	2,201	2,408	2,411
		4.3175	-	2,371	2,732	2,737
		5.1025	-	2,256	2,593	2,597
		5.8875	-	2,240	2,572	2,576
		-0.52333333	-	2,045	2,351	2,353
		-1.04666667	-	2,305	2,662	2,664
		-1.9625	-	2,214	2,547	2,551
		-2.7475	-	2,228	2,556	2,560
		-3.5325	-	2,108	2,412	2,415
		-4.3175	-	2,377	2,736	2,741
		-5.1025	-	2,262	2,597	2,601
		-5.8875	-	2,246	2,576	2,580

				H8SX		
编号	函数	参数 1	参数 2	NRM	ADV	MAX
6	sin	0.523333333	-	2,385	2,459	2,460
		1.046666667	-	2,464	2,537	2,539
		1.9625	-	2,308	2,377	2,380
		2.7475	-	2,650	2,728	2,733
		3.5325	-	2,497	2,571	2,575
		4.3175	-	2,501	2,574	2,578
		5.1025	-	2,361	2,430	2,433
		5.8875	-	2,663	2,741	2,746
		-0.52333333	-	2,397	2,468	2,469
		-1.04666667	-	2,476	2,546	2,548
		-1.9625	-	2,320	2,386	2,389
		-2.7475	-	2,662	2,737	2,742
		-3.5325	-	2,509	2,580	2,584
		-4.3175	-	2,513	2,583	2,587
		-5.1025	-	2,373	2,439	2,442
		-5.8875	-	2,675	2,750	2,755
7	tan	0.3925	-	3,366	3,775	3,771
		1.1775	-	3,566	4,000	3,996
		1.9625	-	3,448	3,854	3,850
		2.7475	-	3,609	4,040	4,036
8	cosh	0.33	-	10,276	9,214	9,214
		0.78	-	10,294	9,237	9,237
		-0.33	-	10,272	9,219	9,219
		-0.78	-	10,299	9,242	9,242
9	sinh	0.33	-	2,413	2,110	2,110
		0.98	-	10,623	9,548	9,548
		-0.33	-	2,413	2,110	2,110
		-0.98	-	10,628	9,553	9,553
10	tanh	0.0033+00	-	1,604	1,553	1,552
11	exp	0.33	-	5,110	4,564	4,556
		0.98	-	5,215	4,667	4,663
		-0.33	-	5,116	4,570	4,562
		-0.98	-	5,221	4,673	4,669
12	frexp	0.3	-	41	42	42
		400	-	41	42	42
13	ldexp	0.3	30	220	196	196
		0.1	100	220	196	196

				H8SX		
编号	函数	参数 1	参数 2	NRM	ADV	MAX
14	log	1.2	-	3,706	3,573	3,573
		2.5	-	3,770	3,636	3,636
		0.999	-	4,058	3,924	3,924
		0.3	-	3,858	3,724	3,724
15	log10	1.2	-	3,884	3,754	3,755
		2.5	-	3,948	3,818	3,818
		0.999	-	4,245	4,115	4,115
		0.3	-	4,029	3,889	3,899
16	modf	256.3	-	535	514	514
		0.032	-	469	450	450
		10000.2345	-	571	550	550
17	pow	2.3	4.2	9,338	8,626	8,634
		45.2	-5	9,492	8,780	8,795
		-4.56	-3	10,016	9,242	9,254
		-85.55	476	9,783	9,033	9,053
18	sqrt	2	-	885	859	859
		3	-	893	867	867
		0.1	-	889	863	863
19	ceil	0.3	-	446	390	390
		-0.6	-	246	215	215
20	fabs	5	-	21	21	21
		-5	-	21	21	21
21	floor	0.3	-	246	215	215
		-0.6	-	445	393	393
22	fmod	11.1	3.2	367	413	415
		500.55	0.4	581	627	629
		1.05E+06	9.54E-07	1,388	1,434	1,436

A.2 双精度浮点操作性能

A.2.1 双精度浮点操作性能（H8/300，H8/300H，H8S/2600）

					H8/300H		H8S/2000,H8S/2600	
编号	函数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
1	acos	0.4	-	88,070	44,762	47,294	20,277	21,016
		1.57075	-	4,646	3,786	4,284	1,814	1,994
		0.6	-	88,396	44,974	47,506	20,383	21,121
		-0.4	-	88,114	44,730	47,262	20,269	21,008
2	asin	0.4	-	86,796	43,666	46,062	19,681	20,369
		1.57075	-	3,542	2,834	3,196	1,290	1,419
		0.6	-	87,104	43,862	46,258	19,779	20,466
		-0.4	-	86,882	43,678	46,074	19,695	20,383
3	atan	0.11	-	29,172	18,570	19,780	7,784	8,156
		0.27	-	41,948	25,560	27,142	11,126	11,590
		0.547	-	41,590	25,512	27,094	11,099	11,563
		0.777	-	43,906	26,862	28,484	11,640	12,114
		0.975	-	41,862	25,714	27,250	11,218	11,669
		54.45	-	53,282	30,720	32,546	13,589	14,113
		154.233	-	53,626	31,070	32,896	13,764	14,288
		-54.45	-	53,368	30,730	32,556	13,602	14,126
		-0.975	-	41,948	25,724	27,260	11,231	11,682
		-0.777	-	43,992	26,872	28,494	11,653	12,127
4	atan2	0.3	0.7	51,604	29,210	31,122	12,919	13,457
		0.2	0.1	62,958	34,532	36,734	15,451	16,062
		0.1	0.9	39,414	22,708	24,248	9,824	10,270
5	cos	0.523333333	-	24,152	15,346	16,078	6,412	6,681
		1.046666667	-	27,734	17,718	18,730	7,411	7,723
		1.9625	-	26,848	17,014	17,944	7,091	7,382
		2.7475	-	25,478	16,430	17,244	6,922	7,212
		3.5325	-	24,488	15,598	16,330	6,538	6,807
		4.3175	-	27,984	17,876	18,888	7,489	7,801
		5.1025	-	26,982	17,064	17,994	7,115	7,406
		5.8875	-	25,488	16,446	17,260	6,930	7,220
		-0.52333333	-	24,238	15,350	16,082	6,422	6,691
		-1.04666667	-	27,796	17,728	18,740	7,424	7,736
		-1.9625	-	26,934	17,024	17,954	7,104	7,395
		-2.7475	-	25,564	16,440	17,254	6,935	7,225
		-3.5325	-	24,574	15,608	16,340	6,551	6,820
		-4.3175	-	28,070	17,886	18,898	7,502	7,814

					H8/300H		H8S/2000,H8S/2600	
编号	函数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
5	cos	-5.1025	-	27,068	17,074	18,004	7,128	7,419
		-5.8875	-	25,574	16,456	17,270	6,943	7,233
6	sin	0.523333333	-	26,522	16,820	17,750	7,000	7,289
		1.046666667	-	25,278	16,214	17,028	6,821	7,109
		1.9625	-	24,278	15,556	16,288	6,524	6,791
		2.7475	-	27,926	17,840	18,852	7,480	7,790
		3.5325	-	26,960	17,002	17,932	7,093	7,382
		4.3175	-	25,590	16,472	17,286	6,951	7,239
		5.1025	-	24,636	15,712	16,444	6,603	6,870
		5.8875	-	27,988	17,908	18,920	7,512	7,822
		-0.52333333	-	26,700	16,858	17,788	7,028	7,317
		-1.04666667	-	25,480	16,300	17,114	6,873	7,161
		-1.9625	-	24,456	15,594	16,326	6,552	6,819
		-2.7475	-	28,104	17,878	18,890	7,508	7,818
		-3.5325	-	27,138	17,040	17,970	7,121	7,410
		-4.3175	-	25,768	16,510	17,324	6,979	7,267
		-5.1025	-	24,814	15,750	16,482	6,631	6,898
		-5.8875	-	28,166	17,946	18,958	7,540	7,850
7	tan	0.3925	-	38,230	21,734	22,712	9,149	9,483
		1.1775	-	39,408	22,136	23,196	9,323	9,677
		1.9625	-	38,490	21,456	22,434	9,017	9,357
		2.7475	-	39,672	22,672	23,732	9,595	9,955
8	cosh	0.33	-	99,902	56,136	58,518	23,476	24,258
		0.78	-	101,046	57,590	59,980	24,901	24,693
		-0.33	-	99,920	56,140	58,522	23,478	24,260
		-0.78	-	101,064	57,594	59,984	23,903	24,695
9	sinh	0.33	-	28,064	17,778	18,546	7,269	7,535
		0.98	-	102,482	57,370	59,838	23,765	24,586
		-0.33	-	28,064	17,778	18,546	7,269	7,535
		-0.98	-	102,500	57,374	59,842	23,765	24,588
10	tanh	0.0033+00	-	109,818	63,362	66,024	26,975	27,838
11	exp	0.33	-	49,318	27,448	28,558	11,505	11,886
		0.98	-	50,186	27,746	28,860	11,503	11,889
		-0.33	-	49,428	27,556	28,666	11,559	11,940
		-0.98	-	50,288	27,782	28,896	11,521	11,907
12	frexp	0.3	-	290	246	274	134	147
		400	-	290	246	274	134	147
13	ldexp	0.3	30	1,792	1,436	1,576	659	721
		0.1	100	1,792	1,436	1,576	659	721

					H8/300H		H8S/2000,H8S/2600	
编号	参数	参数 1	参数 2	H8/300	NRM	ADV	NRM	ADV
14	log	1.2	-	43,214	25,574	26,854	10,931	11,341
		2.5	-	43,360	26,242	27,522	11,265	11,675
		0.999	-	44,000	25,250	26,530	10,769	11,179
		0.3	-	43,580	25,936	27,216	11,112	11,522
15	log10	1.2	-	45,900	27,160	28,580	11,654	12,117
		2.5	-	46,022	27,808	29,228	11,978	12,441
		0.999	-	46,718	26,858	28,278	11,503	11,966
		0.3	-	46,266	27,516	28,936	11,832	12,295
16	modf	256.3	-	4,458	4,044	4,484	1,795	1,945
		0.032	-	4,148	3,712	4,148	1,632	1,780
		10000.2345	-	4,434	3,898	4,338	1,722	1,872
17	pow	2.3	4.2	96,904	56,372	58,948	23,829	24,677
		45.2	-5	97,438	55,556	58,132	23,432	24,280
		-4.56	-3	101,770	59,090	62,090	24,943	25,891
		-85.55	476	100,174	59,292	62,206	25,111	26,039
18	sqrt	2	-	30,274	9,906	10,040	4,940	5,000
		3	-	30,374	9,922	10,056	4,948	5,008
		0.1	-	29,250	9,780	9,914	4,877	4,937
19	ceil	0.3	-	3,720	3,196	3,572	1,451	1,578
		-0.6	-	2,238	1,816	2,034	827	915
20	fabs	5	-	214	166	188	102	112
		-5	-	214	166	188	102	112
21	floor	0.3	-	2,238	1,816	2,034	827	915
		-0.6	-	3,720	3,190	3,566	1,448	1,575
22	fmod	11.1	3.2	2,716	2,070	2,258	1,047	1,127
		500.55	0.4	3,724	2,524	2,712	1,274	1,354
		1.05E+06	9.54E-07	7,624	3,904	4,092	1,964	2,044

A.2.2 双精度浮点操作性能（H8SX）

				H8SX		
编号	参数	参数 1	参数 2	NRM	ADV	MAX
1	acos	0.4	-	16,203	16,305	16,306
		1.57075	-	1,097	1,136	1,135
		0.6	-	16,220	16,323	16,324
		-0.4	-	16,185	16,289	16,291
2	asin	0.4	-	15,881	15,903	15,904
		1.57075	-	738	805	804
		0.6	-	14,895	15,916	15,918
		-0.4	-	14,888	15,910	15,911
3	atan	0.11	-	5,081	5,873	5,872
		0.27	-	8,163	9,066	9,065
		0.547	-	8,089	8,992	8,991
		0.777	-	8,512	9,425	9,424
		0.975	-	8,271	9,159	9,158
		54.45	-	10,528	11,515	11,514
		154.233	-	10,693	11,680	11,679
		-54.45	-	10,534	11,522	11,520
		-0.975	-	8,277	9,166	9,164
		-0.777	-	8,518	9,432	9,430
4	atan2	0.3	0.7	9,791	10,739	10,740
		0.2	0.1	12,161	13,208	13,209
		0.1	0.9	6,978	7,815	7,816
5	cos	0.523333333	-	4,653	4,928	4,927
		1.046666667	-	5,340	5,691	5,691
		1.9625	-	5,147	5,443	5,443
		2.7475	-	5,028	5,355	5,355
		3.5325	-	4,788	5,060	5,060
		4.3175	-	5,418	5,771	5,771
		5.1025	-	5,181	5,479	5,479
		5.8875	-	5,039	5,369	5,368
		-0.52333333	-	4,656	4,931	4,931
		-1.04666667	-	5,341	5,692	5,693
		-1.9625	-	5,151	5,447	5,448
		-2.7475	-	5,032	5,359	5,360
		-3.5325	-	4,792	5,064	5,065
		-4.3175	-	5,422	5,775	5,776
		-5.1025	-	5,185	5,483	5,484
		-5.8875	-	5,043	5,373	5,373

				H8SX		
编号	参数	参数 1	参数 2	NRM	ADV	MAX
6	sin	0.523333333	-	4,665	5,363	5,362
		1.046666667	-	4,601	5,260	5,260
		1.9625	-	4,405	5,040	5,040
		2.7475	-	5,033	5,754	5,754
		3.5325	-	4,764	5,461	5,461
		4.3175	-	4,725	5,384	5,384
		5.1025	-	4,473	5,108	5,108
		5.8875	-	5,061	5,783	5,782
		-0.52333333	-	4,674	5,372	5,372
		-1.04666667	-	4,622	5,281	5,282
		-1.9625	-	4,414	5,049	5,050
		-2.7475	-	5,042	5,763	5,764
		-3.5325	-	4,773	5,470	5,471
		-4.3175	-	4,734	5,393	5,394
		-5.1025	-	4,482	5,117	5,118
		-5.8875	-	5,072	5,792	5,792
7	tan	0.3925	-	7,096	7,418	7,418
		1.1775	-	7,284	7,631	7,631
		1.9625	-	7,050	7,317	7,371
		2.7475	-	7,451	7,797	7,797
8	cosh	0.33	-	16,425	16,725	16,727
		0.78	-	16,918	17,216	17,218
		-0.33	-	16,427	16,727	16,729
		-0.78	-	16,920	17,218	17,220
9	sinh	0.33	-	4,793	4,873	4,873
		0.98	-	16,705	17,003	17,006
		-0.33	-	4,793	4,873	4,873
		-0.98	-	16,707	17,005	17,008
10	tanh	0.0033+00	-	21,563	20,209	20,210
11	exp	0.33	-	8,073	8,249	8,248
		0.98	-	8,113	8,289	8,288
		-0.33	-	8,113	8,289	8,288
		-0.98	-	8,129	8,305	8,304
12	frexp	0.3	-	80	75	75
		400	-	80	75	75
13	ldexp	0.3	30	378	413	413
		0.1	100	378	413	413

				H8SX		
编号	参数	参数 1	参数 2	NRM	ADV	MAX
14	log	1.2	-	8,345	7,889	7,889
		2.5	-	8,640	8,181	8,181
		0.999	-	8,258	7,799	7,799
		0.3	-	8,538	8,079	8,079
15	log10	1.2	-	8,114	8,313	8,316
		2.5	-	8,400	8,599	8,601
		0.999	-	8,035	8,234	8,236
		0.3	-	8,304	8,503	8,505
16	modf	256.3	-	1,226	1,194	1,194
		0.032	-	1,065	1,035	1,035
		10000.2345	-	1,150	1,118	1,118
17	pow	2.3	4.2	17,485	17,294	17,295
		45.2	-5	17,060	16,868	16,870
		-4.56	-3	17,965	17,820	17,820
		-85.55	476	18,237	18,076	18,078
18	sqrt	2	-	3,882	3,912	3,912
		3	-	3,888	3,918	3,918
		0.1	-	3,837	3,867	3,867
19	ceil	0.3	-	892	908	908
		-0.6	-	482	509	509
20	fabs	5	-	51	55	55
		-5	-	51	55	55
21	floor	0.3	-	482	498	498
		-0.6	-	893	894	894
22	fmod	11.1	3.2	688	750	749
		500.55	0.4	921	983	982
		1.05E+06	9.54E-07	1,712	1,774	1,773

H8S, H8/300 系列 C/C++编译程序应用笔记

附加功能

附录 B 附加功能

B.1 2.0 版本与 3.0 版本的附加功能

B.1.1 嵌入式扩展函数的附加

1. entry 函数

在 H8S，H8/300 系列 C/C++ 编译程序 3.0（新版）或以上版本中，#pragma entry 可指定项目 (entry) 函数。在打开电源和复位时，项目函数首先被执行。

项目函数允许在不使用堆栈指针 (SP) 初始值，或嵌入 C/C++ 程序中的汇编语言的情况下，创建 C/C++ 程序。

2. 段地址运算符

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，添加了表示段的起始和终止地址的运算符(_ _sectop 和 _ _secend)。这允许在使用了段的 switch 函数时，数据初始化程序库函数的使用 (_INITSCT)。

3. packed 结构

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，包选项和 #pragma pack 可指定结构成员的边界对齐。

B.1.2 附加和增进的函数

1. C++ 语言函数

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，编译程序可编译 C 和 C++ 语言程序。编译程序通过选项 lang 或文件扩展名来分辨 C 和 C++ 语言程序。

2. 程序库

H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本支持标准程序库中的数学函数（复式或浮点类型）。嵌入式类程序库（ios、istream、ostream、iostream、string、complex，和 new）也被支持。

3. 寄存器参数数量的指定

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，regparam 选项可被用以选取参数寄存器的数量。

4. speed 选项的扩展

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，为选项速度添加了子命令 speed=expression。当指定了 speed=expression 时，代替调用运行时例程，对大部分操作执行了内联扩展。

5. 支持 long 类型位字段

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，长型数据位字段被添加为受支持的数据类型。

6. 有限值的扩展

相较于旧版本（2.0 版本），下列项目的限制值在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中被扩展：

- 符号大小（3.0 或以上版本：250 字符，2.0 版本：31 字符）
- 复合语句的嵌套（3.0 或以上版本：256 级，2.0 版本：32 级）
- 重复语句的嵌套（while、do 和 for 语句）（3.0 或以上版本：256 级，2.0 版本：32 级）
- 选择语句组合的嵌套（if 和 switch 语句）（3.0 或以上版本：256 级，2.0 版本：32 级）
- 切换 (switch) 语句的嵌套（3.0 或以上版本：128 级，2.0 版本：16 级）
- for 语句的嵌套（3.0 或以上版本：128 级，2.0 版本：16 级）
- 行字符数（3.0 或以上版本：8192 字符，2.0 版本：4096 字符）
- 由 malloc 分配的存储器大小（在高级模式中：3.0 或以上版本：size_t，2.0 版本：INT_MAX）

7. 优化列表的输出

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，符号参考计数和优化的信息列表输出函数在执行模块间优化器时被添加。

8. 命令行的输出

由命令行指定的字符串在一个列表中被输出至文件。

9. 加强选项消息

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，任何由消息选项指定的信息消息级的消息，可被排除从而不被输出到文件。

10. 字符代码转换

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，Latin1 选项允许 Latin1 代码被使用于源代码内。

B.1.3 语言规格的修改

1. 为 *((int*)p)++ 输出警告消息

在 H8S，H8/300 系列 C/C++ 编译程序 2.0 版本中，一项错误消息为 *((int*)p)++ 输出。在 3.0 或以上版本中，一项警告消息被输出。

2. 检查原型

在 H8S，H8/300 系列 C/C++ 编译程序 2.0 版本中，若不具参数类型规格的原型声明和具参数类型规格的原型声明被同时指定，将输出错误消息。在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，正确操作在类似情况下被允许。

2.0 版本的实例：

```
void f();

void f(int);              -> 错误 2118 被输出
```

3.0 版本的实例：

```
void f();
void f(int);              <- 正确编译
```

3. 在结构之首不具名称的位字段的描述

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，不具名称的位字段可在结构之首被编写。

2.0 版本的实例：

```
struct S {
   int   :1;
   int a:1;                          -> 错误 2141 被输出
};
```

3.0 版本的实例：

```
            struct S {
   int   :1;
   int a:1;                          -> 正确编译
};
```

4. 禁止结构初始值发生错误

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，结构的分配和声明可被同步完成。

2.0 版本的实例：

```
struct S {
   int a,b;
}s1;
void test()
{
   struct S s2 = s1;                 <- 错误 2130 被输出
}
```

3.0 版本的实例：

```
struct S {
   int a,b;
}s1;
union U {
   int a,b;
}u1;
void test()
{
   struct S s2 = s1;                <- 正确编译
}
```

5. 为 static 函数的未定义的符号错误更改条件

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，仅有一项声明而没有定义的静态函数，将不输出未被参考的符号的错误消息。

2.0 版本的实例：

```
static void func();              <- 由于没有定义，错误 2143 将被无条件输出
there is no definition
void test()
{
}
```

3.0 版本的实例：

```
static void func();              <- 由于没有参考，没有错误被输出
void test()
{
}
```

6. 允许在 C 程序中使用 // 注解

在 H8S，H8/300 系列 C/C++ 编译程序 3.0 或以上版本中，// 注解可被用于 C 程序中。 因此，程序在 3.0 版本中的意义可能有别于 2.0 版本。

2.0 版本的实例：

```
int b = a //* Comment */4;          <- 程序的意义是“int b = a/4；-a；”
  -a;
```

3.0 版本的实例：：

```
int b = a //* Comment */4;          <- 程序的意义是“int b = a -a；”
  -a;
```

B.2 3.0 版本与 4.0 版本的附加功能

B.2.1 一般附加和增进

1. 放宽值的限制

源程序和命令行的限制被大大的放宽：

- 文件名称的长度：251 字节 -> 无限
- 符号的长度：251 字节 -> 无限
- 符号的数量：65,535 -> 无限
- 源程序行数：C/C++：32,767，ASM：65,535 -> 无限
- C 程序行的长度：8,192 字符 -> 16,384 字符
- C 程序字符串文字的长度：512 字符 -> 16,384 字符
- 子命令文件行的长度：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 -> 无限

2. 用于目录和文件名称的破折号

可为目录和文件名称指定破折号（－）。

3. 版权显示的规格

指定商标/无商标 (logo/nologo) 选项，可指定是否显示版权输出。

4. 错误消息的前缀

为了支持 Hitachi 嵌入式工作区 (Hitachi Embedded Workshop) 内的错误帮助函数，一个前缀被添加到编译程序和优化连接编辑程序的错误消息。

B.2.2 附加和增进的编译程序函数

1. 关键字的使用

可以通过使用关键字（_ _interrupt、_ _indirect、_ _entry、_ _ abs8、_ _abs16、_ _regsave、_ _noregsave、_ _inline, 或 _ _register），在函数和变量的声明和定义中指定属性。

2. 向量表的创建

当通过 #pragma interrupt、indirect、entry、_ _interrupt、_ _indirect，或 _ _entry 指定 vect 时，函数的向量表可被自动创建。

3. 支持 _ _evenaccess

以 _ _evenaccess 指定的变量，其在偶数字节边界上的偶数存储器存取被保证。

4. 扩展寄存器参数规格

_ _regparam2 和 **_ _regparam3** 可用以指定函数中寄存器参数的数量。

5. 指定函数单元中的选项

可使用 #pragma 选项指定函数单元中的选项。

6. 分配彼此靠近的数据

使用 **_ _near8** 或 **_ _near16** 来优化数组或结构的地址计算代码。

然而，指针大小不更改。

7. 分配彼此靠近的堆栈

使用 **stack** 来优化堆栈区域的堆栈地址计算代码。

8. 附加的固有函数

添加了下列固有函数。

- 无符号溢出操作

9. 支持 double=float

在新版本中，可指定 **double=float**，以便使被声明为双精度类型的数据和浮点常数均被当作浮点类型来处理。

10. 加强 noregsave 函数

当以 #pragma noregsave 或 _ _noregsave 声明的函数被调用时，寄存器的内容被调用者所保证。

11. 使用环境变量来指定多套包含目录

可使用环境变量 (CH38) 来指定多套包含目录。

12. 将结构参数或返回值分配到寄存器

选项 structreg 被用以将小型结构参数或返回值分配到寄存器。

13. 将 4 字节参数或返回值分配到寄存器 (cpu=300)

选项 longreg 被用以将 4 字节参数或返回值分配到寄存器。

14. 在循环外部移动非易失性变量的条件

迭代条件中的非易失性外部变量，禁止移出循环的外部变量优化，即使迭代条件中不含函数调用或赋值表达式。

15. 支持 speed=loop=1|2

选项 **speed=loop=1|2** 控制循环扩展的优化。

16. 修改边界对齐的数据分配

可为每个边界对齐再分配数据，以使边界对齐所生成的间隙最小化。

17. 附加的隐含声明

_ _HITACHI_ _ 和 _ _HITACHI_VERSION_ _ 被 #define 隐含声明。

18. static 标签名称

使用 #pragma asm 和 #pragma endasm 把标签名称指定为对静态文件标签的参考，及 #pragma inline_asm 被更改为 _ _$ (name)。 然而，在连接列表中，名称被显示为 _ (name)。

19. 语言规格的扩展名与更改

- 在初始化联合时禁止错误。

实例：

```
union{
char c[4];
}uu={ {'a','b','c'} };
```

- enum 可被应用到位字段。

实例：

```
struct{
enum E1{a,b,c}m1:2;
enum E1 m2:2;
};
```

- 在最后的枚举被禁止之后编写逗号“，”时，输出错误消息。

实例：

```
enum E1{a,b,c,}m1;
```

- 联合可在单一语句中被分配和声明。

实例：

```
union U{

int a,b;

}u1;

void test(){

union U u2 = u1;

}
```

- 符号地址表达式转型的错误检查级别被放松了。

实例：

```
int x;

short addr1=(short)&x;
```

- 在 C 程序中编写函数与变量声明，及 #pragma 声明的顺序限制被放宽了。

实例：

```
void f(void);

#pragma interrupt f

void f(void){}  //在函数声明之后的 #pragma 声明是有效的。（在版本 3 中，将会发生错误。）
```

- 在 C++ 程序中编写函数与变量声明，及 #pragma 声明的顺序限制被修改了。

实例：

```
void f(void){}

#pragma interrupt f

void f(void);  //当 #pragma 声明跟随在函数声明之后时，将发生错误。
```

- 异常处理和模板函数也根据 C++ 语言规格被支持。

B.2.3 汇编程序的附加与增进函数

1. BEQU 的外部定义与参考

.BEQU 符号可通过 .BIMPORT 和 .BEXPORT 被外部定义和参考。

B.2.4 优化连接编辑程序的附加与增进函数

1. 支持通配符

通配符可通过输入文件的段名称指定，或以开始 (start) 选项为文件名称指定。

2. 搜索路径

环境变量 (HLNK_DIR) 可被用以指定输入文件或程序库文件的几个搜索路径。

3. 细分装入模块的输出

绝对装入模块文件的输出可被再细分。

4. 更改错误级别

对于信息、警告，和错误级别的消息，其错误级别或输出可被个别更改。

5. 支持二进制与 HEX

二进制文件可被输入和输出。

可选取 Intel® HEX 类型输出。

6. 输出堆栈数目信息

堆栈选项可为堆栈分析工具输出信息文件。

7. 通过 optimize=variable 增进的优化

以 16 位绝对地址空间分配的变量，可通过应用优化以 8 位地址空间分配。

8. 通过 optimize=register 增进的优化

当未指定 **optimize=speed** 的选项时，文件在优化函数间寄存器内容的保存和恢复后被压缩，并将多个寄存器内容的保存和恢复替换为函数调用。

9. 汇编程序的增进优化

包含 .org、.align，或 .data 指令的段可被优化。

10. 删除调试信息

可使用 strip 选项，以从装入模块文件或程序库文件删除调试信息。

B.3 4.0 版本升级到 6.0 版本中的附加与增进功能

（注意： 5.0 版本是不存在的缺失数字。）

B.3.1 附加和增进的编译程序函数

a. 支持新 CPU

支持创建 CPU 类型为 H8SX 的目标文件。

b. 支持 2 字节指针（仅限于 H8SX）

ptr16 关键字或选项 **ptr16** 可用以指定 2 字节指针的使用。

它们在 H8SX 高级模式或 H8SX 最大模式中有效。

c. 指定位字段顺序

#pragma bit_order 或 **bit_order** 选项可用以指定在字段中存储位字段成员的顺序。

d. 扩展存储器间接寻址方式中的函数调用（仅限于 H8SX）

_ _indirect_ex 关键字或 **indirect=extended** 选项可用以声明要在扩展存储器间接寻址方式中调用的函数。同时，#pragma indirect 段可以修改的段名称不仅有 $INDIRECT，存储器间接寻址方式的函数地址区域 (@@aa:8)，还有 $EXINDIRECT，扩展存储器间接寻址方式的函数地址区域 (@@aa:7)。

e. 汇编功能（仅限于 H8SX）

_ _asm 关键字可用以允许汇编语言在 C/C++ 源程序中使用。

f. 禁止 #line 输出

noline 选项可用以在预处理程序扩展上禁止 **#line** 的输出。

g. 为函数 memcpy 和 strcpy 指定内联扩展（仅限于 H8SX）

library 选项可用以指定两个程序库函数，**memcpy** 和 **strcpy** 的内联扩展。

h. 更改错误级别

change_message 选项可用以个别更改信息级和警告级错误消息的错误级别。

i. 指定 8 位绝对区域地址（仅限于 H8SX）

选项 **sbr** 可用以指定要查找的 8 位绝对区域的地址。

j. 加强优化功能（仅限于 H8SX）

优化细节可通过下列附加选项进一步指定：opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate，及 volatile_loop。

k. 附加的固有函数

添加了下列固有函数。

- H8SX 的 64 位乘法（mulsu 和 muluu）
- H8SX 的块转移指令（movmdb、movmdw、movmdl，及 movsd）
- 块转移指令（eepmovb、eepmovw、eepmovi）
- MOVFPE 指令修正的固有函数 (_movfpe)

l. 支持通配符

输入文件可以通配符指定。

m. 编译程序限制的更改

switch 语句的数量限制从 256 更改至 2048。

n. 信息消息显示规格的更改

在 4.0 版本中，只有所有 **message** 和 **nomessage** 选项的最后规格在命令行内有效。 在 6.0 版本中，所有由每个 **nomessage** 选项在命令行中指定的数量联合被制止显示消息。

o. 枚举示例的类型

若指定了 **byteenum** 选项，及若枚举中的所有数目在 0 到 255 的范围内，编译程序将数据处理为 **unsigned char**。

p. 内联扩展

在 H8SX 中，**speed=inline=<numeric value>** 选项中的 <numeric value> 表示内联扩展允许程序大小增加的百分比。在其他 CPU 中，<numeric value> 表示允许执行内联扩展的函数中的最大节点数。

q. 以 1 字节对齐的数据段，及以 4 字节对齐的数据段（仅限于 H8SX）

指定 align=4 选项，将把奇数大小的数据放置到以 1 字节对齐的数据段，及把 4 倍数大小的数据放置到以 4 字节对齐的数据段。

r. 段名称

通过 section 选项，把 P、C、B 或 D 的段名称更改为 S，将导致警告错误。 S 是堆栈区域的保留名称。

s. 附加的隐含声明

_ _ H8SXN_ _、_ _ H8SXM_ _、_ _ H8SXA_ _、_ _ H8SXX_ _、_ _ HAS_MULTIPLIER_ _、_ _HAS_DIVIDER_ _、_ _INTRINSIC_LIB_ _、_ _ DATA_ADDRESS_SIZE_ _、_ _ H8_ _、_ _RENESAS_VERSION_ _，及 _ _ RENESAS_ _ 由编译程序使用 #define 指令隐含声明。

t. 可重入的程序库

若在程序库生成程序中指定 **reent** 选项，将创建可重入的程序库。

u. 支持 Little-endian 空间（仅限于 H8SX）

Little-endian 空间根据 H8SX 的芯片而支持。Little-endian 空间内的 2 或 4 字节已知数，将以其自身数据大小被写读。 为完成此操作，**__evenaccess** 关键字的功能被加强了。

B.3.2 编译程序 6.0 版本优化功能的注意事项

注意下面有关应用在由 6.0 版本优化所创建的 H8SX 目标程序中的优化。 在其他情况下，优化与 4.0 或以下版本相似。

采用最新的编译程序优化技术，允许 6.0 版本中的优化处理，以为指针或外部变量分析别名，及分析数据生命期，包括目前还不可能的控制流程（4.0 或以下版本）。这在语言规格的限制内，提供比 4.0 版本范围更广的优化。

然而，先前因未充分优化而运行的程序可能无法运行，因为它已成为优化的目标。

下面显示目前未被优化，但将在 6.0 版本中成为优化目标的程序实例。

a. 无需 volatile 声明的外部变量或指针变量存取

volatile 声明保证符合易失性标准的变量在任何时候使用时将被存取，因为变量可能在程序序列外部被更新。例如，数据值被中断处理或硬件处理所更改。

编译程序假定不具 **volatile** 声明的变量仅被程序序列，或函数调用的连续处理所更改。

在 4.0 或以下版本中，不具 volatile 声明的外部变量如下例所示般获得优化：

实例：

```
int a;

f() {

    int *ptr=&a;

    *ptr=1; // <-只有这项赋值表达式被删除。

    *ptr=2;

}
```

在 6.0 版本中，优化在下列情况下更进一步执行。

要禁止优化，则以 **volatile** 声明相关变量。

实例 1：

```
int a;
f() {
    int *ptr=&a;
    *ptr &= ~( (0x0080) ); //<- (1)
    while( !( *ptr & (0x0080) ) ) //<- (2)
    {
        :
    }
}
```

在此例中，**while** 语句（2）因优化而变成无穷循环。

. 由于指针的别名分析，（1）和（2）中的 *ptr 被处理为相同值。

. 表达式（1）被传播至表达式（2）。 相应的，表达式（2）被转换如下：

```
while( !( (*ptr & ~( (0x0080) )) & (0x0080) ) ) //<- (2)
-> while(!(*ptr & 0))
-> while(!(0))
-> while(1)
```

因此，有问题的表达式被判定为真实的，判定语句被删除，而上述 **while** 语句变成无穷循环。

实例 2：

```
int a,b;
f() {
    a=1; //<- (1)
    if(a); //<- (2)
    {
        b=1; //<- (3)
    }
}
```

在此例中，**if** 语句（2）被删除，（3）将因为优化的结果被一直执行。

- 由于外部变量的别名分析，（1）和（2）中的 a 被处理为相同值。
- 常数值（1）被传播至表达式 (2)。 相应的，表达式（2）被转换如下：

```
+-> if(1)
```

因此，有问题的表达式被判定为真实的，条件语句被删除，而上述表达式（3）将被一直执行。

实例 3：

```
int a,b,c;

f() {

    a=1; //<- (1)

    if(c); //<- (2)

    {

        b=1; //<- (3)

    }

    a=2; //<- (4)

}
```

在此例中，表达式（1）因为优化的结果而被删除。

- 获取包含 if 语句表达式的条件的控制流程。
- 由于进行控制流程分析和外部变量的别名分析，证实 a 设定在（1）的值是不被使用的。因此，上述表达式（1）是不被参考的冗余表达式，所以被删除。

实例 4：

```
int a;

int b[10];

f() {

    int i; //<- (1)

    for(i=0; i<10; i++) //<- (2)

    {

        b[i]=a; //<- (3)

    }

}
```

在此例中，表达式（3）内的 a 在循环之前被参考一次，并由于优化的结果，始终在循环中被处理为常数值。

- 获取包含 **for** 循环控制表达式的控制流程。
- 由于进行控制流程分析及外部变量的别名分析，（3）内的 a 在循环中被处理为常数值。
- 作为 a 的参考表达式的（3），在 **for** 循环（2）外部被移动如下：

```
temp=a;
for(i=0; i<10; i++) //<- (2)
{
    b[i]=temp; //<- (3)
}
```

因此，表达式（3）内的变量 a 在循环中不更改。

实例 5：

```
int a;
f() {
    a=0; //<- (1)
    while(1); //<- (2)
}
```

在此例中，语句（1）被假定为不需要的，并由于优化的结果被删除。

- 由于（2）是一项无穷循环，此函数被判定为没有出口。
- 因为 a 未在无穷循环中被参考，规格（1）被假定为不需要的编码而被删除。

b. volatile_loop 选项

若循环控制变量为非易失性外部变量，且条件表达式是简单的，**volatile_loop** 选项将把循环控制变量当作是符合**易失性**标准的，以防止创建无穷循环。然而，若循环控制变量不是循环不变的，它将不能被当作符合**易失性**标准来处理。

在 6.0 版本中，以 **volatile** 声明相关变量。

一个样品程序在下面提供。

实例：

```
struct{
    unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
    while (ST.a) { //<- (1)
        if (a) { //<- (2)
            f(); //<- (3)
        }
    }
}
```

在此例中，因为 ST.a 可能在 f() 中被更新，ST.a 在循环中不被假定为循环不变值。 因此，即使使用 **volatile_loop** 选项指定，ST 也不能被当作 **volatile** 来处理。

- 若满足了（2）中的条件，（3）将执行，且 ST.a 值可能被更新。
  相应的，在函数调用之后，ST.a 将被重新装入。
- 若（2）中的条件未被满足，ST.a 值将不被更新，所以在（1）的上一个条件中使用的 ST.a 值可被直接使用。

B.3.3 4.0 版本与 6.0 版本之间的兼容性

要连接由 4.0 版本与 6.0 版本创建的目标程序，必须先满足下列条件。

(1) C 源程序

下列影响函数界面的选项必须被平等指定。

- regparam
- longreg/nolongreg
- double=float
- structreg/nostructreg
- stack
- byteenum
- pack/unpack

(2) 汇编程序

汇编程序必须符合有关描述在“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual)”9.3.2 节，函数调用界面 (Function Calling Interface) 中的函数调用规则。

注意： 1. 有关未在手册中提及的信息，不保证其与升级版本的兼容性。若其中一方或双方的目标程序包含依赖编译程序输出编码的汇编编码，如保存和恢复寄存器内容的顺序，那么 4.0 版本所创建的目标程序，将不能连接到 6.0 版本所创建的目标程序。

2. 要获取有关和 OS、中间软件等进行连接的详细资料，请联系您的销售代理。

B.4. 6.0 版本升级到 6.1 版本中的附加与增进功能

B.4.1 附加和增进的编译程序函数

a. 支持 AE5

AE5 被支持。

b. 加强对 ANSI 标准的遵循

strict_ansi 使浮点运算的联合规则遵循 ANSI 标准。

c. 输出目标代码与 4.0 版本产生的目标代码的兼容性

使用 H8S CPU，**legacy=v4** 支持输出与旧版本编译程序（4.0 版本）所产生的目标代码兼容的目标代码。

d. 以 **legacy=v4** 指定的 **cpuexpand=v6** 扩展指定

当以 **legacy=v4** 指定 **cpuexpand=v6** 时，输出目标代码与 6.00 版本和 **cpuexpand** 选项所产生的目标代码兼容。

e. 寄存器存储类变量的优先分配

enable_register 优先将具有指定寄存器存储类的变量分配给寄存器。

f. 分割优化范围

可通过指定 **scope/noscope** 来选择是否分割函数内的优化范围。

g. 文件间内联扩展

file_inline 用来为跨文件扩展的函数指定内联扩展，而 **file_inline_path** 则用来指定进行内联扩展的文件路径名称。

h. 附加的固有函数

set_vbr 固有函数被用来设定 VBR。

i. #pragma address

#pragma address 可被用来将变量分配到特定绝对地址。

j. 支持 .stack 指令

当指定了 **code=asmcode** 时，编译程序会在汇编源程序内输出一个 .stack 指令。

k. 附加的环境变量

环境变量 **CH38SBR** 可被用来为 SBR 设定初始值

l. 附加的隐含声明

附加了 _ _AE5_ _ 和 _ _ABS16_ _ 的隐含声明。

B.4.2 编译程序 6.01 版本优化功能的注意事项

下面请留意有关在使用 6.01 版本优化所创建的 H8SX 与 H8S（不含 **legacy=v4** 选项）目标程序中的优化应用。在其他情况下，优化与 4.0 或以下版本相似。

采用 H8SX 和 H8S 的最新编译程序优化技术，允许 6.01 版本中的优化处理为指针或外部变量分析别名，及分析数据生命期，包括目前还不可能的控制流程（4.0 或以下版本）。这在语言指定的限制内，提供了比 4.0 版本范围更广的优化。

因此，当 H8S 的开发使用 H8C 6.0 版本来完成，同时工程被更新到 6.01 版本时，所生成的代码将因上述新的优化技术而与旧工程有很大的不同。

但是，因为使用了新的优化方法，一个原先可以运行的程序经过新编译器的编译后可能无法运行。

下面显示目前未被优化，但将在 6.01 版本中成为优化目标的程序实例。

a. 无需 volatile 声明的外部变量或指针变量存取

volatile 声明保证符合易失性标准的变量在任何时候使用时将被存取，因为变量可能在程序序列外部被更新。 例如，数据值被中断处理或硬件处理所更改。

编译程序假定不具 **volatile** 声明的变量仅被程序序列，或函数调用的连续处理所更改。

在 4.0 或以下版本中，不具 **volatile** 声明的外部变量如下例所示般获得优化：

实例：

```
int a;
f() {
        int *ptr=&a;
        *ptr=1; //<-只有这项赋值表达式被删除。
        *ptr=2;
}
```

在 6.01 版本中，优化在下列情况下更进一步执行。

要禁止优化，则以 **volatile** 声明相关变量。

实例 1：

```
int a;
f() {
        int *ptr=&a;
        *ptr &= ~( (0x0080) ); //<- (1)
        while( !( *ptr & (0x0080) ) ) //<- (2)
        }
                :
        }
}
```

在此例中，**while** 语句（2）因优化而变成无穷循环。

- 由于指针的别名分析，（1）和（2）中的 *ptr 被处理为相同值。
- 表达式（1）被传播至表达式（2）。相应的，表达式（2）被转换如下：
  while(!((*ptr & ~((0x0080))) & (0x0080))) //<- (2)
  -> while(!(*ptr & 0))
  -> while(!(0))
  -> while(1)

因此，不确定的表达式被判定为真，判定语句被删除，而上述 **while** 语句变成无穷循环。

实例 2：

```
int a,b;
f() {
        a=1; //<- (1)
        if(a) //<- (2)
        {
              b=1; //<- (3)
        }
}
```

在此例中，**if** 语句（2）被删除，（3）将因为优化而总是被执行。

- 由于外部变量的别名分析，（1）和（2）中的 a 被处理为相同值。
- 常数值（1）被传播至表达式 (2)。 相应的，表达式（2）被转换如下：
  -> if(1)

因此，不确定的表达式被判定为真，条件语句被删除，而上述表达式（3）将总是被执行。

实例 3：

```
int a,b,c;
f() {
    a=1; //<- (1)
    if(c) //<- (2)
    {
        b=1; //<- (3)
    }
    a=2; //<- (4)
}
```

在此例中，表达式（1）因为优化的结果而被删除。

- 获取包含 if 语句表达式的条件的控制流程。
- 由于进行控制流程分析和外部变量的别名分析，证实 a 设定在（1）的值是不被使用的。因此，上述表达式（1）是不被参考的冗余表达式，所以被删除。

实例 4：

```
int a;
int b[10];
f() {
    int i; //<- (1)
    for(i=0; i<10; i++) //<- (2)
    {
        b[i]=a; //<- (3)
    }
}
```

在此例中，表达式（3）内的 a 在循环之前被参考一次，并由于优化的结果，始终在循环中被处理为常数值。

- 获取包含 **for** 循环控制表达式的控制流程。
- 由于进行控制流程分析及外部变量的别名分析，（3）内的 a 在循环中被处理为常数值。
- 作为 a 的参考表达式的（3），将被移动到 **for** 循环（2）外部，如下所示：

```
temp=a;
for(i=0; i<10; i++) //<- (2)
{
    b[i]=temp; //<- (3)
}
```

因此，表达式（3）内的变量 a 在循环中不更改。

实例 5：

```
int a;
f() {
    a=0; //<- (1)
```

```
    while(1); //<- (2)
}
```

在此例中，语句（1）被假定为不需要的，并由于优化而被删除。

- 由于（2）是一项无穷循环，此函数被判定为没有出口。
- 因为 a 未在无穷循环中被参考，语句（1）被假定为不需要的编码而被删除。

b. volatile_loop 选项

若循环控制变量为非易失性外部变量，且条件表达式是简单的，**volatile_loop** 选项将把循环控制变量当作是符合**易失性**标准的，以防止创建无穷循环。然而，若循环控制变量不是循环不变的，它将不能被当作符合**易失性**标准来处理。

在 6.01 版本中，以 **volatile** 声明相关变量。

一个样品程序在下面提供。

实例：

```
struct{
    unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
    while (ST.a) { //<- (1)
        if (a) { //<- (2)
            f(); //<- (3)
        }
    }
}
```

在此例中，因为 ST.a 可能在 f() 中被更新，ST.a 在循环中不被假定为循环不变值。 因此，即使使用 **volatile_loop** 选项指定，ST 也不能被当作 **volatile** 来处理。

- 若满足了（2）中的条件，（3）将执行，且 ST.a 值可能被更新。 相应的，在函数调用之后，ST.a 将被重新装入。
- 若（2）中的条件未被满足，ST.a 值将不被更新，所以在（1）的上一个条件中使用的 ST.a 值可被直接使用。

B.4.3 4.0 版本与 6.01 版本 的兼容性

要连接由 4.0 版本与 6.01 版本创建的目标程序，必须先满足下列条件。

(1) C 源程序

下列影响函数界面的选项必须被平等指定。

- regparam
- longreg/nolongreg
- double=float

- structreg/nostructreg
- stack
- byteenum
- pack/unpack

(2) 汇编程序

汇编程序必须符合有关描述在“H8S，H8/300 系列 C/C++ 编译程序、汇编程序、优化连接编辑程序用户手册 (H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual)”9.3.2 节，函数调用界面 (Function Calling Interface) 中的函数调用规则。

注意： 1. 有关未在手册中提及的信息，不保证其与升级版本的兼容性。若其中一方或双方的目标程序包含依赖编译程序输出编码的汇编编码，如保存和恢复寄存器内容的顺序，那么 4.0 版本所创建的目标程序，将不能连接到 6.01 版本所创建的目标程序。

2. 要获取有关和 OS、中间软件等进行连接的详细资料，请联系您的销售代理。

H8S, H8/300 系列 C/C++编译程序应用笔记

版本升级的注意事项

附录 C 版本升级的注意事项

本节描述当为旧版本（H8S，H8/300 系列 C/C++ 编译程序封装 4.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

当使用升级版本开发了程序时，程序的操作可能更改。

当程序被创建时，请注意下列事项，并充分的测试您的程序。

(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 增加顺序未定义。                    */
f(i++,i++) ;            /* 参数值根据增加顺序而更改。              */
                        /* 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。 操作将根据版本而更改。

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。 操作将根据版本而更改。

文件 1：

```
int f(double d)
{
      :
}
```

文件 2：

```
int g(void)
{
      f(1);
}
```

调用函数的参数是 int 类型，但调用目标函数的参数是复式类型。因此，值无法被正确参考。

在此提供的信息不包含所有可能发生的情形。请谨慎使用此编译程序，并记取版本差异，及充分测试您的程序。

C.2 与旧版本的兼容性

下列备注涵盖了要将使用编译程序（3.x 或 以下版本）生成的文件，连接到旧版本生成的文件，或由汇编程序（2.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的状况。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

标准目标文件的格式从 SYSROF 更改至 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（3.x 或以下版本）或汇编程序（2.x 或以下版本） 输出的目标文件 (SYSROF) 要被输入到优化连接编辑程序时，使用文件转换器来将它转换到 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创建的 C++ 目标文件不能被连接。请确保重新编译这类文件。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（3.x 或以下版本）或汇编程序（2.x 或以下版本） 输出的目标文件不是模块间优化的目标。请确保对这类文件进行重新编译及重新汇编，以便它们成为模块间优化的目标。

H8S, H8/300 系列 C/C++编译程序应用笔记

限制列表

附录 D 限制列表

H8S 和 H8/300 C/C++ 编译程序 6.01 版本具有下列限制：

编号	类别	项目	限制
1	编译程序启动	可在单一操作中编译的源程序数量	无限制 *1
2		可在 Define 选项中指定的宏名称总数	无限制
3		文件名称长度	无限制（视 OS 而定）
4	源程序行数	行的长度	32768 字符 (H8SX/H8S) 16384 字符 (300H,300)
5		每个文件的源程序行数	无限制
6		可编译的源程序行数	无限制
7	预处理程序	由 #include 语句创建的文件嵌套级别深度	无限制
8		由 #define 语句定义的宏名称总数	无限制
9		可在宏定义及宏调用中指定的参数总数	无限制
10		宏名称的替换数量	无限制
11		#if、#ifdef、#indef、#else，及 #elif 语句的嵌套级别深度	无限制
12		可在 #if 或 #elif 语句中指定的运算符及操作数的总数	无限制
13	声明	函数定义的数量	无限制
14		外部连接的标识符（外部名称）数量	无限制
15		可在函数中使用的标识符（内部名称）数量	无限制
16		指针类型中的声明总数，符合基本类的数组类型和函数类型	16 个声明
17		数组次元的数量	6 次元
18		数组或结构体大小 *2	
		H8SX 普通模式， H8S/2600 普通模式、 H8S/2000 普通模式、 H8S/300H 普通模式、 H8/300	65535 字节
		H8SX 中级模式， H8SX 高级模式（包含 ptr16 选项）， H8SX 最高模式（包含 ptr16 选项）	32767 字节
		H8S/300H 高级模式	16777215 字节
		H8SX 高级模式（不含 ptr16 选项）， H8SX 最高模式（不含 ptr16 选项） H8S/2600 高级模式、 H8S/2000 高级模式	2147483647 字节 4294967295 （若指定了 legacy=v4） 字节

编号	类别	项目	限制
19	语句	复合语句嵌套级别的深度	无限制
20		合并反复语句（while、do，及 for 语句）与选择语句（if 和 switch 语句）时的嵌套级别深度	4096 级 (H8SX/H8S) 256 级 (300H/300)
21		可在函数中指定的转至 (goto) 标签数	2147483646 个标签 (H8SX/H8S) 511 个标签 (300H/300)
22		switch 语句的数量	2048 个语句
23		switch 语句的嵌套级别深度	2048 级 (H8SX/H8S) 128 级 (300H/300)
24		case 标签数	2147483646 个标签 (H8SX/H8S) 511 个标签 (300H/300)
25		for 语句的嵌套级别深度	2048 级 (H8SX/H8S) 128 级 (300H/300)
26	表达式	字符串长度	32766 字符
27		可在函数定义或函数调用中指定的参数数量	2147483646 个参数 (H8SX/H8S) 63 个参数 (300H/300) *3
28		可在表达式中指定的运算符与操作数的总数	约 500
29	标准包含	可使用打开函数一次打开的文件数量	变量 *4

注意： 1. 对于 PC，因为命令行的限制，最多可输入 127 字符。
2. 在高级模式中，若指定了位宽度的地址空间，与指定位宽度对应的地址空间大小将取得优先级。
3. 在非静态函数成员的情况下，最大数目为 62。
4. 可指定使用 *open* 函数一次打开的文件数量。

H8S, H8/300 系列 C/C++编译程序应用笔记

ASCII 代码表

附录 E　ASCII 代码表

表 E.1 ASCII 代码表

	上端四位							
下端四位	**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