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.

M16C 族 C 编译器应用笔记

前言

前言

这些应用笔记说明如何使用 NC308WA V.5.20 Release 02 和 NC30WA V.5.30 Release 02 有效建立可利用 Renesas Technology M16C 族的功能与性能之应用程序的方法。

要获取 C 编译程序的详细规格，请参考 NC308 用户手册和 NC30 用户手册。

相关手册

- Renesas Tecnology M16C 族硬件手册 (Renesas Tecnology M16C Family Hardware Manual)
- 高性能嵌入式工作区用户手册 (High-performance Embedded Workshop User's Manual)
- NC308 用户手册
- NC30 用户手册
- M16C/80 及 M32C/80 系列编程指导 <C 语言>

本应用笔记所使用的符号以及惯例。
[]: 表示可省略格子内的项目。

(RET): 表示需要按下回车 (Enter) 键。

Δ: 表示一个或多个空格或制表符。

abc: 加粗项目必须由用户输入。

<>: 须指定括号内的项目。

… : 表示紧邻在前的项目被指定一次或多次。

H: 后面标有 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。

0b: 前面标有 0b 的整数常数为二进制。

Linux 是 Linus Torvalds 的商标。

Turbolinux 及其商标是 Turbolinux, Inc. 的商标。

Solaris 是 Sun Microsystems, Inc. 的注册商标。

建议依照以下方式阅读这些应用笔记。

M16C 族 C 编译器应用笔记

目录

目录

M16C 族 C 编译器应用笔记

概述

第 1 节 概述

1.1 摘要

NC30WA 及 NC308WA 编译程序汲取 Renesas Technology M16C 族应用于嵌入式应用程序的单片微型计算机功能与性能，使其 C 程序语言的建立更具效率。

本文档将说明使用这些 C 编译程序建立应用程序的步骤。

1.1.1 规格摘要

语言规格	ANSI 标准
INT 类型	16 位
支持的数据类型	8、16、32 和 64 位整数类型 32 和 64 位浮点类型
功能	多存储器型号 日文字符支持 高 ROM 效率

1.2 功能

下表显示 M16C 族编译程序 NC30WA 及 NC308WA 的功能。

性能	顶级 ROM 效率。
存储器型号	可使用 "near/far" 来详细指定存储器。
可扩充性	使用 #pragma 强力支持嵌入式功能，如：中断函数和地址声明。
地域字符	支持 EUC 字符编码，能够以欧洲及亚洲语言指定字符串常数和注解。
实用程序	支持 ROM 压缩实用程序。
随附工具	集成开发环境（TM 及 HEW） 支持结构代码的汇编程序 模拟程序

1.3 安装方法

1.3.1 PC 版本

要执行安装，请启动安装程序然后依照所显示的说明进行。开始安装前请确定先检查许可 ID，因为在安装期间中需要输入此 ID。

安装过程中输入的数据用于建立用户注册文件（该文件仅为 PC 版本建立）。

1.3.2 UNIX 版本

依照下列步骤执行 UNIX 版本的安装。下列说明以 Solaris 版本为根据。要在另一台主机上执行安装，请依照需要更改目录名称。

加下划线的区域表示从键盘输入的文本。

- 定位至包含安装程序的目录

  定位至 CD-ROM 上包含您要安装的产品之安装程序的目录。在此情形下，所安装的 CD-ROM 上的目录为 /cdrom。将此目录名称更改为您系统上的适当名称。

  ```
  >cd /cdrom/nc30wa/solaris
  ```

- 启动安装程序

  执行安装命令。

  ```
  >./setup
  ```

- 输入许可 ID

  - 输入所列出的许可 ID，包括所有的连字号 (-)。
  - 输入产品随附的许可 ID 证书上所列出的 ID。
  - 如果许可 ID 不正确，下列信息将会显示。在此情形下，请再次输入许可 ID。

  非法许可 ID。'1234-5678-9012-3456-7890'

```
Tool Installation Utility V.1.20.00
COPYRIGHT(C) 1998 (2003) RENESAS TECHNOLOGY CORPORATION
AND RENESAS SOLUTIONS CORPORATION
ALL RIGHTS RESERSVED
Please type license ID.[xxxx-xxxx-xxxx-xxxx-xxxx]
```

- 指定压缩文件

  - 默认指定为 “data.0”。按下 [返回 (Return)] 键以继续。

  ```
  Program package file name[data.0]
  ```

- 指定安装目录
  - 如果目录已经存在，它必须可以写入并且具有足够的磁盘空间。

  ```
  Install Directory[/usr/local/mtool] /usr/common
  ```

- 检查安装目录
  - 如果指定的目录不存在，系统将会询问您是否要建立目录。输入 “y”（是）或 “n”（否）。

  目录 '/usr/common/tool' 不存在。
  是否要建立该目录 (y/n) y

  即使您输入 “y”，还是有可能无法建立该目录。在此情形下，终止安装，使用 “mkdir” OS 命令建立该目录，然后再次执行安装。

- 指定工作目录
  - 这是用于安装程序操作的临时目录。此目录必须可以写入并且具有足够的磁盘空间。

  默认指定为 /tmp。如果 /tmp 目录具有足够的磁盘空间，您可以按下 [返回 (Return)] 键以继续。

  ```
  Installer Temporary Directory[/tmp]
  ```

- 开始文件转移
  - 转移之文件的名称将按照顺序显示。可能需要一些时间才可显示出来。
  - 如果输入的许可 ID 与要安装的产品不符，将不会进行文件转移，下列信息将会显示，而处理程序也会停止。在此情形下，请再次运行安装程序。

  ```
  License ID error 'data.0'
  ```

  ```
  Setup made a file, named 'toolinfo.txt', at the directory
  "/usr/common/tool/support/nc30wa".
  And Setup recorded product information in this file.
  Please use this file if you need calling us.
  ```

- 停止文件转移
  - 包含产品信息的文件将会在信息中显示的文件内建立（请参阅信息以获得名称和目录）。
  - 目录的名称将取决于安装目录。

  ```
  Installation completed.
  ```

1.4 执行方法

1.4.1 启动编译程序

- 编译驱动器的命令输入格式

编译驱动器会运行编译程序命令、汇编程序命令和连接命令，以及建立机器语言数据文件。运行编译驱动器时将需要下列信息（输入参数）：

1. C 源文件
2. 汇编源文件
3. 可再定位目标文件
4. 命令行选项（按需要指定的项目）

在命令行上输入这些项目。至少输入 1、2 和 3 之中的其中一个项目。图 1.1 显示输入格式，而图 1.2 显示输入实例。在输入实例中，下列操作将会执行，绝对模块文件“sample.x30”也将会建立：

1. 启动程序“ncrt0.a30”已汇编。
2. C 源程序“sample.c”已编译/汇编。
3. 可再定位目标文件“ncrt0.a30”已连接到“sample.r30”。

启动选项如下：

- 指定绝对模块文件“sample.x30”的名称的选项： "-o"
- 指定汇编期间输出的列表文件 (*.lst) 的选项： -as30 "-l"
- 指定连接期间输出的映像文件 (*.map) 的选项： -ln30 "-ms"

```
% nc30{Δ}[启动选项]{Δ}<[汇编语言源文件的名称]{Δ}
        [可再定位目标文件的名称]{Δ}[C 源文件的名称]>
%: 表示命令提示符。
< >: 表示必要的项目。
[ ]: 表示可选择的项目。
Δ: 表示一个空格。
```

图 1.1 用于编译驱动器的输入格式

```
% nc30 -osample -as30 "-l" -ln30 "-ms" ncrt0.a30 sample.c{return}
{return}: 表示已输入“返回”（Return）键。
连接时请确定先指定启动程序。
```

图 1.2 编译驱动器命令的输入实例

1.4.2 启动嵌入式工作区

正确执行安装后，“嵌入式工作区”连同“嵌入式工作区”的每个可执行程序的快捷方式将会放置在名为 [瑞萨高性能嵌入式工作区 (Renesas High-performance Embedded Workshop)] 的文件夹中，它位于 Windows [开始 (Start)] 菜单中的 [程序 (Programs)] 文件夹内。请注意，[开始 (Start)] 菜单中显示的项目将取决于所安装的工具而有所不同。

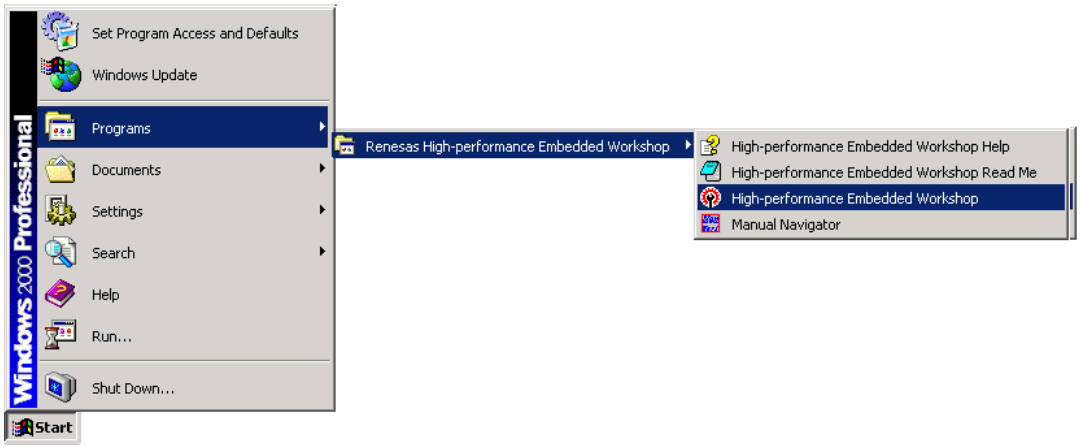


图 1.3 从“开始”(Start) 菜单启动“嵌入式工作区”

从“开始”(Start) 菜单，当您单击“嵌入式工作区”(Embedded Workshop) 时，启动信息将会显示，然后显示 [欢迎! (Welcome!)] 对话框（如图 1.4 所示）。

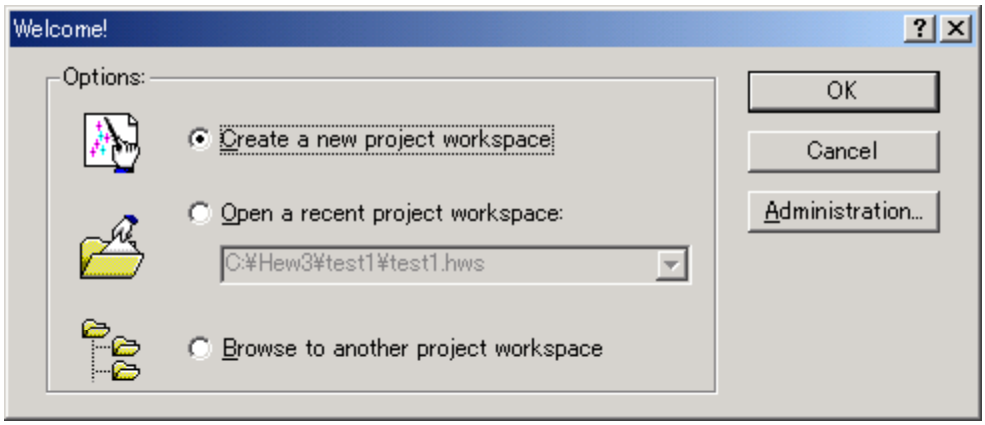


图 1.4 [欢迎 (Welcome!)] 对话框

如果是第一次使用“嵌入式工作区”(Embedded Workshop)，或是要建立新的工程，请选取 [建立新的工程工作空间 (Create a new project workspace)]，然后单击 [确定 (OK)] 按钮。否则，若要在已建立的工程中工作，请选取 [打开最近使用的工程工作空间 (Open a recent project workspace)] 或 [浏览至另一个工程工作空间 (Browse to another project workspace)]，然后单击 [确定 (OK)] 按钮。您也可以单击 [管理... (Administration...)] 按钮，添加或删除与“嵌入式工作区”配合使用的系统工具。

1.5 程序开发的步骤

图 1.5 显示使用 NC308 进行程序开发的流程实例。以下是此程序的概述（项目 1 至 4 与图 1.5 中的项目相应）。

(1) C 源程序 ("AA.c") 由 nc308 编译以及由汇编程序 as308 汇编，而可再定位目标文件 ("AA.r30") 将会建立。

(2) 段位置、段大小的设定，以及中断向量表将会更改以符合已嵌入包含文件 “sect308.inc” 的系统。此文件包含启动程序 “ncrt0.a30” 的代码，也包含关于段的信息。

(3) 更改的启动程序将会汇编，由此，可再定位目标文件 “ncrt0.r30” 也会建立。

(4) 两个可再定位目标文件 “AA.r30” 和 “ncrt0.r30” 将会由从 nc308 执行的连接编辑程序 ln308 连接。接着，便会建立绝对模块文件 ("AA.x30")。

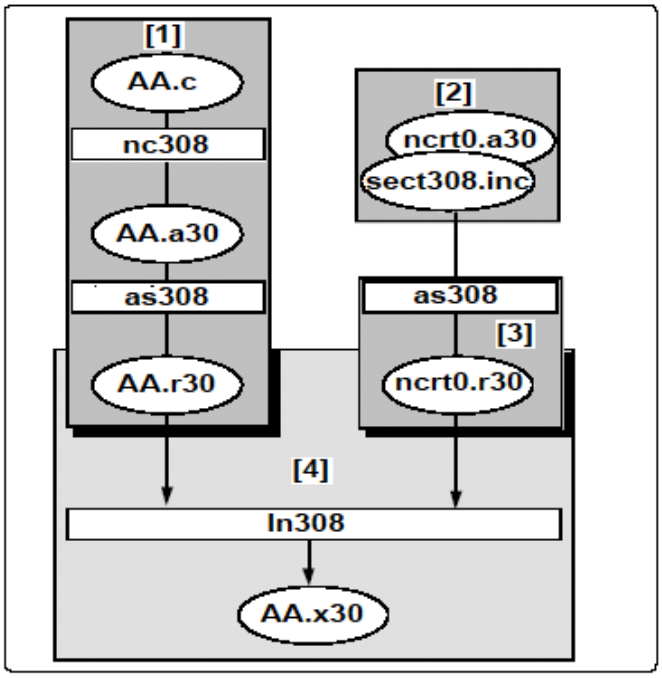


图 1.5 程序开发的流程

备忘录

M16C 族 C 编译器应用笔记

建立程序的步骤

第 2 节 建立程序的步骤

2.1 建立工程

(1) 指定工程

当您在 [欢迎! (Welcome!)] 对话框中选取 [建立新的工程工作空间 (Create a new project workspace)] 单选按钮和单击 [确定 (OK)] 后，用于建立新的工作空间和工程的 [新的工程工作空间 (New Project Workspace)] 对话框（图 2.1）将会启动。您可以在这个对话框中指定工作空间名称（建立新的工作空间时，工程名称在默认情形下将会一样）、CPU 家族和工程类型等等。例如，当您在 [工作空间名称 (Workspace Name)] 字段中输入 “tutorial”，那么 [工程名称 (Project Name)] 字段将显示为 “tutorial” 且 [目录 (Directory)] 字段也将显示为 “C: \Hew3\tutorial”。如果您要更改工程名称，请在 [工程名称 (Project Name)] 字段中人工输入一个新的工程名称。如果您要更改用作新的工作空间的目录，可单击 [浏览… (Browse…)] 按钮指定一个目录，或直接在 [目录 (Directory)] 字段中人工输入目录路径。

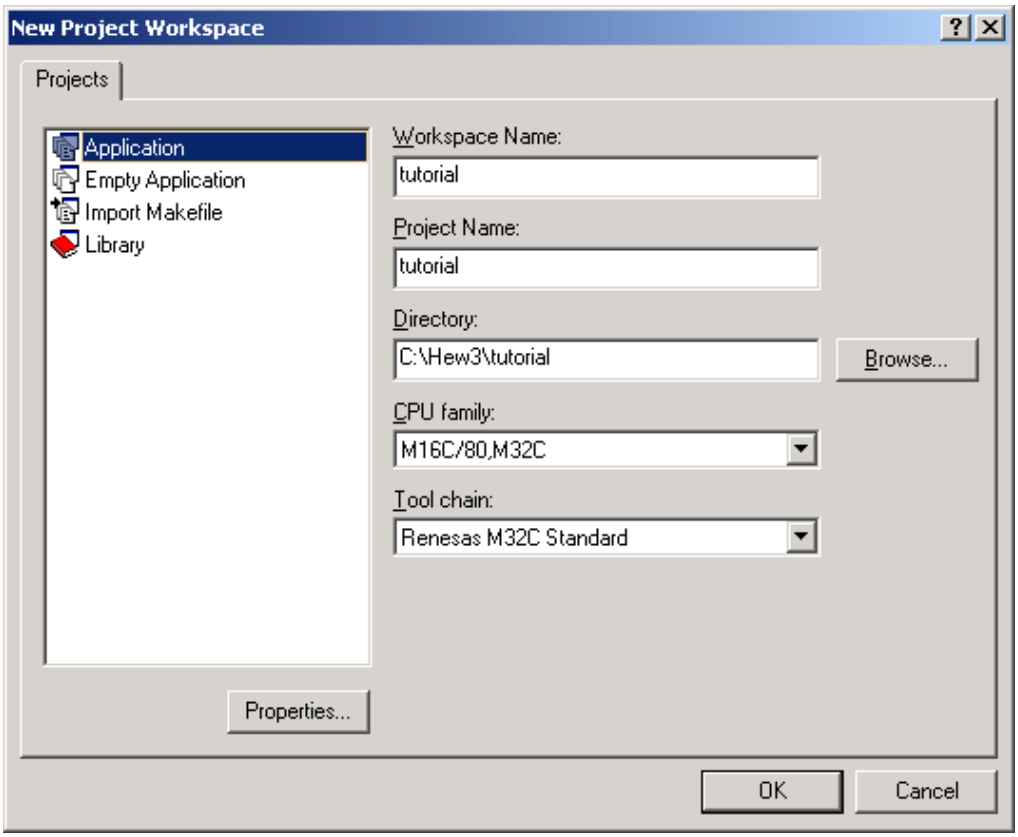


图 2.1 “新的工程工作空间”(New Project Workspace) 对话框

(2) 选择目标 CPU

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

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

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

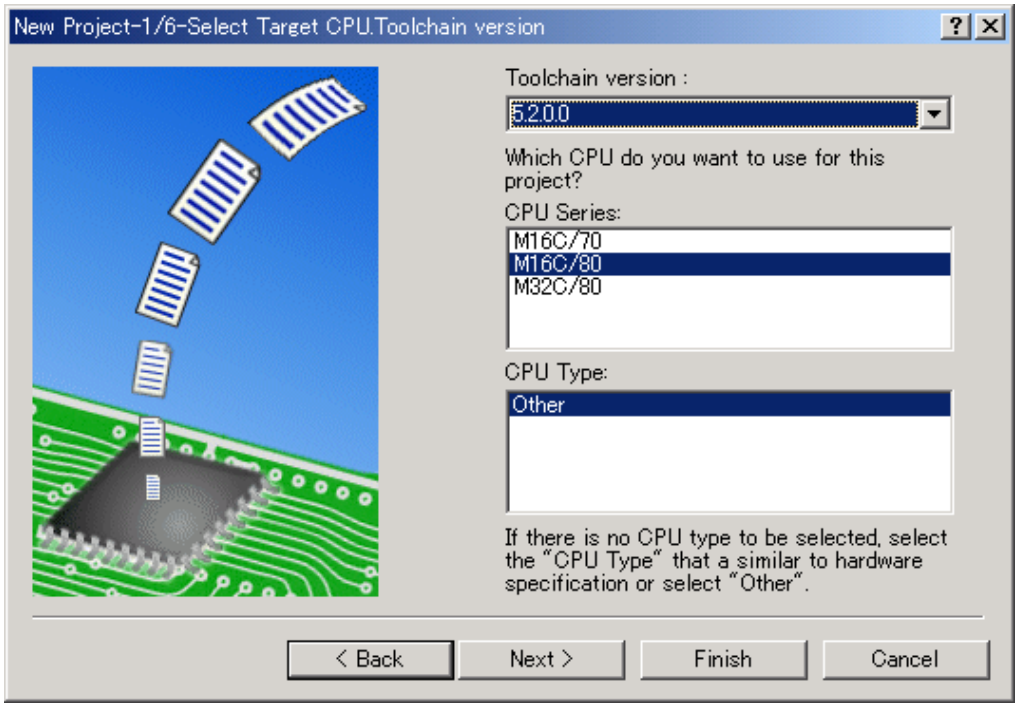


图 2.2 “新的工程步骤 1”(New Project Step 1) 对话框

(3) 选择 RTOS

在“步骤 1”(Step-1) 画面上单击 [下一步> (Next>)] 按钮将打开图 2.3 中显示的对话框。在此窗口中，您可以选择是否要使用 RTOS，以及是否要使用默认的启动文件或用户指定的文件。

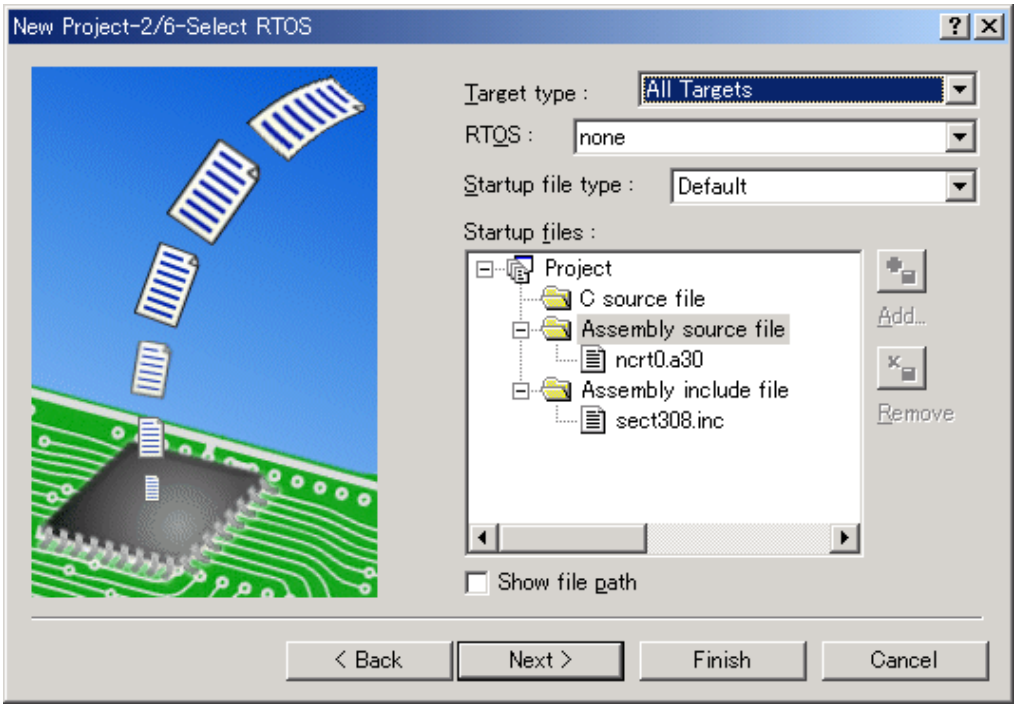


图 2.3 “新的工程步骤 2”(New Project Step 2) 对话框

(4) 选择输入/输出程序库，以及堆大小

在“步骤 2”(Step-2) 画面上单击 [下一步> (Next>)] 按钮将打开图 2.4 中显示的对话框。在此窗口中，您可以选择是否要使用 I/O 程序库、是否要生成“main”函数文件，以及堆区域的大小。

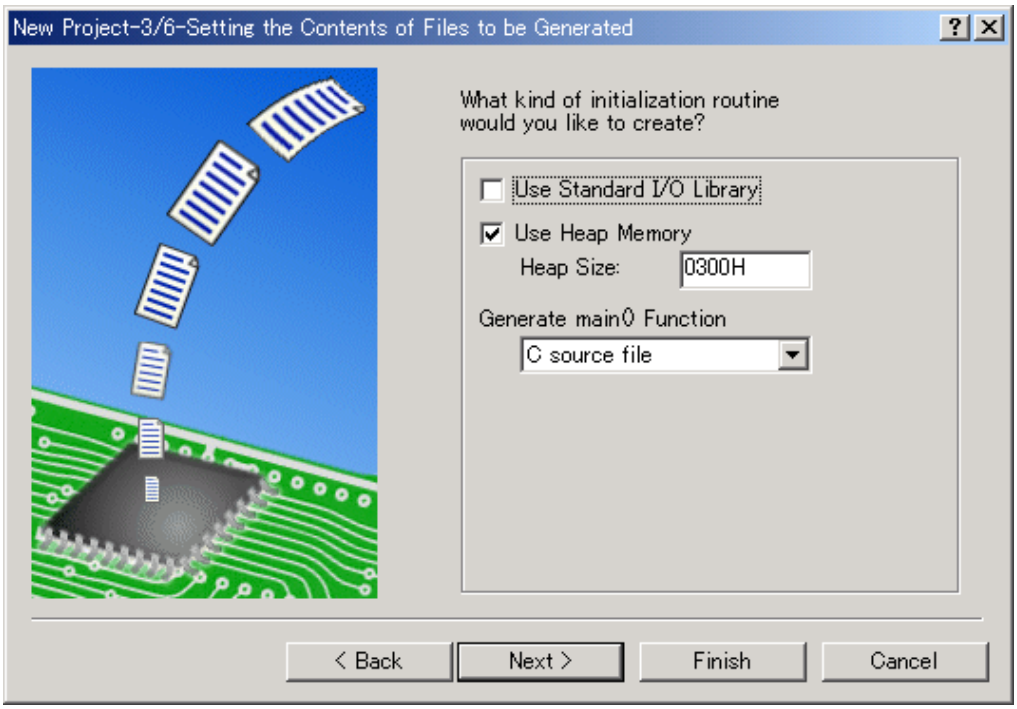


图 2.4 “新的工程步骤 3”(New Project Step 3) 对话框

(5) 设定堆栈区域

在“步骤 3”(Step-3) 画面上单击 [下一步> (Next>)] 按钮将打开图 2.5 中显示的对话框。您可以使用此窗口来设定堆栈大小和中断堆栈大小。

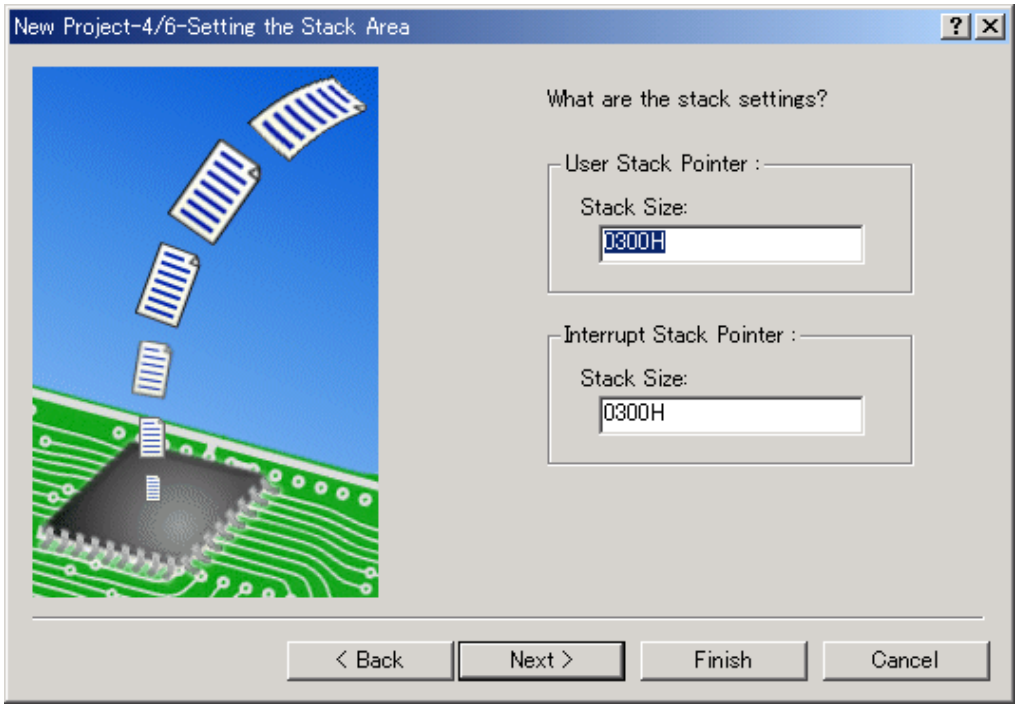


图 2.5 “新的工程步骤 4”(New Project Step 4) 对话框

(6) 设定调试程序选项

在“步骤 4”(Step-4) 画面上单击 [下一步> (Next>)] 按钮，图 2.6 中显示的画面将会出现。您可以使用此窗口来执行外部封装调试程序的设定。在此窗口上， 通过[目标: (Targets:)]下选项来选择调试程序目标。您也可以不选择调试程序目标。

注意，您同样可以选择外部调试程序 (Extenal Debugger)，假如存在的话。

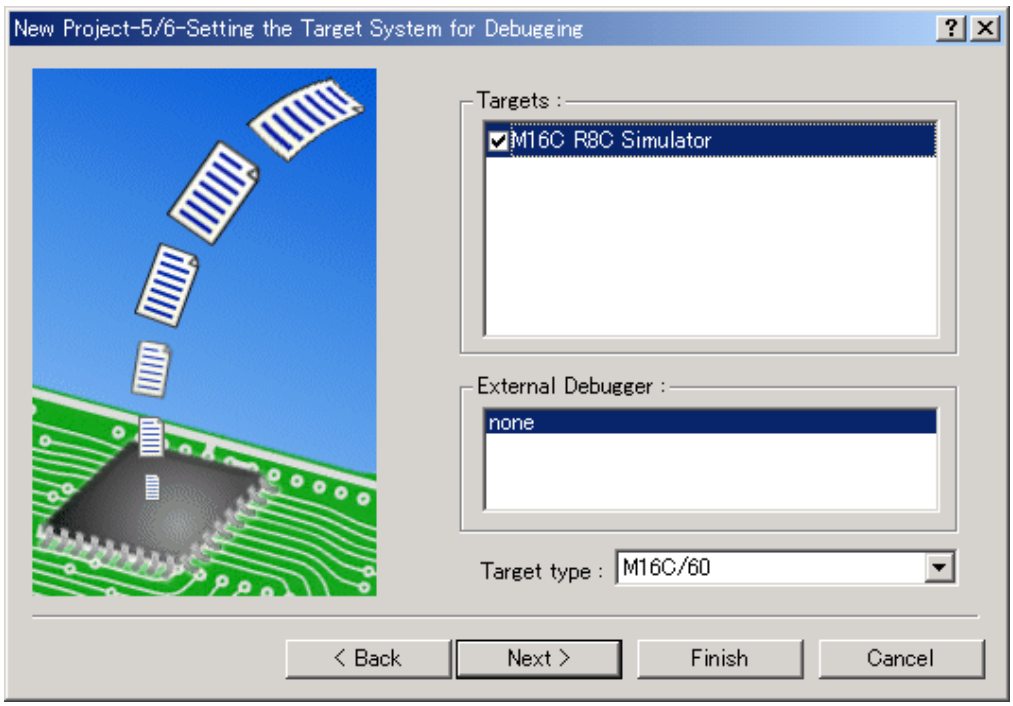


图 2.6 “新的工程步骤 5”(New Project Step 5) 对话框

(7) 设定配置文件 (Configuration File) 名

在“步骤 5”(Step-5) 画面上单击 [下一步> (Next>)] 按钮将显示图 2.7 中显示的画面。

在此窗口，为每一个选择的目标设定配置文件名。

配置文件指的是储存了非目标高性能嵌入式工作区状态的文件。

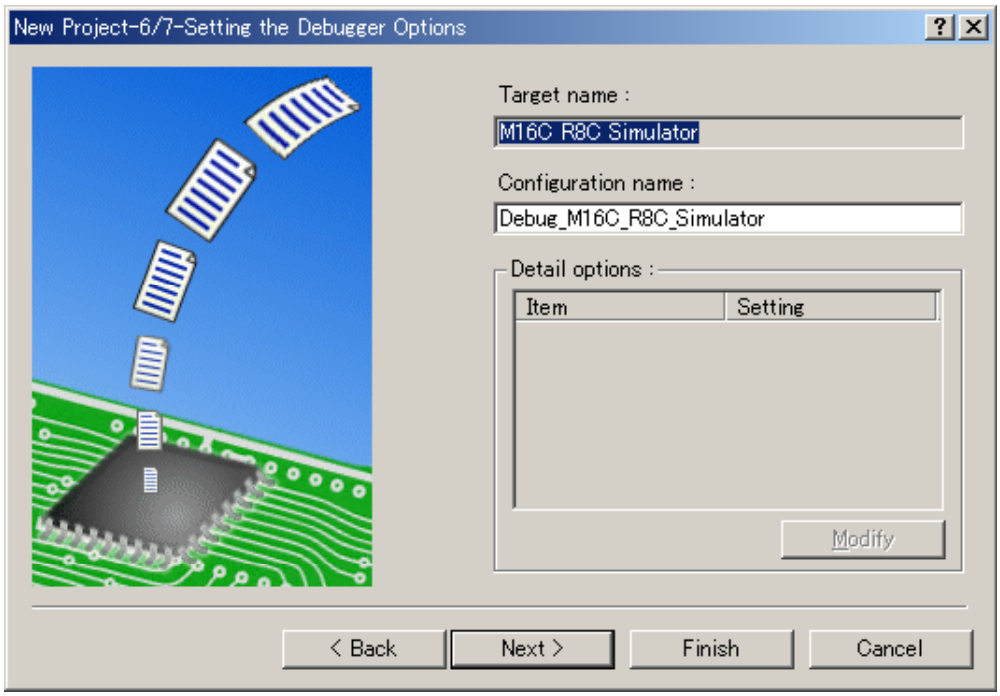


图 2.7 “新的工程步骤 6”(New Project Step 6) 对话框

(8) 确认设定（“摘要”对话框）

在“步骤 6”(Step-6) 画面上单击 [下一步> (Next>)] 按钮将显示图 2.8 中显示的画面。此窗口显示所要建立的工程的源文件信息。确认后，单击 [完成 (Finish)]。

在图 2.8 中显示的画面上单击 [完成 (Finish)] 时，工程生成程序将会在 [摘要 (Summary)] 对话框中显示所生成文件的列表（图 2.9）。确认该对话框的内容然后单击 [确定 (OK)]。

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

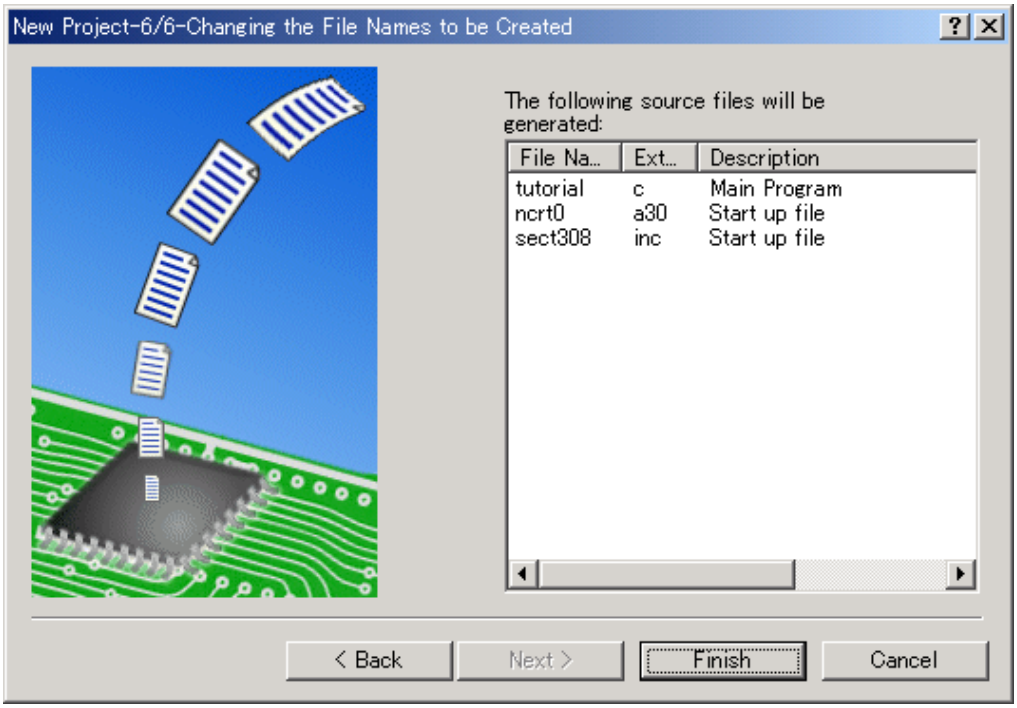


图 2.8 “新的工程步骤 7”(New Project Step 7) 对话框

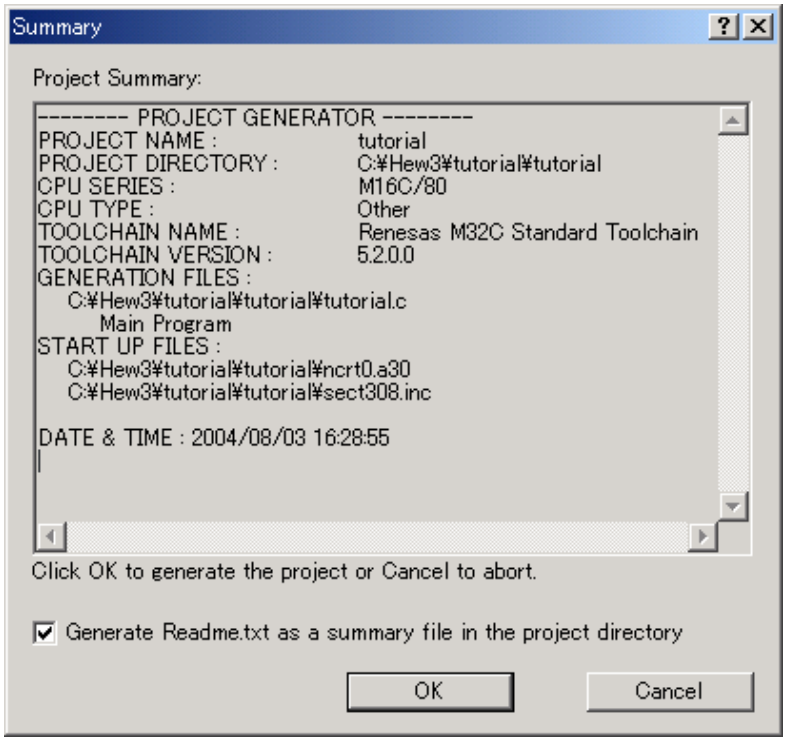


图 2.9 “摘要”(Summary) 对话框

2.2 启动程序

2.2.1 启动程序的用途

让内建程序可以正确运行，进行前，微型计算机必须初始化，而堆栈区域也必须设定。由于这些处理程序通常不能使用 C 代码执行，因此使用 C 源程序以外的程序通过汇编代码来执行初始化和设定。它称为*启动程序*。以下说明由 NC308 预备的 “ncrt0.a30” 和 “sect308.inc” 的样品启动程序。

启动程序的用途：

① 保护堆栈区域

② 为微型计算机执行初始设定

③ 初始化静态变量空间

④ 设定中断表寄存器 INTB

⑤ 调用 `main` 函数

⑥ 设定中断向量表

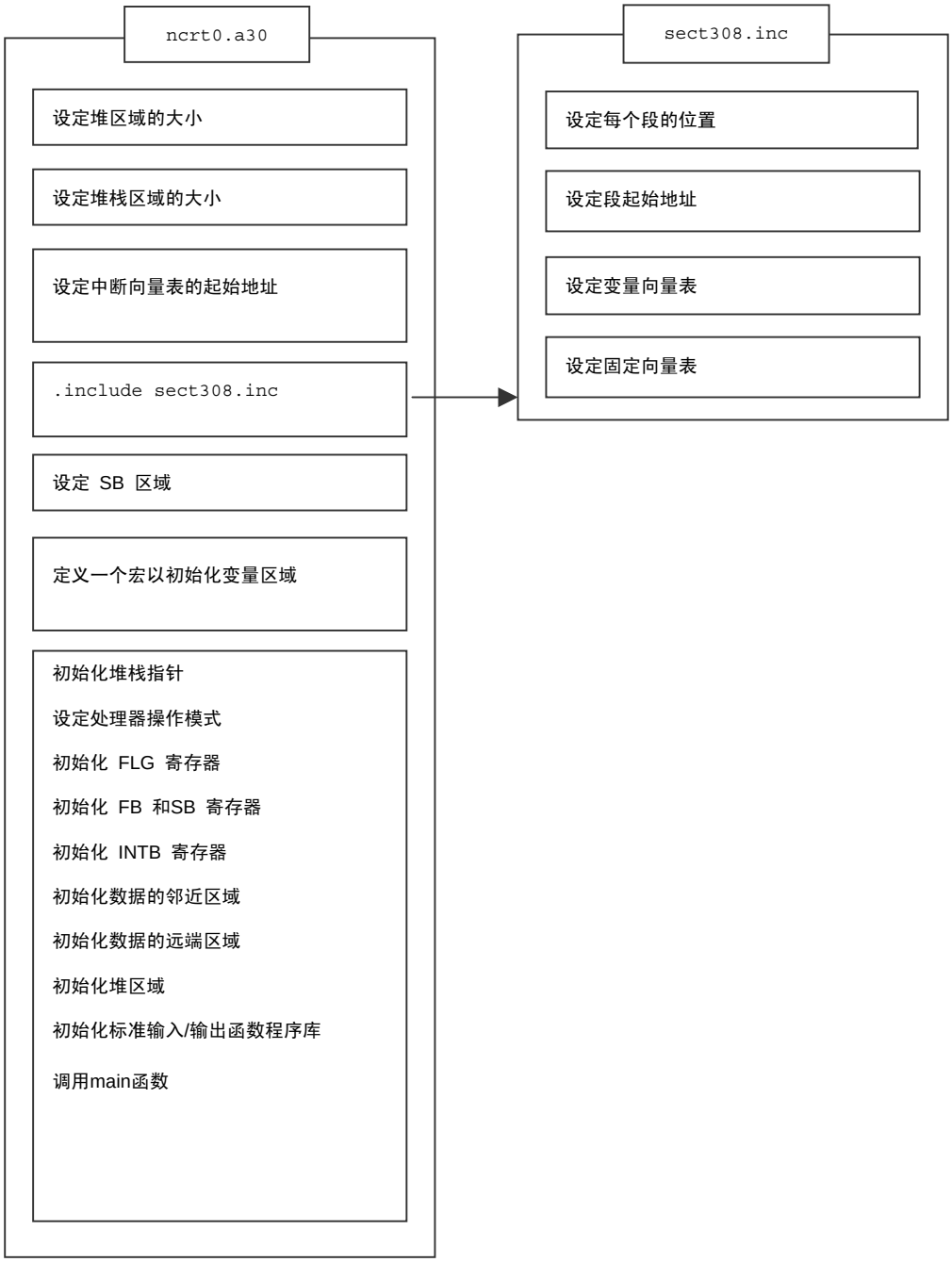


图 2.10 启动程序的结构

2.2.2 设定启动程序

(1) 添加段名称

由 NC308 建立的段将会在段定义文件 “sect308.inc” 中定义。当您更改 “#pragma SECTION” 中的段名称时，将会添加由 NC308 建立的段基本名称。由此，您必须在段定义文件 “sect308.inc” 中添加这些定义。

```
;------------------------------------------------------------------------
; 段的编排
;------------------------------------------------------------------------
;------------------------------------------------------------------------
; 靠近 RAM 数据区域
;------------------------------------------------------------------------
; SBDATA area
        .section data_SE,DATA
        .org 400H
data_SE_top:
;
        .section bss_SE,DATA
bss_SE_top:
...
        .section data_NO,DATA
data_NO_top:
;
.     section    new_data_NE,DATA
new_data_NE_top:
;
      section    new_bss_NE,DATA
new_bss_NE_top:
...
;------------------------------------------------------------------------
; 代码区域
;------------------------------------------------------------------------
        .section interrupt
;
        .section program
;
        .section new_program
...
```

定义在 “#pragma SECTION.” 中更改的附加段名称。

定义在 “#pragma SECTION.” 中更改的附加段名称。

定义在 “#pragma SECTION.” 中更改的附加段名称。

图 2.11 添加段名称 (sect308.inc)

添加数据段或 bss 段时，除了添加段名称的定义外，您还需要包含每个区域的初始值转移处理和零清除处理。因此，请确定将这些添加到 “ncrt0.a30”。

```
;=====================================================================
; 靠近区域初始化
;---------------------------------------------------------------------
; bss 零清除
;---------------------------------------------------------------------
      BZERO    bss_SE_top,bss_SE
      BZERO    bss_SO_top,bss_SO
      BZERO    bss_NE_top,bss_NE
      BZERO    bss_NO_top,bss_NO"
      BZERO    new_bss_NE_top,new_bss_NE
;---------------------------------------------------------------------
; 初始化数据段
;---------------------------------------------------------------------
      BCOPY    data_SEI_top,data_SE_top,data_SE
      BCOPY    data_SOI_top,data_SO_top,data_SO
      BCOPY    data_NEI_top,data_NE_top,data_NE
      BCOPY    data_NOI_top,data_NO_top,data_NO
      BCOPY    new_data_NEI_top,new_data_NE_top,new_data_NE
;
```

为新的 bss 段添加零清除处理。

为新的数据段添加初始值转移处理。

图 2.12 为所添加的段添加初始化处理

(2) 注册中断函数

要正确使用中断，您需要指定中断处理函数，以及在中断向量表中将它注册。以下说明如何在中断向量表中执行注册。

要指定中断处理函数，请通过更改样品启动程序“sect308.inc”中的中断向量表以执行注册。

对中断向量表执行下列更改：

① 使用“.glb”指令命令为中断处理函数制作一个外部参考声明。

② 对于要使用的中断，将虚设函数名称 dummy_int 改成中断处理函数的名称。

```
;-----------------------------------------------------------------------
;    变量向量段
;-----------------------------------------------------------------------
      .section vector ; variable vector table
      .org VECTOR_ADR
;
.     lword dummy_int ; vector (BRK)
      .org ( VECTOR_ADR + 32 )
      .lword dummy_int ; DMA0 (software int 8)
      .lword dummy_int ; DMA1 (software int 9)
      .lword dummy_int ; DMA2 (software int 10)
      .lword dummy_int ; DMA3 (software int 11)
      .glb _ta0
      .lword _ta0 ; TIMER A0 (software int 12)
      .lword dummy_int ; TIMER A1 (software int 13)
      .lword dummy_int ; TIMER A2 (software int 14)
      .lword dummy_int ; TIMER A3 (software int 15)
      .lword dummy_int ; TIMER A4 (software int 16)
      ...
```

注册 TA0 中断的函数 ta0()

图 2.13 中断向量表 (sect308.inc)

M16C 族 C 编译器应用笔记

编译程序

第 3 节 编译程序

3.1 中断函数

3.1.1 编写中断处理函数

NC308 可以让您将中断处理编写为 C 函数。该过程包含四个步骤。

① 编写中断处理函数

② 在中断向量表上注册该函数

③ 设定中断允许标志（I 标志）

- 使用内联汇编。

④ 设定中断优先级

- 允许中断前先设定中断优先级。

本小节将描述如何根据中断处理类型编写函数。

(1) 编写硬件中断 (#pragma INTERRUPT)

```
#pragma INTERRUPT 中断函数名称
```

除了正规函数步骤外，此声明将使编译程序在指定函数的入口和出口点生成下列指令： 该指令用于保存和恢复函数与 reit 指令内所使用的所有寄存器。只有 void 类型可用作中断处理函数的参数和返回值。如果声明含有任何其他类型，编译期间将会输出一则警告。

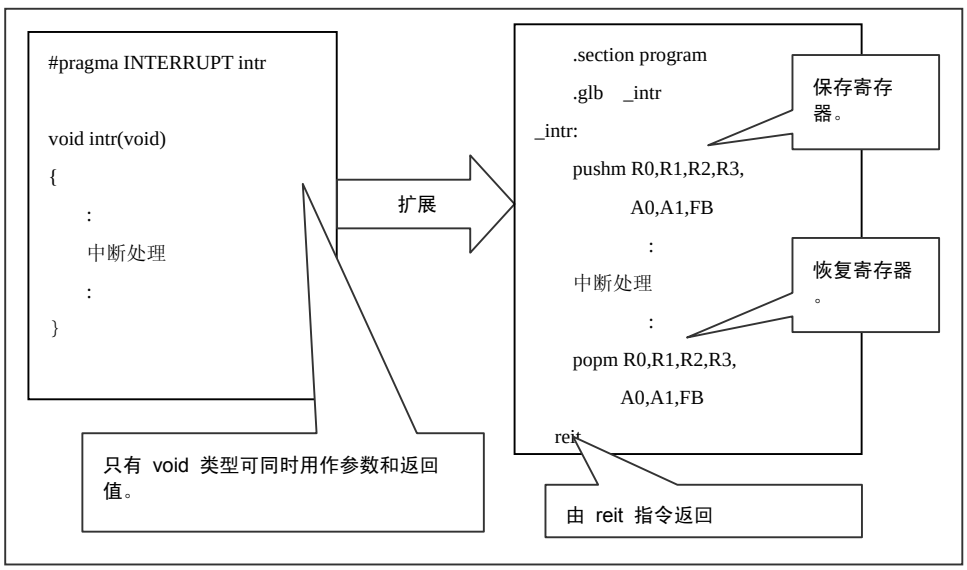


图 3.1 中断处理函数的扩展

(2) 使用寄存器库编写中断 (#pragma INTERRUPT/B)

对于 M16C/80 系列，您可以切换寄存器库来减少启动中断处理所需的时间，同时维护寄存器的内容。要使用此功能，请声明下列编写：

```
#pragma INTERRUPT/B 中断函数名称
```

上述编写将使编译程序生成切换寄存器库的指令，而不是用于保存和恢复寄存器的指令。但是，您只可以指定一个中断，因为 M16C/80 系列产品提供寄存器库 0 和 1。请将此功能用于需要更快启动的中断。

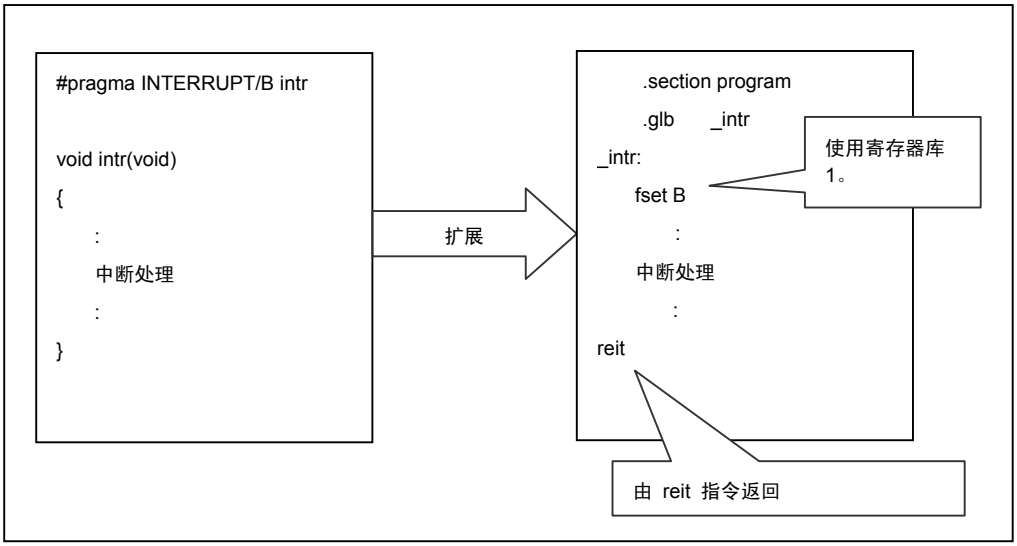


图 3.2 使用寄存器库的中断处理函数的扩展

(3) 编写允许多个中断的中断 (#pragma INTERRUPT/E)

当 M16C/80 系列编译程序接收中断请求时，中断允许标志（I 标志）将会设定为 “0”，并禁止中断。通过将中断处理函数入口点的 I 标志设定为 “1”（紧随输入中断处理函数后）可允许多个中断，您可以由此提高中断响应。要使用此功能，请依照下列方式编写：

```
#pragma INTERRUPT/E 中断函数名称
```

上述编写将使编译程序在中断处理函数入口点生成将 I 标志设定为 “1” 的指令（紧随输入中断处理函数后）。但是，如果您要在中断处理函数的中间允许多个中断，请声明 “#pragma INTERRUPT”，然后在中断处理函数的中间使用 asm() 函数将 I 标志设定为 “1”。

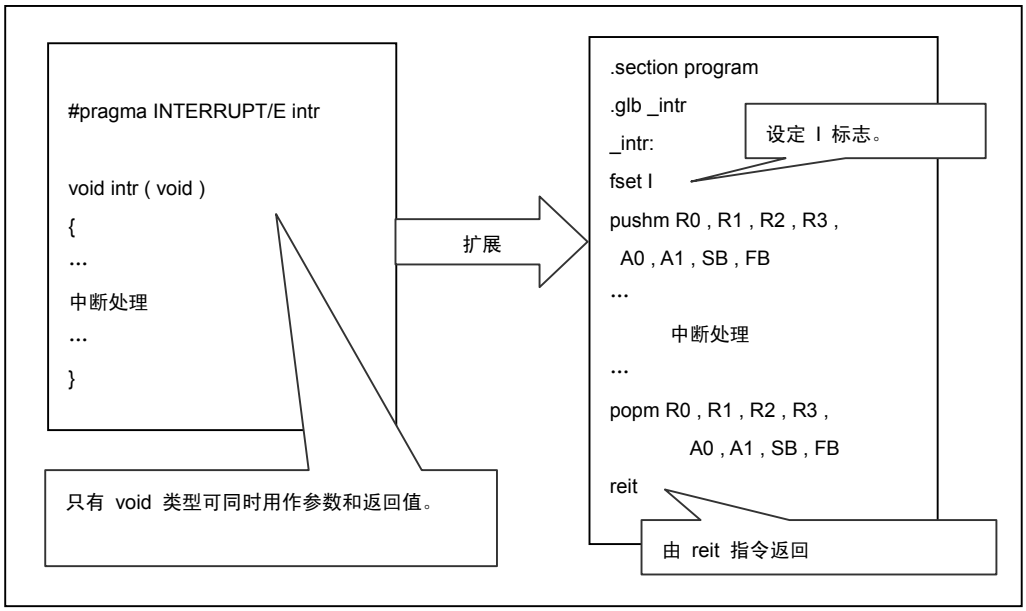


图 3.3 允许多个中断的中断处理函数的扩展

3.1.2 编写快速中断处理函数

NC308 可以让您将快速中断处理编写为 C 函数，它可以在五个周期内响应中断，以及在三个周期内从中断返回。但是，您只可以将中断优先级 7 的单个中断设定为快速中断。该过程包含五个步骤。

① 编写快速中断处理函数

② 设定快速中断的中断优先级

- 允许中断前先设定中断优先级。

③ 设定快速中断位

④ 设定向量寄存器 (VCT)

⑤ 设定中断允许标志（I 标志）

- 使用内联汇编程序。

本小节将描述如何根据快速中断处理类型编写函数。

(1) 编写快速硬件中断 (#pragma INTERRUPT/F)

```
#pragma INTERRUPT/F 中断函数名称
```

除了正规函数步骤外，上述声明将使编译程序在指定函数的入口和出口点生成下列指令： 该指令用于保存和恢复函数以及从快速中断例程返回的 freit 指令内所使用的所有寄存器。只有 void 类型可用作中断处理函数的参数和返回值。如果声明含有任何其他类型，编译期间将会输出一则警告。

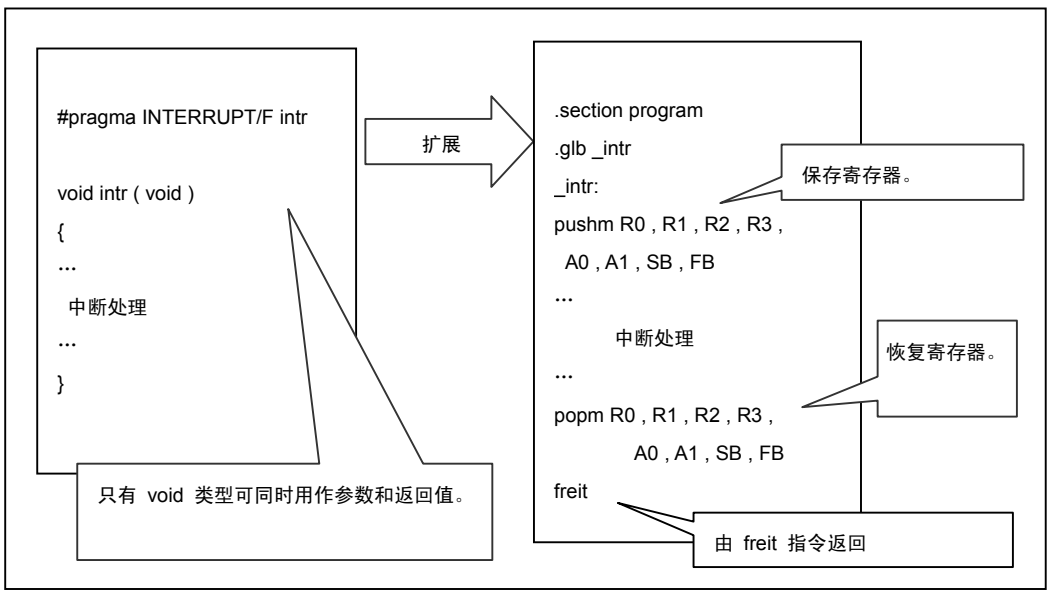


图 3.4 快速中断处理函数的扩展

(2) 使用寄存器库编写快速中断 (#pragma INTERRUPT/F/B)

对于 M16C/80 系列，您可以切换寄存器库来减少启动快速中断处理所需的时间，同时维护寄存器的内容。要使用此功能，请声明下列编写：

```
#pragma INTERRUPT/F/B 中断函数名称
```

上述编写将使编译程序生成切换寄存器库的指令，而不是用于保存和恢复寄存器的指令。但是，您只可以指定一个中断，因为 M16C/80 系列产品提供寄存器库 0 和 1。请将此功能用于需要最快启动的中断。

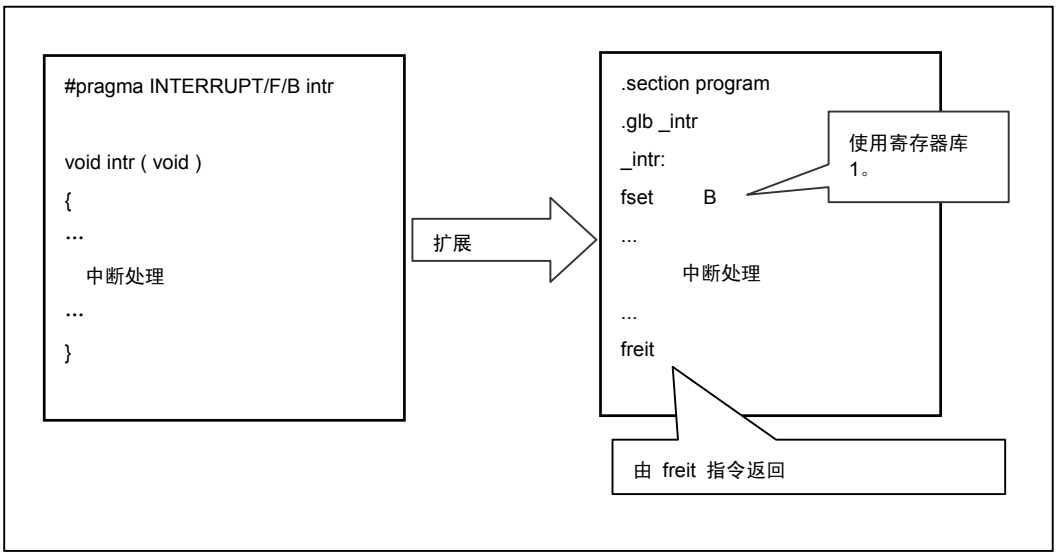


图 3.5 使用寄存器库的快速中断处理函数的扩展

3.1.3 编写软件中断（INT 指令）处理的函数

(1) 编写将调用汇编语言函数的软件中断 (#pragma INTCALL)

要使用 M16C/80 系列的软件中断（INT 指令），请指定 “#pragma INTCALL”。此功能可以让您在调试期间生成伪中断。

编写方法将可以根据由软件中断调用的函数主体是以汇编语言或以 C 编写而有所不同。

如果要调用的函数主体是以汇编语言编写，请依照下列方式编写：

```
#pragma INTCALL 软件中断编号汇编语言函数名称 (寄存器名称, 寄存器, ... )
```

如果要调用的函数主体是以汇编语言编写，您可以通过寄存器传递参数。您还可以收到结构或联合类型以外的返回值。

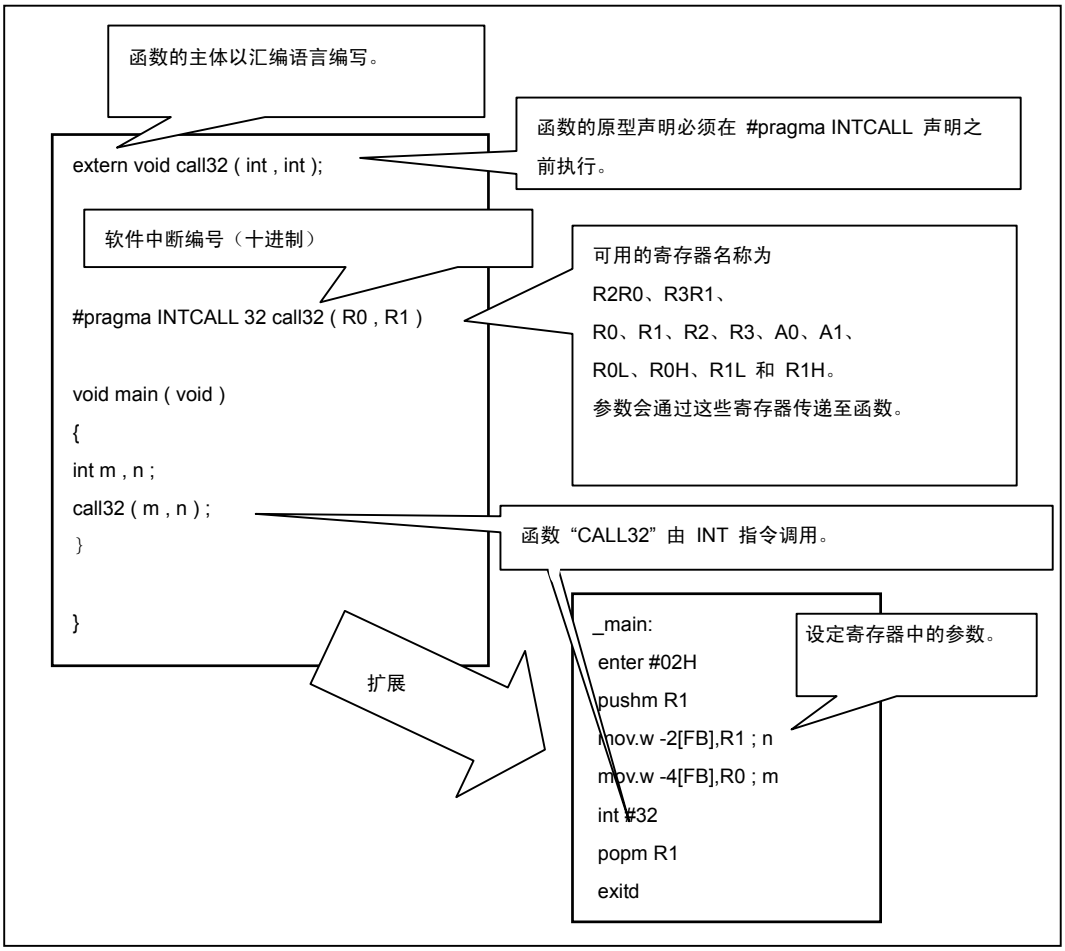


图 3.6 将调用汇编语言函数的 #pragma INTCALL 的编写实例

(2) 编写将调用 C 函数的软件中断 (#pragma INTCALL)

如果软件中断（INT 指令）要调用的函数主体是以 C 编写，请依照下列方式编写：

```
#pragma INTCALL 软件中断编号 C-函数名称 ()
```

如果要调用的函数主体是以 C 编写，根据参数传递规则，您只可以指定通过寄存器传递所有参数的函数。该编写不能包含声明 “#pragma INTCALL” 的函数的任何参数。将会收到结构或联合类型以外的返回值。

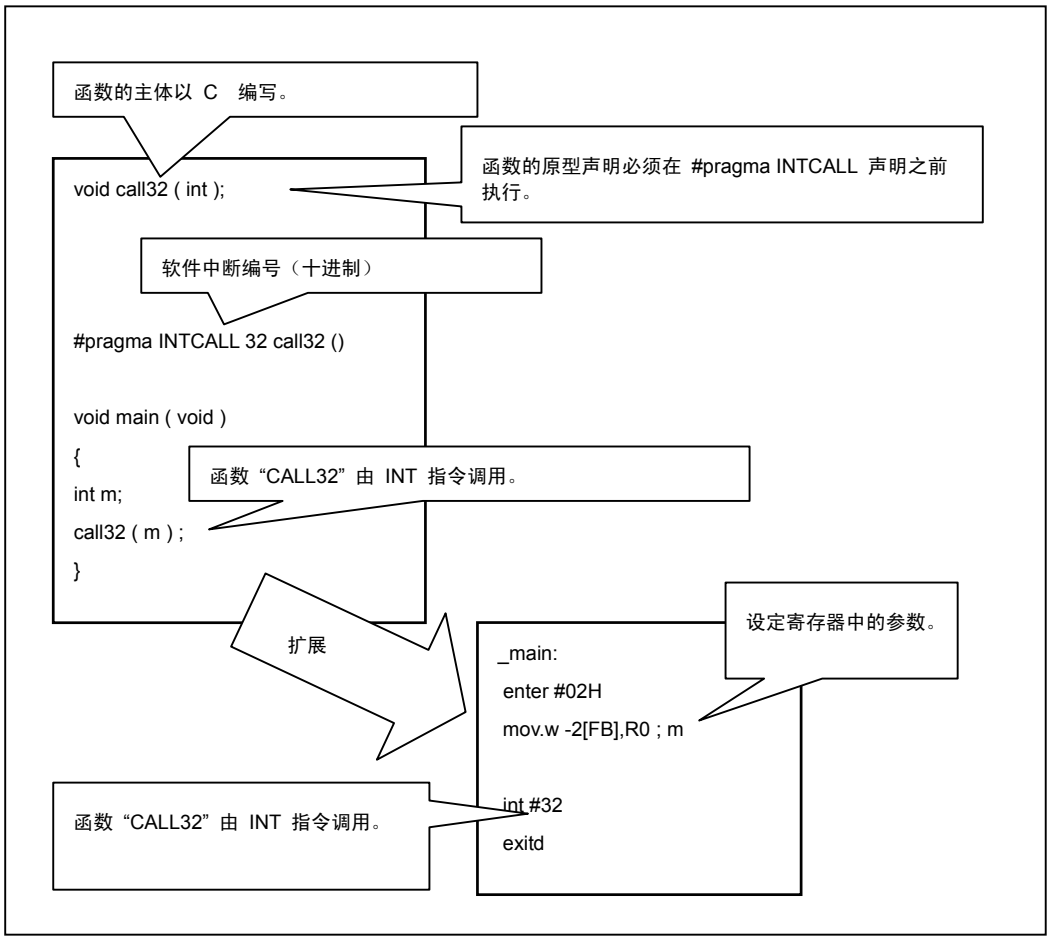


图 3.7 编写将调用 C 函数的 #pragma INTCALL

3.1.4 注册中断处理函数

要正常使用中断，您必须指定中断处理函数，以及在中断向量表中将它们注册。

本小节将描述如何将中断处理函数注册到中断向量表中。

(1) 在中断向量表中注册函数

当您指定中断处理函数时，请通过更改样品启动程序“sect308.inc”中的中断向量表以将它注册。

要更改中断向量表：

① 使用 .glb 指令声明用于外部参考的中断处理函数的名称。

② 将中断的虚设函数名称“dummy_int”改成中断处理函数的名称。

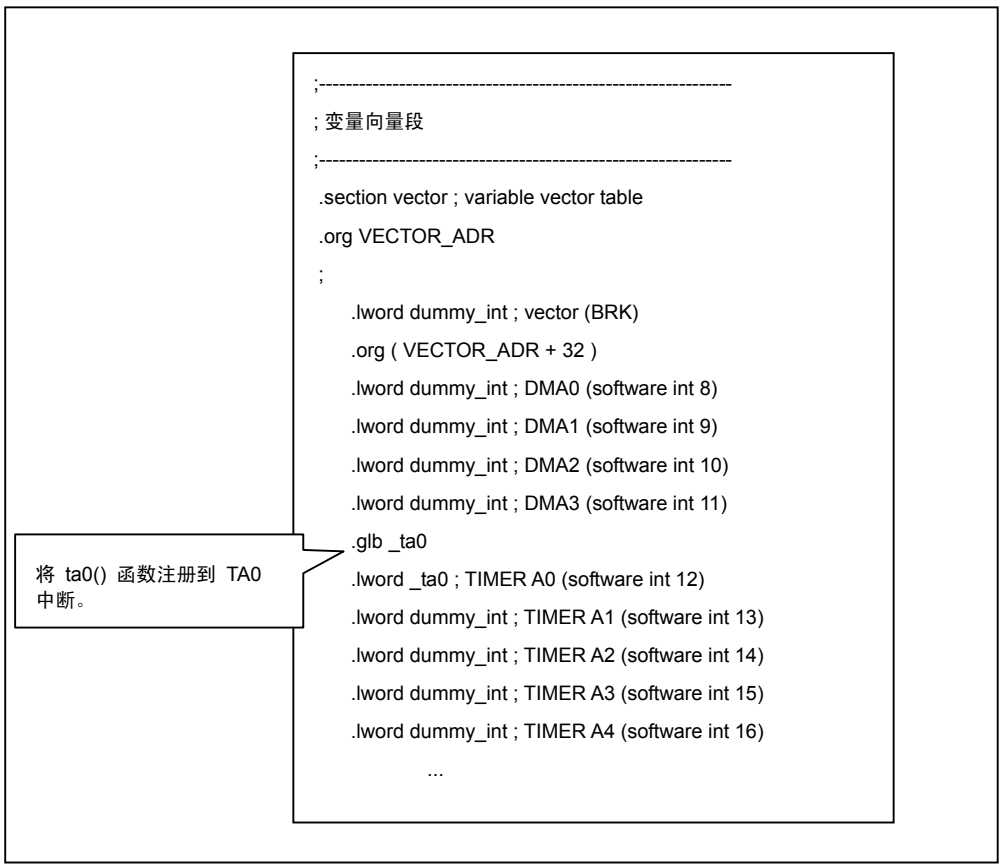


图 3.8 中断向量表 (sect308.inc)

3.1.5 中断处理函数的编写实例

本小节提供如何对程序进行编写的实例，该程序会在每次 INT0 中断发生时将“counter”的内容清除为零，以及在每次 INT1 中断发生时增大“counter”的内容。

(1) 中断处理函数的编写实例

```
/* 原型声明 ********************/
void int0 ( void ) ;
void int1 ( void ) ;
#pragma   INTERRUPT/F   int0
#pragma   INTERRUPT   int1
/*****************************/
unsigned   int   counter ;
void int0 ( void ) /* 快速中断函数 */
{
     counter = 0 ;
}
void int1 ( void ) /* 中断函数 */
{
     if ( counter < 9 ) {
        counter ++ ;
     }
     else {
        counter = 0 ;
     }
}

void main ( void )
{
     INT0IC = 0x07; /* 设定快速中断优先级 */
     RLVL = 0x08; /* 快速中断指定 */
     asm ( " LDC #_int0,VCT " ); /* 设定向量寄存器 */
     INT1IC = 0x01 ; /* 设定中断优先级 */
     asm ( " fset i " ) ; /* 允许中断 */
     while (1) ; /* 中段等待循环 */
}
```

图 3.9 中断处理函数的编写实例

(2) 在中断向量表中注册函数

图 3.10 显示在中断向量表中注册函数的实例。

```
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－
; 变量向量段
;－－－－－－－－－－－－－－－－－－－－－－－－－－－－
.section vector ; variable vector table
.org VECTOR_ADR
      ...
.org ( VECTOR_ADR + 32 )
.lword dummy_int ; DMA0 (software int 8)
.lword dummy_int ; DMA1 (software int 9)
.lword dummy_int ; DMA2 (software int 10)
.lword dummy_int ; DMA3 (software int 11)
.lword dummy_int ; TIMER A0 (software int 12)
.lword dummy_int ; TIMER A1 (software int 13)
.lword dummy_int ; TIMER A2 (software int 14)
.lword dummy_int ; TIMER A3 (software int 15)
.lword dummy_int ; TIMER A4 (software int 16)
.lword dummy_int ; uart0 trance (software int17)
.lword dummy_int ; uart0 receive (software int18)
.lword dummy_int ; uart1 trance (software int19)
.lword dummy_int ; uart1 receive (software int 20)
.lword dummy_int ;TIMER B0 (software int 21)
.lword dummy_int ;TIMER B1 (software int 22)
.lword dummy_int ;TIMER B2 (software int 23)
.lword dummy_int ;TIMER B3 (software int 24)
.lword dummy_int ;TIMER B4 (software int 25)
.lword dummy_int ; INT5 (software int 26)
.lword dummy_int ; INT4 (software int 27)
.lword dummy_int ; INT3 (software int 28)
.lword dummy_int ; INT2 (software int 29)
.glb _int1
.lword _int1 ; INT1 (software int 30)
.glb _int0
.lword _int0 ; INT0 (software int 31)
.lword dummy_int ; TIMER B5 (software int 32)
      ...
```

图 3.10 在中断向量表中注册函数的实例

3.2 汇编程序宏

NC308 可以让您将一些汇编语言指令编写为 C 函数（这些函数称为汇编程序宏函数）。

汇编程序宏函数可以让您在 C 程序中直接编写汇编语言指令，这些指令是 NC308 在普通 C 编写中无法扩展的。此功能可以让您更简易地调谐程序。

本小节将描述如何指定和使用汇编程序宏函数。

3.2.1 可使用汇编程序宏函数指定的汇编语言指令

NC308 可以让您使用汇编程序宏函数指定 18 种汇编语言指令。

汇编程序宏函数名称与以大写字母显示的汇编语言指令相同。这些函数名称后面有“_b”、“_w”或“_l”标示运算的位长度。表 3.1 和 3.2 显示可使用汇编程序宏函数指定的汇编语言指令。

表 3.1 可使用汇编程序宏函数指定的汇编语言指令 (1)

汇编语言指令	汇编程序宏函数名称	描述	格式
DADD	dadd_b	返回 val1 加 val2 的十进制加法的结果。	char dadd_b(char val1,char val2)
	dadd_w		int dadd_w(int val1,int val2)
DADC	dadc_b	返回进位 val1 加 val2 的十进制加法的结果。	char dadc_b(char val1,char val2)
	dadc_w		int dadc_w(int val1,int val2)
DSUB	dsub_b	返回 val1 减 val2 的十进制减法的结果。	char dsub_b(char val1, char val2);
	dsub_w		int dsub_w(int val1, int val2);
DSBB	dsbb_b	返回借入 val1 减 val2 的十进制减法的结果。	char dsbb_b(char val1, char val2);
	dsbb_w		int dsbb_w(int val1, int val2);
RMPA	rmpa_b	返回积和运算的结果，使用 init 为初始值，以 counter 为次数，以及 p1 和 P2 作为存储乘数的起始地址。	long rmpa_b(long init, int count, char *p1, char *p2);
	rmpa_w		long rmpa_w(long init, int count,int *p1, int *p2);
MAX	max_b	返回值 val1 或 val2，视何者比较大。	char max_b(char val1, char val2);
	max_w		int max_w(intr val1, int val2);
MIN	min_b	返回值 val1 或 val2，视何者比较小。	char min_b(char val1, char val2);
	min_w		int min_w(intr val1, int val2);

表 3.2 可使用汇编程序宏函数指定的汇编语言指令 (2)

汇编语言指令	汇编程序宏函数名称	描述	格式
SMOVB	smovb_b	以 count 为次数，后退方向将字符串从地址 p1 转移到地址 p2。	void smovb_b(char *p1, char *p2, unsigned int count);
	smovb_w		void smovb_w(int *p1, int *p2, unsigned int count);
SMOVF	smovf_b	以 count 为次数，前进方向将字符串从地址 p1 转移到地址 p2。	void smovf_b(char *p1, char *p2, unsigned int count);
	smovf_w		void smovf_w(int *p1, int *p2, unsigned int count);
SMOVU	smovu_b	以 count 为次数，前进方向将字符串从地址 p1 转移到地址 p2 直到检测到零。	void smovu_b(char *p1, char *p2);
	smovu_w		void smovu_w(int *p1, int *p2);
SIN	sin_b	以 count 为次数，前进方向将字符串从固定地址 p1 转移到地址 p2。	void sin_b(char *p1, char *p2, unsigned int count);
	sin_w		void sin_w(int *p1, int *p2, unsigned int count);
SOUT	sout_b	以 count 为次数，后退方向将字符串从地址 p1 转移到地址 p2。	void sout_b(char *p1, char *p2, unsigned int count);
	sout_w		void sout_w(int *p1, int *p2, unsigned int count);
SSTR	sstr_b	存储字符串，使用要存储的数据 val、地址 p 和以 count 为次数转移数据。	void sstr_b(char val, char *p,unsigned int count);
	sstr_w		void sstr_w(int val, int *p,unsigned int count);
ROLC	rolc_b	返回向左循环 1 位后的 val 值，包括进位。	unsigned char rolc_b(unsigned char val);
	rolc_w		unsigned int rolc_w(unsigned int val);
RORC	rorc_b	返回向右循环 1 位后的 val 值，包括进位。	unsigned char rorc_b(unsigned char val);
	rorc_w		unsigned int rrlc_w(unsigned int val);
ROT	rot_b	以 count 为次数，返回循环后的 val 值。	unsigned char rot_b(signed char count,unsigned char val);
	rot_w		unsigned int rot_w(signed char count,unsigned int val);
SHA	sha_b	以 count 为次数，返回算术移位后的 val 值。	unsigned char sha_b(signed char count, unsigned char val);
	sha_w		unsigned int sha_w(signed char count, unsigned int val);

汇编语言指令	汇编程序宏函数名称	描述	格式
	sha_l		unsigned long sha_l(signed char count, unsigned longval);
SHL	shl_b	以 count 为次数，返回逻辑移位后的 val 值。	unsigned char shl_b(signed char count, unsigned char val);
	shl_w		unsigned int shl_w(signed char count, unsigned int val);
	shl_l		unsigned long shl_l(signed char count, unsigned longval);

3.2.2 使用汇编程序宏函数“dadd_b”的十进制加法

当您调用和使用 NC308 汇编程序宏函数时，您必须包含可定义汇编程序宏函数的“asmmacro.h”文件。

图 3.11 显示使用汇编程序宏函数“dadd_b”的十进制加法的实例。

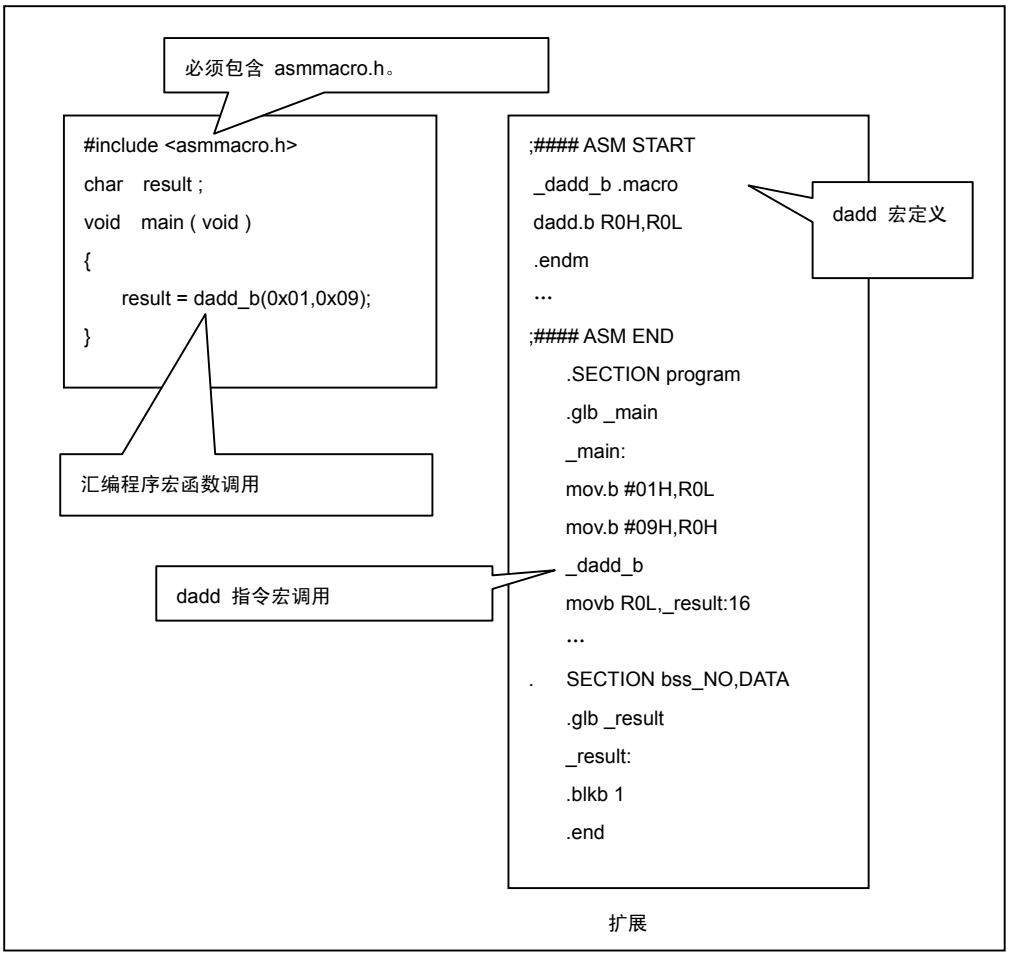


图 3.11 使用汇编程序宏函数“dadd_b”的十进制加法

3.2.3 使用汇编程序宏函数“smovf_b”转移字符串

图 3.12 显示使用汇编程序宏函数“smovf_b”转移字符串的实例。

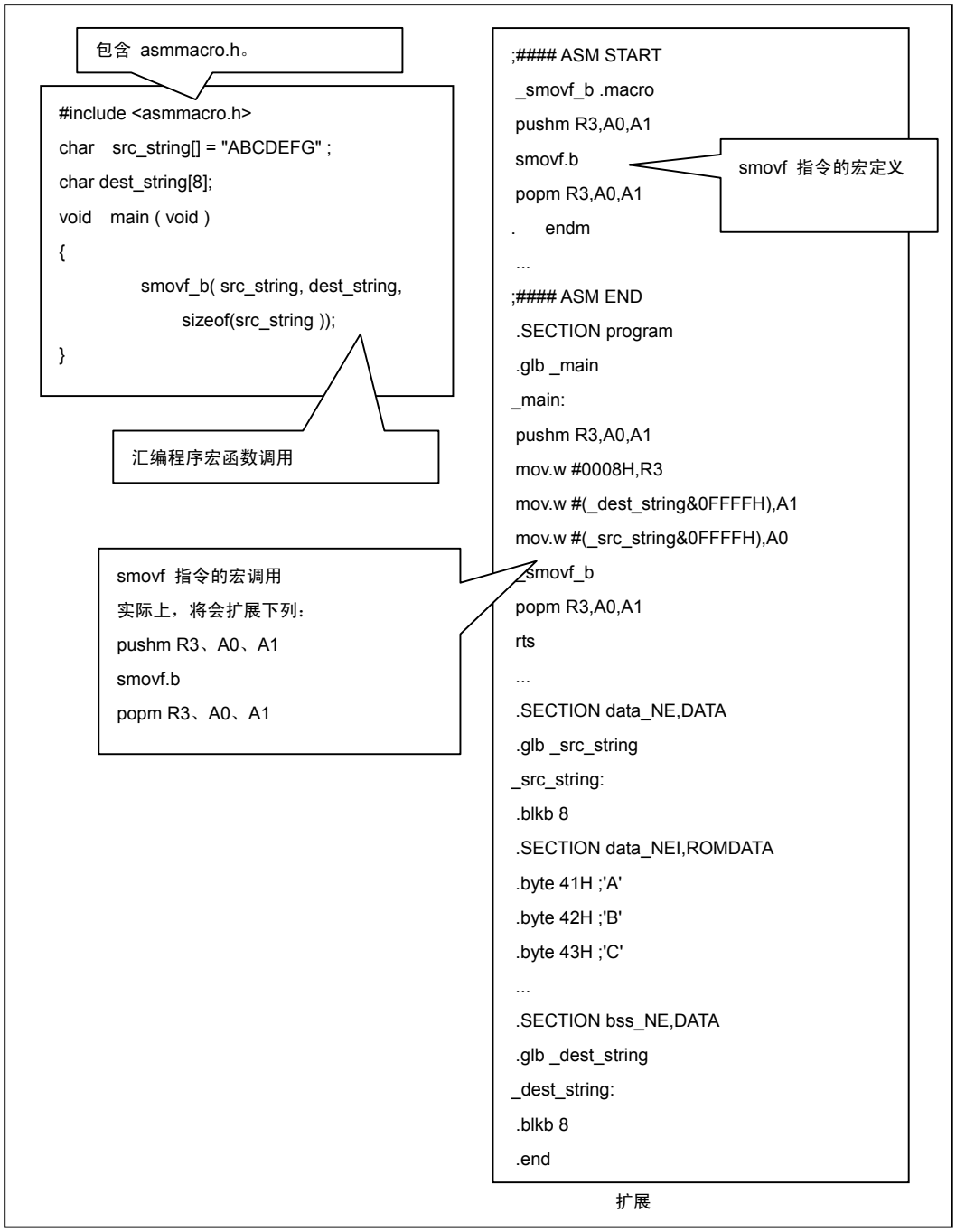


图 3.12 使用 smovf_b 汇编程序宏函数转移字符串

3.2.4 使用汇编程序宏函数“rmpa_w”的积和运算

图 3.12 显示使用汇编程序宏函数“rmpa_w”的积和运算的实例。

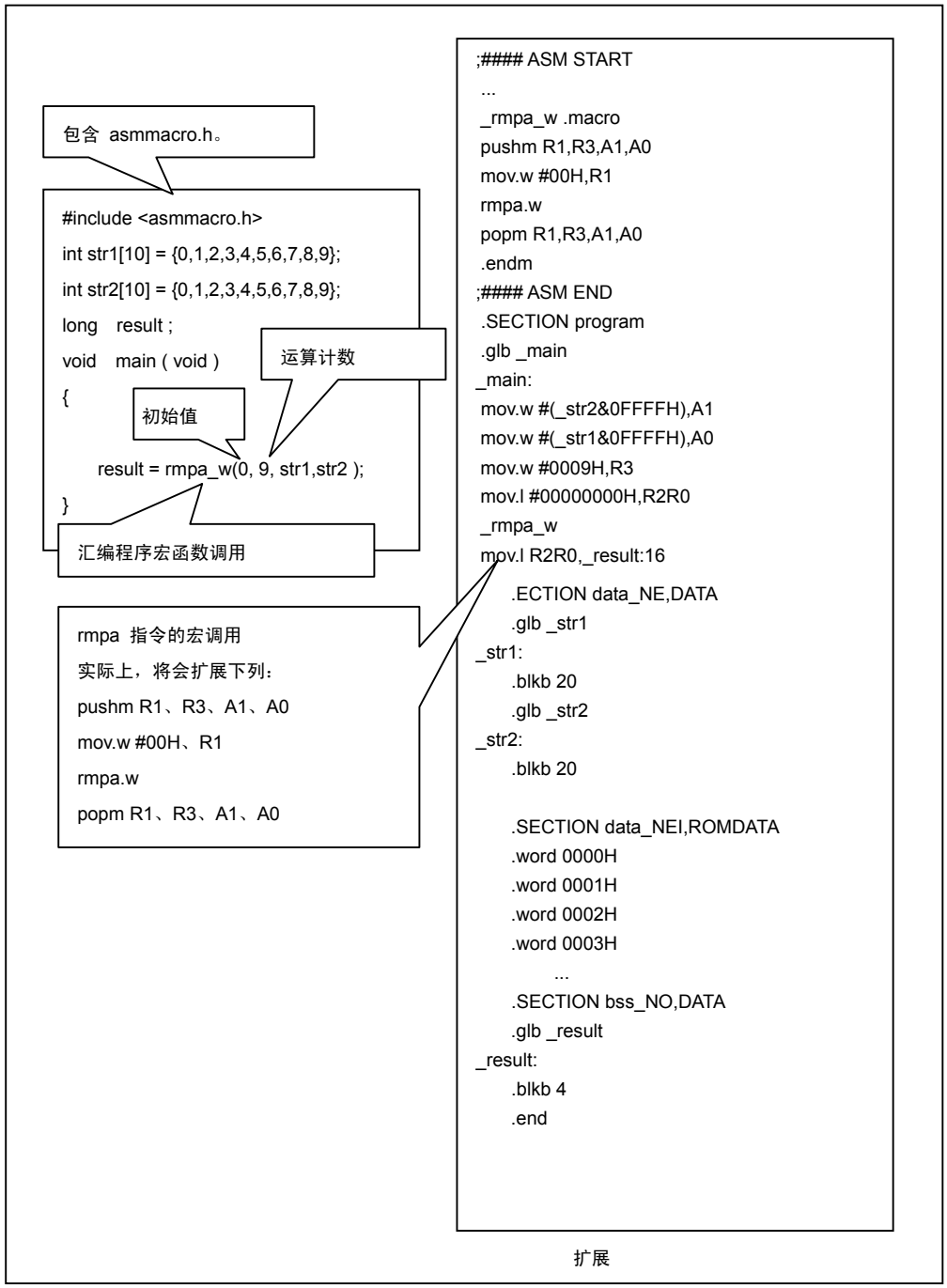


图 3.13 使用汇编程序宏函数“rmpa_w”的积和运算

3.2.5 #pragma __ASMMACRO

对于 NC308，您可以使用 #pragma__ASMMACRO 将任何汇编程序指令字符串设成汇编程序宏函数。

函数

声明由汇编程序宏定义的函数。

语法

#pragma __ASMMACRO *函数名称* (*寄存器名称*,...)

规则

1. 您必须在声明 #pragma __ASMMACRO 之前输入原型声明。您必须将汇编程序宏函数声明为静态。
2. 您不能声明不具有任何参数的函数。参数会通过寄存器传递。指定符合参数类型的寄存器（根据 #pragma PARAMETER）。
3. 当您定义汇编程序宏时，宏名称必须是所声明的函数名称并以下划线 (_) 为前缀。
4. 返回值将根据函数调用规则设定，如下显示。您不能将复合类型（结构和联合类型）声明为返回值。

   char 和 _Bool 类型： R0L　　浮点类型： R2R0

   int 和短类型： R0　　双精度类型： R3R2R1R0

   长类型： R2R0　　加长类型： R3R1R2R0
5. 将会在汇编程序宏中更改内容的寄存器必须在开始汇编程序宏时保存。寄存器必须紧挨在返回之前恢复。（您不需要保存或恢复存储返回值的寄存器）。

实例

```
static long mul( int, int ); /* 请确定声明“静态”（static）。*/
#pragma __ASMMACRO mul( R0, R2 )
#pragma ASM
_mul .macro
mul.w R2,R0 ; 返回值会在 R2R0 中设定。
.endm
#pragma ENDASM
long l;
void test_func( void )
{
    l=mul(2,3);
}
```

图 3.14 使用汇编程序宏的实例

3.3 缩减 ROM 区域的 Pragma 函数和选项

表 3.3 缩减 ROM 区域的 Pragma 函数和选项

小节	名称	描述
3.3.1	#pragma SBDATA	使用 SB 相对寻址存取变量。
3.3.2	#pragma SB16DATA	使用 2 字节 SB 相对寻址存取变量。
3.3.3	#pragma BIT	在 16 位绝对寻址模式中生成一位操作指令。
3.3.4	#pragma SPECIAL	将跳转子例程指令从四字节压缩为二字节。
3.3.5	-fjsrw	使用 JSR.W 指令调用函数。
3.3.6	-OR	执行 ROM 效率的最大优化。
3.3.7	-fno_align	不对齐函数的起始地址。
3.3.8	-Wno_used_function	输出未使用函数的警告。

3.3.1 #pragma SBDATA

此选项使用 SB 寄存器在相对寻址模式中存取指定的变量。将频繁存取的变量的存取模式改为 SB 相对寻址，您可以提高代码的效率。指定用 SB 相对寻址来存取的变量将分配到 SBDATA 属性段，并用相对于 SBDATA 属性段起始地址（存于 SB 寄存器中）的偏移量来参考。这将使扩展的代码比加载地址作为参考的代码更简练，同时也帮助提高 ROM 效率。其格式如下：

#pragma SBDATA *变量名称*

基于 SB 寄存器寻址的最大存取区域是自 SB 寄存器起 256 字节以内。地址规格仅需要一个字节。为常用变量指定此类型的寻址将缩减 ROM 区域。

使用 SBDATA 之前	使用 SBDATA
int a; a=1;	#pragma SBDATA a int a; a=1;
.GLB __SB__ .SB __SB__ .FB 0 ... mov.w #0001H,_a	.GLB __SB__ .SB __SB__ .FB 0 .SBSYM_a ... mov.w #0001H,_a

图 3.15 使用寻址模式和 SBDATA 的实例

3.3.2 #pragma SB16DATA

SBDATA 指定使用从 SB 寄存器的一字节偏移的寻址。SB16DATA 指定使用二字节偏移的寻址。对于此类型的寻址，最大存取区域是 SB 寄存器的 64 千字节。地址规格仅需要二个字节。为常用变量指定 #pragma SB16DATA 将缩减 ROM 区域。其格式如下：

#pragma SB16DATA *变量名称*

使用 SB16DATA 之前	使用 SB16DATA
int a; a=1;	#pragma SB16DATA a int a; a=1;
.GLB __SB__ .SB__SB__ .FB 0 ... mov.w #0001H,_a	.GLB __SB__ .SB__SB__ .FB 0 .SBSYM16 _a ... mov.w #0001H,_a

图 3.16 使用寻址模式和 SB16DATA 的实例

3.3.3 #pragma BIT

此选项声明了所指定的外部变量位于可在 16 位绝对寻址模式下使用 1 位操作指令的范围内（00000H 到 01FFFH）。这将可以让您在 16 位绝对寻址模式中生成一位操作指令。（此函数仅可用于 NC30WA）。

其格式如下：

#pragma BIT *变量名称*

使用 #pragma BIT 之前	使用 #pragma BIT
int sym sym=0x01|sym	#pragma BIT sym int sym sym=0x01|sym
or.w #01H,_sym	bset 0,_sym

图 3.17 使用 #pragma BIT 的位运算

3.3.4 #pragma SPECIAL

special 页将跳转子例程指令从四字节的 JSR.A _func 压缩为二字节的 JSRS 编号。这将缩减 ROM 区域。

其格式如下：

#pragma SPECIALΔ[/ C]Δ*调用编号*Δ*函数名称*()

#pragma SPECIALΔ[/ C]Δ*函数名称*(vect= *调用编号*)

在 #pragma SPECIAL 中声明的函数将映像至 special 页向量表中设定的地址加上 0FF0000H 所得到的地址，因此会受到 special 页子例程调用的影响。您可以在声明中指定下列切换：

[/C]

此切换将生成在调用所声明的函数时用于保存需要的寄存器的代码。

您可以在声明中指定一个调用编号。

为编译源文件指定一个调用编号和编译选项 -fmake_special_table (-fMST)。编译程序将会自动生成一个 special 页向量表。

使用 #pragma SPECIAL 声明的函数将会映像至 program_S 段。

您必须将 program_S 段映像至 0FF0000H 和 0FFFFFFH 之间。

您可以指定从 18 至 255 的调用编号，仅限十进制。作为标签，"_SPECIAL_*calling-number*:"（_SPECIAL_*调用编号*：） 将输出至使用 #pragma SPECIAL 声明的函数的起始地址。在启动文件的 special 页子例程表中设定此标签。

如果已指定 -fmake_special_table (-fMST) 选项，将不需要进行上述设定。

如果您为同一函数指定不同的调用编号，最后声明的调用编号将会被使用。

实例：

```
#pragma SPECIAL func(vect=20)

#pragma SPECIAL func(vect=30)// 调用编号 30 将被使用
```

如果要在某个文件中定义函数，而在另一个文件中定义调用函数，您必须同时在两个文件中指定此声明。

使用 #pragma SPECIAL 之前	使用 #pragma SPECIAL

void func(unsigned int, unsigned int);
void main()
{
int i, j;
i = 0x7FFD;
j = 0x007F;
func(i, j);
}
	#pragma SPECIAL 20 func()
	void func(unsigned int, unsigned int);
	void main()
	{
	int i, j;
	i = 0x7FFD;
	j = 0x007F;
	func(i, j);
	}

push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsr $func	push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsrs #20

图 3.18 使用 #pragma SPECIAL 声明的实例

3.3.5 -fjsrw

编译程序使用 JSR.A 指令调用已在文件以外定义的函数。但是，如果程序不是太大，多数函数都可以通过 JSR.W 指令调用。

在此情形下，您可以通过以下方式缩减 ROM 中的代码数量：

使用 -fJSRW 选项编译源文件。然后，使用“#pragma JSRA *函数名称*”声明在连接期间导致错误的函数。

注意： 当您使用 -OGJ 选项时，最适当的 jmp 指令是在连接期间选取。

C 源	不使用 -fjsrw	使用 -fjsrw

/*file 1*/
void f() {}
 	
/* file 2*/
int main()
{
f();
}
	.GLB __SB__
	.SB __SB__
	.FB 0
	...
	jsr _f
		.OPTJ JSRW
		.GLB __SB__
		.SB __SB__
		.FB 0
		...
		jsr _f

图 3.19 使用 –fjsrw 的实例

3.3.6 -OR

此选项执行最大优化以将 ROM 中的代码数量减至最少，虽然可能要牺牲速度。此选项可使用 -g 和 -O 选项指定。通过此选项的优化可部分修改源行信息。这可能会导致程序在调试期间有不同的行为。如果您不要修改源行信息，请使用 -Ono_break_source_debug (-ONBSD) 选项来制止优化。

图 3.20 显示使用 -OR 选项进行优化的实例。公用表达式将会统一以缩减 ROM 中的代码数量。

C 源	不使用优化	使用优化

If(b==1){
sub();
return a;
} else if(b==2){
sub()
return a;
} else {
return 0;
}
	;## # C_SRC : if(b==1)
	cmp.w #0001H,_b:16
	jne L1
	;## # C_SRC : sub();
	jsr _sub
	;## # C_SRC : return a;
	mov.w _a:16,R0
	rts
	;## # C_SRC : else if(b==2)
	L1:
	cmp.w #0002H,_b:16
	jne L11
	;## # C_SRC : sub();
	jsr _sub
	;## # C_SRC : return a;
	mov.w _a:16,R0
	rts
	;## # C_SRC : else
	L11:
	;## # C_SRC : return 0;
	mov.w #0000H,R0
	rts
		;## # C_SRC : if(b==1)
		mov.w _b:16,R0
		cmp.w #0001H,R0
		jne L5
		;## # C_SRC : sub();
		L31:
		jsr _sub
		;## # C_SRC : return a;
		mov.w _a:16,R0
		rts
		;## # C_SRC : else if(b==2)
		L5:
		cmp.w #0002H,R0
		jeq L31
		;## # C_SRC : return 0;
		mov.w #0000H,R0
		rts

图 3.20 使用 –OR: 统一公用表达式进行优化的实例

3.3.7 -fno_align

此选项不会对齐函数的起始地址。此选项防止 .align 插入函数的起始，由此缩减 ROM 区域。

C 源	不使用 -fno_align	使用 -fno_align

int f()
{
return 0;
}
	;## # FUNCTION f
	;## # ARG Size(0)Auto Size(0) Context Size(4)
	.SECTION program,CODE,ALIGN
	._inspect 'U', 2, "program", "program", 0
	._file 'C:/Hew3/fno_align/fno_align/fno_align.c'
	._type 256,'x',16,0
	._func 'f','G',0,256,_f,0
	._inspect 'F', 's', "f", "_f", 'G', 4
	.align
	;## # C_SRC : {
	.glb _f
	_f:
		;## # FUNCTION f
		;## # ARG Size(0)Auto Size(0) Context Size(4)
		.SECTION program,CODE
		._inspect 'U', 2, "program", "program", 0
		._file 'C:/Hew3/fno_align/fno_align/fno_align.c'
		._type 256,'x',16,0
		._func 'f','G',0,256,_f,0
		._inspect 'F', 's', "f", "_f", 'G', 4
		;## # C_SRC : {
		.glb _f
		_f:

图 3.21 使用 –fno_align 的实例

3.3.8 -Wno_used_function

此函数显示连接期间未使用的全局函数。删除未使用的函数将缩减 ROM 区域。

C 源	警告信息

void f()
{
}
int main()
{
}
	C:¥Hew3¥test2¥test2¥test2.c(20) : Warning (ln308): Global function 'f'is never used

图 3.22 使用 –Wno_used_function 的实例

3.4 加快微处理器处理速度的 Pragma 函数和选项

表 3.4 加快微处理器处理速度的 Pragma 函数和选项

小节	名称	描述
3.4.1	#pragma STRUCT	执行结构对齐以提高存取速度。
3.4.2	-Ostack_frame_align	执行堆栈帧对齐以提高存取速度。
3.4.3	-OS	执行速度的最大优化。
3.4.4	-Oloop_unroll[=*count*]	解开循环。
3.4.5	-Ofloat_to_inline	对浮点运算执行运行时例程的内联扩展。

3.4.1 #pragma STRUCT

此选项执行结构对齐以提高存取速度。

格式 1.#pragma STRUCT *结构标签名称* unpack

2.#pragma STRUCT *结构标签名称* arrange

在编译程序中，结构都是包型的。例如，图 3.23 中的结构成员都在没有任何填充的情形下按照它们声明的顺序编排。

图 3.23 映像结构成员的实例 (1)

编译程序的扩展函数可以让您控制结构成员的映像。图 3.24 显示使用 #pragma STRUCTunpack 在图 3.23 中包装结构时禁止映像结构成员的实例。

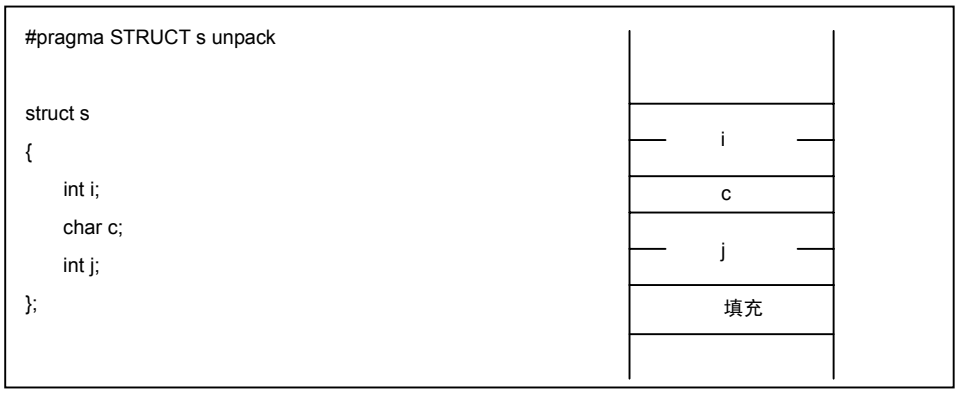


图 3.24 映像结构成员的实例 (2)

如图 3.24 所示，如果结构成员的总大小是奇数字节，#pragma STRUCTunpack 将会在最后一个成员后面添加 1 字节作为包装。因此，如果您使用 #pragma STRUCTunpack 来禁止填充，所有结构的大小将会是偶数字节。

编译程序的扩展函数可以让您先映像所有的奇数大小的结构成员，然后是偶数大小的成员。图 3.25 显示使用 #pragma STRUCT arrange 在图 3.23 中编排结构时的映像实例。

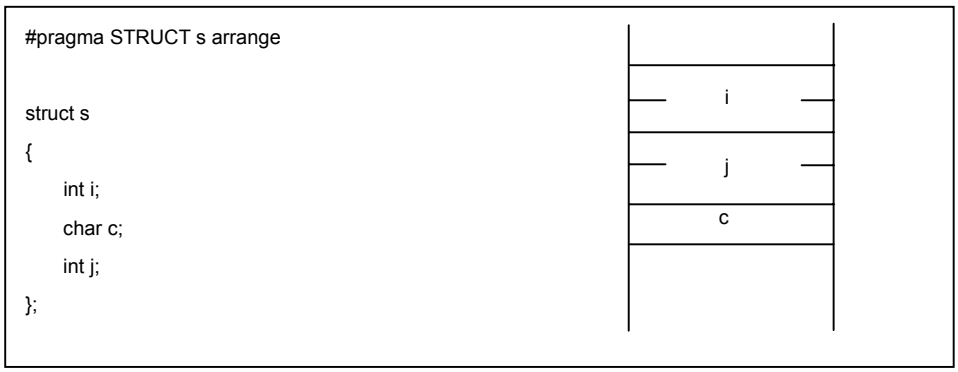


图 3.25 映像结构成员的实例 (3)

一起使用 #pragma STRUCT unpack 和 #pragma STRUCT arrange 对齐偶数大小的成员。

默认	unpack

struct A
{
int a;
char b;
int c;
};
 	
f()
{
struct A a,b;
a.a=1; a.b=2; a.c=3;
b.a=4; b.b=5; b.c=6;
}
	#pragma STRUCT A unpack
	struct A
	{
	int a;
	char b;
	int c;
	};
	
	f()
	{
	struct A a,b;
	a.a=1; a.b=2; a.c=3;
	b.a=4; b.b=5; b.c=6;
	}

;## # C_SRC : a.a=1; mov.w #0001H,-10[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-7[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-3[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-2[FB] ; b	;## # C_SRC : a.a=1; mov.w #0001H,-12[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-10[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-9[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-4[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b

图 3.26 使用 #pragma STRUCT 的实例 (1)

arrange	arrange+unpack

#pragma STRUCT A arrange
struct A
{
int a;
char b;
int c;
};
 	
f()
{
struct A a,b;
a.a=1; a.b=2; a.c=3;
b.a=4; b.b=5; b.c=6;
}
	#pragma STRUCT A arrange
	#pragma STRUCT A unpack
	struct A
	{
	int a;
	char b;
	int c;
	};
	
	f()
	{
	struct A a,b;
	a.a=1; a.b=2; a.c=3;
	b.a=4; b.b=5; b.c=6;
	}

;## # C_SRC : a.a=1; mov.w #0001H,-10[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-6[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-8[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-1[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b	;## # C_SRC : a.a=1; mov.w #0001H,-12[FB] ; a ;## # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-10[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-2[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-4[FB] ; b

图 3.27 使用 #pragma STRUCT 的实例 (2)

3.4.2 -Ostack_frame_align

如果偶数大小的自动变量映像至奇数地址，存储器的存取将需要比当变量映像至偶数地址时多一个周期。此选项将对齐偶数大小的自动变量至偶数地址的映像。这将允许快速存储器存取。（此选项仅可用于 NC30WA）。

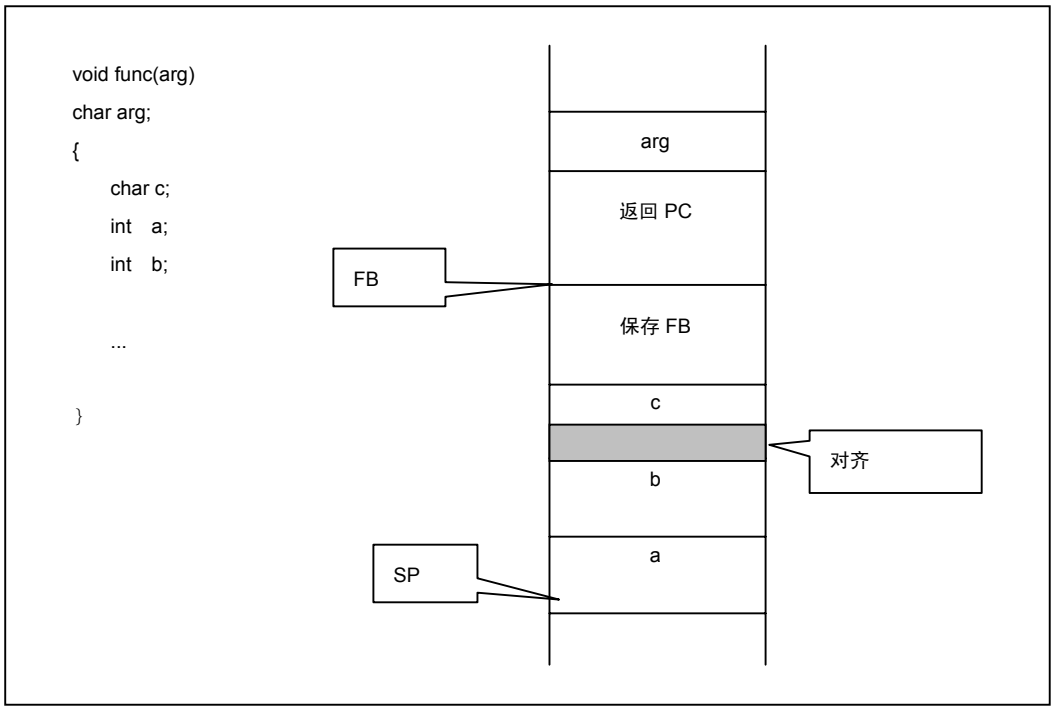


图 3.28 通过 –Ostack_frame_align 对齐的实例

C 源	不使用 -Ostack_frame_align	使用 -Ostack_frame_align

void f()
{
int a;
int b;
a=1;
b=2;
...
}
	;## # C_SRC : a=1;
	mov.w #0001H,-2[FB] ; a
	;## # C_SRC : b=2;
	mov.w #0002H,-4[FB] ; b
		;## # C_SRC : a=1;
		mov.w #0001H,-5[FB] ; a
		;## # C_SRC : b=2;
		mov.w #0002H,-3[FB] ; b

图 3.29 –Ostack_frame_align 的实例

3.4.3 -OS

此选项执行最大优化以获得可能的最快速度，虽然 ROM 中的代码数量可能会增加。此选项可以和 -g 及 -O 选项一起指定。

C 源	使用优化	不使用优化
for(i=0;i<100;i++) a[i]=l*4;	;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: ;## # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ;## # C_SRC : a[i]=l*4; mov.w _l:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 jmp L1 L5:	mov.w #0000H,_i:16 mov.w _l:16,R0 shl.w #2,R0 L3: ._line 26 ;## # C_SRC : a[i]=l*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L3

图 3.30 通过 –OS 优化的实例

3.4.4 -Oloop_unroll[=*count*]

此选项可在不循环循环语句的情形下依照循环计数解开代码。您可以省略循环计数。如果您省略循环计数，此选项将应用至多达五个循环语句。此选项可删除转移和 counter 的计算以提高执行速度。但是，这将降低 ROM 效率。

C 源	不使用优化	使用优化

for(i=0;i<3;i++)
{
a[i]=i;
}
	;## # C_SRC : for(i=0;i<3;i++)
	mov.w #0000H,-2[FB] ; i
	L1:
	;## # C_SRC : for(i=0;i<3;i++)
	cmp.w #0003H,-2[FB] ; i
	jge L5
	;## # C_SRC : a[i]=i;
	indexwd.w -2[FB] ; i
	mov.w -2[FB],_a:16 ; i
	add.w #0001H,-2[FB] ; i
	jmp L1
	L5:
		mov.w #0000H,-2[FB] ; i
		mov.w #0002H,A0
		mul.w #0000H,A0
		mov.w A0,A0
		mov.w #0000H,_a:16[A0]
		mov.w #0001H,-2[FB] ; i
		mov.w #0002H,A0
		mul.w #0001H,A0
		mov.w A0,A0
		mov.w #0001H,_a:16[A0]
		mov.w #0002H,-2[FB] ; i
		;## # C_SRC : a[i]=i;
		mov.w #0002H,R0
		indexwd.w R0
		mov.w #0002H,_a:16
		mov.w #0003H,-2[FB] ; i

图 3.31 解开循环的实例

3.4.5 -Ofloat_to_inline

此选项执行浮点运行库的内联扩展以加快浮点运算的处理（仅用于比较和乘法）。此选项仅可用于 M32C/80 系列。使用此选项时，您也必须指定编译选项 “-M82”。

源代码	不使用 -Ofloat_to_inline	使用 -Ofloat_to_inline
float f; long l; l=f;	push.l _f:16 jsr.a __f4toi4 add.l #04H,SP mov.l R2R0,_l:16	;## # C_SRC : l=f; mov.w _f+2:16,R0 mov.w _f:16,R2 btst 7,R0H scc R3 mov.w R0,R1 shl.w #-07H,R1 and.w #0ffH,R1 xchg.w R0,R2 and.w #07fH,R2 mov.w R1,R1 jne ?+ mov.l R2R0,R2R0 jeq M1 ?: btst 7,R1L jc ?+ cmp.w #07fH,R1 jne M1 ?: add.w #062H,R1 tst.w #0ff00H,R1 jeq M2 mov.w R3,R3 jeq ?+ mov.l #80000000H,R2R0 jmp M3 ?: mov.l #7fffffffH,R2R0 jmp M3 M2: dec.w R1 or.w#0080H,R2 shlnc.l #8H,R2R0 ?: inc.w R1 cmp.w #0100H,R1 jeq M4 shlnc.l #-1,R2R0 jmp ?- M4: cmp.w #1,R3 jne M3 not.w R0 not.w R2 add.l #01H,R2R0 jmp M3 M1: mov.l #0,R2R0 M3:

图 3.32 使用 –Ofloat_to_inline 优化的实例

3.4.6 -Ostatic_to_inline

此选项将静态函数（声明为静态的函数）当作是内联函数（声明为内联的函数），并生成内联扩展汇编代码。

满足下列条件时，编译程序会将静态函数当作是内联函数并生成内联扩展汇编代码：

(1) 此选项可应用至函数调用之前指定实体的静态函数。

（函数调用和该函数的实体必须包含在同个源文件中。）

（如果您指定 -Oforward_function_to_inline 选项则可忽略此条件。）

(2) 目标静态函数的地址采集会在程序中省略。

(3) 未执行目标静态函数的递归调用。

(4) 未在编译程序的汇编代码输出中执行帧的架设（自动变量的预留，等等）。（是否执行帧架设将取决于目标函数的描述内容以及其他优化选项。）

（如果您指定 -Oforward_function_to_inline 选项则可忽略此条件。）

C 源	不使用 -Ostatic_to_inline	使用 -Ostatic_to_inline

static int f(int a,int b)
{
return a+b;
}
int c;
void main()
{
c=f(2,3);
}
	push.w #0003H
	mov.w #0002H,R0
	jsr $f
	add.l #02H,SP
	mov.w R0,_c:16
		mov.w #0005H,_c:16

图 3.33 使用 –Ostatic_to_inline 的实例

3.5 缩减 ROM 区域和加快微处理器处理速度的 Pragma 函数和选项

表 3.5 缩减 ROM 区域和加快微处理器处理速度的 Pragma 函数和选项

小节	名称	描述
3.5.1	-O[1-5]	执行优化。
3.5.2	-Osp_adjust	每次都执行修正堆栈指针的优化。
3.5.3	-fuse_DIV	使用 div 指令执行除法。
3.5.4	-Wno_used_argument	输出未使用参数的警告。
3.5.5	-fsmall_array	以 16 位计算数组的索引。
3.5.6	-fdouble_32	将双精度数据当作浮点数据处理。

3.5.1 -O[1-5]

此选项执行最大优化以获得更快的处理和缩减 ROM 中的代码数量。此选项可使用 -g 选项指定。如果您不指定任何数字（级别），系统将假设为 -O3。

-O1： 与 -O3、-Ono_bit、-Ono_break_source_debug、-Ono_float_const_fold 和 -Ono_stdlib 有效时相同。

-O2： 与 -O1 相同。

-O3： 执行最大优化以获得更快的处理和缩减 ROM 中的代码数量。

-O4： 确认 -O3 和 -Oconst。

-O5： 执行可提高公用子表达式的最大优化（同时指定 -OR 选项时）、转移字符串，以及比较（同时指定 -OS 选项时）。

但是，在满足下列条件时，可能无法输出正常代码：

- 不同变量同时指向同个存储器位置。
- 在同个函数中使用这些变量。

```
实例：
int a=3;
int *p=&a;

test()
{
    int b;
    *p=9;
    a=10;
    b=*p; //优化将会以 “9” 替换 “p”。
    print(“b=%d(expect b=10)¥n”,b);
}

Result)
b=9(expect b=10)
```

图 3.34 若指定 "-O5" 将导致不正确运算的代码实例

注意：
您不能使用 BTSTC 或 BTSTS 位操作指令写入或读取 SFR 区域内之寄存器的数据。如果您使用优化选项 (-O5)，编译程序可能会生成汇编程序代码的位操作指令（BTSTC 和 BTSTS）。如果您在下列编译指定中使用 -O5 优化选项，中断请求位将无法正确确定，从而导致未预期的运算。

```
[实例：  不可以和优化选项一起使用的 C 源]
  #pragma ADDRESS TA0IC 006Ch /* M16C/80 定时器 A0 中断控制寄存器
*/
   struct {
        char ILVL : 3;
        char IR : 1; /* 一个中断请求位 */
        char dmy : 4;
      } TA0IC;
      void wait_until_IR_is_ON(void)
      {
        while (TA0IC.IR == 0) /* 等待 1 */
        {
        ;
        }
        TA0IC.IR = 0; /* 将 1 返回 0  */
      }
```

图 3.35 无法使用优化选项时的实例

如果您发现位操作指令（BTSTC 和 BTSTS）输出至 SFR 区域，执行编译之前请先采取下列措施。确保所生成的代码在其设定中没有任何问题：

- 使用 -O5 以外的优化选项。
- 使用 ASM 函数直接在程序中指定指令。
- 添加 -Ono_asmopt（或 -ONA）选项。

3.5.2 -Osp_adjust

此选项通过结合函数调用之后的堆栈修正代码执行优化。通常，在每次调用函数时编译程序会修正堆栈指针以释放函数的参数区域。指定此选项时，堆栈指针将会全体修正，而不是为每个函数调用修正。-Osp_adjust 选项可缩减在 ROM 中使用的代码数量以及加快微处理器处理速度。但是，所使用的堆栈数量可能会增加。

C 源	不使用 -Osp_adjust	使用 -Osp_adjust

main()
{
f(1.1);
g(1.1);
}
	_main:
	;## # C_SRC : f(1.1);
	push.l #3ff19999H
	push.l #9999999aH
	jsr _f
	add.l #08H,SP
	;## # C_SRC : g(1.1);
	push.l #3ff19999H
	push.l #9999999aH
	jsr _g
	add.l #08H,SP
	;## # C_SRC : }
	rts
		_main:
		;## # C_SRC : f(1.1);
		push.l #3ff19999H
		push.l #9999999aH
		jsr _f
		;## # C_SRC : g(1.1);
		push.l #3ff19999H
		push.l #9999999aH
		jsr _g
		add.l #010H,SP
		;## # C_SRC : }
		rts

图 3.36 使用 -Osp_adjust 的实例

3.5.3 -fuse_DIV

此选项更改生成的除法运算代码。

编译程序生成用于下列除法的 div.w (divu.w) 和 div.b (divu.b) 微型计算机指令：

- 被除数是一个 4 字节值，除数是一个 2 字节值，以及结果是一个 2 字节值。

- 被除数是一个 2 字节值，除数是一个 1 字节值，以及结果是一个 1 字节值。

如果在指定此选项时除法结果溢出，编译程序的运算可能会和 ANSI 中所规定的不一样。如果除法结果溢出，对于 M16C 的 div 指令，结果将会在发生溢出时无法预期。

因此，当 NC308 以默认设定编译程序时，它会调用运行库来修正此问题，即使是在被除数是一个 4 字节值，除数是一个 2 字节值，以及结果是一个 2 字节值的情形下。

源程序	默认	使用 -fuse_DIV 时
int k,j; long l; k=l/j;	mov.w _j:16,R1 exts.w R1 push.l R3R1 mov.l _l:16,R2R0 glb __i4div jsr.a __i4div add.l #4H,SP mov.w R0,_k:16 输出代码，考虑溢出。	mov.l _l:16,R2R0 div.w _j:16 mov.w R0,_k:16 使用 div 指令。

图 3.37 使用 -fuse_DIV 的实例

3.5.4 -Wno_used_argument

如果已定义具有参数的函数，此选项将输出未使用参数的警告。通过根据警告来修正源程序，您可以节省存储器空间和加快微处理器处理速度。

C 源	警告信息

int f(int a,int b,int c)
{
return a+c;
}
	C:¥Hew3¥test1¥test1¥test1.c(68) : [Warning(ccom)] function "f()" has no-used argument(b).

图 3.38 执行 -Wno_used_argument 的实例

3.5.5 -fsmall_array

在参考编译期间总大小未知的远型数组时，此选项将以 16 位计算索引，并假设该数组的总大小在 64 千字节内。在参考未知大小之远型数组的元素时，在默认情形下，编译程序将以 32 位计算索引，从而可以处理 64 千字节或更长的数组。请参考下列实例：

extern int array[]:

int i = array[j];

在此情形下，由于数组的总大小对于编译程序是未知数，因此索引 “j” 将以 32 位计算。

指定此选项时，编译程序将假设数组的总大小是 64 千字节或更少并且以 16 位计算索引 “j”。这将提高处理速度和缩减代码的数量。

我们建议您在单个数组的大小不超过 64 千字节时使用此选项。

源程序	不使用 -fsmall_array	使用 -fsmall_array
extern far int a[]; int i,j; i=a[j];	mov.w -2[FB],R0 ; j exts.w R0 mov.l R2R0,A0 shl.l#1,A0 mov.w _a[A0],-2[FB] ; i	indexws.w -4[FB] ; j mov.w:g_a,-2[FB] ; i

图 3.39 使用 -fsmall_array 的实例

3.5.6 -fdouble_32

此选项使编译程序将双精度类型当作浮点类型处理。

注意：

1. **指定此选项时，您必须声明一个函数原型。如果没有原型声明，编译程序可能会生成无效代码。**
2. **指定此选项时，双精度类型的调试信息将当作浮点类型处理。因此，双精度数据将会在调试程序 PD308 和模拟程序 PD308SIM 的 C 监视窗口和全局窗口中显示为浮点类型。**

C 源	不使用 -fdouble_32	使用 -fdouble_32
float data; data=data+123.456;	push.l _data:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H .glb __f8add jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 .glb __f8tof4 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0,_data:16	push.l _data:16 push.l #42f6e979H .glb __f4add jsr.a __f4add add.l #08H,SP mov.l R2R0 _data:16

图 3.40 使用 -fdouble_32 选项的实例

3.6 其他 Pragma 函数和选项

3.6.1 其他 Pragma 函数

(1) 用于存储器映像的扩展函数

扩展函数	描述
#pragma ROM	将指定变量映像至 rom 段。 语法： #pragma ROMΔ *变量名称* 实例： #pragma ROM val 注意： 提供此功能的目的是维护与 NC77 和 NC79 的兼容性。该变量一般上必须位于使用 const 限定符的 rom 段中。
#pragma SECTION	更改由编译程序生成的段名称。 语法： #pragma SECTIONΔ*现有的段名称*Δ*新的段名称* 实例： #pragma SECTION bss nonval_data

(2) 用于和目标设备配合使用的扩展函数

扩展函数	描述
#pragma ADDRESS (#pragma EQU)	将变量分配到绝对地址。 语法： #pragma ADDRESSΔ*变量名称*Δ*绝对地址* 实例： #pragma ADDRESS port0 2H
#pragma BITADDRESS	将变量分配到指定绝对地址的比特位位置。 语法： #pragma BITADDRESSΔ*变量名称*Δ*比特位位置,绝对地址* 实例： #pragma BITADDRESS io 1,100H
#pragma DMAC	为外部变量分配 DMAC 寄存器。（仅限于 NC308WA） 语法： #pragma DMACΔ*变量名称*Δ*DMAC 寄存器名称* 实例： #pragma DMAC dma0 DMA0
#pragma INTCALL	声明要被软件中断（int 指令）调用的函数。 Switch [/c]（切换）将生成在调用函数时用于保存需要的寄存器的代码。 语法 1： #pragma INTCALLΔ[/C]Δ *INT-编号*Δ*汇编程序函数名称 (寄存器名称)* 实例 1： #pragma INTCALL 25 func(R0, R1) #pragma INTCALL /C 25 func(R0, R1) 语法 2： #pragma INTCALLΔ *INT-编号*Δ*C-函数名称* ()

扩展函数	描述
	实例 2： #pragma INTCALL 25 func() #pragma INTCALL /C 25 func() 注意： 输入此声明之前您必须先声明函数的原型。
#pragma INTERRUPT (#pragma INTF)	声明以 C 编写的中断处理函数。此声明将使编译程序在该函数的入口和出口点生成代码，它可执行中断处理函数的过程。 语法： #pragma INTERRUPTΔ[/B¦/E¦/F]Δ *中断处理函数名称* #pragma INTERRUPTΔ[/B¦/E¦/F]Δ *中断向量号*Δ *中断处理函数名称* #pragma INTERRUPTΔ[/B¦/E¦/F]Δ *中断处理函数名称*(vect= *中断向量号*) 实例： #pragma INTERRUPT int_func #pragma INTERRUPT /B int_func #pragma INTERRUPT 10 int_func #pragma INTERRUPT /E 10 int_func #pragma INTERRUPT int_func (vect=10) #pragma INTERRUPT /F int_func (vect=20) 注意： 您也可以使用 #pragma INTF 来维护与 C77 的兼容性。
#pragma PARAMETER	声明在调用汇编程序函数时让参数通过指定的寄存器传递。 Switch [/C]（切换）将生成在调用函数时用于保存需要的寄存器的代码。 语法： #pragma PARAMETERΔ[/C]Δ*函数名称*(*寄存器名称*) 实例： #pragma PARAMETER asm_func(R0, R1) #pragma PARAMETER /C asm_func(R0, R1) 注意： 输入此声明之前您必须先声明函数的原型。

(3) 用于 MR308 支持的扩展函数

扩展函数	描述
#pragma ALMHANDLER	声明 MR308 警报处理器的名称。 语法： #pragma ALMHANDLER∆*函数名称* 实例： #pragma ALMHANDLER alm_func
#pragma CYCHANDLER	声明 MR308 循环启动处理器的名称。 语法： #pragma CYCHANDLER∆*函数名称* 实例： #pragma CYCHANDLER cyc_func
#pragma INTHANDLER #pragma HANDLER	声明 MR308 中断处理器的名称。 语法 1： #pragma INTHANDLER∆[/E]∆*函数名称* 语法 2： #pragma HANDLER∆[/E]∆*函数名称* 实例： #pragma INTHANDLER int_func
#pragma TASK	声明 MR308 任务启动处理器的名称。 语法： #pragma TASK∆*任务启动函数名称* 实例： #pragma TASK task1

(4) 其他扩展函数

扩展函数	描述
#pragma ASM #pragma ENDASM	指定用汇编语言编写的区域。 语法： #pragma∆ASM #pragma∆ENDASM 实例： #pragma ASM mov.w R0,R1 add.w R1,02H #pragma ENDASM
#pragma JSRA	将 JSR.A 指令用作 JSR 指令来调用函数。 语法： #pragma JSRA ∆*函数名称* 实例： #pragma JSRA func
#pragma JSRW	将 JSR.W 指令用作 JSR 指令来调用函数。 语法： #pragma JSRW∆*函数名称* 实例： #pragma JSRW func

扩展函数	描述
#pragma PAGE	指定汇编程序列表文件中的新页点。 语法： #pragmaΔPAGE 实例： #pragma PAGE
#pragma __ASMMACRO	声明由汇编程序宏定义的函数。 语法： #pragma __ASMMACROΔ*函数名称* (*寄存器名称*) 实例： #pragma __ASMMACRO mul(R0,R2)

3.6.2 其他选项

(1) 用于控制编译驱动器的选项

选项	功能
-c	建立可再定位文件（扩展名 .r30），以及结束处理。
-D *标识符*	定义标识符。此选项具有 #define 的相同功能。
-I *目录名称*	指定包含将由 #include 预处理命令参考的文件之目录。您可以指定多达 16 个目录。
-E	仅处理预处理命令和输出结果至标准输出。
-P	仅启动预处理命令和建立文件（扩展名 .i）。
-S	建立汇编源文件（扩展名 .a30），以及结束处理。
-U *预定义的宏名称*	使预定义的宏成为未定义。
-silent	制止启动时的版权信息。
-dsource	生成汇编源文件（扩展名 .a30）并具备 C 源列表输出作为注解。（此文件即使在汇编后也不会被删除。）
-dsource_in_list	除了执行“-dsource”函数外，还生成汇编语言列表文件（扩展名 .lst）。

(2) 用于指定输出文件的选项

选项	功能
- o 文件名	指定由 ln308 生成的文件（绝对模块文件、映像文件，等等）的名称。您也可以指定包含目录名称的路径名称。请不要指定文件扩展名。
-dir 目录名称	指定由 ln308 生成的文件（绝对模块文件、映像文件，等等）的输出目标目录。

(3) 用于显示版本信息和命令行的选项

选项	功能
-v	显示已执行的命令程序的名称和命令行。
-V	显示编译程序的启动信息，然后结束处理（在没有编译任何项目的情形下）。

(4) 用于调试的选项

选项	功能
-g	将调试信息输出至汇编源文件（扩展名 .a30）。这将允许 C 语言层调试。
-genter	在函数调用期间永远输出一个回车指令。使用调试程序的堆栈跟踪功能时您必须指定此选项。
-gno_reg	制止输出寄存器变量的调试信息。

(5) 优化选项

选项	功能
-Oconst	通过以常数替代常数修改变量的参考来执行优化。
-Ono_bit	根据位操作的分组来制止优化。
-Ono_break_source_debug	制止可影响源行信息的优化。
-Ono_float_const_fold	制止浮点数字的常数合并处理。
-Ono_stdlib	制止标准程序库函数的内联填充以及修改程序库函数。
-Ono_logical_or_combine	制止将连续 OR 放在一起的优化。
-Ono_asmopt	通过汇编程序优化器“aopt30”制止优化。
-Ocompare_byte_to_word	以字 (word) 的方式比较位于邻近地址的数据之连续字节。
-Oforward_function_to_inline	对所有内联函数执行内联扩展。
-Oglb_jmp	优化外部参考到跳转指令。

(6) 用于修改生成的代码之选项

选项	功能
-fansi	确认 -fnot_reserve_far_and_near、-fnot_reserve_asm、-fnot_reserve_inline，以及 -fextend_to_int。
-fnot_reserve_asm	从预定字排除 asm（只有“_asm”有效）。
-fnot_reserve_far_and_near	从预定字排除“far”和“near”（只有“_far”和“_near”有效）。
-fnot_reserve_inline	从预定字排除“内联”（inline）（只有“_inline”将是预定字）。

选项	功能
-fextend_to_int	将数据从字符类型扩展成整数类型之后执行运算（使用符合 ANSI 标准的扩展）。
-fchar_enumerator	将计数类型作为无符号的字符类型处理，而不是整数类型。
-fno_even	在没有将奇数数据从偶数数据分开的情形下，将所有输出数据分配到奇数属性段。
-ffar_RAM	将 RAM 数据的默认属性更改成 “far”。
-fnear_ROM	将 ROM 数据的默认属性更改成 “near”。
-fnear_pointer	将指针和地址的默认属性更改成 “near”。
-fconst_not_ROM	不会处理以常数指定为 ROM 数据的类型。
-fnot_address_volatile	不会将由 #pragma ADDRESS (#pragma EQU) 指定的变量当作是由 volatile 指定的变量。
-fenable_register	确认寄存器存储类。
-finfo	输出 “检查器”（Inspector）、“Stk 阅读器”（Stk Viewer）、“映像阅读器”（Map Viewer）和 utl30 需要的信息。
-M82	生成 M32C/80 系列的代码。
-fswitch_other_section	将为 switch 语句生成的表跳转输出至程序段以外的段。
-ferase_static_function=*函数名称*	如果在此选项中指定的函数是一个静态函数则不会生成任何代码。
-fno_switch_table	为 switch 语句生成比较后转移的代码。
-fmake_vector_table	自动生成变量向量表。
-fmake_special_table	自动生成 special 页向量表。

(7) 程序库指定选项

选项	功能
-l library-file-name	指定要在连接期间使用的程序库。

(8) 警告选项

选项	功能
-Wnon_prototype	在使用不具备原型声明的函数时输出警告。
-Wunknown_pragma	在使用不支持的 #pragma 时输出警告。
-Wno_stop	即使在发生错误时也不会停止编译。
-Wstdout	将错误信息输出至主机的标准输出 (stdout)。
-Werror_file<fileame>	输出标签文件。
-Wstop_at_warning	在编译期间出现警告时停止编译。
-Wnesting_comment	对含有“/*”的注解输出警告。
-Wccom_max_warnings =*警告计数*	允许您指定 ccom308 可以输出警告的最多次数。
-Wall	显示所有可检测的警告（但是，不包括由“-Wlarge_to_small”和“-Wno_used_argument”输出的警告）。
-Wmake_tagfile	出现错误和警告时为每个文件输出标签文件。
-Wuninitialize_variable	对尚未初始化的自动变量输出警告。
-Wlarge_to_small	对以大小递减顺序进行的隐含变量分配输出警告。
-Wno_warning_stdlib	此选项与“-Wnon_prototype”或“-Wall”一起指定，可禁止“不具备原型声明的标准程序库的警告”（warning for standard libraries that do not have prototype declaration）。
-Wno_used_static_function	显示不需要代码生成的静态函数名称。
-Wundefined_macro	在 #if 中使用未定义的宏时显示警告。
-Wstop_at_link	在连接期间出现警告时制止绝对模块文件的生成。

(9) 汇编和连接选项

选项	功能
-as308Δ<*选项*>	指定 as308 汇编命令的选项。要传递两个或更多选项，请将它们放入双引号 (")。
-ln308Δ<*选项*>	指定 ln308 连接命令的选项。要传递两个或更多选项，请将它们放入双引号 (")。

3.7 段

3.7.1 由 NC308 管理的段

NC308 将数据和代码映像区域作为段管理。

本小节将描述 NC308 所管理的段类型，以及如何管理它们。

(1) 段的结构

NC308 根据作为个别段的类型管理数据。表 3.6 显示 NC308 所管理之段的结构。

表 3.6 NC308 的段之结构

段基本名称	内容
data	存储具有初始值的静态变量。
bss	存储不具有初始值的静态变量。
rom	存储字符串和常数。
program	存储程序。
program_s	存储在 #pragma SPECIAL 中指定的程序。
vector	变量向量区域（编译程序不会生成此区域）。
fvector	固定向量区域（编译程序不会生成此区域）。
stack	堆栈区域（编译程序不会生成此区域）。
heap	堆区域（编译程序不会生成此区域）。

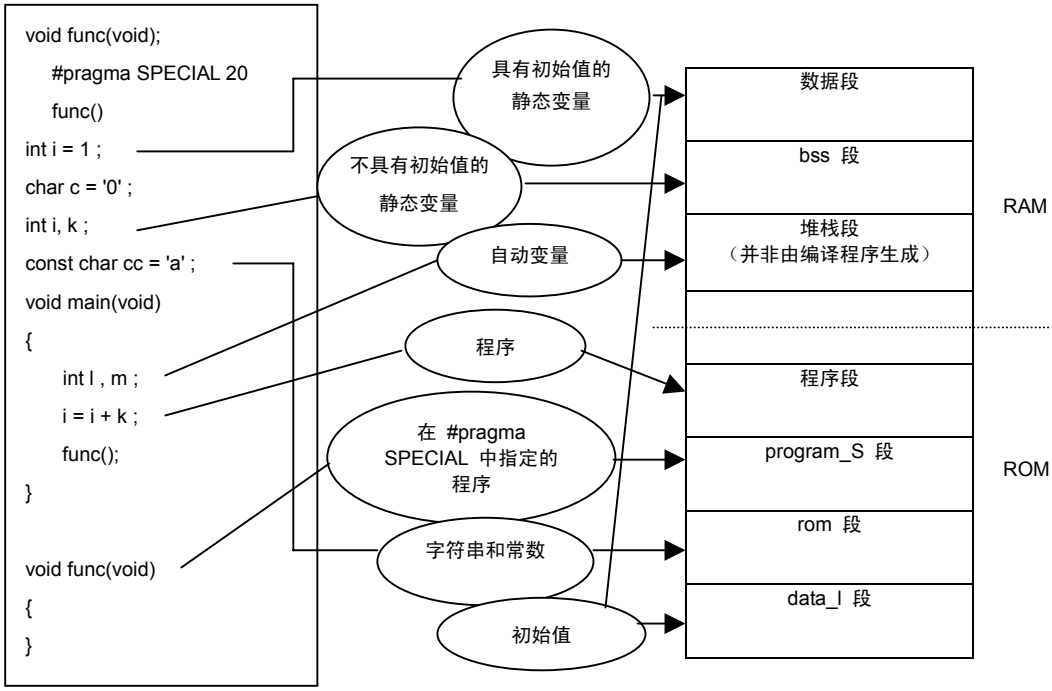


图 3.41 根据数据类型映像段

(2) 段属性

由 NC308 生成的段会根据它们的属性进一步分类，包括它们是否具有初始值、它们所映像的区域，以及它们的数据大小。

表 3.7 显示标示属性的符号。

表 3.7 段属性

属性	内容	可应用的段基本名称
I	包含数据之初始值的段	data
N/F/S	N：“near”属性（从 000000H 至 00FFFFH 的区域） F：“far”属性（从 000000 至 FFFFFFH 的区域）	data、bss、rom
	S：SBDATA 属性（SB 相对寻址的可用区域）	data、bss
E/O	E：数据大小是一个偶数。 O：数据大小是一个奇数。	data、bss、rom

(3) 段的命名规则

由 NC308 所生成之段的名称由段基本名称和属性决定。

图 3.42 显示段基本名称和属性的结合。

段名称 = 段基本名称_属性

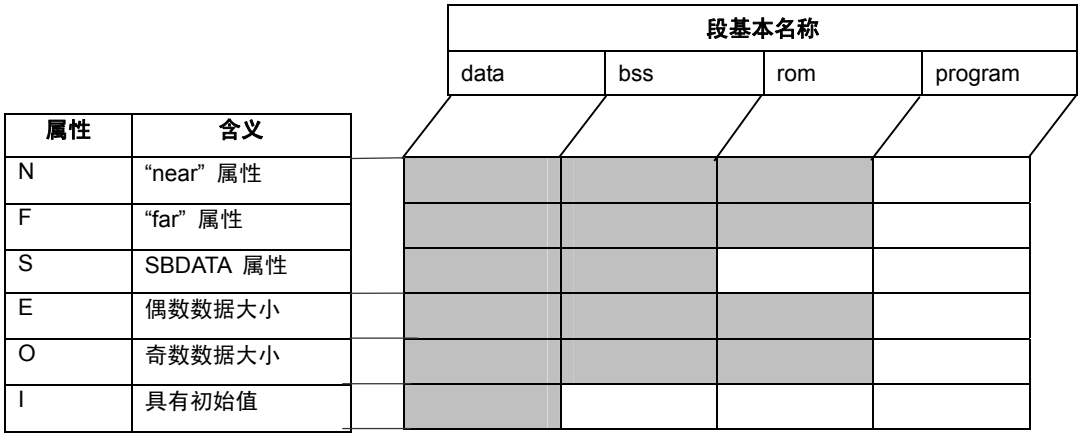


图 3.42 段名称的命名规则

3.8 交叉软件的相关事项

3.8.1 汇编语言程序的相关事项

几乎所有类型的程序都能够以 C 编写。然而，您可以在您要提高性能或使用特殊指令时使用汇编语言。本节复习在将 C 程序连接到汇编语言程序时您必须牢记的事项。

(1) 从 C 程序调用汇编程序函数

(a) 不具有参数的汇编程序函数

从 C 程序调用汇编程序函数时，请按照调用以 C 编写的函数之相同方式使用汇编程序函数的名称。

汇编程序函数中的第一个标签名称前面必须有一个下划线 (_) 。要从 C 程序调用汇编程序函数，请使用没有下划线的第一个标签名称。调用的 C 程序必须包含汇编程序函数的原型声明。

图 3.43 显示调用汇编程序函数 asm_func 的实例。

```
extern void asm_func( void );        ← 汇编程序函数的原型声明
void main()
{
    :
    （被省略）
    :
asm_func();                          ← 调用汇编程序函数

}
```

图 3.43 显示调用不具有参数的汇编程序函数 (smp1.c) 的实例

```
.glb _main
_main:
:
（被省略）
:
jsr _asm_func   ← 调用汇编程序函数（前面标有“_”）
rts
```

图 3.44 smp1.c (Excerpt) (smp1.a30) 的编译结果

(b) 将参数传递至汇编程序函数

将参数传递至汇编程序函数时，请使用扩展函数 #pragma PARAMETER。

#pragma PARAMETER 会通过 32 位一般用途寄存器（R2R0、R3R1）、16 位一般用途寄存器（R0、R1、R2、R3）、8 位一般用途寄存器（R0L、R0H、R1L、R1H），以及地址寄存器（A0、A1）将参数传递至汇编程序函数。

以下显示使用 #pragma PARAMETER 来调用汇编程序函数的步骤：

① 在声明 #pragma PARAMETER 之前输入汇编程序函数的原型声明。
您也必须声明参数类型。

② 对于 #pragma PARAMETER，声明要在汇编程序函数的参数列表中使用的寄存器名称。

图 3.45 显示使用 #pragma PARAMETER 来调用汇编程序函数 asm_func 的实例。

```
extern unsigned int asm_func(unsigned int, unsigned int);
#pragma PARAMETER asm_func(R0, R1) ← 通过 R0 和 R1 寄存器将参数传递至汇编程序函数。
void main()
{
    int i = 0x02;
    int j = 0x05;
    asm_func(i, j); ← 调用汇编程序函数
}
```

图 3.45 显示调用具有参数的汇编程序函数 (smp2.c) 的实例

```
.glb _main
_main:
enter #04H
pushm R1
._line6
;## # C_SRC : int i = 0x02;
mov.w #0002H,-4[FB] ; i
._line7
;## # C_SRC : int j = 0x05;
mov.w #0005H,-2[FB] ; j
._line9
;## # C_SRC : asm_func(i, j);
mov.w -2[FB],R1 ; j ← 通过 R0 和 R1 寄存器将参数传递至汇编程序函数。
    mov.w -4[FB],R0 ; i
    jsr _asm_func ← 调用汇编程序函数（前面标有 “_”）
    ._line10
;## # C_SRC : }
    popm R1
    exitd
```

图 3.46 smp2.c (Excerpt) (smp2.a30) 的编译结果

(c) #pragma PARAMETER 声明中的参数限制

您不可以在 #pragma PARAMETER 声明中声明下列参数类型：

- 结构类型和联合类型的参数
- 64 位整数类型（加长）参数
- 双精度浮点类型（双精度）参数

您不能对汇编程序函数定义结构或联合类型的返回值。

(2) 编写汇编程序函数

(a) 编写要调用的汇编程序函数

以下描述编写汇编程序函数的入口处理的步骤：

① 使用汇编程序伪指令 .SECTION 指定段名称。

② 使用汇编程序伪指令 .GLB 将函数名称标签指定为全局。

③ 将一个下划线 (_) 添加到函数名称中以将它编写为标签。

④ 如果您要修改 B 或 U 标志，请将标志寄存器保存到堆栈中。

⑤ 保存可能会在函数中被破坏的寄存器。

以下描述编写汇编程序函数的出口处理的步骤：

⑥ 恢复在函数入口处理期间所保存的寄存器。

⑦ 如果您在函数中修改了 B 和 U 标志，请从堆栈恢复标志寄存器。

⑧ 编写 RTS 指令。

不要更改汇编程序函数中的 SB 和 FB 寄存器的内容。

如果您更改 SB 和 FB 寄存器的内容，请在进入函数时将它们保存到堆栈中，然后在退出函数时从堆栈恢复它们。

图 3.47 显示编写汇编程序函数的实例。在此实例中，段名称为 "program" ，和编译程序所输出的段名称相同。

```
.SECTION program          ← ①
.GLB _asm_func            ← ②
_asm_func:                ← ③
 PUSHC FLG                ← ④
 PUSHM R3,R1              ← ⑤
 MOV.L SYM1, R3R1
 POPM R3,R1               ← ⑥
 POPC FLG                 ← ⑦
 RTS                      ← ⑧
 .END
* ① 至 ⑧ 与上述步骤相应。
```

图 3.47 编写汇编程序函数的实例

(b) 从汇编程序函数返回值

整数、指针和浮点类型的值可以通过寄存器从汇编程序函数返回至 C 程序。表 3.8 显示返回值的调用规则。图 3.48 显示编写汇编程序函数以返回值的实例。

表 3.8 返回值的调用规则

返回值类型	规则
_Bool 类型 字符类型	R0L 寄存器
整数类型 near 指针类型	R0 寄存器
浮点类型 长类型	返回值时，16 低顺序位将会在 R0 寄存器中存储，而 16 高顺序位则存储在 R2 寄存器中。

返回值类型	规则
双精度类型 长双精度类型	返回值时，每个以 16 位存储，从 MSB 开始，以 R3、R2、R1 和 R0 寄存器的顺序存储。
加长类型	返回值时，每个以 16 位存储，从 MSB 开始，以 R3、R1、R2 和 R0 寄存器的顺序存储。
结构类型 联合类型	调用函数之前，标示用来存储返回值的区域之 “far” 地址会即刻推到堆栈中。返回调用程序之前，所调用的函数会将返回值写入已推到堆栈中以 “far” 地址来标示的区域。

```
.SECTION program
.GLB _asm_func
_asm_func:

（被省略）

    MOV.I #01A000H, R2R0
    RTS
    .END
```

图 3.48 编写汇编程序函数以返回长类型返回值的实例

(c) 参考 C 变量

由于汇编程序函数是在和 C 程序不同的文件中编写，因此只有 C 全局变量可以参考。

要在汇编程序函数中包含 C 变量名称，请在它们的前面添加一个下划线 (_) 。您也需要使用汇编程序伪指令 .GLB 在汇编程序语言程序中声明外部参考的变量。

图 3.49 显示从汇编程序函数 asm_func 参考 C 程序全局变量 “counter” 的实例。

```
[C 程序]
unsigned int counter;          ←C 程序全局变量

main()
{
     :
（被省略）
     :
}
[汇编程序函数]
  .GLB _counter                ← 声明外部参考的 C 程序全局变量
asm_func:
       :
（被省略）
       :
  MOV.W _counter, R0           ← 参考
```

图 3.49 参考 C 全局变量

(d) 在汇编程序函数中指定中断处理的注意事项

必须在程序（函数）的入口和出口点执行下列操作以进行中断处理：

1. 在入口点保存寄存器（R0、R1、R2、R3、A0、A1 和 FB）。
2. 在出口点恢复寄存器（R0、R1、R2、R3、A0、A1 和 FB）。
3. 使用 REIT 指令从函数返回。

图 3.50 显示编写汇编程序函数以进行中断处理的实例。

```
.section program
.glb _func
_func:
pushm R0,R1,R2,R3,A0,A1,FB     ← 保存所有寄存器。
MOV.B #01H, R0L
:
（被省略）
:
popm R0,R1,R2,R3,A0,A1,FB      ← 恢复所有寄存器。
reit                           ← 返回至 C 程序。
.END
```

图 3.50 编写中断处理汇编程序函数的实例

(e) 从汇编程序调用 C 函数的注意事项

从汇编语言程序调用以 C 编写的函数时请注意下列事项：

① 要调用 C 函数，请使用前面标有下划线 (_) 或元 ($) 的标签名称。

② 对于 NC308，R0 寄存器和用于返回值的寄存器不会在函数的入口处理期间保存。您必须在从汇编程序调用 C 函数之前保存这些寄存器。

对于 NC30，C 函数不会保存或恢复寄存器。调用 C 函数之前，先保存在汇编程序函数中使用的寄存器。从 C 函数返回之后恢复它们。

(3) 编写汇编程序函数的注意事项

编写从 C 程序调用的汇编语言函数（子例程）时请注意下列事项。

(a) 处理 B 和 U 标志的注意事项

从汇编程序函数返回至 C 程序时，将 B 和 U 标志恢复到它们在函数调用时的相同状态。

(b) 处理 FB 寄存器的注意事项

如果您在汇编程序函数中修改 FB（帧基址寄存器）的值，您通常无法返回调用的 C 程序。因此，不要在汇编程序函数中修改 FB值。如果您因为系统设计而需要更改 FB 寄存器，请在函数的开始保存它，然后在从所调用的函数返回该函数时将它恢复。

(c) 处理一般用途寄存器和地址寄存器的注意事项

在汇编程序函数中修改一般用途寄存器（R1、R2 和 R3，除了 R0）和地址寄存器（A0、A1）的内容时，您必须在汇编程序函数的入口处理保存它们，然后在出口处理时将它们恢复。

但是，如果汇编程序函数是使用 #pragma PARAMETER /C 声明，用于保存和恢复寄存器的代码将会在调用的程序中生成。因此，不需要在汇编程序函数中保存和恢复寄存器。（代码数量会变得稍微大些。）

(d) 将参数传递至汇编程序函数的注意事项

如果您要将参数传递至以汇编语言编写的函数，请使用 #pragma PARAMETER 函数通过寄存器传递这些参数。图 3.51 显示该格式（“asm_func” 是汇编程序函数的名称）

```
unsigned int near asm_func(unsigned int, unsigned int);
        ↑汇编程序函数的原型声明
#pragma PARAMETER asm_func(R0, R1)
```

图 3.51 汇编程序函数的编写实例

#pragma PARAMETER 会通过 16 位一般用途寄存器（R0、R1、R2、R3）、8 位一般用途寄存器（R0L、R0H、R1L、R1H），以及地址寄存器（A0、A1）将参数传递至汇编程序函数。此外，16 位一般用途寄存器会通过传递至汇编程序函数的参数与地址寄存器结合以形成 32 位寄存器（R3R1、R2R0、A1A0）。

您必须在声明 #pragma PARAMETER 之前输入汇编程序函数的原型声明。

但是，您不可以在 #pragma PARAMETER 声明中声明下列参数类型：

- 结构类型和联合类型的参数
- 64 位整数类型（加长）参数
- 双精度浮点类型（双精度）参数

您不能对汇编程序函数定义结构或联合类型的返回值。

3.9 加长类型

编译程序支持“加长”（long long）和“无符号加长”（unsigned long long）类型数据。

此数据类型以“加长”（long long）表示有分正负的整数，和以“无符号加长”（unsigned long long）表示无正负之分的整数。

要建立加长类型整数常数，请在整数数值后面附加后缀 LL。要建立无符号加长类型整数常数，请在整数数值后面附加后缀 ULL。

表 3.9 整数类型和值范围

类型	值范围	数据大小
char	0 至 255	1 字节
signed char	-128 至 127	1 字节
unsigned char	0 至 255	1 字节
short	-32768 至 32767	2 字节
unsigned short	0 至 65535	2 字节
int	-32768 至 32767	2 字节
unsigned int	0 至 65535	2 字节
long	-2147483648 至 2147483647	4 字节
unsigned long	0 至 4294967295	4 字节
long long	-9223372036854775808 至 9223372036754775807	8 字节
unsigned long long	0 至 18446744073709551615	8 字节

3.10 “near/far” 类型

M16C/80 系列的最大存取区域是 16 兆字节。NC308 将此区域划分成一个 near 区域（从 000000H 至 00FFFFH）和一个 far 区域（从 000000H 至 FFFFFFH）进行管理。本小节将描述如何将变量和函数映像至这些区域以及如何存取这些区域。

3.10.1 Near 和 Far 区域

为了管理最大 16 兆字节的存取区域，NC308 将此区域划分成一个 near 和 far 区域。表 3.10 显示这些区域的特征。

表 3.10 Near 和 Far 区域

区域	描述
Near 区域	此区域可帮助 M16C/80 系列高效存取数据。 此 64 千字节的区域在 000000H 至 00FFFFH 的绝对地址上扩展。堆栈和内部 RAM 映像至此区域。
Far 区域	可从 M16C/80 系列存取整个存储器空间。此 16 兆字节的区域在 000000H 至 FFFFFFH 的绝对地址上扩展。 内部 ROM 映像至此区域。

3.10.2 “near” 和 “far” 属性的默认值

对于 NC308，映像至 near 区域的变量和函数具有 “near” 属性，而那些映像至 far 区域的则具有 “far” 属性。表 3.11 显示变量和函数的默认属性。

表 3.11 “near” 和 “far” 属性的默认值

类别	属性
程序	固定为 “far”
RAM 数据	“near”（但是，指针类型具有 “far” 属性）
ROM 数据	“far”
堆栈数据	固定为 “near”

要更改 near 和 far 属性的默认值，请在启动 NC308 时指定下列选项：

-ffar_RAM (-fFRAM)： 将 RAM 数据的默认属性更改成 “far”。

-fnear_ROM (-fNROM)： 将 ROM 数据的默认属性更改成 “near”。

-fnear_pointer (-fNP)： 将指针类型数据的默认属性更改成 “near”。

3.10.3 函数的 “near” 和 “far” 指定

因为 M16C/80 系列的体系结构，NC308 函数的属性将固定为 far 区域。如果您指定 “near”，NC308 将会在编译期间输出警告并强制将函数映像至 far 区域。

3.10.4 变量的 “near” 和 “far” 指定

[*存储类*] *类型说明符* near / far *变量名称*;

如果您在类型声明中没有指定 “near” 或 “far”，RAM 数据将会映像至 near 区域。具有常数限定语符的数据和 ROM 数据将会映像至 far 区域。

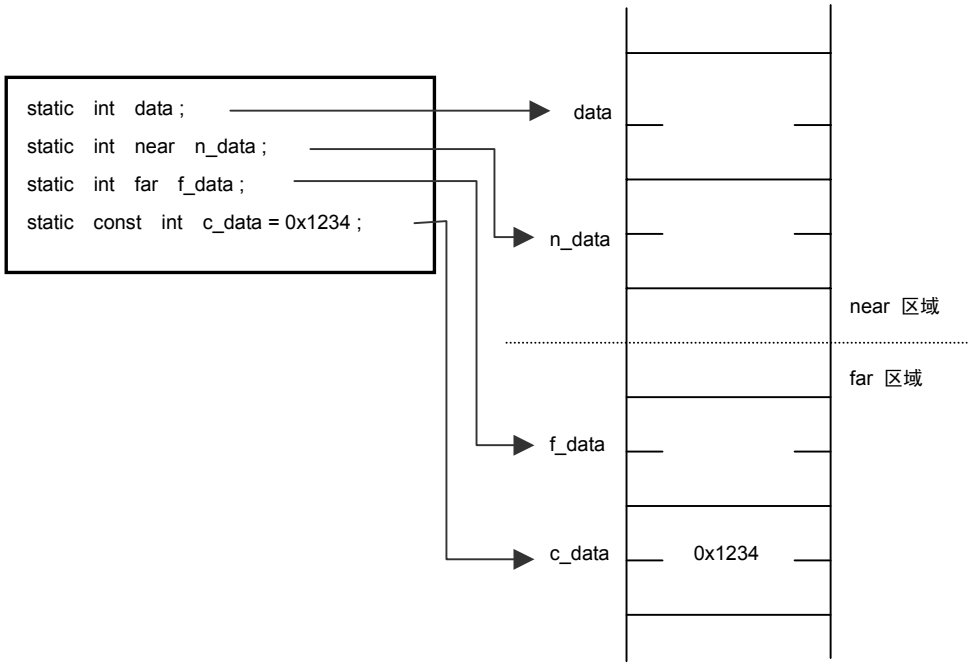


图 3.52 静态变量的 near/far

“near”或“far”的指定不会影响自动变量的映像，因为它们全部会映像至堆栈区域。

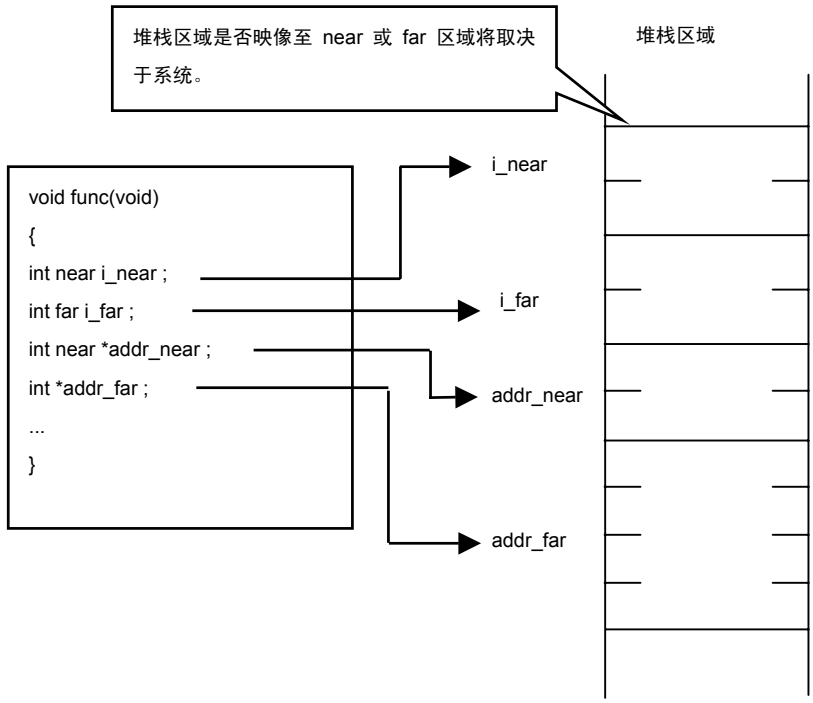


图 3.53 自动变量的 near/far

3.10.5 指针的“near”和“far”指定

指定指针的“near”或“far”，可以指定要在指针中存储的地址的大小，指针所映像的区域。

(1) 指定要在指针中存储的地址的大小

如果您没有指定任何项目，指针将会当作一个指向 far 区域中的变量的 32 位（4 字节）指针变量处理。

```
[存储类]  类型说明符  near/far  * 变量名称;
```

near：要在指针变量中存储的地址的大小为 16 位。

far： 要在指针变量中存储的地址的大小为 32 位。

```
int   near   *near_data ;
int   far    *far_data ;
```

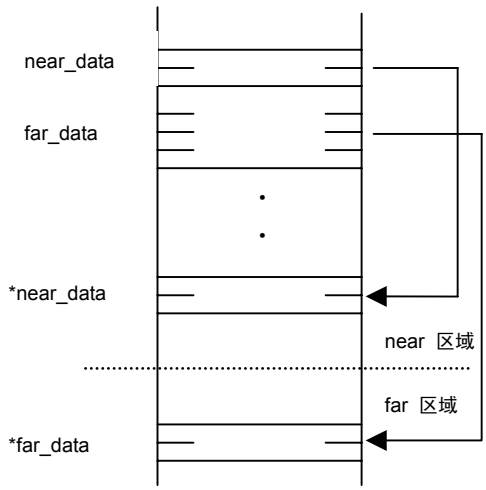


图 3.54 指定存储在指针中的地址大小

(2) 指定要映像指针的区域

如果您没有指定任何项目，指针变量将会映像至 near 区域。

[*存储类*] *类型说明符* * near/far *变量名称*;

near：将指针变量的区域映像至 near 区域。

far： 将指针变量的区域映像至 far 区域。

```
int   *near   near_data ;
int   *far    far_data ;
```

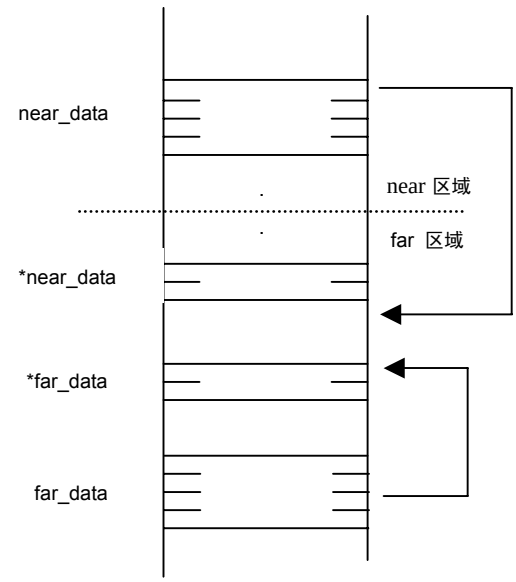


图 3.55 指定指针映像区域

3.10.6 NC308 和 NC30 之间的指针 “near/far” 指定的差异

对于 M16C/60 和 M16C/20 系列的 NC30 C 编译程序，如果未指定 “near” 或 “far”，所有指针将假设为 “near” 属性。对于 NC308，如果没有为要在指针中存储的地址的大小指定任何项目，指针变量的大小将假设为 32 位（4 字节）。指针随后即当作一个指向 far 区域中的变量之指针变量处理。

3.10.7 将 far 区域中的变量地址分配到 “near” 指针

如果尝试将 far 区域中的变量地址分配到 “near” 指针，NC308 将会输出一则警告信息，说明该指针将会分配，但会忽略地址的较高顺序部分。

NC308 也会输出一则警告信息，说明 “far” 指针会明确或隐含地转换为 “near” 指针。

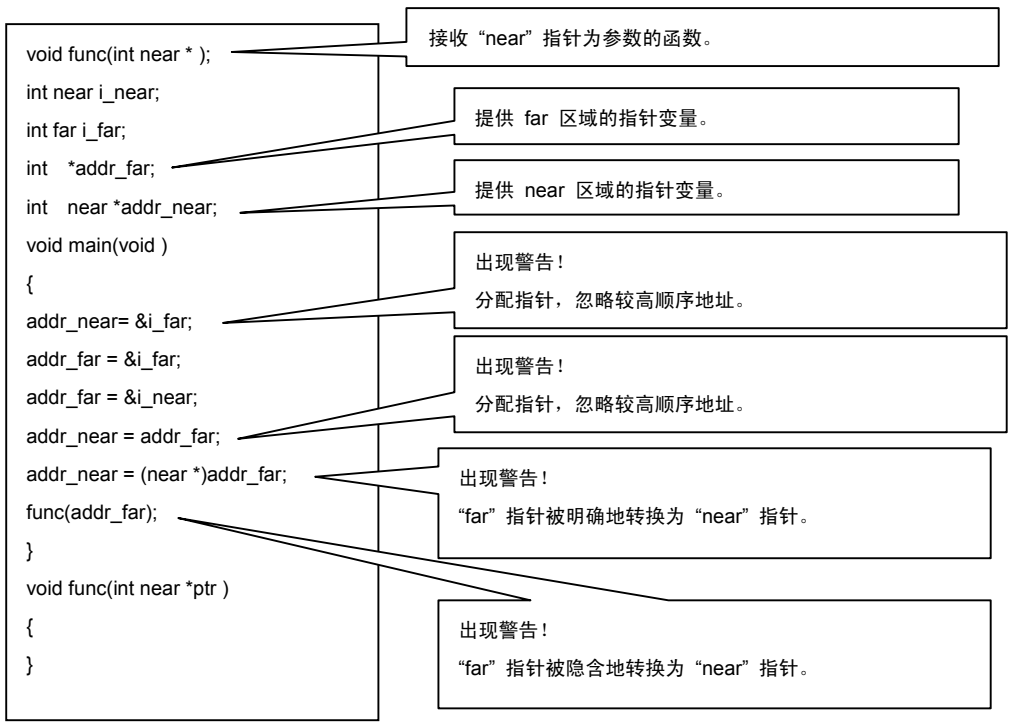


图 3.56 将 far 区域中的变量地址分配到 near 指针

3.11 内联扩展

您可以用在 C++ 中的相同操作方式来指定内联存储类。通过指定函数的内联存储类，您可以执行该函数的内联扩展。

3.11.1 内联存储类的概述

内联存储类说明符声明函数可进行内联扩展。对于为内联存储类指定的函数，代码会在汇编语言层直接嵌入。

3.11.2 内联存储类声明的格式

在声明中，使用静态的相同语法 extern 类型存储类说明符，输入一个内联存储类说明符。图 3.57 显示声明格式。

内联Δ*类型说明符*Δ*函数*;

图 3.57 内联存储类的声明

图 3.58 显示函数声明的实例。图 3.59 显示编译的结果。

```
inline int func(int i)                    ← 声明和定义内联函数的一部分
{
    return i++;
}
void main()
{
    int s;
    s = func(s);                          ←  调用内联函数的一部分
    }
```

图 3.58 内联函数的样品程序

```
._LANG 'C','X.XX.XX','REV.X'
;## NC308 C Compiler OUTPUT
;## ccom308 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS
RESERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,
2000,2001,2002,2003
;## COMMAND_LINE: ccom308 D:¥MTOOL¥nc308wa5¥TMP¥sss.i -o .¥smp.a30
d S
;## Normal Optimize O F F
;## ROM size Optimize O F F
;## Speed Optimize O F F
;## Default ROM is f a r
;## Default RAM is near
        .GLB __SB__
        .SB __SB__
        .FB 0
;## # FUNCTION func
;## # FUNCTION main
;## # FRAME AUTO ( s) size 2, offset -4
;## # FRAME AUTO ( i) size 2, offset -2
;## # ARG Size(0) Auto Size(4) Context Size(8)
        .SECTION program,CODE,ALIGN
        ._file 'smp.c'
        .align
        ._line 7
;## # C_SRC : {
        .glb _main
_main:
        enter #04H
        pushm R1
        ._line 9
;## # C_SRC : s = func(s);
        mov.w -4[FB],R0 ; s
        ._line 2
;## # C_SRC : {
        mov.w R0,-2[FB] ; i
        ._line 3
;## # C_SRC : return i++;          ← 内联函数已嵌入。

        mov.w R0,R1
        add.w #0001H,R0
._line 9
;## # C_SRC : s = func(s);
        mov.w R1,-4[FB] ; s
        ._line 10
;## # C_SRC : }
        popm R1
        exitd
E1:
        .END
;## Compile End Time Tue Jul 16 13:12:00 20xx
```

图 3.59 样品程序的编译结果

3.11.3 内联存储类的规则

指定内联存储类时请注意下列事项：

- 关于内联函数参数

  您不能使用结构或联合类型作为内联函数的参数。

  如果您使用这些类型，将会发生编译错误。

- 关于内联函数的间接调用

  您不能执行内联函数的间接调用。间接调用的指定将会导致编译错误。

- 关于内联函数的递归调用

  您不能执行内联函数的递归调用。递归调用的指定将会导致编译错误。

- 关于内联函数的定义

  当您为函数指定内联存储类时，除了声明之外，您还必须指定函数的主体。此主体定义必须包含在该函数所在的同个文件中。

  图 3.60 显示编译程序处理为错误时的编写。

```
inline void func(int i);
void main( void )
{
    func(1);
}

[错误信息]
[Error(ccom):smp.c,line 5] inline function's body is not declared previously
===> func(1);
Sorry, compilation terminated because of these errors in main().
```

图 3.60 内联函数的不当编写的实例 (1)

如果您将函数作为普通函数使用，然后将它定义为内联函数，该内联指定将会禁止，并且所有函数将会作为静态函数处理。在此情形下，编译程序将会输出一则警告。

```
int func(int i);
void main( void ){
 func(1);
}
inline int func(int i){
      return i;
}
[警告信息]
[Warning(ccom):smp.c,line 9] inline function is called as normal function before
,change to static function
```

图 3.61 内联函数的不当编写的实例 (2)

- 关于内联函数的地址

  由于内联函数不具备地址，对内联函数使用 & 运算符将会导致错误。

```
inline int func(int i)
{
      return i;
}
main()
{
      int (*f)(int);
      f = &func;
}
[错误信息]
[Error(ccom):smp.c,line 10] can't get inline function's address by '&' operator
===> f = &func;
Sorry, compilation terminated because of these errors in main().
```

图 3.62 内联函数的不当编写的实例 (3)

- 静态数据的声明

  如果在内联函数中声明静态数据，所声明的静态数据的主体将会以文件单位分配。基于此原因，如果内联函数在两个或更多文件中扩展，不同的区域将会被存取。

  要在内联函数中使用的静态数据必须在函数外声明。如果在内联函数中发现静态声明，编译程序将会输出警告。我们不建议内联函数中的静态声明。

```
inline int func( int j)
{
static int i = 0;
i++;
return i + j;
}
[警告信息]
[Warning(ccom):smp.c,line 3] static valuable in inline function
===> static int i = 0;
```

图 3.63 内联函数的不当编写的实例 (4)

- 调试信息

  编译程序不会输出内联函数的 C 语言层调试信息。

  这意味着内联函数在汇编语言层进行调试。

M16C 族 C 编译器应用笔记

使用 HEW

第 4 节 使用 HEW

4.1 指定 HEW 选项

您可以从 [选项（Options）] 菜单指定选项。以下显示如何从瑞萨集成开发环境（Renesas Integrated Development Environment）指定选项。从 [选项（Options）] 菜单选取 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 。

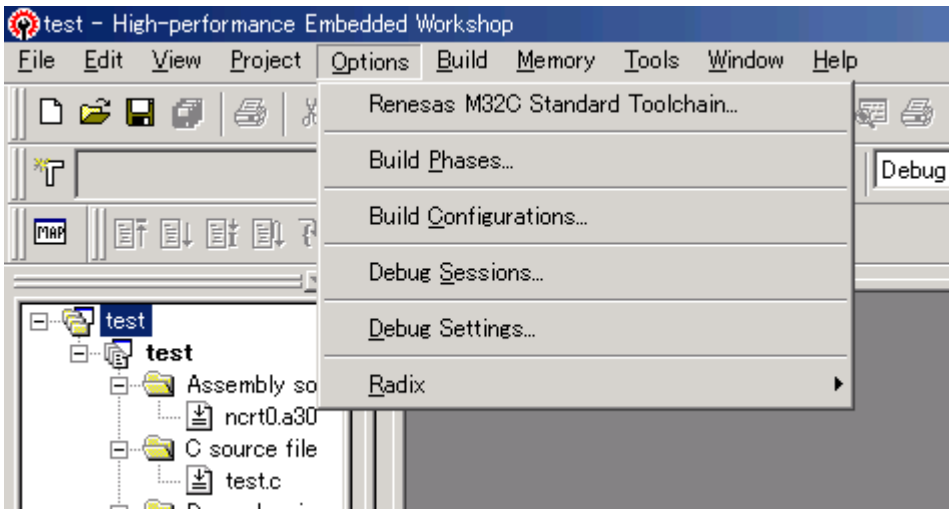


图 4.1 HEW 选项菜单

4.1.1 C 编译程序选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [C] 标签。

– 类别（Category）:[源（Source）]

表 4.1 类别（Category）:[源（Source）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项
显示有关项目:	
包含文件目录	I<目录名称>
定义	D<sub> <sub> : <宏名称> [= <字符串>]
预定义	U<sub> <sub> : <预定义的宏名称>

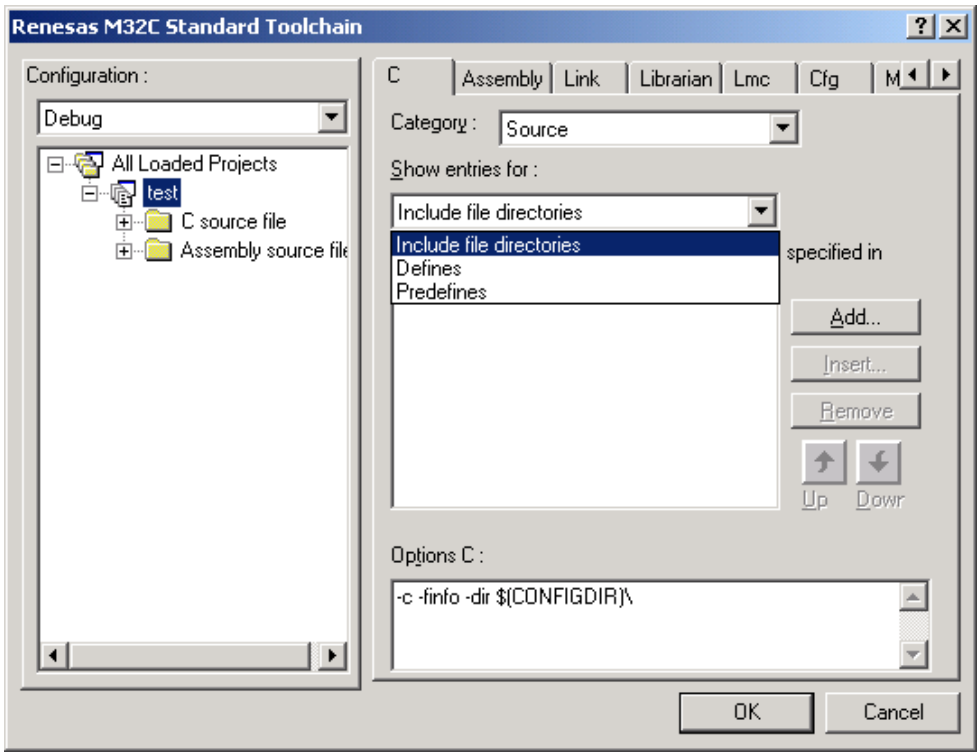


图 4.2 类别（Category）:[源（Source）] 对话框

– 类别（Category）:[目标（Object）]

表 4.2 类别（Category）:[目标（Object）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项
输出文件类型：	
[-c] 可再定位文件 (*.r30)	c
[-S] 汇编语言源文件 (*.a30)	S
[-P] 预处理后的源文件 (*.i)	P
[-E] 预处理输出	E
调试选项：	
[-finfo] 输出检查器、Stk 阅读器和 utl30 需要的信息	finfo
[-g] 输出调试信息以便您进行 C 语言层调试	g
[-genter] 在调用函数时永远输出一个回车指令。使用调试程序的堆栈跟踪功能时请确定指定此选项	genter
[-gno_reg] 制止输出寄存器变量的调试信息	gno_reg
[-dir] 指定要输出文件的目录：	dir<目录名称>

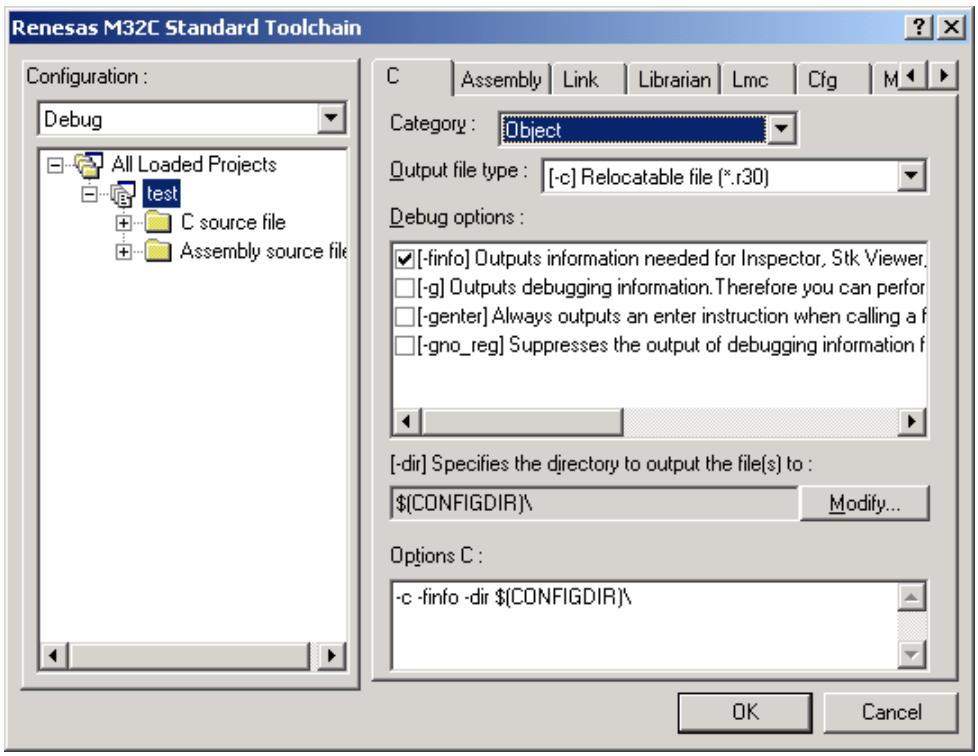


图 4.3 类别（Category）:[目标（Object）] 对话框

– 类别（Category）:[列表（List）]

表 4.3 类别（Category）:[列表（List）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项	快捷方式
[-dS] 在输出汇编语言源列表中输出 C 源代码作为注解。	dsource	dS
[-dSL] 在输出汇编语言列表中输出 C 源代码作为注解。	dsource_in_list	dSL

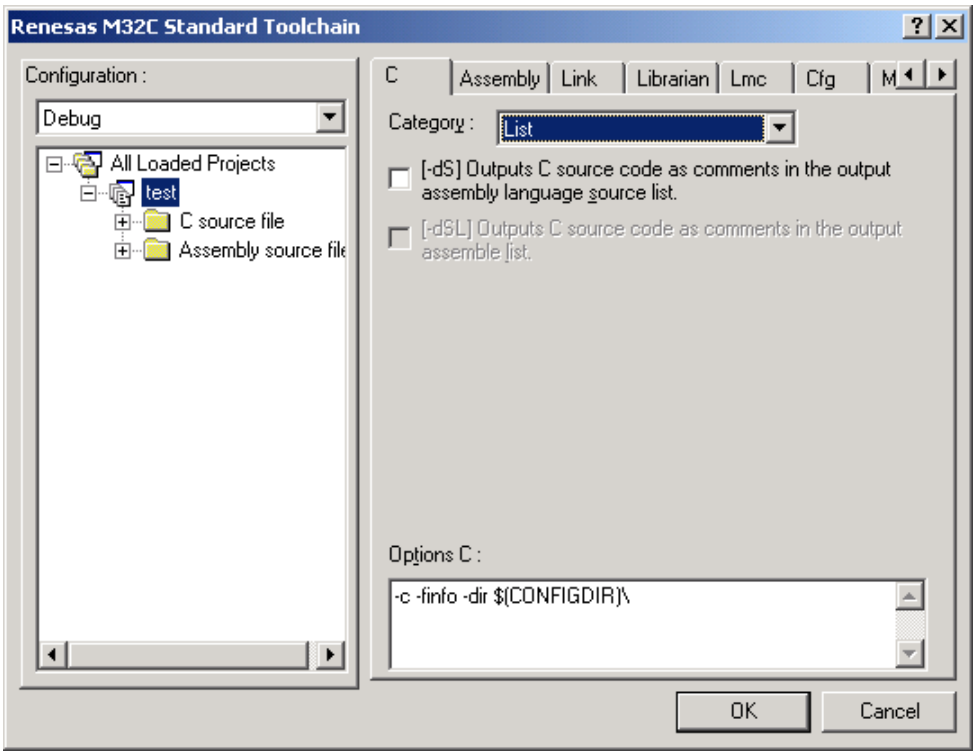


图 4.4 类别（Category）:[列表（List）] 对话框

– 类别（Category）:[优化（Optimize）]

表 4.4 类别（Category）:[优化（Optimize）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项	快捷方式
优化级别：		
[-O1] 使 –O3、-ONB、-ONBSD、-ONFCF 和 –ONS 有效	O1	无
[-O2] 使和 –O1 无差异	O2	无
[-O3] 对速度和 ROM 大小进行最大优化	O3	无
[-O4] 使 -O3 和 Oconst 有效	O4	无
[-O5] 实现可能的最佳优化	O5	无
大小或速度：		
[-OR] 先 ROM 大小后速度	OR	无
[-OS] 先速度后 ROM 大小	OS	无

对话框	选项	快捷方式
杂项：		
[-OC] 通过以常数替代常数标准外部变量 (const_qualified external variables) 的参考来执行优化	Oconst	OC
[-OCBTW] 以字 (word) 的方式比较位于邻近地址的数据之连续字节	Ocompare_byte_to_word	OCBTW
[-OFFTI] 对前面描述的函数执行内联部署。	Oforward_function_to_inline	OFFTI
[-OFTI] 开发浮点运行库函数。	Ofloat_to_inline	OFTI
[-OGJ] 优化参考全局标签的转移指令。	Oglb_jmp	OGJ
[-ONA] 制止汇编程序优化器 aopt308 的执行	Ono_asmopt	ONA
[-ONB] 制止基于位操纵分组的优化	Ono_bit	ONB
[-ONBSD] 制止可影响源行信息的优化	Ono_break_source_debug	ONBSD
[-ONFCF] 制止浮点数字的常数合并处理	Ono_float_const_fold	ONFCF
[-ONLOC] 制止将连续 OR 放在一起的优化	Ono_logical_or_combine	ONLOC
[-ONS] 禁止标准程序库函数的内联填充以及修改程序库函数	Ono_stdlib	ONS
[-OSA] 执行优化以删除调用函数之后的堆栈修正代码	Osp_adjust	OSA
[-OSTI] 静态函数被当作内联函数	Ostatic_to_inline	OSTI
[-OLU] 根据循环次数展开循环语句，而不执行循环。	Oloop_unroll=<numeric value>	OLU

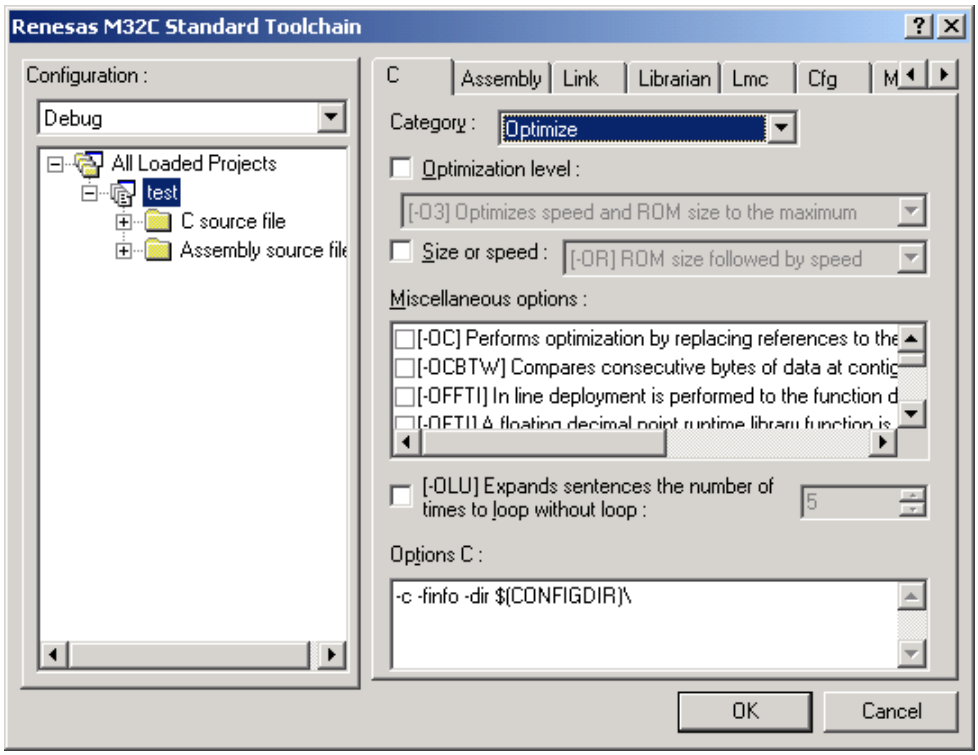


图 4.5 类别（Category）:[优化（Optimize）] 对话框

– 类别（Category）:[代码修改（Code Modification）]

表 4.5 类别（Category）:[代码修改（Code Modification）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项	快捷方式
杂项：		
[-fansi] 使 -fNRA、-fNRFAN、-fNRI 和 -fETI 有效	fansi	无
[-fCE] 将枚举类型作为无符号的字符类型处理，而不是整数类型	fchar_enumerator	fCE
[-fCNR] 不会将以 const 指定的类型作为 ROM 数据来处理	fconst_not_ROM	fCNR
[-fD32] 将双精度类型当作浮点类型处理	fdouble_32	fD32
[-fER] 使寄存器存储类可用	fenable_register	fER
[-fETI] 将字符类型数据扩展成整数类型之后执行运算。	fextend_to_int	fETI

对话框	选项	快捷方式
（根据 ANSI 标准扩展。） [-fFRAM] 将 RAM 数据的默认属性更改成 far	ffar_RAM	fFRAM
[-fJSRW] 将调用函数的默认指令更改成 JRSW	fJSRW	无
[-fMST] 生成 special 页向量表。	fmake_special_table	fMST
[-fMVT] 生成变量向量表。	fmake_vector_table	fMVT
[-fNA] 不对齐函数的起始地址	fno_align	fNA
[-fNAV] 不会将由 #pragma ADDRESS 指定的变量当作是由 volatile 所指定	fnot_address_volatile	fNAV
[-fNE] 输出时，在没有将奇数数据从偶数数据分开的情形下，将所有数据分配到奇数段。	fno_even	fNE
[-fNP] 将指针数据的默认类型更改成 near。	fnear_pointer	fNP
[-fNRA] 从预定字排除 asm。 （只有 _asm 有效。）	fnot_reserve_asm	fNRA
[-fNRFAN] 从预定字排除 far 和 near。 （只有 _far 和 _near 有效。）	fnot_reserve_far_and_near	fNRFAN
[-fNRI] 从预定字排除“inline”。 （只有 _inline 有效。）	fnot_reserve_inline	fNRI
[-fNROM] 将 ROM 数据的默认属性更改成 near	fnear_ROM	fNROM
[-fNST] 对于 switch 语句，它永远会进行比较然后生成转移的代码。	fno_switch_table	fNST
[-fSA] 在参考远型数组时，如果该数组的总大小在 64 千字节内，此选项将以 16 位计算索引。	fsmall_array	fSA
[-fSOS] 从与程序段不同的段对应的 ROM 表输出	fswitch_other_section	fSOS
[-fUD] 使用除法运算时忽略溢出。	fuse_DIV	fUD
[-fESF] 如果所指定的函数是静态函数，则不生成代码。	ferase_static_function=<函数名称>	fESF=<函数名称>

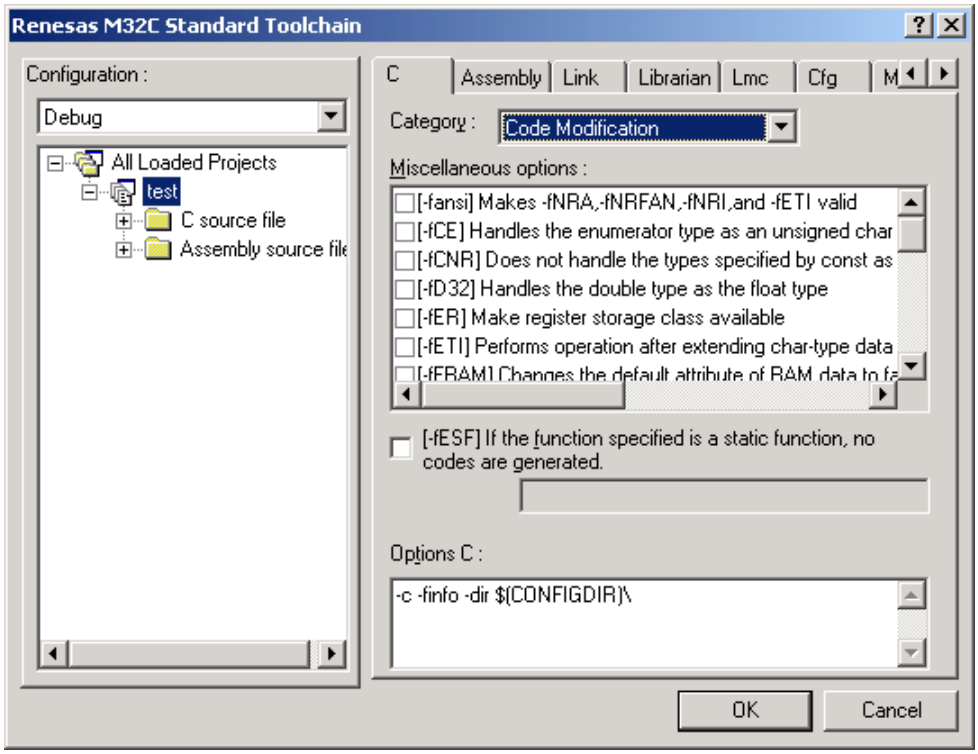


图 4.6 类别（Category）:[代码修改（Code Modification）] 对话框

– 类别（Category）:[警告（Warning）]

表 4.6 类别（Category）:[警告（Warning）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项	快捷方式
杂项：		
[-Wall] 所有警告选项有效	Wall	无
[-WLTS] 对从大的大小转移到较小的大小之隐含转移输出警告	Wlarge_to_small	WLTS
[-WMT] 输出错误信息到每个文件	Wmake_tagfile	WMT
[-WNC] 对含有 /* 的注解输出警告	Wnesting_comment	WNC
[-WNP] 对不具备原型声明的函数输出警告信息	Wnon_prototype	WNP
[-WNS] 防止编译程序在发生错误时停止	Wno_stop	WNS
[-WNUA] 对具有未使用参数的函数输出警告	Wno_used_argument	WNUA
[-WNUF] 对未使用的函数名称输出警告。	Wno_used_function	WNUF
[-WNUSF] 输出不需要代码生成的静态函数名称	Wno_used_static_function	WNUSF
[-WNWS] 制止缺失警告，包括使用标准程序库的文件。	Wno_warning_stdlib	WNWS
[-WSAL] 连接处理会在连接的时候出现警告时停止。	Wstop_at_link	WSAL
[-WSAW] 出现警告时停止编译处理	Wstop_at_warning	WSAW
[-Wstdout] 将错误信息输出至主机的标准输出(stdout)	Wstdout	无
[-WUM] 对 #if 中未定义的宏输出警告。	Wundefined_macro	WUM
[-WUP] 对不支持的 #pragma 输出警告信息	Wunknown_pragma	WUP
[-WUV] 对未初始化的自动变量输出警告	Wuninitialize_variable	WUV
[-WCMW] 指定由 ccom30 输出的最多警告数量：	Wccom_max_warnings=<数字值>	WCMW=<数字值>
[-WEF] 输出错误信息到指定的文件：	Werror_file=<文件名>	WEF=<文件名>

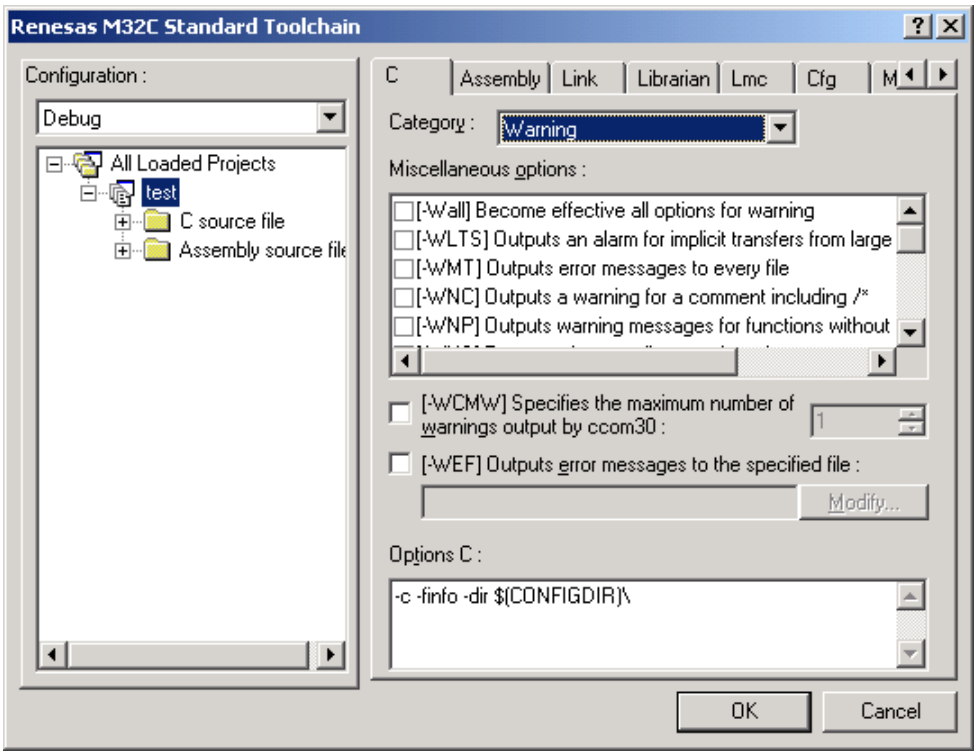


图 4.7 类别（Category）:[警告（Warning）] 对话框

– 类别（Category）:[其他（Other）]

表 4.7 类别（Category）:[其他（Other）] 对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项
杂项： [-silent] 制止启动时显示的版权信息 [-v] 显示执行期间的命令程序和命令行的名称 [-V] 显示编译程序的启动信息，然后退出处理（不编译）。	 silent v V
[-l] 程序库文件：	l<文件名>
用户定义的选项：	

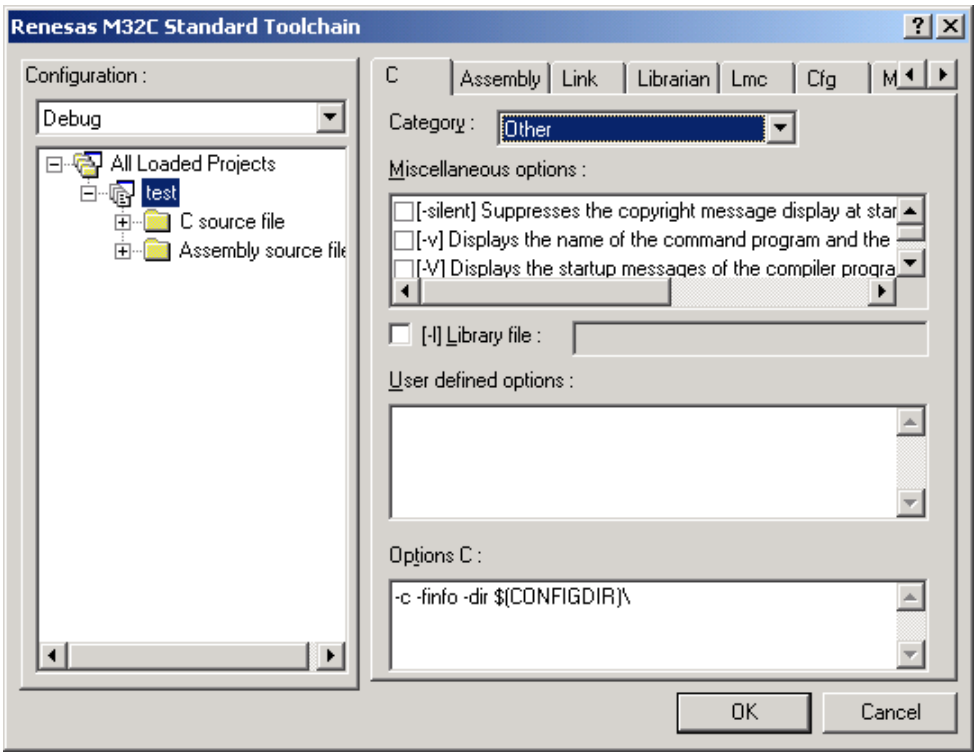


图 4.8 类别（Category）:[其他（Other）] 对话框

4.1.2 汇编程序选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [汇编（Assembly）] 标签。

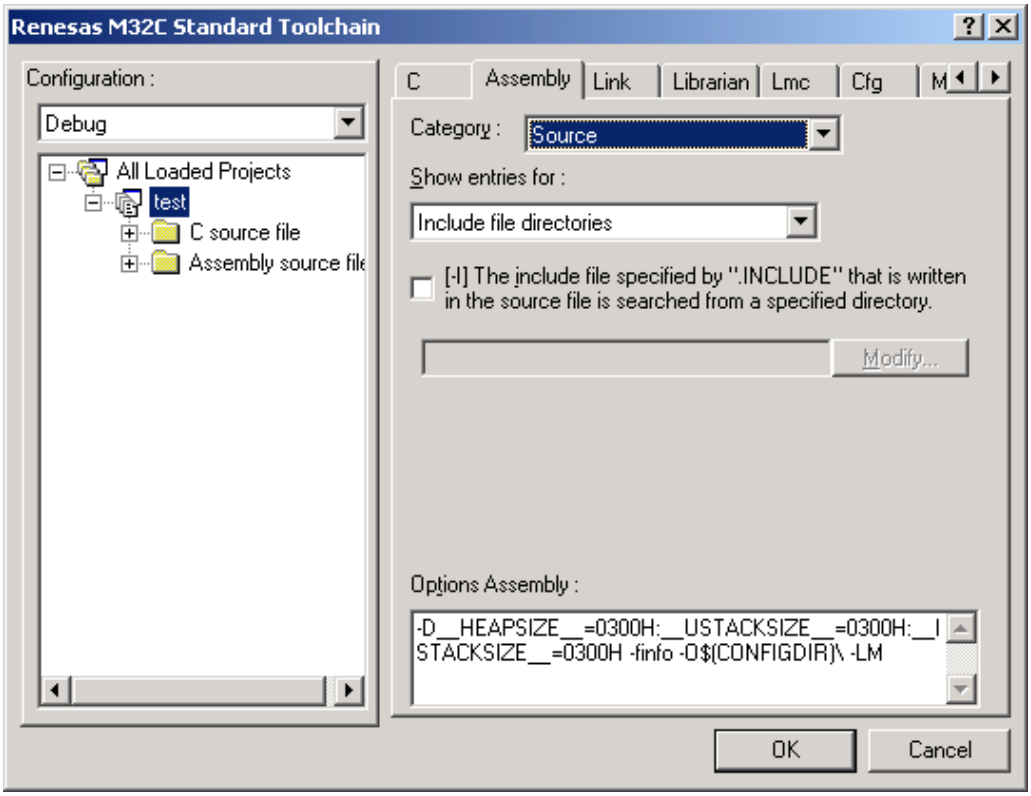


图 4.9 [汇编（Assembly）] 标签对话框

– 类别（Category）:[源（Source）]

表 4.8 类别（Category）:[源（Source）] 对话框中的项目和汇编程序选项（Assembler Option）之间的对应

对话框	选项
显示有关项目： 包含文件目录 [-I] 源文件中由“.INCLUDE”指定的包含文件将会从指定的目录搜索。	I<目录名称>
定义 [-D__HEAP__=1] 启动时禁用堆 (ncrt0.a30)。	D__HEAP__=1
[-D__STANDARD_IO__=1] 启用标准 I/O 程序库的初始化。	D__STANDARD_IO__=1
[-D] 将常数设定为符号：	D<sub> <sub> : <宏名称> [= <字符串>]

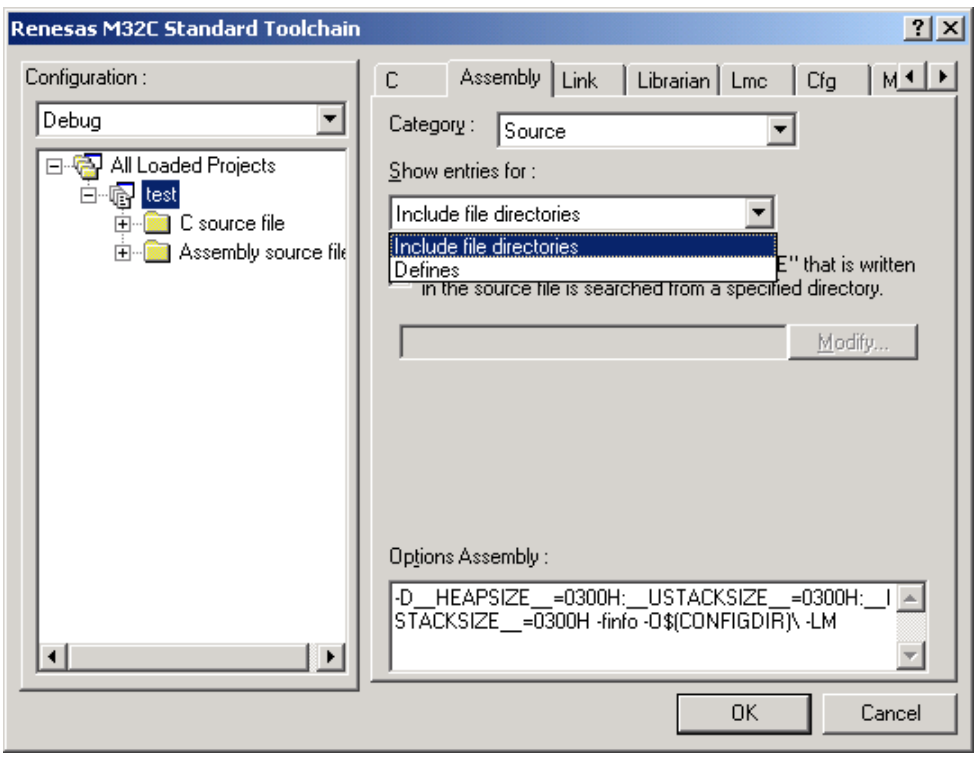


图 4.10 类别（Category）:[源（Source）] 对话框

– 类别（Category）:[目标（Object）]

表 4.9 类别（Category）:[目标（Object）] 对话框中的项目和汇编程序选项（Assembler Option）之间的对应

对话框	选项
[-S] 指定要输出的本地符号信息。	S
[-SM] 指定系统标签和本地符号信息输出。	SM
[-finfo] 生成检查器信息。	finfo
[-N] 禁止输出宏命令行信息。	N
[-mode60] 使用此参数运行 AS308 来处理以 AS30 编写的程序将允许某些代码由 AS308 来汇编。	mode60
[-mode60p] 运行有结构的处理程序 (pre30) 和处理参数 – mode60。	mode60p
[-M] 生成有结构的字节类型描述命令变量。	M
[-O] 输出文件目录：	O<目录名称>

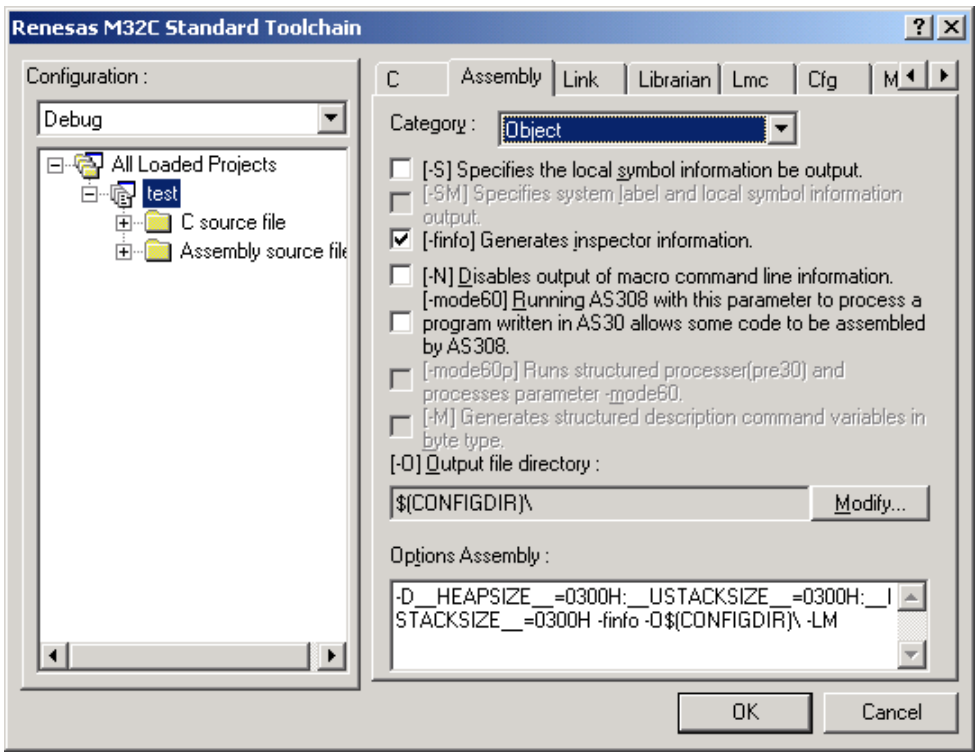


图 4.11 类别（Category）:[目标（Object）] 对话框

– 类别（Category）:[列表（List）]

表 4.10 类别（Category）:[列表（List）] 对话框中的项目和汇编程序选项（Assembler Option）之间的对应

对话框	选项
[-L] 生成汇编语言列表文件。	L
文件格式：*1	
[+C] 行连接将按原样直接输出至列表文件	LC
[+D] .DEFINE 前面的信息会被替换并输出至列表文件	LD
[+I] 在条件汇编中结果为“假”条件的偶数程序段输出至汇编程语言列表文件	LI
[+M] 偶数宏描述扩展段输出至汇编语言列表文件	LM
[+S] 偶数结构描述扩展段输出至汇编语言列表文件	LS
[-H] 标头信息不输出至汇编语言列表文件。	H
*1: “L”将自动附加到为该选项选取的项目的前面。 例如： 如果选取项目 [+C]、[+D]、[+I]，选项将是 –LCDI。	

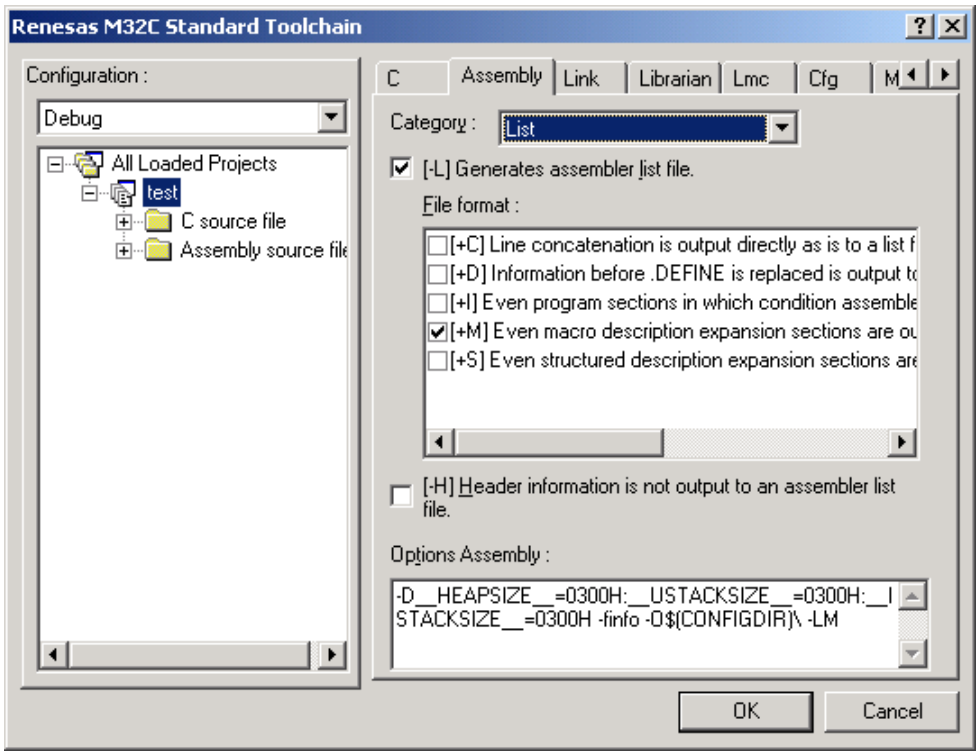


图 4.12 类别（Category）:[列表（List）] 对话框

– 类别（Category）:[调谐（Tuning）]

表 4.11 类别（Category）:[调谐（Tuning）] 对话框中的项目和汇编程序选项（Assembler Option）之间的对应

对话框	选项
[-fMST] 生成 special 页向量表。	fMST
[-fMVT] 生成变量向量表。	fMVT
[-abs16] 选取 16 位绝对寻址。	abs16
[-JOPT] 优化参考全局标签的转移工具。	JOPT
[-PATCH_TA] 避开三相电机控制的定时器函数上的警惕 No.1： 编号：	PATCH_TA[*n*] [*n*]：在“编号”中指定的数值

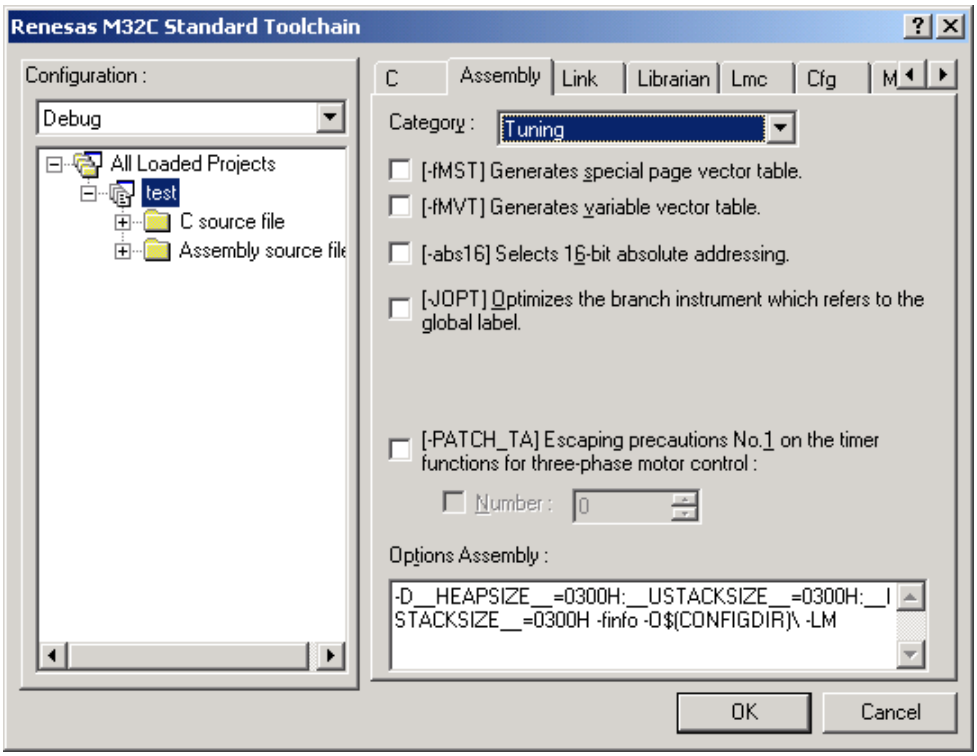


图 4.13 类别（Category）:[调谐（Tuning）] 对话框

– 类别（Category）:[其他（Other）]

表 4.12 类别（Category）:[其他（Other）] 对话框中的项目和汇编程序选项（Assembler Option）之间的对应

对话框	选项
杂项：	
[-.] 禁止信息输出至显示画面	.
[-C] 在 as30 调用 mac30 和 asp30 时标示命令行的内容	C
[-F] 将 ..FILE 扩展的文件名设定为源文件名	F
[-T] 生成汇编程序错误标签文件	T
[-V] 标示汇编程序系统程序的版本	V
用户定义的选项：	

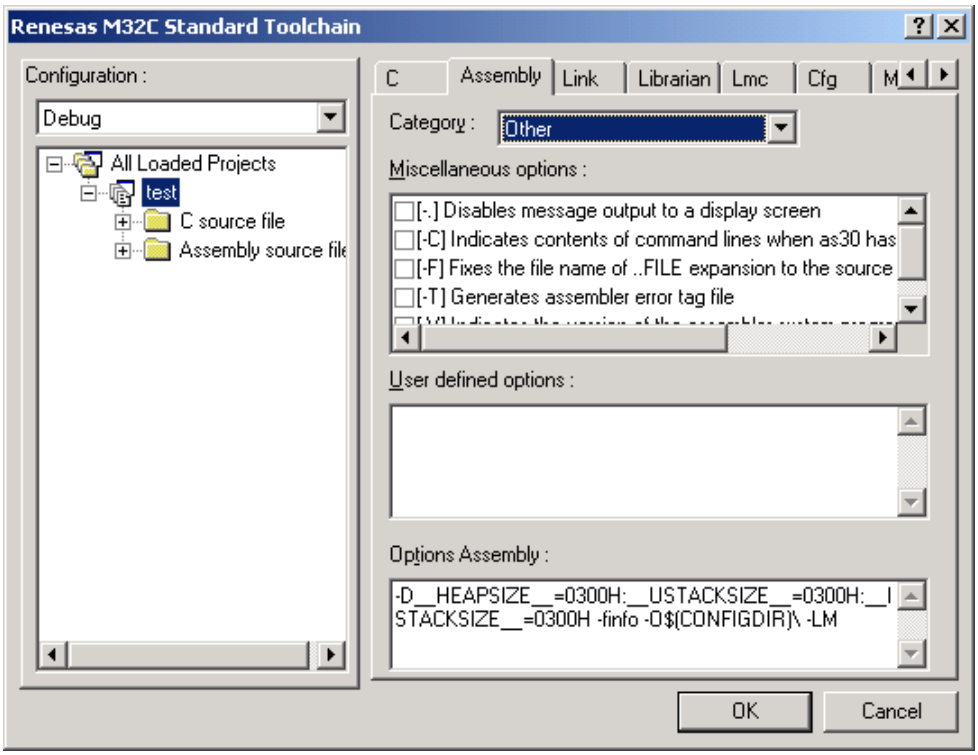


图 4.14 类别（Category）:[其他（Other）] 对话框

4.1.3 连接编辑程序选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [连接（Link）] 标签。

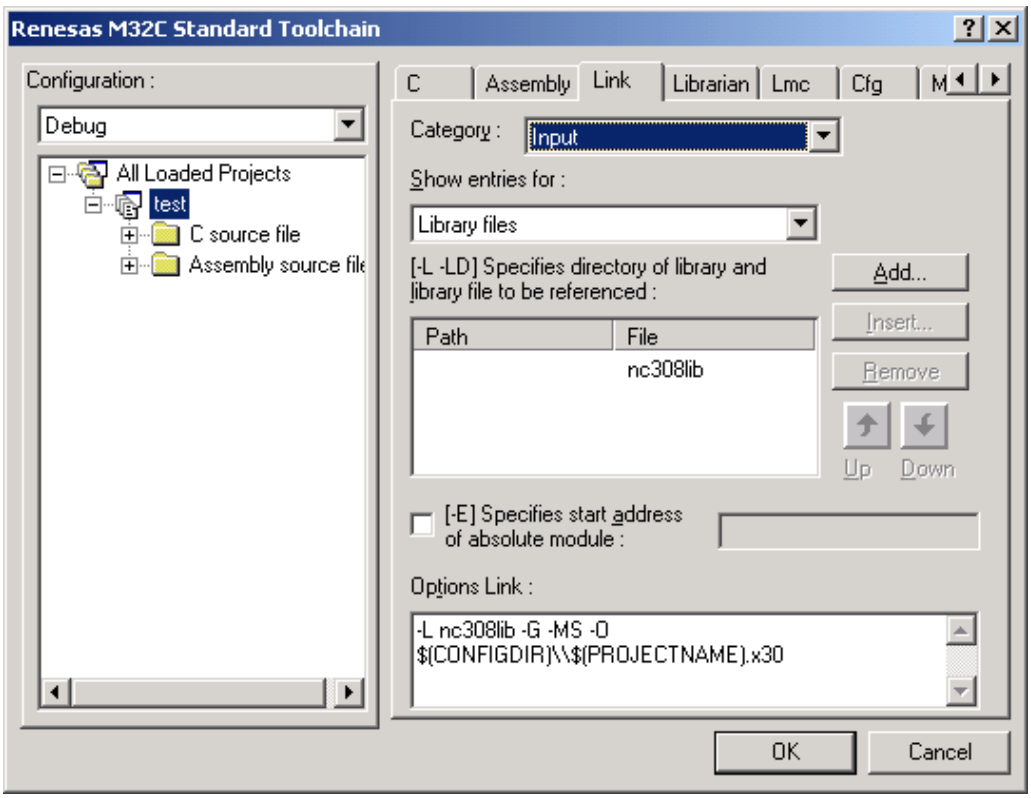


图 4.15 [连接（Link）] 标签对话框

– 类别（Category）:[输入（Input）]

表 4.13 类别（Category）:[输入（Input）] 对话框中的项目和连接编辑程序选项（Linkage Editor Option）之间的对应

对话框	选项
显示有关项目： 程序库文件	 LΔ<文件名> LDΔ<目录名称>
[-E] 指定绝对模块的起始地址：	EΔ<sub> <sub> : <数值> | <标签名称>

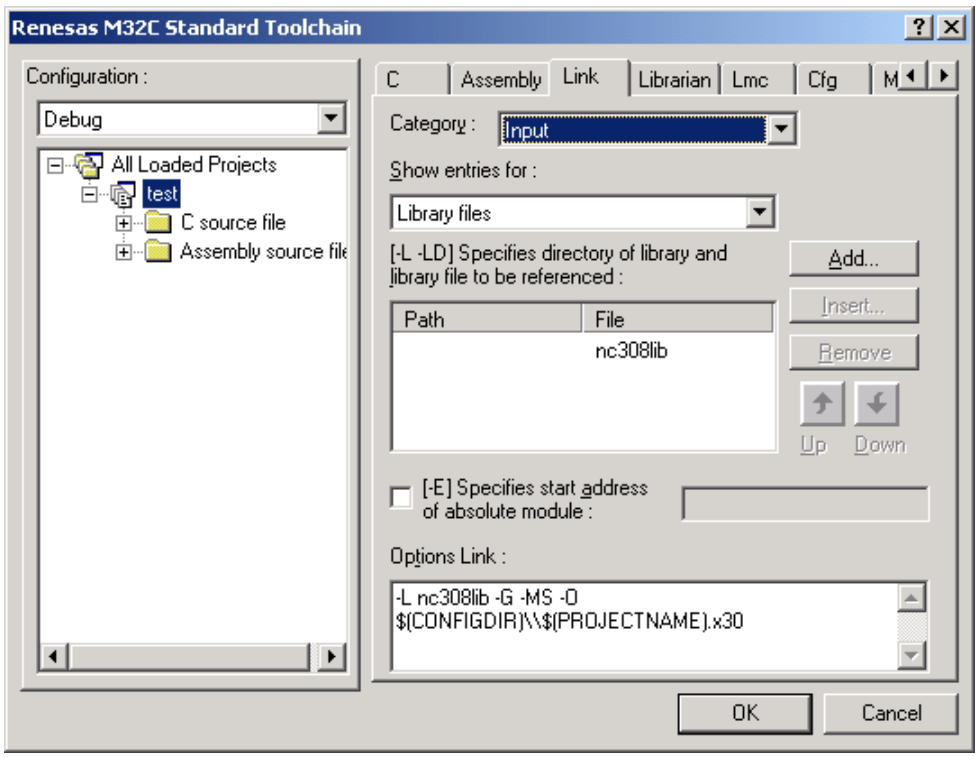


图 4.16 类别（Category）:[输入（Input）] 对话框

– 类别（Category）:[输出（Output）]

表 4.14 类别（Category）:[输出（Output）] 对话框中的项目和连接编辑程序选项（Linkage Editor Option）之间的对应

对话框	选项
[-G] 将源调试信息输出至绝对模块文件。	G
[-U] 对未使用的函数名称输出警告。	U
[-W] 连接处理会在连接的时候出现警告时停止。	W
生成映像文件： 无 [-M] 生成映像文件 [-MS] 包含符号信息 [-MSL] 包含符号信息的全名	 - M MS MSL
[-O] 指定绝对模块文件名：	OΔ<文件名>

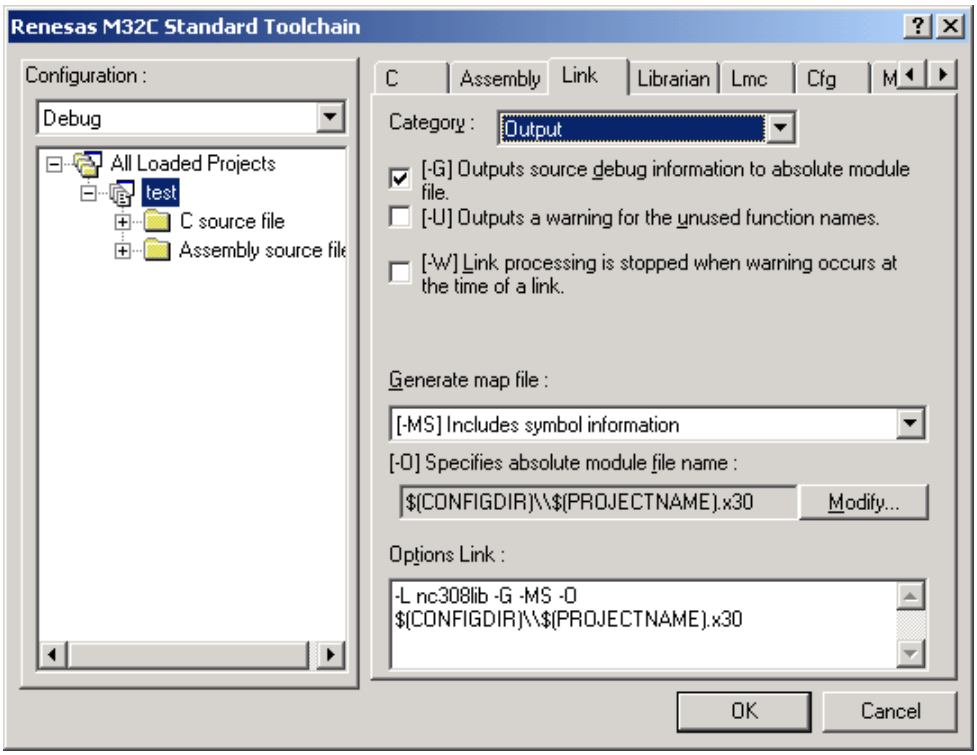


图 4.17 类别（Category）:[输出（Output）] 对话框

– 类别（Category）:[调谐（Tuning）]

表 4.15 类别（Category）:[调谐（Tuning）] 对话框中的项目和连接编辑程序选项（Linkage Editor Option）之间的对应

对话框	选项
[-fMST] 生成 special 页向量表。	fMST
[-fMVT] 生成变量向量表。	fMVT
[-VECT] 设定执行自动生成变量向量表之后所产生的可用区域的地址。	VECTΔ<sub> <sub> : <数值> / <标签名称>
[-JOPT] 优化参考全局标签的转移工具。	JOPT

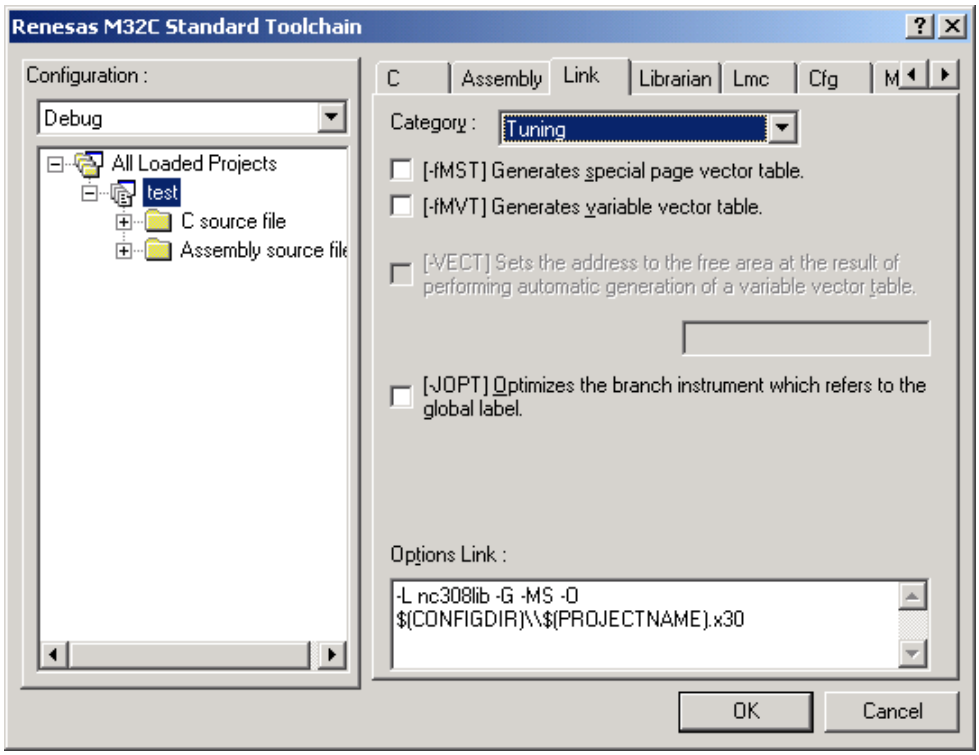


图 4.18 类别（Category）:[调谐（Tuning）] 对话框

– 类别（Category）:[段（Section）]

表 4.16 类别(Category):[段(Section)] 对话框中的项目和连接编辑程序选项(Linkage Editor Option)之间的对应

对话框	选项
显示有关项目：	
段顺序	ORDERΔ<sub>[,…] <sub> : <段名称>[=地址]
段位置	LOCΔ<sub>[,…] <sub> : <段名称>=<地址>

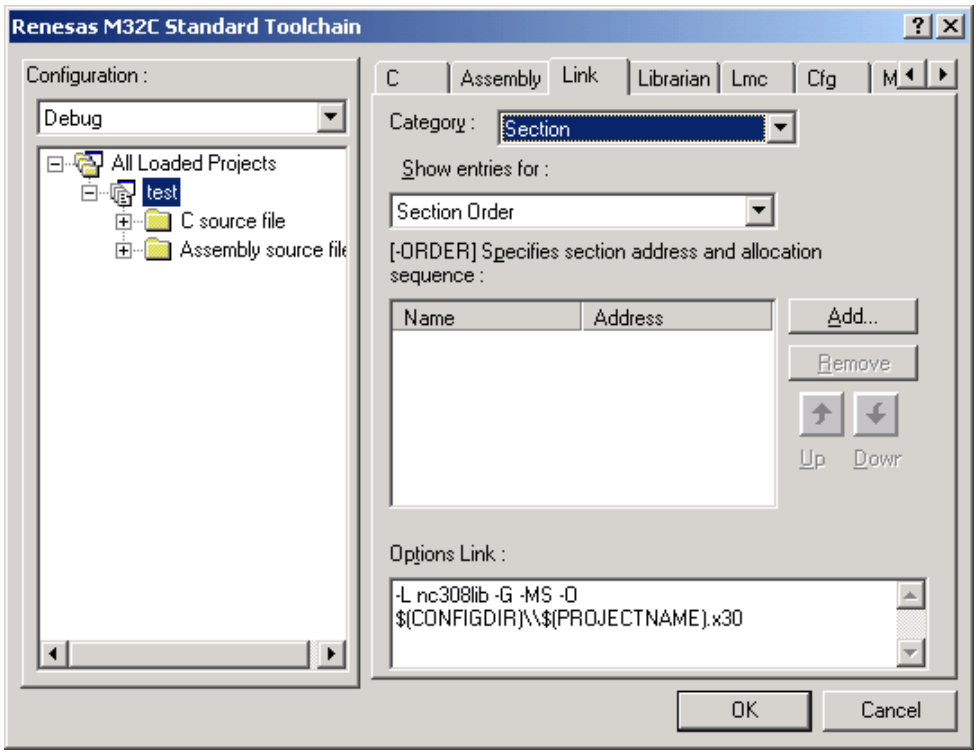


图 4.19 类别（Category）:[段（Section）] 对话框

– 类别（Category）:[其他（Other）]

表 4.17 类别（Category）:[其他（Other）] 对话框中的项目和连接编辑程序选项（Linkage Editor Option）之间的对应

对话框	选项
杂项：	
[-.] 禁止信息输出至画面	.
[-NOSTOP] 将所遇到的所有错误输出至画面	NOSTOP
[-T] 生成连接错误标签文件	T
[-V] 标示连接编辑程序的版本号	V
用户定义的选项：	

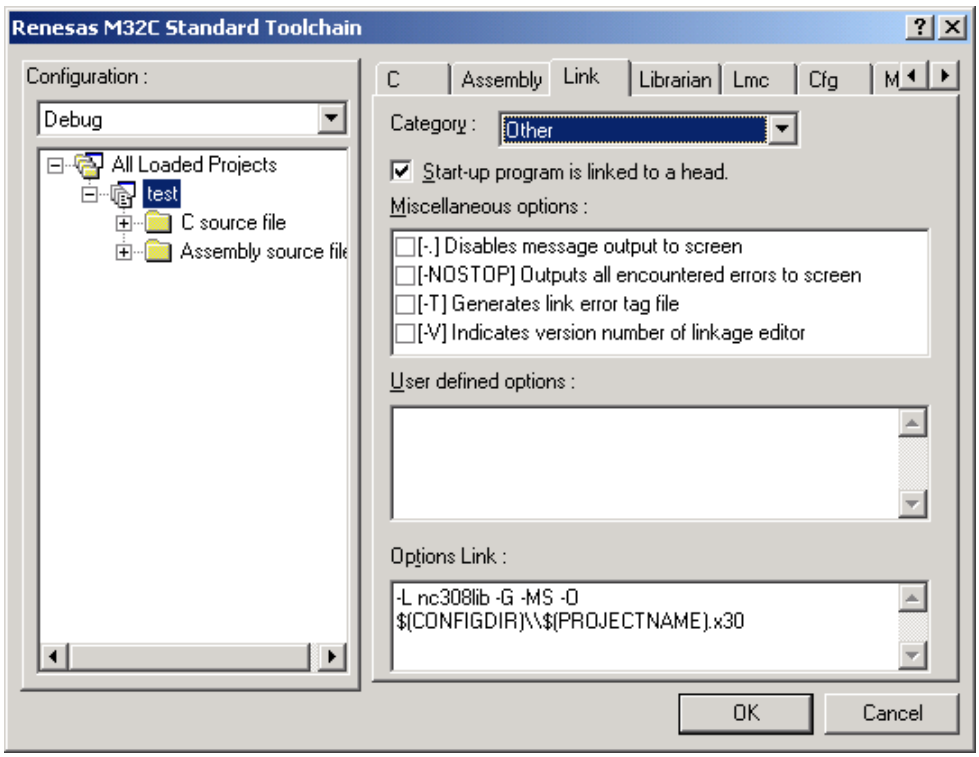


图 4.20 类别（Category）:[其他（Other）] 对话框

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

表 4.18 类别（Category）:[子命令文件（Subcommand file）] 对话框中的项目和连接编辑程序选项（Linkage Editor Option）之间的对应

对话框	选项
[@] 使用外部子命令文件。	@<文件名>

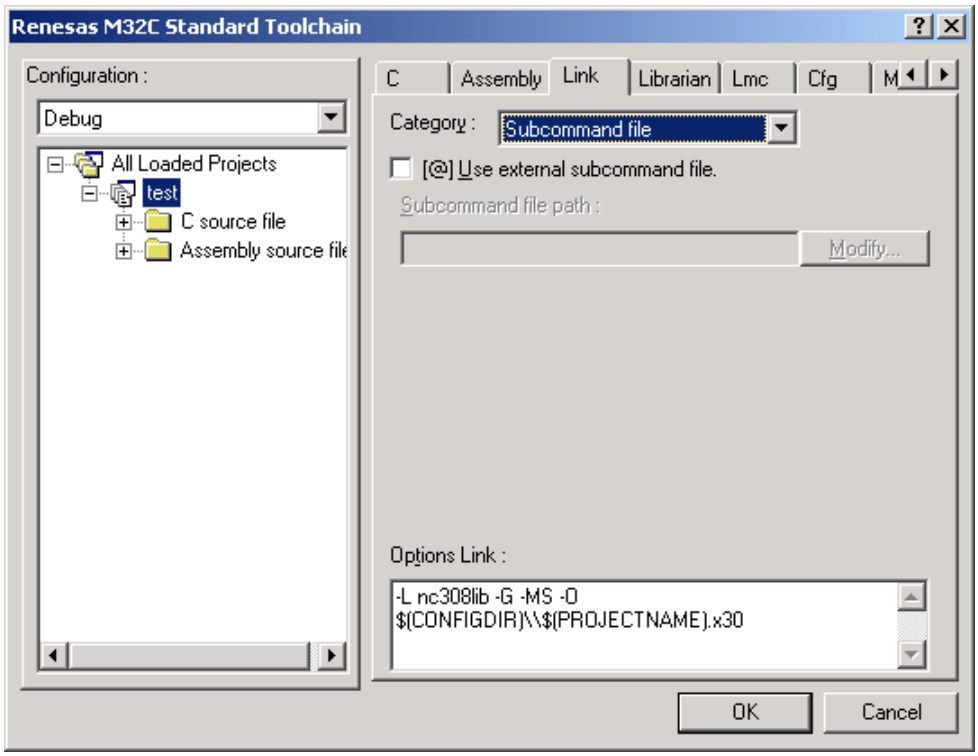


图 4.21 类别（Category）:[子命令文件（Subcommand file）] 对话框

4.1.4 库管理程序选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [库管理程序（Librarian）] 标签。

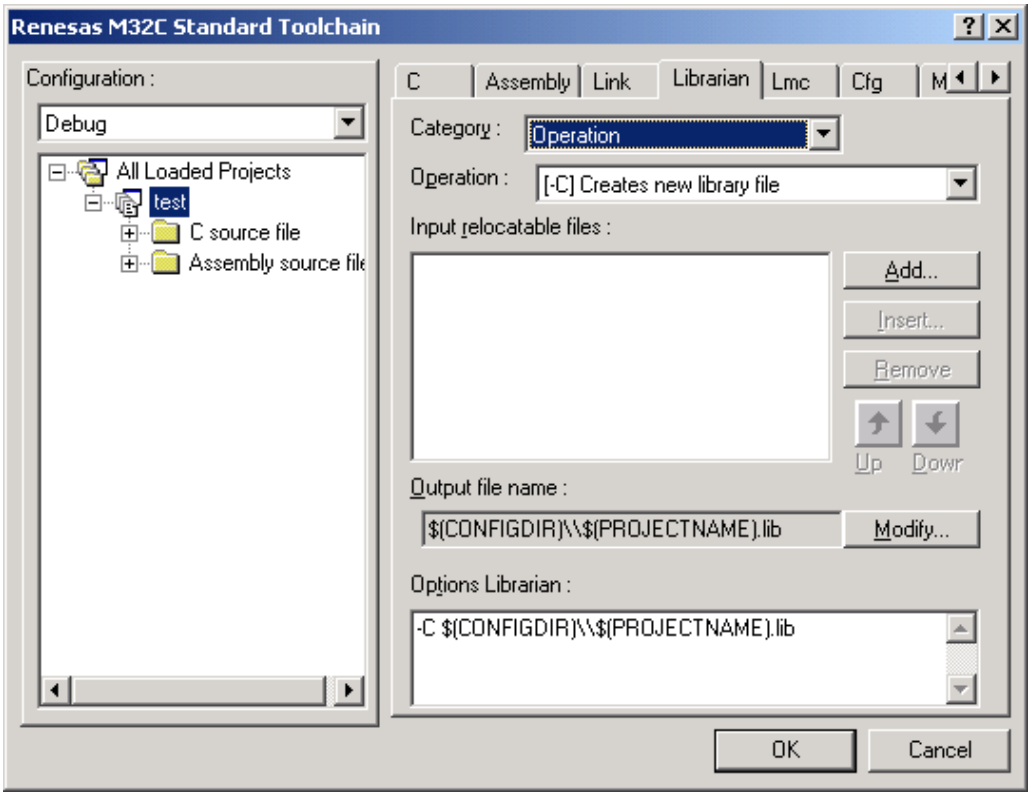


图 4.22 [库管理程序（Librarian）] 标签对话框

– 类别（Category）:[操作（Operation）]

表 4.19 类别（Category）:[操作（Operation）] 对话框中的项目和库管理程序选项（Librarian Option）之间的对应

对话框	选项
操作（Operation）	
[-A] 添加模块到程序库文件	A
[-C] 建立新程序库文件	C
[-D] 从程序库文件删除模块	D
[-L] 生成程序库列表文件	L
[-R] 替换模块	R
[-U] 更新模块	U
[-X] 提取	X

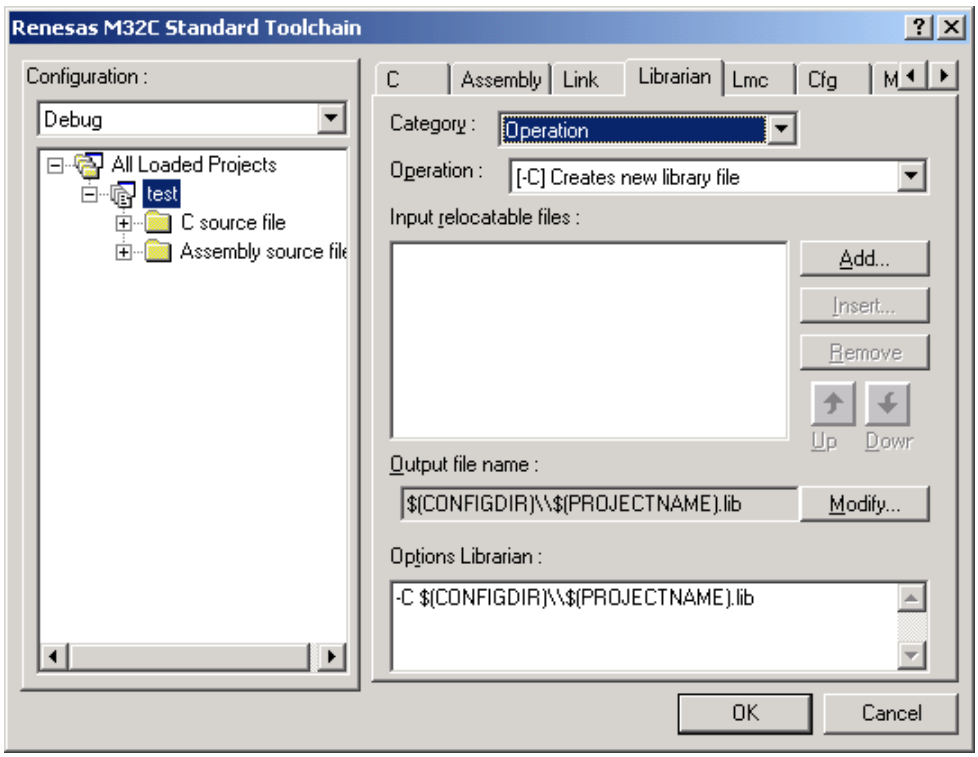


图 4.23 类别（Category）:[操作（Operation）] 对话框

– 类别（Category）:[其他（Other）]

表 4.20 类别（Category）:[其他（Other）] 对话框中的项目和库管理程序选项（Librarian Option）之间的对应

对话框	选项
杂项：	
[-.] 禁止信息输出至画面	.
[-V] 标示库管理程序的版本	V
用户定义的选项：	

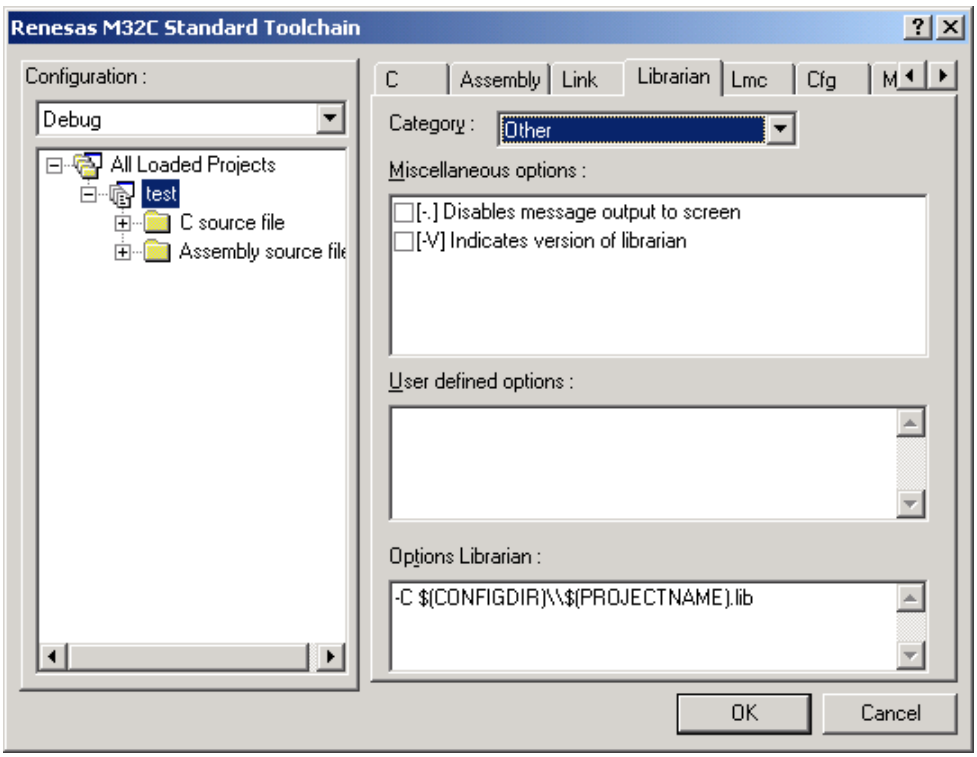


图 4.24 类别（Category）:[其他（Other）] 对话框

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

表 4.21 类别（Category）:[子命令文件（Subcommand file）] 对话框中的项目和库管理程序选项（Librarian Option）之间的对应

对话框	选项
[@] 使用外部子命令文件。	@<文件名>

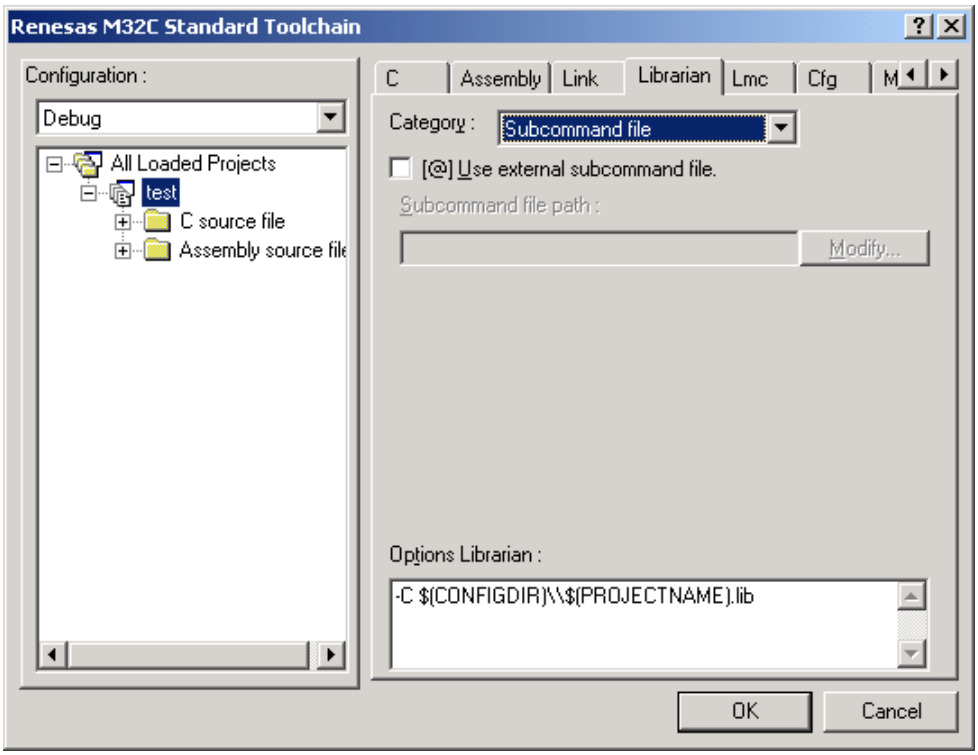


图 4.25 类别（Category）:[子命令文件（Subcommand file）] 对话框

4.1.5 装入模块转换器选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [Lmc] 标签。

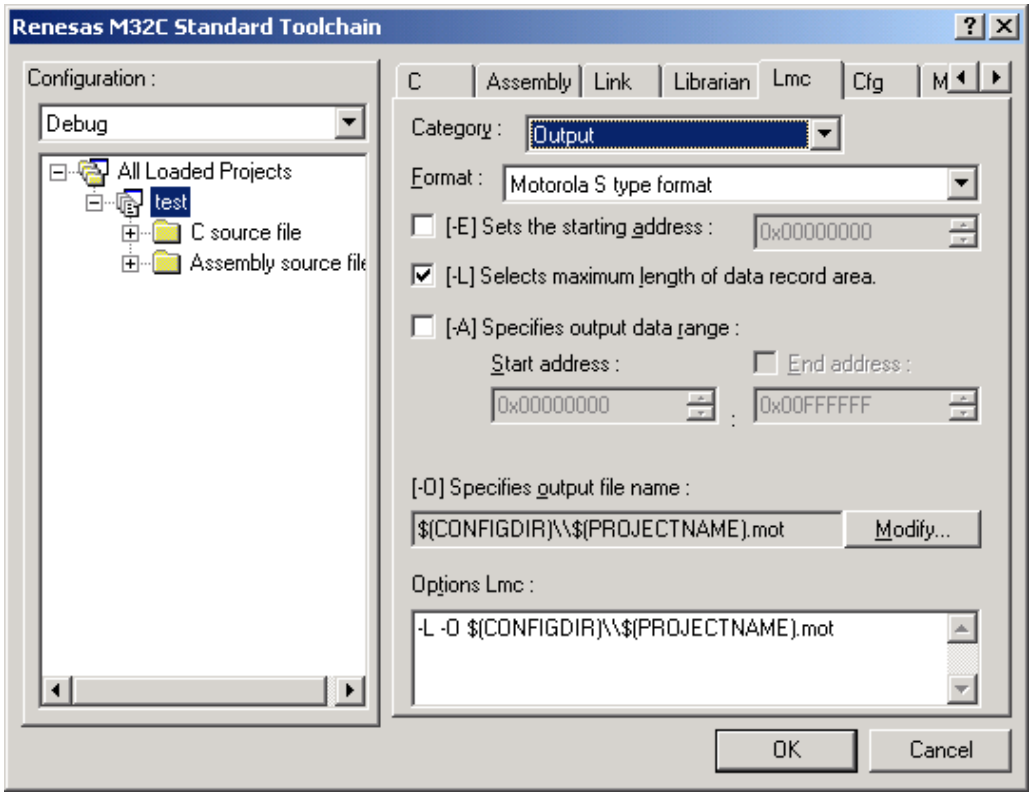


图 4.26 [Lmc] 标签对话框

– 类别（Category）:[输出（Output）]

表 4.22 类别（Category）:[输出（Output）] 对话框中的项目和装入模块转换器选项（Load Module Converter Option）之间的对应

对话框	选项
格式： Motorola S 型格式 [-H] 转换文件信息 Intel HEX 格式	 - H
[-E] 设定起始地址：	EΔ<地址>
[-L] 选取数据记录区的最大长度。	L
[-A] 指定输出数据范围：	AΔ<起始地址>[:终止地址]
[-O] 指定输出文件名：	OΔ<文件名>

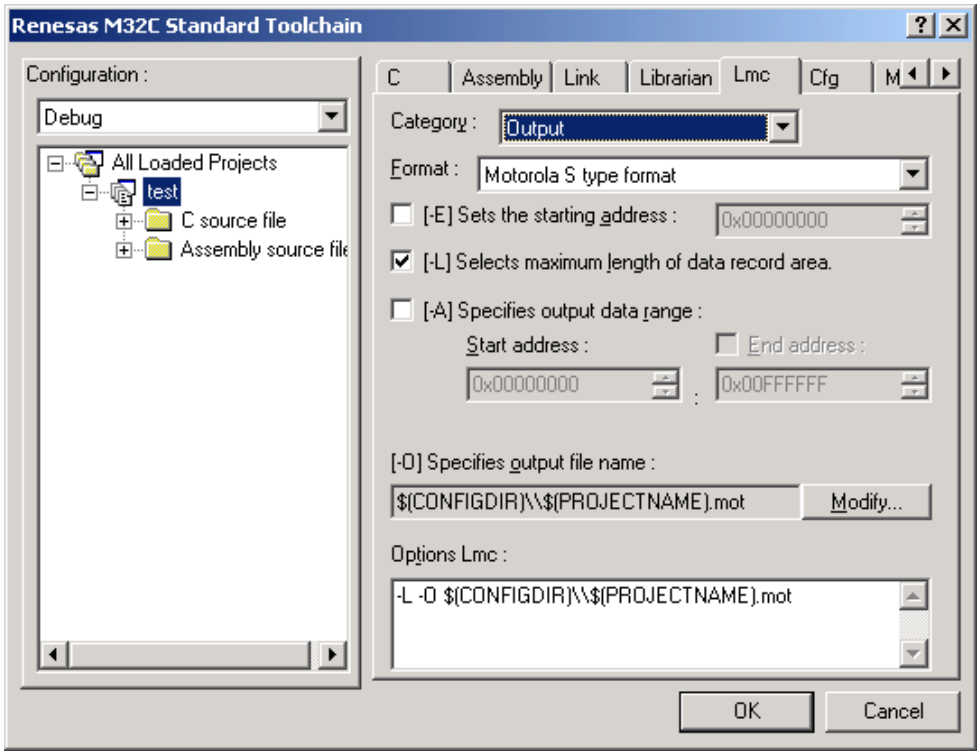


图 4.27 类别（Category）:[输出（Output）] 对话框

– 类别（Category）:[代码（Code）]

表 4.23 类别（Category）:[代码（Code）] 对话框中的项目和装入模块转换器选项（Load Module Converter Option）之间的对应

对话框	选项
[-ID] ID 代码检查 ID 代码设定：	ID[sub] [sub] : <代码保护设定> | <#数值>
ROM 代码保护函数： [-protect1] 级别 1 设定 [-protect2] 级别 2 设定 [-protectx] 设定 ROM 代码保护值	 protect1 protect2 protectxΔ<数值>
[-F] 设定可用区域中的填充数据：	FΔ<可用区域中设定的数据值>[sub] [sub] : <:起始地址>[:终止地址]

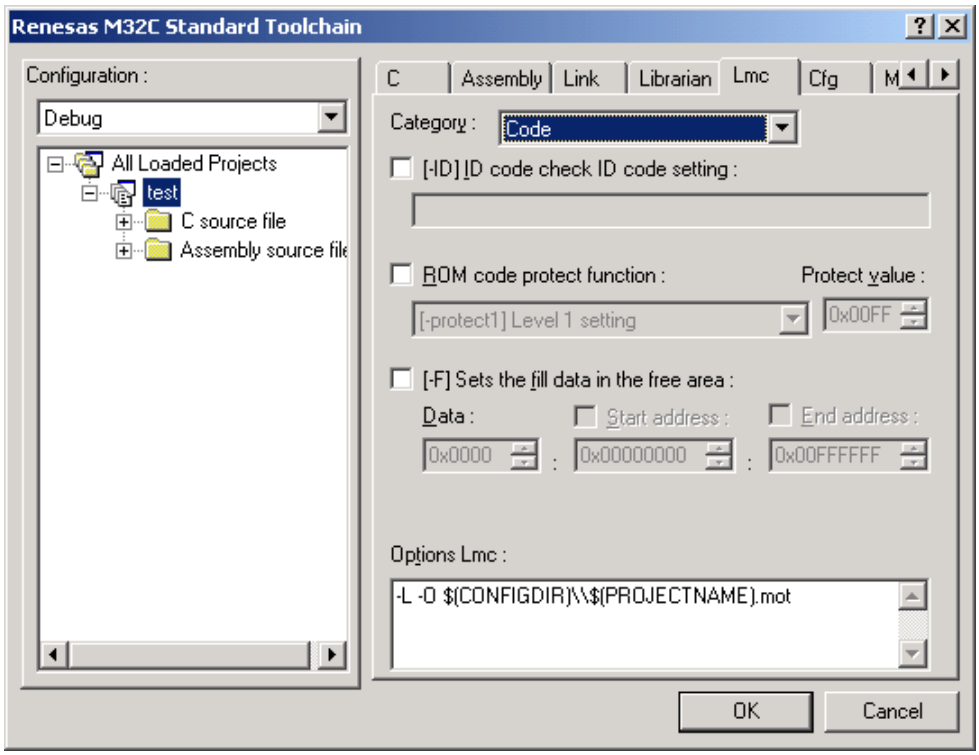


图 4.28 类别（Category）:[代码（Code）] 对话框

– 类别（Category）:[其他（Other）]

表 4.24 类别（Category）:[其他（Other）] 对话框中的项目和装入模块转换器选项（Load Module Converter Option）之间的对应

对话框	选项
杂项：	
[-.] 禁止信息输出至画面	.
[-V] 标示装入模块转换器的版本	V
用户定义的选项：	

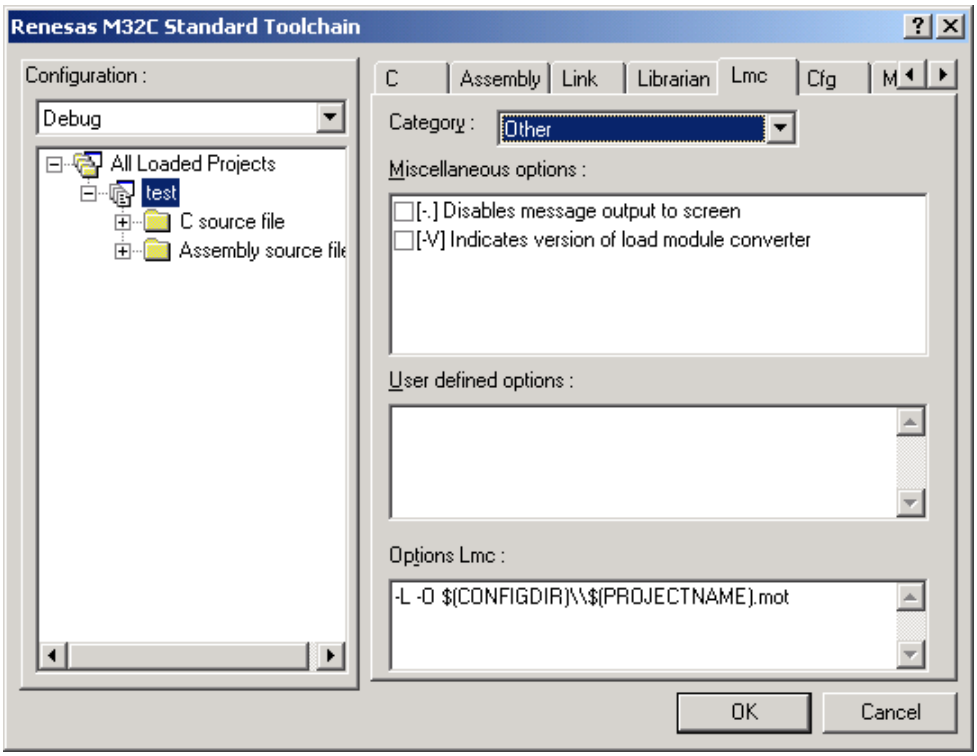


图 4.29 类别（Category）:[其他（Other）] 对话框

4.1.6 配置选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [Cfg] 标签。

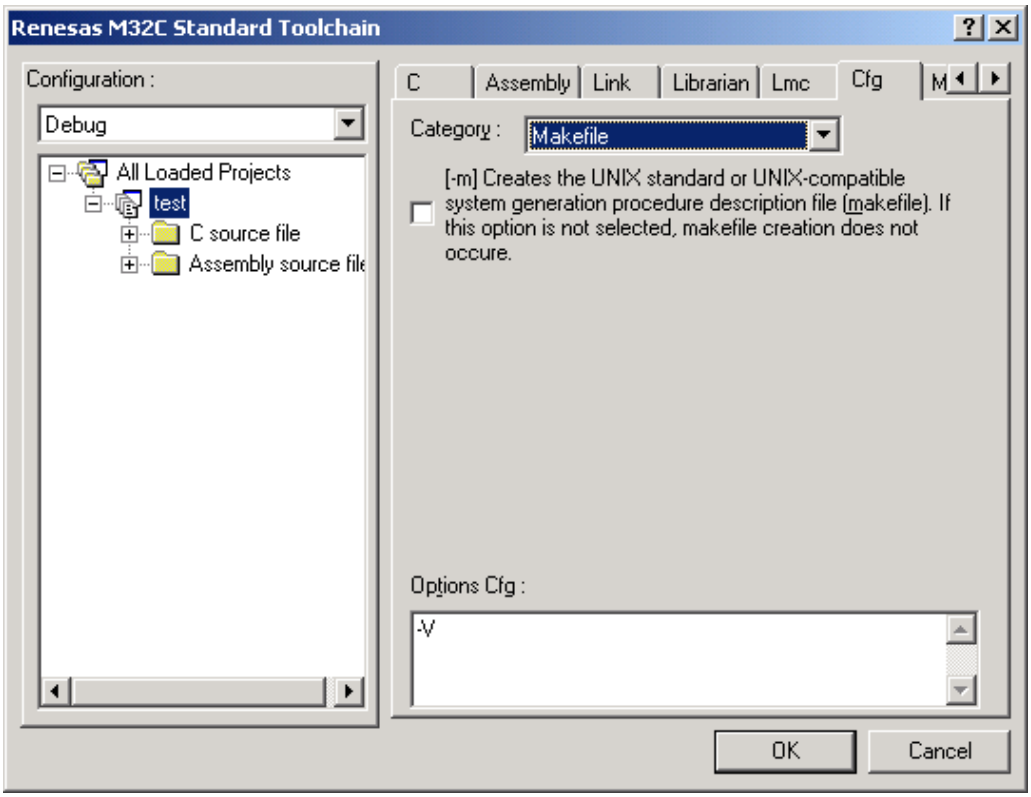


图 4.30 [Cfg] 标签对话框

– 类别（Category）:[命令描述文件（Makefile）]

表 4.25 类别（Category）:[命令描述文件（Makefile）] 对话框中的项目和配置程序选项（Configurator Option）之间的对应

对话框	选项
[-m] 建立 UNIX 标准或 UNIX 兼容的生成过程描述文件（即命令描述文件）。	m

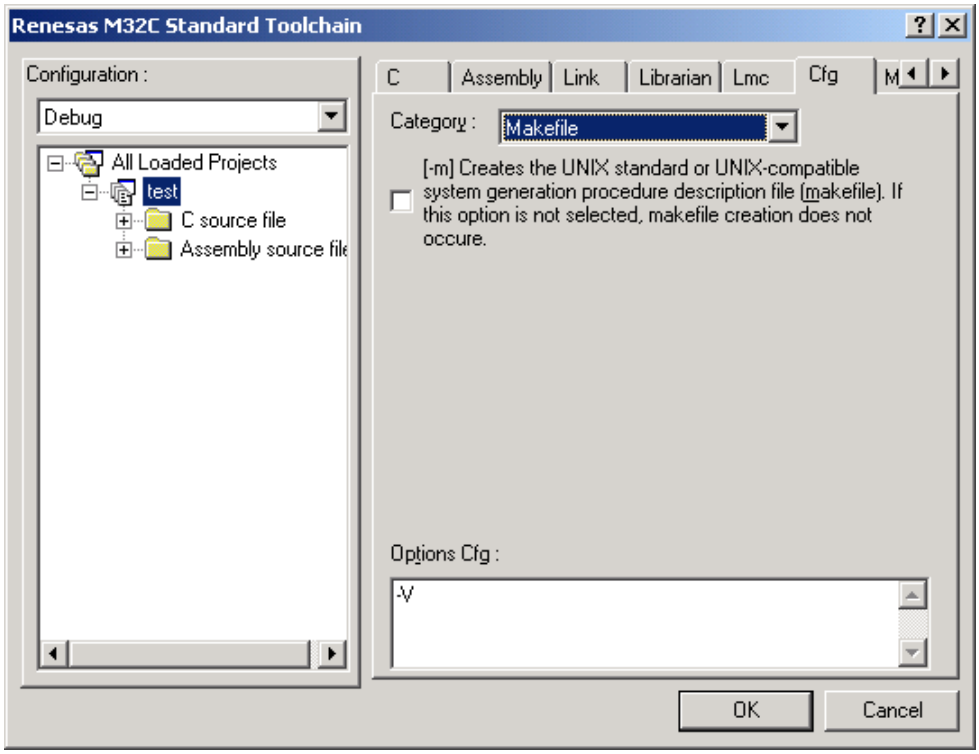


图 4.31 类别（Category）:[命令描述文件（Makefile）] 对话框

– 类别（Category）:[其他（Other）]

表 4.26 类别（Category）:[其他（Other）] 对话框中的项目和配置程序选项（Configurator Option）之间的对应

对话框	选项
杂项：	
[-V] 显示命令所生成之文件上的信息	V
[-v] 显示命令选项描述和版本中的详尽信息	v
用户定义的选项：	

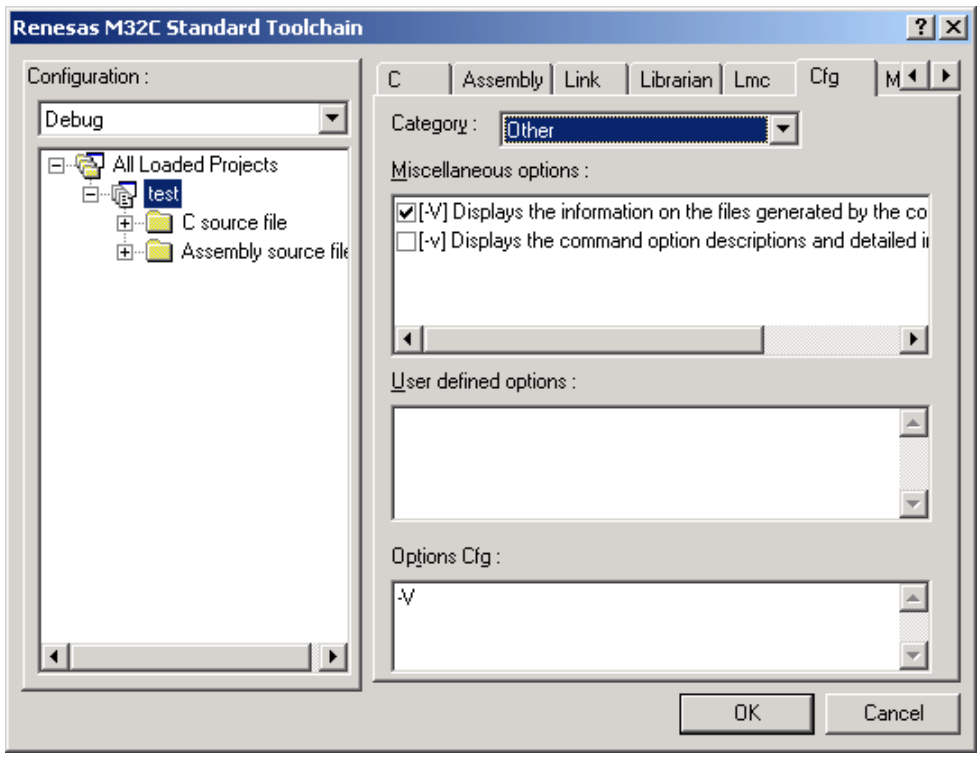


图 4.32 类别（Category）:[其他（Other）] 对话框

4.1.7 CPU 选项

在 [瑞萨 M32C 标准工具链（Renesas M32C Standard Toolchain）] 对话框中选取 [CPU] 标签。

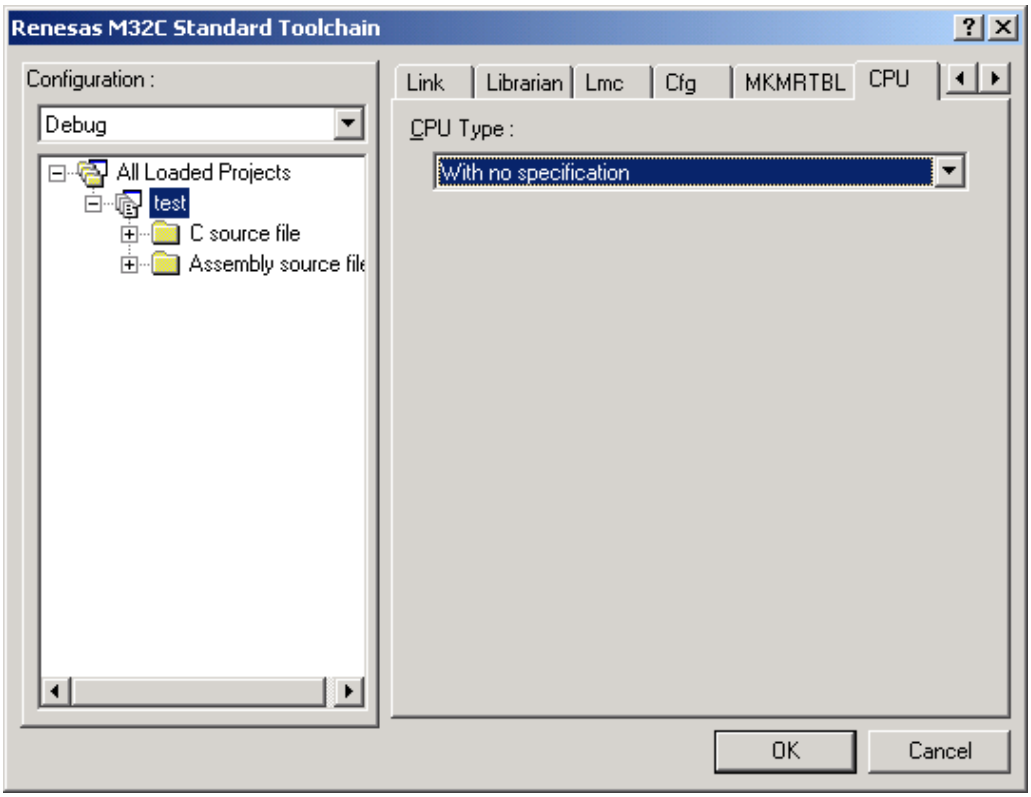


图 4.33 [CPU] 标签对话框

表 4.27 [CPU] 标签对话框中的项目和编译程序选项（Compiler Option）之间的对应

对话框	选项
CPU 类型：	
不具备规格	-
生成 M32C/80 系列的代码	M82

4.2 创建

4.2.1 命令描述文件的输出

■ 描述：

HEW 可以让您根据当前的选项设定来建立命令描述文件。

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。

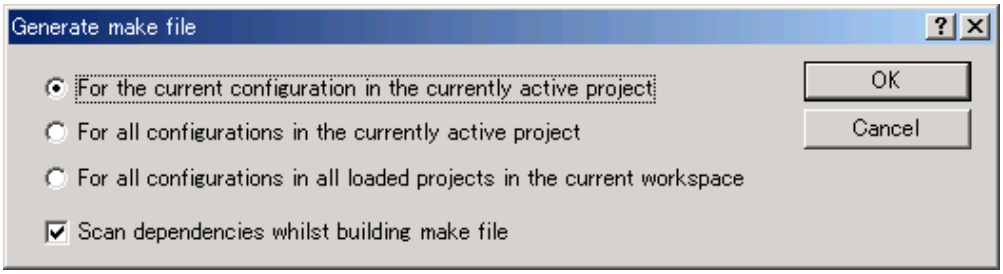
■ 如何生成命令描述文件：

– 确保生成命令描述文件的工程是当前工程。

– 确保创建工程的创建配置是当前配置。

– 选择 [创建 (Build) > 生成命令描述文件 (Generate Makefile)]。

– 以下对话框将会显示。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建立“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，可以使用 HEW 安装目录（例如 c: \hew3）内的可执行文件 HMAKE.EXE 执行。然而，用户修改的命令描述文件将不能执行。

■ 如何执行命令描述文件：

1. 打开 [命令 (Command)] 窗口，然后前往包含所生成的命令描述文件的“命令描述”(make) 目录。

2. 执行 HMAKE。在命令行上，输入 HMAKE.EXE <命令描述文件名称>。

4.2.2 命令描述文件的输入

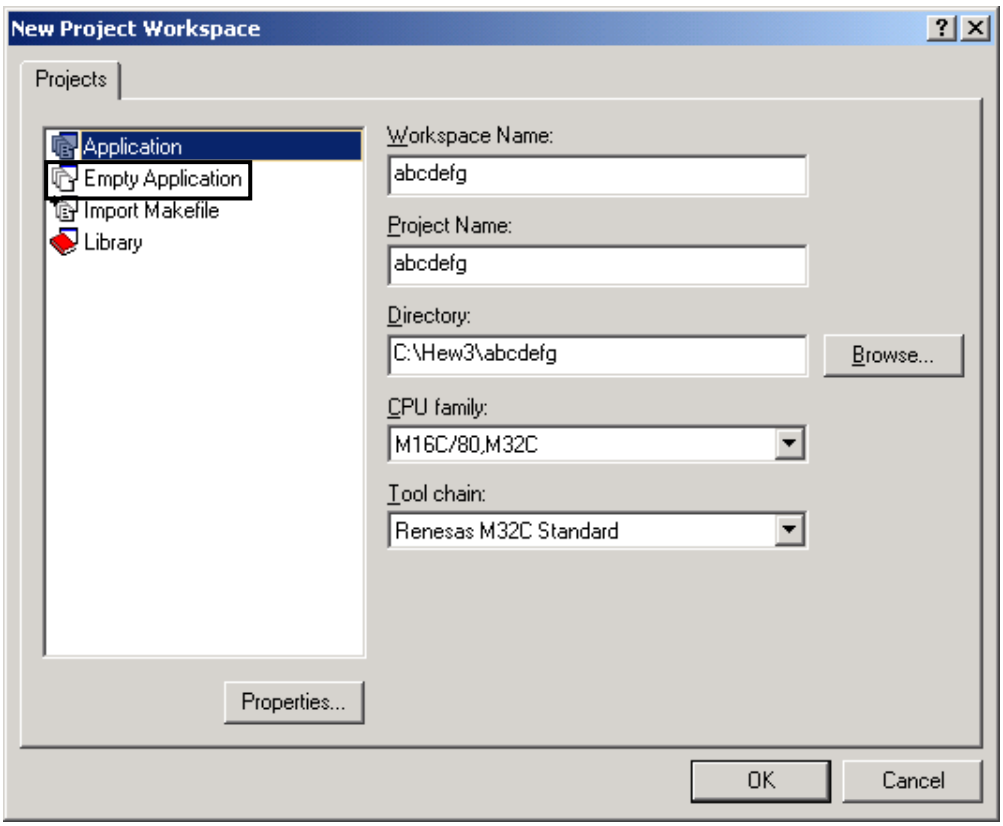
■ 描述：

HEW 可以让您输入由 HEW 生成或在 UNIX 环境中使用的命令描述文件。

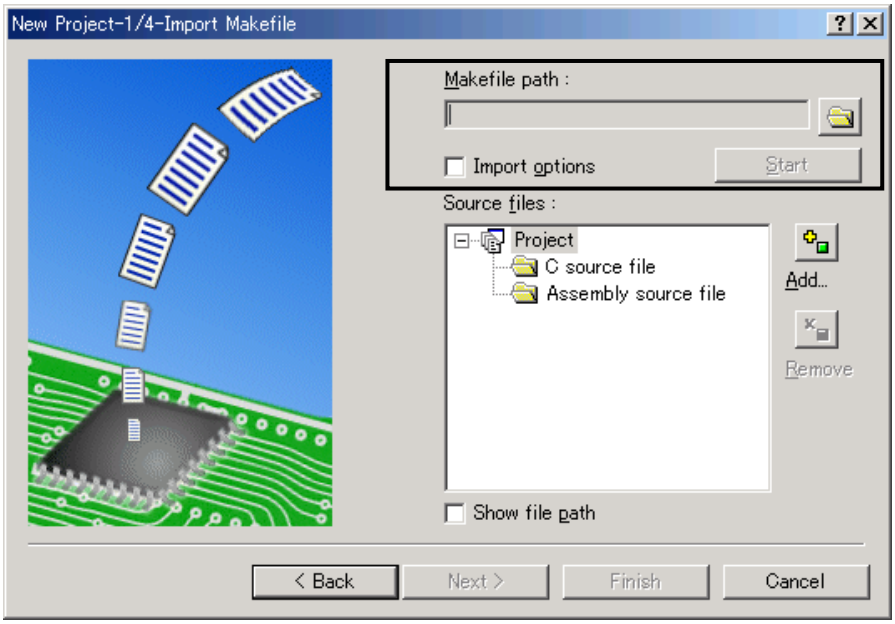
从命令描述文件，您可自动获取工程的**文件结构**。（不过，您不能获取选项设定或类似的规格。） 这方便了从命令行至 HEW 的转移。

■ 如何输入命令描述文件：

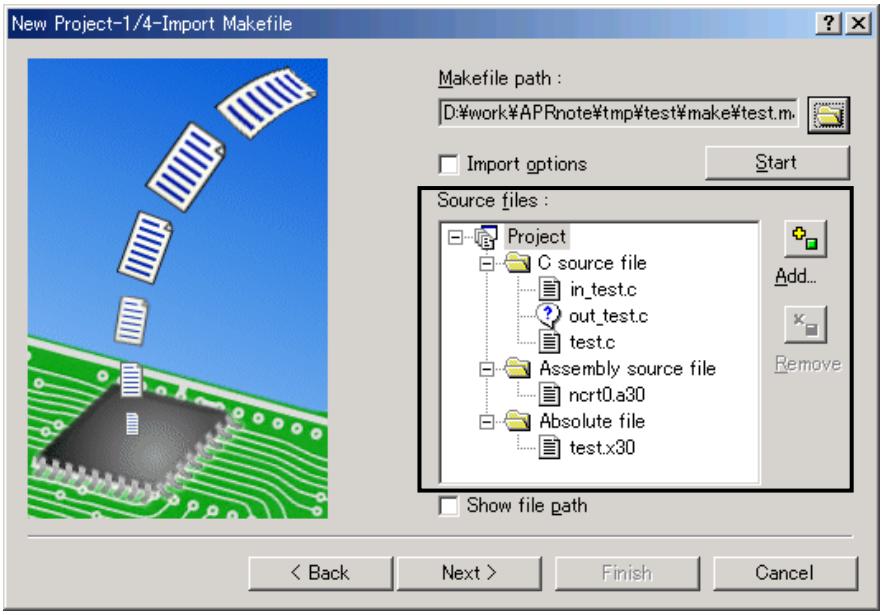
1. 当建立新的工作空间时，在 [新的工程工作空间 (New Project Workspace)] 对话框内，从工程类型选项选取 [导入命令描述文件 (Import Makefile)]。



2. 在 [新的工程－导入命令描述文件 (New Project-Import Makefile)] 对话框内的 [命令描述文件路径 (Makefile path)] 字段中，指定命令描述文件路径，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。在此结构图中，任何含有 标志的文件都经过分析以确定它不含任何实体。此文件将不会添加到工程（它会被忽略）。



4. 跟随向导，指定 CPU 和其他选项，然后打开工作空间。之后您就可以开始进行开发工作。

4.2.3 建立定制的工程类型

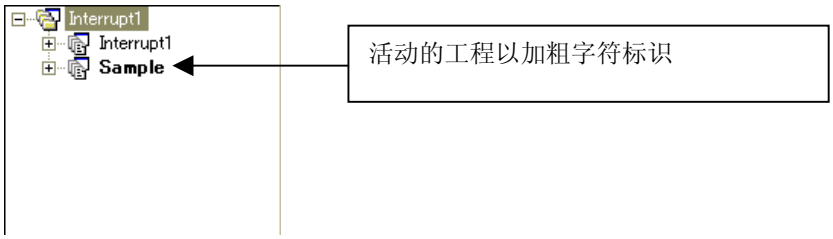
■ 描述：

此功能允许用户使用由其他用户建立的工程，并在其他机器上作为程序开发的模板。

该模板可能包含关于工程的所有信息，包括工程文件结构、创建选项，以及调试程序设定。

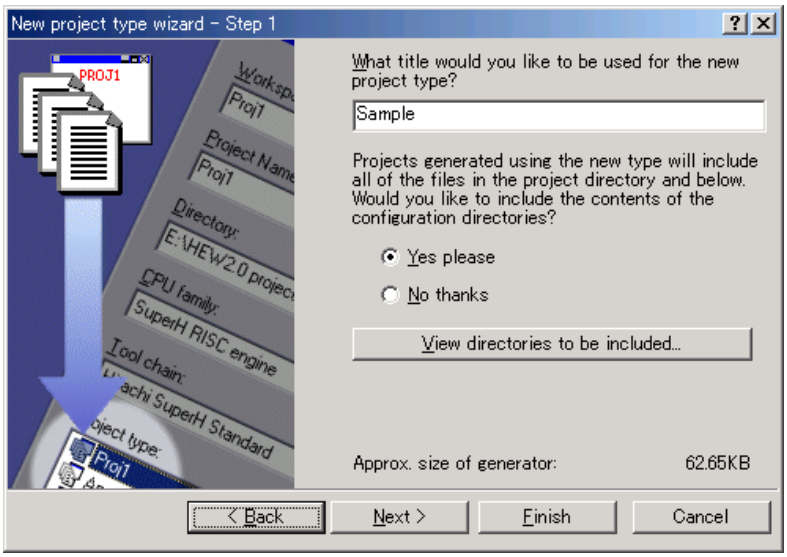
■ 如何保存工程类型：

1. 启动目标工程。这是因为关于在打开工作空间时活动的工程之信息将会保存。要启动工程，请选择 [工程 (Project) -> 设定当前工程 (Set Current Project)] 然后选取工程。



2. 选择 [工程 (Project) -> 建立工程类型 (Create Project Type)]。以下工程类型向导将会显示。输入您要用作模板的工程类型的名称，然后指定是否要将包含后创建可执行文件的配置目录和其他资源包含在模板中。

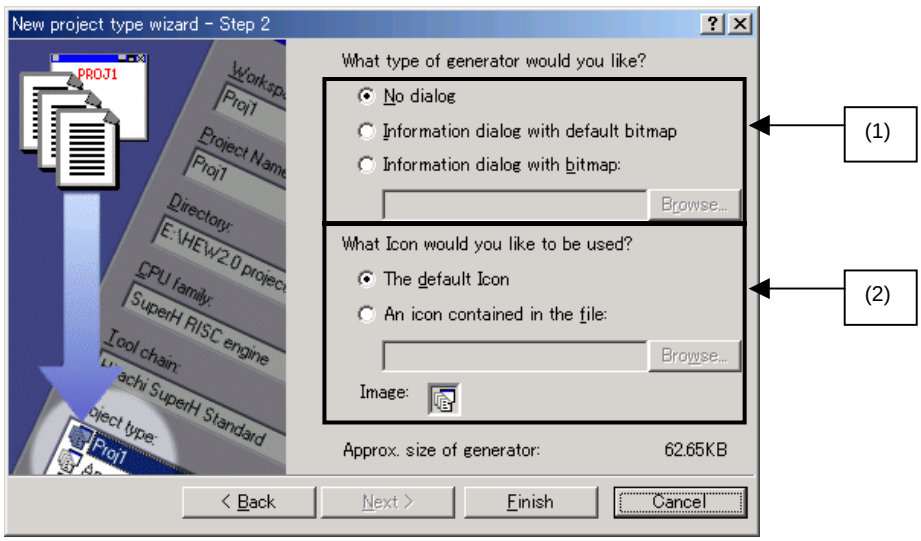
   在这里，您可以单击 [完成 (Finish)] 按钮以退出工程类型向导。



3. 在 [新的工程类型向导 － 步骤 1 (New project type wizard － Step 1)] 对话框中，单击 [下一步 (Next)] 按钮。以下向导将会显示。在以下的（1）中，指定在打开工程类型模板 时是否要显示工程信息和点阵图。

   在（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。

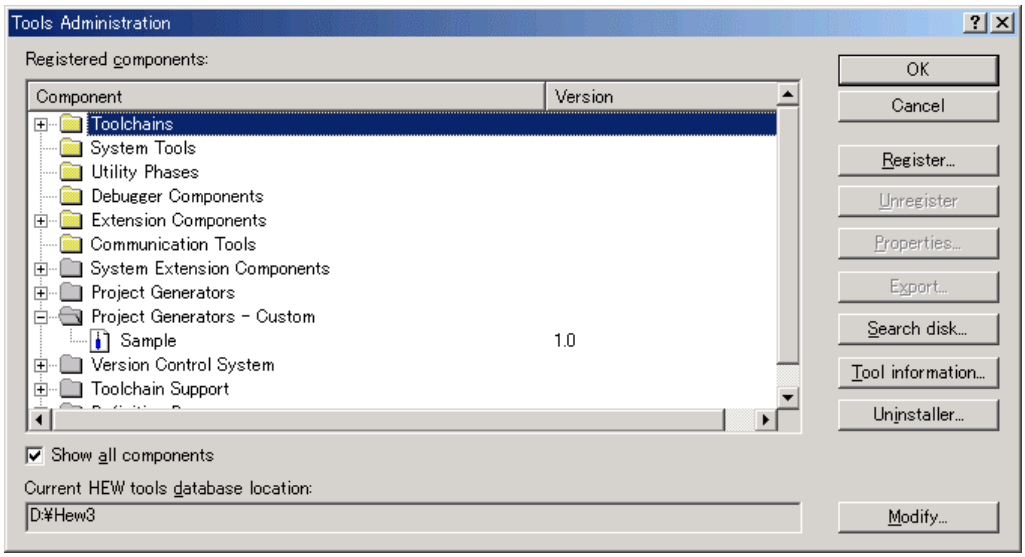
   这些设定并不是强制性的。



4. 上述操作将建立一个命名为“定制工程生成程序” (Custom Project Generator)的工程类型模板。要在其他机器上使用此模板，请选择 [工具 (Tools) -> 管理 (Administration)]。以下对话框将会显示。

   选取 [显示全部组件 (Show all components)] 复选框以显示 [工程生成程序 － 定制 (Project Generators － Custom)] 文件夹。

   在此文件夹中，单击已建立的工程类型，然后单击 [导出... (Export...)] 按钮。



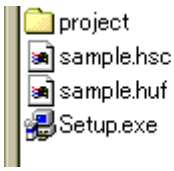
以下对话框将会显示。选取您要用来输出定制工程生成程序模板的目录。该目录必须是空的。此操作将完成工程类型的保存。

■ 安装定制工程生成程序：

以下显示如何将上述步骤建立的“定制工程生成程序”模板安装到其他机器上。

1. 以下安装环境将会在*如何保存工程类型*步骤 5 中所建立的目录中建立：

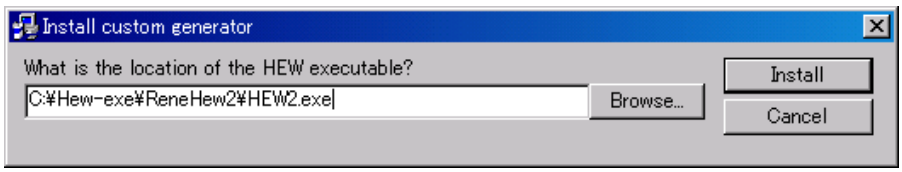
（安装环境目录）



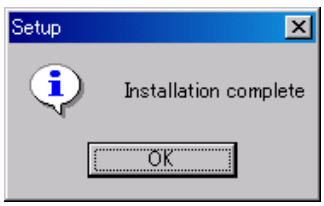
2. 复制上述安装环境，再将副本安装到其他机器上。

当您运行 Setup.exe 时，以下对话框将会显示。指定要安装 HEW 的位置，然后单击 [安装(Install)] 按钮。

（目录实例： C:\Hew-exe\ReneHew2\HEW2.exe）



3. 此操作将完成环境的创建。

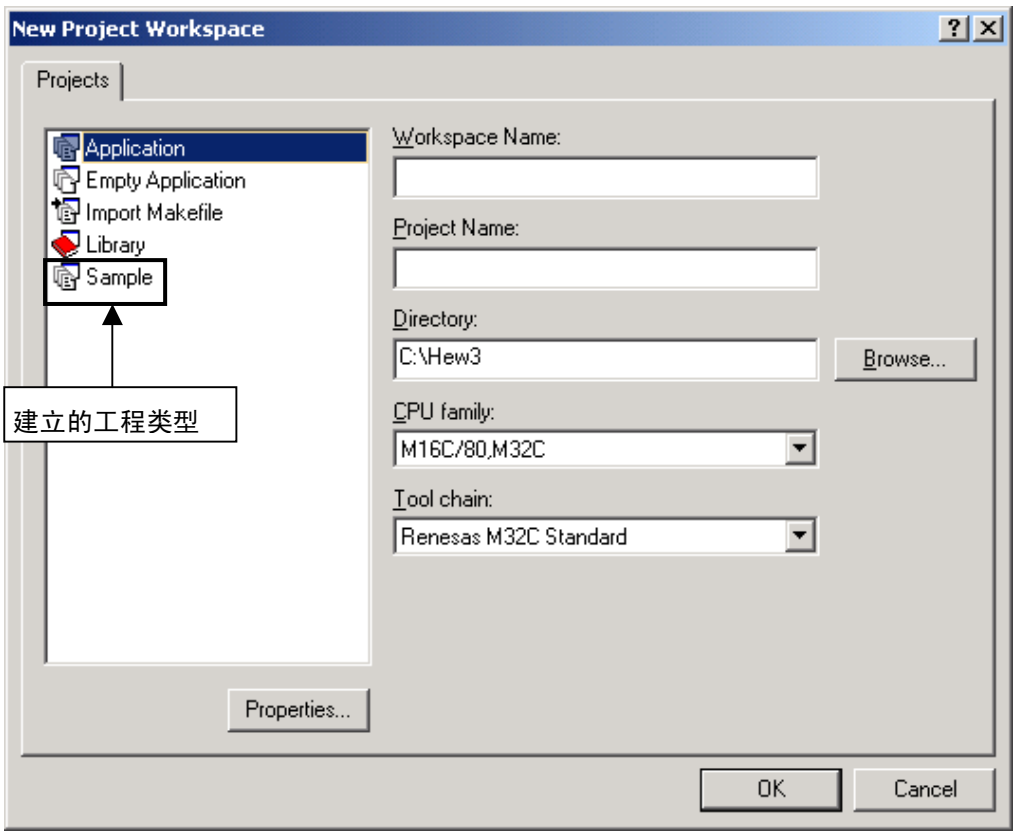


■ 使用定制工程生成程序的实例：

以下提供使用已安装的定制工程生成程序模板的实例。

1. 启动 HEW，在 [欢迎使用！(Welcome!)] 对话框中选择 [建立新的工程工作空间 (Create a new project workspace)]。已安装的工程类型将会添加到 [工程 (Projects)] 列表。单击工程类型，然后单击 [确定 (OK)] 按钮。

   您可以开始使用已存储的工程模板，为任何新工程进行程序开发。



4.2.4 多个 CPU 功能

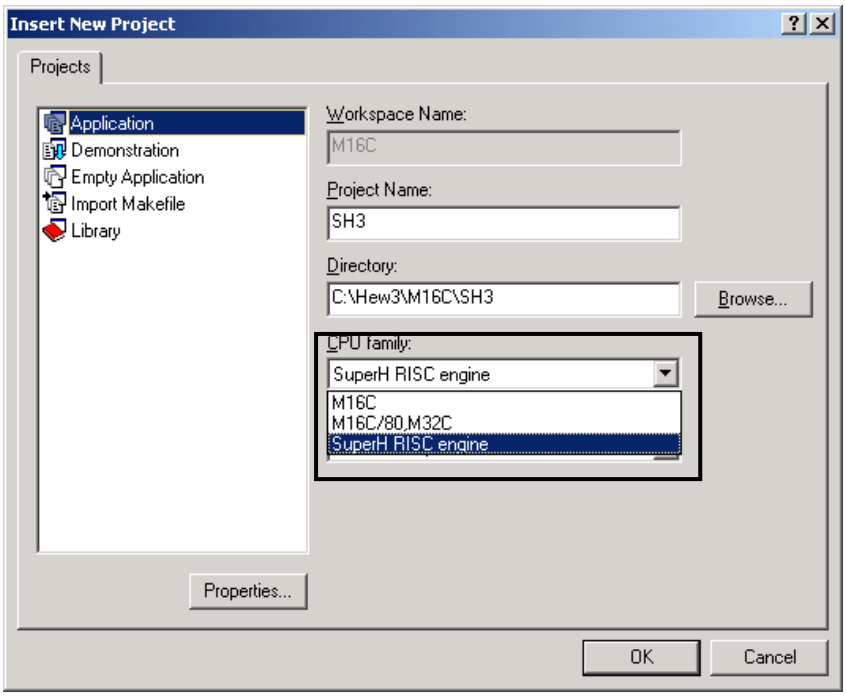
■ 描述：

在工作空间中插入新工程时，您可以插入其他类型的 CPU。

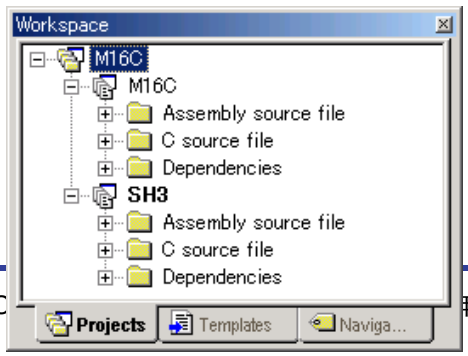
这将允许您在单一工作空间内管理不同的工程，如 M16C 和 SH。

■ 插入不同 CPU 家族的实例：

1. 在 M16C (SH) 工程打开时，单击 [工程 (Project) -> 插入工程... (Insert Project...)]。在 [插入工程 (Insert Project)] 对话框中，选取一个新工程，然后单击 [确定 (OK)] 按钮。
2. [插入新工程 (Insert New Project)] 对话框将会显示。在工程名称和 CPU 家族中选取 M16C (SH) ，然后单击 [确定 (OK)] 按钮。在工作空间内，除了当前的 CPU 类型，您还可以放置其他的 CPU 类型。



3. 通过以上步骤，您可以在单一工作空间内安装 M16C 和 SH 工程。



4.2.5 网络功能

■ 描述：

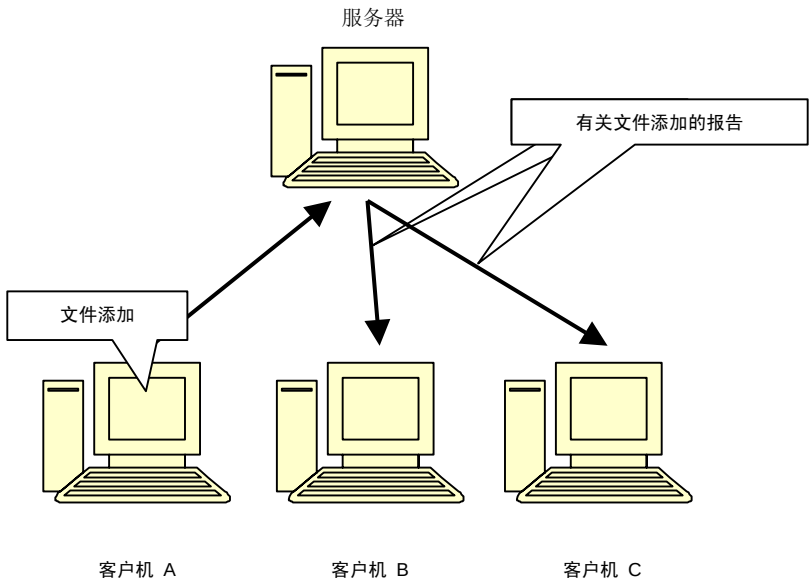
HEW 允许不同用户通过网络共享工作空间和工程。

此功能可以让用户通过同时操控共享的工程，获知其他用户所做出的更改。

此系统将一台计算机用作其服务器。

例如，如果某台客户机将新文件添加到工程中，服务器机器将会获得通知。然后服务器将会通知其他客户机关于此添加操作。

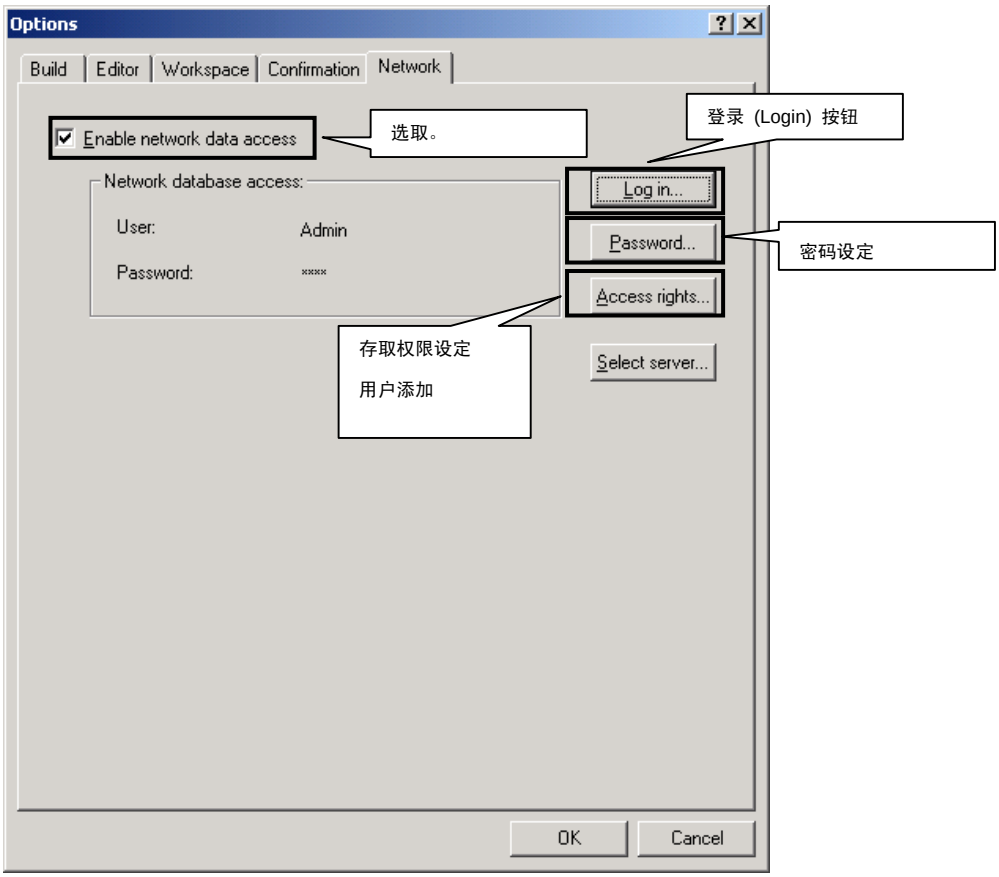
此外，用户可被授予特定工程或文件的存取权限。



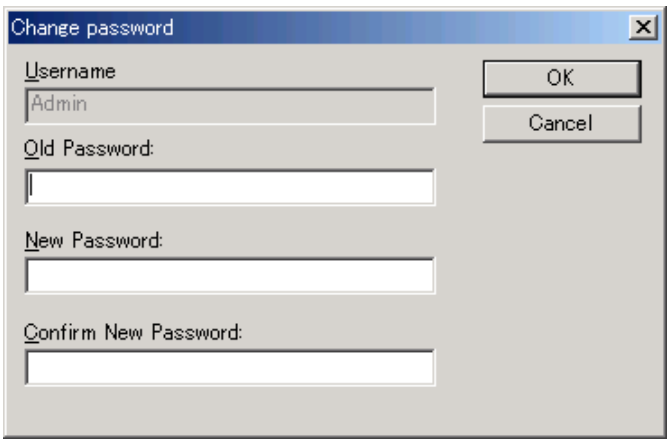
■ 设定网络存取：

1. 选择 [工具 (Tools) -> 选项 (Options)]，然后选取 [网络 (Network)] 标签。选取 [允许网络数据存取 (Enable network data access)] 复选框。
2. 一位管理员将会添加。由于管理员开始时没有密码，您将需要指定密码。管理员将被授予最高的存取权限。
3. 单击 [密码 (Password)] 按钮，然后为管理员指定密码。
4. 单击 [确定 (OK)] 按钮。此操作可以让管理员存取网络。

[选项 (Options)] 对话框的 [网络 (Network)] 标签



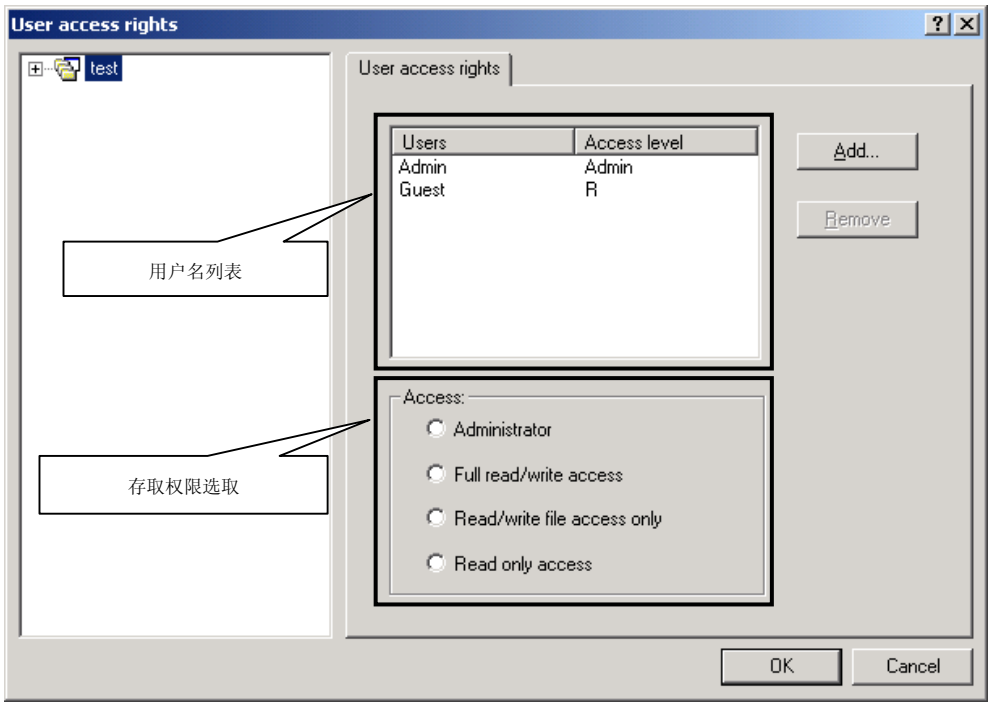
[更改密码 (Change password)] 对话



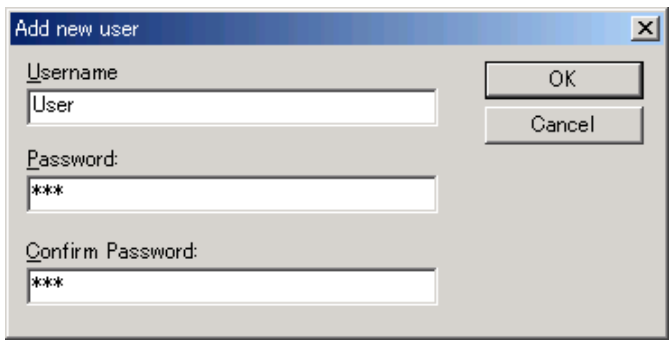
■ 添加新用户：

默认情形下，将会添加一位管理员和一位访客。您可以注册新用户。

1. 单击前页所示的 [登录... (Log in...)] 按钮。以具有管理员存取权限的用户身份登录。
2. 单击 [存取权限 (Access rights)] 按钮。[用户存取权限 (User access rights)] 对话框将会显示。



3. 单击 [添加… (Add…)] 按钮。[添加新用户 (Add new user)] 对话框将会显示。
4. 输入新用户名称和密码（密码指定是强制性的）。



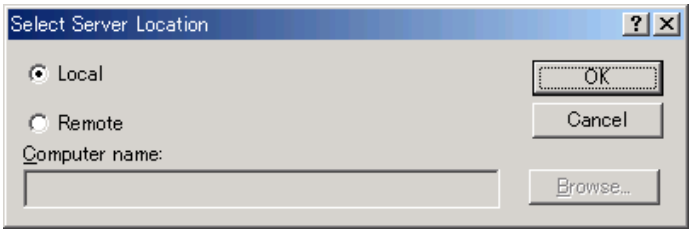
5. 接着，新用户名就会添加到用户列表中。选取用户名和指定该用户的存取权限。
   单击 [确定 (OK)] 按钮以启用您的指定。

■ 选取服务器机器：

选取将被用作服务器的机器。如果您希望以自己的机器作为服务器，您不需要执行任何操作。

如果您要将其他机器指定为服务器，请单击 [网络 (Network)] 标签中的 [选取服务器 (Select server)] 按钮。在以下对话框中选取 [远程 (Remote)]，然后指定计算机名称。

单击 [确定 (OK)] 按钮以启用您的指定。



■ **注意：**
使用此功能将降低 HEW 的性能。

备忘录

M16C 族 C 编译器应用笔记

有效的编程技术

第 5 节 有效的编程技术

在 NC308WA 编译程序执行它本身的优化 的同时，聪颖而又富有智慧的编程也可以产生提高的性能。本章将描述数种用户用于建立更有效率之程序的技术。程序的评估可以使用两个标准进行：它可以多快执行，以及它有多小。以下是建立有效程序的重要原理：

(1) 最大化执行速度

执行速度同时由经常执行的语句和复杂的语句决定。了解这些语句的处理方式以及如何选择性地改进它们，是非常重要的。

(2) 最小化程序大小

要使程序保持尽可能的小，应该共享相似的处理段以及尽可能简化复杂的函数。

由于编译程序的优化功能，有关执行速度的结果可能不同于它们在理论上的执行速度。因此，在过程中请使用编译程序上的各种方法来提高性能和测试它们。

表 5.1 列出本章所说明的有效编程技术。

表 5.1 有效编程技术的列表

编号	项目		RAM 效率	ROM 效率	执行速度
5.1	参数的寄存器传递		✓	✓	✓
5.2	使用寄存器变量		✓	✓	✓
5.3	使用 M16C 指定的指令		--	✓	✓
5.4	使用位运算转移的“进位”（carry）标志		--	✓	✓
5.5	将循环内的确定项目移到循环外		--	--	✓
5.6	SBDATA 声明和 SPECIAL 页函数声明实用程序	SBDATA 声明	--	✓	--
		SPECIAL 页函数声明	--	✓	×
5.7	使用 “switch” 而不是 “else if”		--	--	✓
5.8	循环 counter 的比较运算符		--	✓	✓
5.9	限制		--	✓	✓
5.10	使用 _Bool		--	✓	--
5.11	明确地初始化自动变量		✓	✓	✓
5.12	初始化数组		--	--	✓
5.13	增量/减量		✓	✓	✓
5.14	Switch 语句		--	--	✓
5.15	紧靠浮点		--	✓	✓

编号	项目	RAM 效率	ROM 效率	执行速度
5.16	零清除外部变量	--	✓	✓
5.17	编排启动	--	✓	✓
5.18	使用循环内的临时值	×	--	✓
5.19	使用 32 位数学函数	--	✓	✓
5.20	尽可能使用无符号	--	✓	✓
5.21	数组索引类型	✓	✓	✓
5.22	使用原型声明	✓	✓	✓
5.23	对仅返回字符类型值的函数使用字符类型	--	✓	--
5.24	注出 bss 区域的清除处理	--	✓	✓
5.25	缩减所生成的代码	✓	✓	--

5.1 参数的寄存器传递

有两种方法可以将参数传递到函数：*堆栈传递*，参数通过推到堆栈传递；以及*寄存器传递*，参数通过分配到寄存器传递。然而堆栈传递必须承担推至堆栈和从堆栈弹出的成本，而寄存器传递却可在承受相同成本的情形下更快地执行。寄存器传递在下列三个条件下使用：

① 存在函数的原型声明。

② The … 变量参数未在原型声明中使用。

③ 函数参数类型符合下表列出的其中一个。

图 5.1 显示使用寄存器传递或堆栈传递时现有原型声明更改的条件。

表 5.2 寄存器传递的类型

编译程序	第一个参数	第二个参数
NC30WA	_Bool type char type int type near pointer type	int type near pointer type
NC308WA	_Bool type char type int type near pointer type	无。

请注意，使用寄存器传递时的寄存器分配如下：

表 5.3 寄存器传递的参数分配

参数类型	编译程序	第一个参数	第二个参数	第三个和随后的参数
_Bool type	NC30WA	R1L	Stack	Stack
char type	NC308WA	R0L		
int type	NC30WA	R1	R2	Stack
near pointer type	NC308WA	R0	Stack	
Other types	NC30WA	Stack	Stack	Stack
	NC308WA			

之前	之后

`int main()`
`{`
`f(3);`
`}`
`int f(a)`
`int a;`
`{`
`…`
`}`
	`int f(int a);`
	`int main()`
	`{`
	`  f(3);`
	`}`
	`int f(int a)`
	`{`
	`…`
	`}`

`push.w#0003H` `jsr_f`	`mov.w #0003H,R0` `jsr$f`

图 5.1 参数的寄存器传递实例

5.2 使用寄存器变量

为了将经常使用的变量分配到寄存器，您可以添加 register 限定语符到变量声明，以大大加快程序。但是，如果您过量使用寄存器限定语符，寄存器空间将变得不足，从而在实际上使程序变慢。此外，使用 NC30WA 时，如果从函数调用余下的变量分配到寄存器，寄存器的保存/恢复指令将会在函数调用之前或之后建立，从而也会使程序变慢。要使 register 限定语符生效，需要在编译时指定使用“-fER”选项。图 5.4 提供此类改进的实例。

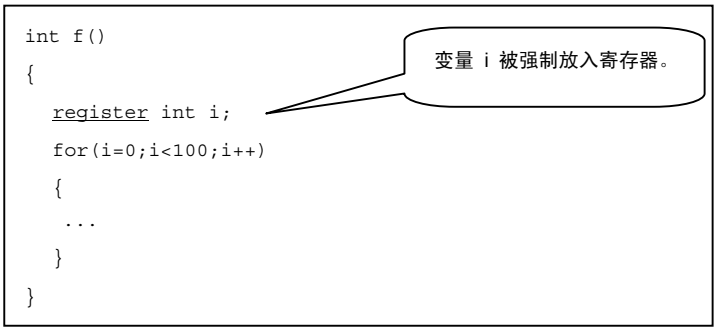


图 5.2 声明寄存器变量

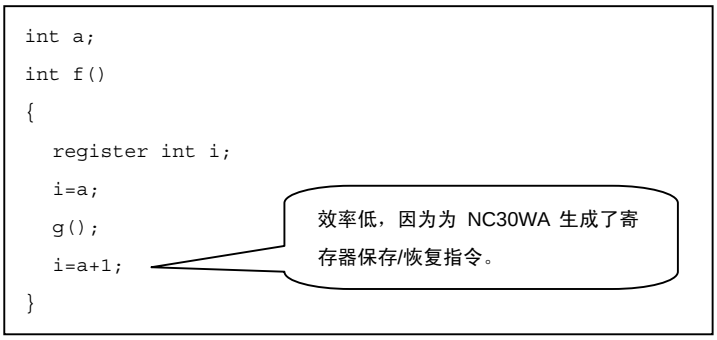


图 5.3 寄存器保存/恢复

之前	之后

int i;
sum=0;
for(i=0;i<100;i++)
{
sum+=a[i];
}
	register int i;
	sum=0;
	for(i=0;i<100;i++)
	{
	sum+=a[i];
	}

;## # C_SRC : sum=0; mov.w #0000H,-4[FB] ; sum ;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,-4[FB] ; i L1: ;## # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,-4[FB] ; i jge L5 ;## # C_SRC : sum+=a[i]; mov.w -4[FB],A0 ; i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5:	;## # C_SRC : sum=0; mov.w #0000H,-2[FB] ; sum ;## # C_SRC : for(i=0;i<100;i++) mov.w#0000H,R0 L1: ;## # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,R0 ; i jge L5 ;## # C_SRC : sum+=a[i]; mov.w R0,A0 ; i i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,R0 ; i jmp L1 L5:

图 5.4 使用寄存器变量

5.3 使用 M16C 指定的指令

通过使用同时有“if”和“else”的语句而不是只用“if”语句来分配代码的变量值，您可以通过扩展“STZX”指令缩减转移以及提高 ROM 效率。相关内容如下所示。

之前	之后

void main(void)
{
  int i=2;
  int port;
 	
  if(port == 1){
  i = 3;
  }
}
	void main(void)
	{
	  int i;
	  int port;
	
	  if(port == 1){
	  i = 3;
	  }else{
	  i = 2;
	  }
	}

mov.w #0002H,R0 ; i cmp.w #0001H,-2[FB] ; port jne L3 mov.w #0003H,R0 ; i L3:	cmp.w #0001H,-2[FB] ; port stzx.w #0003H,#0002H,R0 ; i

图 5.5 使用 M16C 指定的指令

5.4 使用位运算转移的“进位”（carry）标志

在如下所示的代码中，应该使用 & (|) 而不是 && (||)。

之前

```
struct A
{
   int a:1;
   int b:1;
   int c:1;
} a;

main()
{
   if(a.a && a.b && a.c) func();
}
```

```
   btst 00H,-2[FB]  ;  a
   jz L1
   btst 01H,-2[FB]  ;  a
   jz L45
   btst 02H,-2[FB]  ;  a
   jz L47
   jsr _func
  L47:
  L45:
L1:
```

之后

```
struct A
{
   int a:1;
   int b:1;
   int c:1;
} a;

main()
{
   if(a.a & a.b & a.c ) func();
}
```

```
   btst 00H,-2[FB] ;  a
   band 01H,-2[FB] ;  a
   band 02H,-2[FB] ;  a
   jnc L29
   jsr _func
L29:
```

图 5.6 使用位运算转移的“进位”（carry）标志

5.5 将循环内的确定项目移到循环外

在如下所示的代码中，您可以通过移动循环内的确定表达式使它们在循环外，减少需要的计算次数，从而加快程序。此操作可以通过启用编译程序中的优化项目自动执行。

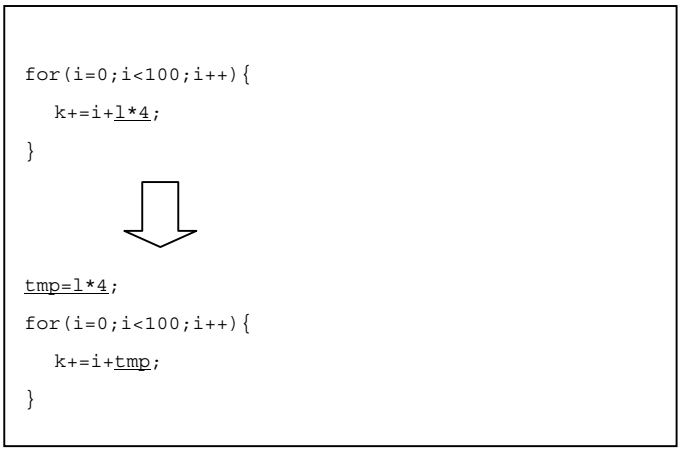


图 5.7 将循环内的确定项目移到循环外

C 源	优化之前	优化之后
for(i=0;i<100;i++) a[i]=l*4;	;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: ._line 18 ;## # C_SRC for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ;## # C_SRC : a[i]=l*4; mov.w _l:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 jmp L1 L5:	;## # C_SRC :for(i=0;i<100;i++) mov.w#0000H,_i:16 mov.w l:16,R0 shl.w #2,R0 L1: ;## # C_SRC : a[i]=l*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L1

图 5.8 执行优化将循环内的确定项目移到循环外

5.6 SBDATA 声明和 SPECIAL 页函数声明实用程序

utl308、SBDATA 声明和 SPECIAL 页函数声明实用程序，处理绝对模块文件 (*.x30)，以及输出下列项目：

1. SBDATA 声明

   此声明用于执行从常用变量至 SB 区域的分配。

   (#pragma SBDATA)

2. SPECIAL 页函数声明

   此声明用于执行从常用函数至 special 页区域的分配。

   (#pragma SPECIAL)

要使用 utl308，编译时需指定编译驱动器中的命令行选项 “-finfo” 以生成绝对模块文件 (*.x30)。

图 5.9 显示 NC308 的处理流程。您可以利用此工具获得 SBDATA 功能和 SPECIAL 页功能的最佳使用方式。要获取详细资料，请参考《NC308 用户手册》（NC308 User's Manual）中的*附录 G*。

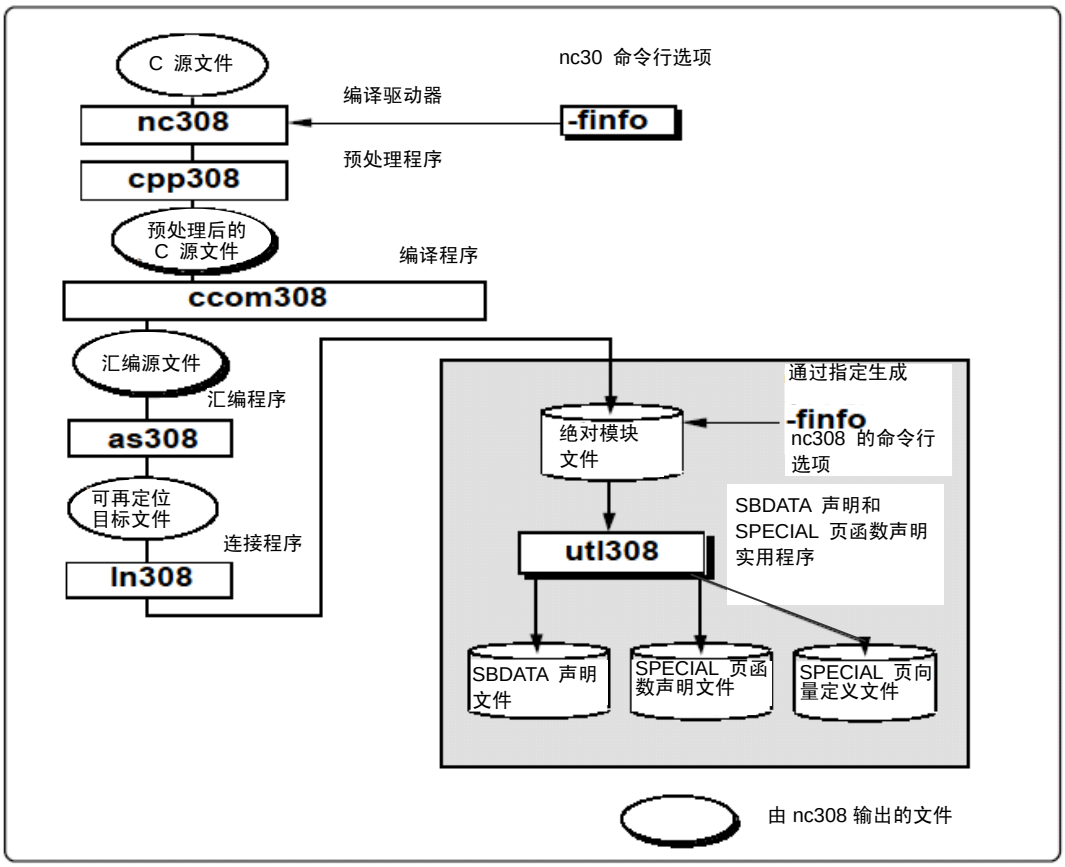


图 5.9 SBDATA 声明和 SPECIAL 页函数声明实用程序

5.7 使用“switch”而不是“else if”

需要多次比较数组或其他结构时，使用“switch”语句会比使用“else if”语句来得快。这是因为“else if”语句是通过间接寻址执行比较，而“switch”语句可以保留空间以及在寄存器中执行比较。

之前

```
if(a[i]==0){
 ...
}  else if(a[i]==1){
 ...
} else if(a[i]==2){
 ...
}  else {
    ...
}
```

```
;## # C_SRC :       if(a[i]==0)
    mov.w R0,R1 ;  i  i
    shl.w #1,R0
    mov.w R0,A0
    mov.w _a:16[A0],R0
    jne L1
  ...
  ;## # C_SRC : else if(a[i]==1)
  L1:
    mov.w R1,A0 ;  i  i
    shl.w #1,A0
    cmp.w #0001H,_a:16[A0]
    jne L11
     ...
  ;## # C_SRC :        else if(a[i]==2)
  L11:
    mov.w R1,A0 ;  i  i
    shl.w #1,A0
    cmp.w #0002H,_a:16[A0]
...
```

之后

```
switch(a[i])
{
   case 0:
       ...
      break;
   case 1:
       ...
       break;
   case 2:
       ...
       break;
   default:
       ...
       beak;
}
```

```
   indexws.w R0 ;  i
   mov.w:g _a:16,R0
   cmp.w #0000H,R0
   jeq L3
   cmp.w #0001H,R0
   jeq L5
   cmp.w  #0002H,R0
   jeq L7
   jmp L9
   ...
```

图 5.10 使用“switch”而不是“else if”

5.8 循环 counter 的比较运算符

在循环中，对 0 进行比较可使用较少 ROM 并执行得更快。

之前	之后

for(i=0;i<10;i++){
  ...
}
	for(i=10;i!=0;i--){
	  ...
	}

add.w #0001H,_i cmp.w #000aH,_i jlt label	**adjnz.w #-1,_i,label**

图 5.11 循环 counter 的比较运算符

5.9 限制

您可以使用 restrict 限定语符使优化方便进行，只要您确定没有其他变量指向同一区域。请注意，如果您在指向同个区域作为变量时使用限制限定语符，将可能发生故障。

之前	之后

int a;
int *q;
void f()
{
  a=10;
  *q=20;
  sub(a);
}
装入的需要。
	int a;
	int * restrict q;
	void f()
	{
	  a=10;
	  *q=20;
	  sub(a);
	}
	这将替换为 a=10。

_f:
;## # C_SRC : a=10;
  mov.w #000aH,_a:16
;## # C_SRC : *q=20;
  mov.w #0014H,[_q:16]
;## # C_SRC : sub(a);
  mov.w _a:16,R0
  jsr $sub
;## # C_SRC : }
rts
	_f:
	;## # C_SRC : a=10;
	  mov.w#000aH,_a:16
	;## # C_SRC : *q=20;
	  mov.w #0014H,[_q:16]
	## # C_SRC : sub(a);
	  mov.w #000aH,R0
	  jsr $sub
	;## # C_SRC : }
	rts

图 5.12 限制的使用实例

5.10 使用 _Bool

_Bool 是一个布尔类型，它可以是 0 或 1。您可以使用 _Bool 标志来预防不必要的异常处理。

之前	之后

char flag;
 	
if(flag==0){
  ...
} else if(flag==1) {
  ...
}else{
// 异常处理
}
	_Bool flag;
	
	if(flag==0){
	  ...
	} else {
	  ...
	}

图 5.13 _Bool 的使用实例

5.11 明确地初始化自动变量

初始化自动变量时，它会被分配到寄存器中。否则，它会存储在堆栈中。此优化操作需要优化选项来进行。

之前

```
extern unsigned int    *p, max_val, min_val;
void func(void)
{
   unsigned int    max = 0;
   unsigned int    min; // 推到
                        // 堆栈
   while (1) {
        if (max == 0) min = max = *p;
        if (max < *p) max = *p;
        if (min > *p) min = *p;
        p++;
        if (*p == 0) break;
   }
   max_val = max;
   min_val = min;
}
```

之后

```
extern unsigned int    *p, max_val, min_val;
void func(void)
{
   unsigned int    max = 0;
   unsigned int    min=*p; // 分配
                           // 到寄存器
   while (1) {
        if (max == 0) min = max = *p;
        if (max < *p) max = *p;
        if (min > *p) min = *p;
        p++;
        if (*p == 0) break;
   }
   max_val = max;
   min_val = min;
}
```

```
  .glb_func
_func:
 enter #02H
 pushm R1
 ;## # C_SRC :  unsigned int  max = 0;
   mov.w #0000H,R0 ;  max
 ;## # C_SRC :  while (1)
 L3:
 ;## # C_SRC : if (max == 0) min =
    max = *p;
    cmp.w #0000H,R0 ;  max
    jne L7
    mov.w [_p:16],R0
    mov.w R0,R1
    mov.w R1,-2[FB] ;  min
 L7:
 ;## # C_SRC : if (max < *p) max = *p;
   cmp.w [_p:16],R0;  max
   jgeu L17
   mov.w [_p:16],R0
 L17:
 ;## # C_SRC : if (min > *p) min = *p;
   cmp.w [_p:16],-2[FB] ;  min
   mov.w [_p:16],-2[FB] ;  min
 L27:
 ;## # C_SRC : p++;
   add.l #00000002H,_p:16
 ;## # C_SRC : if (*p == 0) break;
   mov.w [_p:16],R1
   jne L3
 ;## # C_SRC : max_val = max;
   mov.w R0,_max_val:16 ;  max
 ;## # C_SRC : min_val = min;
   mov.w -2[FB],_min_val:16 ;  min
 ;## # C_SRC : }
   popm R1
 exitd
```

```
  .glb _func
_func:
 pushm   R1
 ;## # C_SRC : unsigned int  max = 0;
   mov.w #0000H,R0 ;  max
 ;## # C_SRC :  unsigned int    min=*p;
   mov.w [_p:16],R1 ;  min
 ;## # C_SRC : while (1)
 L3:
 ;## # C_SRC : if (max == 0) min = max = *p;
   cmp.w #0000H,R0 ;  max
   jne L7
   mov.w [_p:16],R0
   mov.w  R0,R1
 L7:
  ._line  27
 ;## # C_SRC : if (max < *p) max = *p;
   cmp.w [_p:16],R0 ;  max
   jgeu L17
   mov.w [_p:16],R0
 L17:
 ._line 28
 ;## # C_SRC : if (min > *p) min = *p;
   cmp.w [_p:16],R1 ;  min
   jleu L27
   mov.w [_p:16],R1
 L27:
 ;## # C_SRC : p++;
   add.l #00000002H,_p:16
 ;## # C_SRC : if (*p == 0) break;
   cmp.w #0000H,[_p:16]
   jne L3
 ;## # C_SRC : max_val = max;
   mov.w R0,_max_val:16    ;  max
 ;## # C_SRC : min_val = min;
   mov.w R1,_min_val:16    ;  min
 ;## # C_SRC : }
   popm R1
rts
```

图 5.14 明确地初始化自动变量

5.12 初始化数组

在一个循环语句内初始化两个数组时，请将其中一个移到不同的循环。这将允许使用 sstr 指令来扩展循环，从而提高执行速度。

但是，当您初始化三个或更多数组时，请在同个循环内执行初始化来提高 ROM 效率。这是因为 sstr 指令需要对每个寄存器进行初始设置，所以在初始化 3 个或更多数组时会降低 ROM 效率。

之前	之后

/* 在同个循环内初始化 */
void loop2_1(void)
{
  int i;
  for(i = 0; i < 10; i++){
  a[i] = 0x0a;
  b[i] = 0x0b;
  }
}
	/* 在不同的循环内初始化 */
	void loop2_2(void)
	{
	  int i;
	  for(i = 0; i < 10; i++)
	  a[i] = 0x0a;
	  for(i = 0; i < 10; i++)
	  b[i] = 0x0b;
	}

_loop2_1: pushm A0 mov.w #0000H,R0 L3: mov.w R0,A0 mov.b #0aH,_a:16[A0] mov.b #0bH,_b:16[A0] add.b #01H,A0 mov.w A0,R0 cmp.w #000aH,R0 jlt L3 popm A0 rts	_loop2_2: pushm R3,A1 mov.b #0aH,R0L mov.w #(_a&0FFFFH),A1 mov.w #0aH,R3 sstr.b mov.b #0bH,R0L mov.w #(_b&0FFFFH),A1 mov.w #0aH,R3 sstr.b popm R3,A1 rts

图 5.15 初始化数组

5.13 增量/减量

从表达式分开增量和减量。编译程序将会尝试保留在执行增量/减量之前的值。

之前	之后

a[i++]=b[j++];	a[i]=b[j];
	i++;
	j++;

mov.w -22[FB],R0 ; j add.w #0001H,-22[FB] ; j indexws.w R0 mov.w:g -22[FB],R0 ; b indexwd.w -2[FB] ; i mov.w R0,-22[FB] ; a add.w #0001H,-2[FB] ; i	indexws.w -4[FB] ; j mov.w:g -24[FB],R0 ; b indexwd.w -2[FB] ; i mov.w R0,-24[FB] ; a add.w #0001H,-2[FB] ; i add.w #0001H,-4[FB] ; j

图 5.16 增量/减量

5.14 Switch 语句

为了获得更有效率的代码，在“switch”的具有数个 case 值的语句中，尽可能缩短 case 值之间的间隔。这是因为大间隔的代码会扩展成 “if else” 格式，而小间隔则扩展为转移表。对 case 值使用枚举类型也是一个好办法。

之前	之后

switch(a)
{
case 100:
...
case 200:
...
case 300:
...
}
	switch(a)
	{
	case 1:
	...
	case 2:
	...
	case 3
	...
	}

cmp.w #0064H,R0 jeq L5 cmp.w #00c8H,R0 jeq L7 cmp.w #012cH,R0 jeq L9 ...	S2: jmpi.w L47 L47: .word L45-S2&0ffffH .word L23-S2&0ffffH .word L25-S2&0ffffH ...

图 5.17 Switch 语句

5.15 即时浮点

当精度不是重要关注时，将 f 后缀附加到浮点数。不具有 f 后缀的数将当作双精度类型处理。

之前	之后
float f; f=f+123.456;	float f; f=f+123.456f;
push.l _f:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0, f:16	push.l _f:16 push.l #42f6e979H jsr.a __f4add add.l #08H,SP mov.l R2R0,_f:16

图 5.18 即时浮点

5.16 零清除外部变量

在启动期间，所有外部变量的初始值将会被零清除，即使是在不明确设定为如此执行的情形下。当您明确地零清除一个变量时，由于 0 会保留在 ROM 中并且在启动期间转移，因此会浪费 ROM 空间。所以，不要零清除外部变量的初始值。

之前	之后
`int i=0`	`int i;`

`.SECTION data_NE,DATA`
`._inspect 'U', 1, "data_NE", "data_NE", 0`
`.glb _i`
`_i:`
`.blkb 2`
 	
`.SECTION data_NEI,ROMDATA`
`._inspect 'U', 1, "data_NEI", "data_NEI", 0`
`;## # init data of i.`
`.word 0000H`
	`.SECTION bss_NE,DATA`
	`._inspect 'U', 3, "bss_NE", "bss_NE", 0`
	`.glb _i`
	`_i:`
	`.blkb 2`

图 5.19 零清除外部变量

5.17 编排启动

如果您知道将不会使用标准 I/O 函数或存储器管理函数，您可以通过省略它们的初始设置来加快程序。在汇编时间通过对 ncrt0.a30 指定 “-D__STANDARD_IO__=1” 和 “-D__HEAP__=1”，您可以禁止这些函数的初始化处理。图 5.21 显示设定 “-D__STANDARD_IO__=1” 和 “-D__HEAP__=1” 的 HEW 对话框。

```
;------------------------------------------------------------------------
; 堆大小定义
;------------------------------------------------------------------------
.if __HEAP__ == 1      ; for HEW

HEAPSIZE  .equ   0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE  .equ   300h

.else             ; for HEW

HEAPSIZE  .equ   __HEAPSIZE__

.endif
.endif

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

;========================================================================
; 初始化标准 I/O
;------------------------------------------------------------------------
.if __STANDARD_IO__ != 1
    .glb  _init
    .call _init,G
    jsr.a _init
.endif
```

图 5.20 编排启动

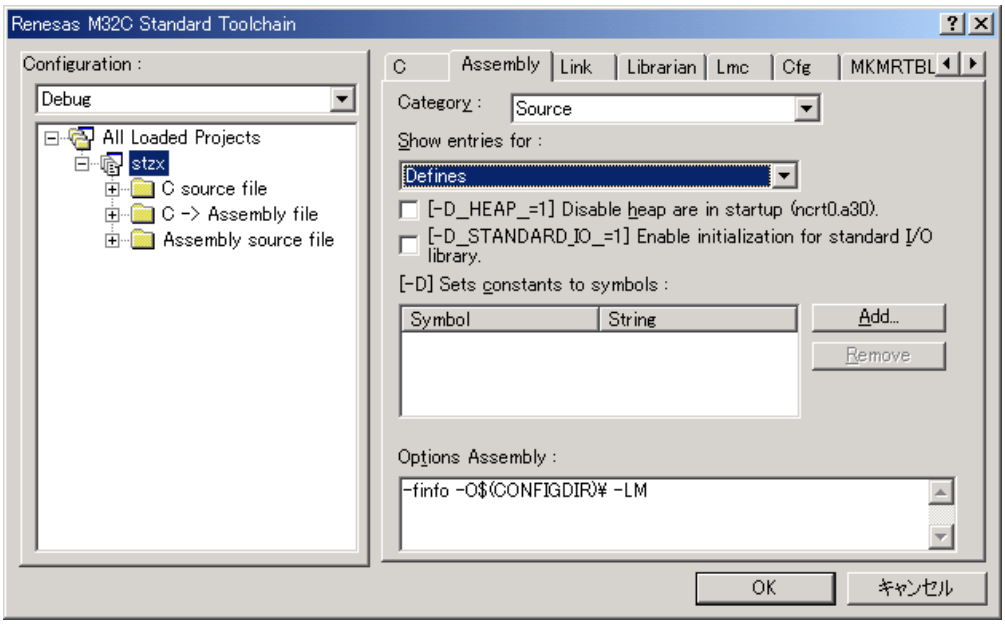


图 5.21 编排启动的 HEW 对话框

5.18 使用循环内的临时值

在循环内使用外部变量时，存储器将会在每个迭代中参考。您可以通过使用临时变量使寄存器中的计算方便执行。

之前

```
int total;
void func(int p[])
{
   int i;
   total=0;
   for(i=0;i<100;i++){
      total+=p[i];  //存储器每次均被
                    // 参考。
   }
}
```

```
 _func:
   enter #02H
   pushm R2,A0
 ;## # C_SRC : total=0;
   mov.w #0000H,_total:16
 ;## # C_SRC : for(i=0;i<100;i++){
   mov.w  #0000H,-2[FB] ;  i
 L1:
 ;## # C_SRC : for(i=0;i<100;i++){
   cmp.w  #0064H,-2[FB] ;  i
   jge L5
 ;## # C_SRC : total+=p[i];
 //存储器每次均被参考。
   mov.w  -2[FB],R0 ;  i
   exts.w R0
   mov.l  R2R0,A0
   shl.l  #1,A0
   add.l  8[FB],A0 ;  p
   add.w  [A0],_total:16
   add.w  #0001H,-2[FB] ;  i
   jmp L1
 L5:
 ;## # C_SRC : }
   popm R2,A0
   exitd
```

存储器会在循环中的每个迭代被参考。

之后

```
int total;
void func(int p[])
{
   int i,tmp;
   tmp=0;
   for(i=0;i<100;i++){
      tmp+=p[i];  // 在寄存器中执行的
                  // 计算。
   }
   total=tmp;
}
```

```
 _func:
   enter #00H
   pushm R1,R2,R3,A0,A1
   mov.l 8[FB],A0 ;  p  p
 ;## # C_SRC : tmp=0;
   mov.w #0000H,R1 ;  tmp
 ;## # C_SRC : for(i=0;i<100;i++)
   mov.w #0000H,R0 ;  i
 L1:
 ;## # C_SRC :tmp+=p[i];
 // 在寄存器中执行的计算。
   mov.w R0,R3 ;  i  i
   exts.w R0
   mov.l R2R0,A1
   shl.l #1,A1
   add.l A0,A1 ;  p
   add.w [A1],R1
   add.w #0001H,R3 ;  i
   mov.w R3,R0 ;  i  i
   cmp.w #0064H,R0 ;  i
   jlt L1
 ;## # C_SRC : total=tmp;
   mov.w R1,_total:16 ;  tmp
 ;## # C_SRC : }
   popm  R1,R2,R3,A0,A1
   exitd
```

由寄存器中的一个计算替代。

图 5.22 使用循环内的临时值

5.19 使用 32 位数学函数

您可以对浮点类型的变量使用 32 位数学函数来提高执行速度。使用标准数学函数时，参数将会先转换成双精度类型，然后，所形成的双精度类型值将会在该变量被替换时转换回浮点类型值。使用 32 位数学函数时，所有的计算将会使用浮点类型执行。

<table>
<tr><th>之前</th><th>之后</th></tr>
<tr><td>

```
#include<math.h>
float fdata,result;
result=sin(fdata);
```

双精度转换

浮点转换

</td><td>

```
#include<mathf.h>
float fdata,result;
result=sinf(fdata)
```

计算为浮点

</td></tr>
<tr><td>

```
;## # C_SRC : result=sin(fdata);
push.l _fdata:16
.glb __f4tof8
jsr.a __f4tof8
add.l #04H,SP
pushm R0,R1,R2,R3
jsr _sin
add.l #08H,SP
pushm R3,R2,R1,R0
.glb __f8tof4
jsr.a __f8tof4
add.l #08H,SP
mov.l R2R0,_result:16
```

</td><td>

```
;## # C_SRC : result=sinf(fdata);
push.l _fdata:16
jsr _sinf
add.l #04H,SP
mov.l  R2R0,_result:16
```

</td></tr>
</table>

图 5.23 使用 32 位数学函数

5.20 尽可能使用无符号

尽可能使用无符号整数，可提高转移指令的代码效率（但是，请注意，这样做会恶化仅用于 M16C/20 和 M16C/60 系列的代码效率，因为在 NC308 中添加“无符号”（unsigned）会导致符号扩展）。此外，在比较有符号的整数时，尽可能使用 != 和 == 运算符而不是 <=、>=、<、> 运算符来提高代码效率。

之前	之后

`int data[100];`
`void main()`
`{`
`  int i;`
`  int sum=0;`
`  for(i=0;i<100;i++) sum+=data[i];`
`}`
	`int data[100];`
	`void main()`
	`{`
	`  unsigned int i;`
	`  int sum=0;`
	`  for(i=0;i!=100;i++)sum+=data[i];`
	`}`

`;## # C_SRC : int sum=0;`
`  mov.w #0000H,-4[FB] ; sum`
`;## # C_SRC :for(i=0;i<100;i++)sum+=`
`  data[i];`
`  mov.w #0000H,-2[FB] ; i`
`L1:`
`;## # C_SRC : for(i=0;i<100;i++)sum+=`
`  data[i];`
`  cmp.w #0064H,-2[FB] ; i`
`  jge L5`
`  mov.w -2[FB],A0 ; i`
`  shl.w #1,A0`
`  add.w _data:16[A0],-4[FB] ; sum`
`  add.w #0001H,-2[FB] ; i`
`  jmp L1`
`L5:`
	`;## # C_SRC : int sum=0;`
	`  mov.w #0000H,-4[FB] ; sum`
	`;## # C_SRC :for(i=0;i!=100;i++)sum+=`
	`  data[i];`
	`  mov.w #0000H,-2[FB] ; i`
	`L1:`
	`;## # C_SRC :for(i=0;i!=100;i++)sum+=`
	`  data[i];`
	`  cmp.w #0064H,-2[FB] ; i`
	`  jeq L5`
	`  mov.w -2[FB],A0 ; i`
	`  shl.w #1,A0`
	`  add.w _data:16[A0],-4[FB] ; sum`
	`  add.w #0001H,-2[FB] ; i`
	`  jmp L1`
	`L5:`

图 5.24“无符号”（Unsigned）的使用实例

5.21 数组索引类型

类型扩展在计算数组索引期间执行，并以数组内的元素之大小为根据。

(1) 对于 2 字节或更大的大小（用于字符类型或有符号字符类型以外的类型）：

通过将索引扩展成整数类型执行计算。

(2) 对于 64K 或更大的远型数组：

通过将索引扩展成长型执行计算。

同样的，当您将字符类型变量用作数组索引时，它必须扩展成整数类型，形成效率低的代码。所以，对于数组索引，使用整数类型而不是字符类型。请注意，此优化仅限于 NC30WA。

之前	之后

`char i;`
`int a[10];`
`a[i]=0;`
	`int i;`
	`int a[10];`
	`a[i]=0;`

`mov.b -1[FB],R0L ; i`
`mov.b #00H,R0H`
`shl.w #01H,R0`
`mova -21[FB],A0 ; a`
`add.w R0,A0`
`mov.w #0000H,[A0]`
	`mov.w -2[FB],R0 ; i`
	`shl.w #01H,R0`
	`mova -22[FB],A0 ; a`
	`add.w R0,A0`
	`mov.w #0000H,[A0]`

图 5.25 数组索引类型

5.22 使用原型声明

使用此编译程序，您可以利用函数原型声明执行更有效的函数调用。也就是说，如果您没有在这个编译程序中使用函数原型声明，当这些函数被调用时，它们的参数将会按照下表显示的规则推到堆栈区域。

图 5.4 适用于参数的堆栈使用规则

数据类型	推到堆栈时的规则
字符类型 有符号的字符类型	扩展成整数类型。
浮点类型	扩展成双精度类型。
其他类型	无扩展。

这意味着在不使用函数原型声明时，可能会执行冗余类型扩展。

您可以使用函数原型声明来提高函数调用的效率、避免冗余类型扩展，以及允许参数分配到寄存器。

之前	之后

int main()
{
  char data1;
  char data2;
  char data3;
  int data;
  data=f(data1,data2,data3);
}
int f(i, j, k)
char i,j,k;
{
  ...
}
	int f(char ,char ,char);
	
	int main()
	{
	  char data1;
	  char data2;
	  char data3;
	  int data;
	  data=f(data1,data2,data3);
	}
	
	int f(i, j,k)
	char i,j,k;
	{
	...
	}

extz -2[FB],R0 ; data3 push.w R0 extz -2[FB],R0 ; data2 push.w R0 extz -2[FB],R0 ; data1 push.wR0 jsr _f	push.b -2[FB] ; data3 push.b -2[FB] ; data2 mov.b -2[FB],R0L ; data1 jsr $f

图 5.26 使用原型声明

5.23 对仅返回字符类型值的函数使用字符类型

您可以通过对仅返回字符类型值的函数之返回值使用字符类型来减少 ROM 空间的数量。

同时，尽可能使用更小的数据类型。

之前

```
int func2(void)
{
      switch (x) {
      case 0:
          return 255;
      case 1:
          return 254;
      default:
          return 253;
      }
 }
```

```
   .glb _func2
  _func2:
  ;## # C_SRC : switch (x) {
    mov.w _x:16,R0
    jeq L3
    cmp.w #0001H,R0
    jeq L5
    jmp L7
  ;## # C_SRC : case 0:
  L3:
  ;## # C_SRC : return 255;
    mov.w #00ffH,R0
    rts
  ;## # C_SRC : case 1:
  L5:
  ;## # C_SRC : return 254;
    mov.w #00feH,R0
    rts
  ;## # C_SRC : default:
  L7:
  ;## # C_SRC : return 253;
    mov.w #00fdH,R0
  rts
```

之后

```
char func2(void)
{
      switch (x) {
      case 0:
          return 255;
      case 1:
          return 254;
      default:
          return 253;
      }
 }
```

```
   .glb  _func2
  _func2:
  ;## # C_SRC : switch (x) {
    mov.w _x:16,R0
    jeq L3
    cmp.w #0001H,R0
    jeq L5
    jmp L7
  ;## # C_SRC : case 0:
  L3:
  ;## # C_SRC : return 255;
    mov.b #0ffH,R0L
    rts
  ;## # C_SRC : case 1:
  L5:
  ;## # C_SRC : return 254;
    mov.b #0feH,R0L
    rts
  ;## # C_SRC : default:
  L7:
  ;## # C_SRC : return 253;
    mov.b #0fdH,R0L
  rts
```

图 5.27 对仅返回字符类型值的函数使用字符类型

5.24 注出 bss 区域的清除处理

启动程序 ncrt0.a30 包含 bss 区域清除处理。此处理是为了满足未初始化的变量应该具有 0 作为其初始值的 C 语言规格而存在。

例如，由于图 5.28 中显示的代码包含没有初始值的变量，启动期间需要处理（bss 区域清除处理）将 0 设定为初始值。

```
static int i;
```

图 5.28 不具备初始值的变量之声明实例

根据应用程序而定，不具备初始值的变量可能不需要进行零清除。在此情形下，您可以通过将启动程序中执行 bss 区域清除处理的部分注出来加快启动处理。

```
;======================================================================
; “近”（NEAR）区域初始化。
;----------------------------------------------------------------------
; bss 零清除
;----------------------------------------------------------------------
;           N_BZERO bss_SE_top,bss_SE
;           N_BZERO bss_SO_top,bss_SO
;           N_BZERO bss_NE_top,bss_NE
;           N_BZERO bss_NO_top,bss_NO
  :
 （被省略）
  :
;======================================================================
; “远”（FAR）区域初始化。
;----------------------------------------------------------------------
; bss 零清除
;----------------------------------------------------------------------
;           BZERO bss_FE_top,bss_FE
;           BZERO bss_FO_top,bss_FO
```

图 5.29 注出 bss 区域的清除处理的实例

5.25 缩减所生成的代码

您可以缩减使用整数类型声明的数据所生成的代码数量，并且在以下范围之内：

从 0 至 255：更改成无符号的字符类型。

从 -128 至 127：更改成有符号的字符类型。

通过缩减变量类型的大小，您可以缩减比较、加法，以及其他指令的大小，从而提高 ROM 效率。

之前

```
int type

int data[128];

void main(void)
{

   int cnt = 128;
   int i;

   int sum=0;

   for( i = 0 ; i < cnt ; i++ ){
     sum += data[i];
   }

}
```

```
   ;## # C_SRC : int cnt = 128;
     mov.w #0080H,-6[FB]   ;  cnt
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     mov.w#0000H,-4[FB] ;  i
   L1:
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     cmp.w -6[FB],-4[FB] ;  cnt  i
     jge L5
   ;## # C_SRC : sum += data[i];
     mov.w -4[FB],A0 ;  i
     shl.w #1,A0
     mova -262[FB],A1 ;  data
     add.l A1,A0
     add.w [A0],-2[FB] ;  sum
     add.w #0001H,-4[FB] ;  i
     jmp L1
   L5:
   ;## # C_SRC : }
     popm A0,A1
     exitd
```

之后

```
unsigned char type

void main(void)
{

   unsigned char cnt = 128;
   unsigned char i;

   int data[128];
   int sum;

   for( i = 0 ; i < cnt ; i++ ){
     sum += data[i];
   }

}
```

```
   ;## # C_SRC : unsigned char cnt = 128;
   mov.b #80H,-2[FB] ;  cnt
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     mov.b #00H,-1[FB] ;  i
   L1:
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     cmp.b -2[FB],-1[FB]  ;  cnt  i
     jgeu L5
   ;## # C_SRC : sum += data[i];
     mov.b -1[FB],A0 ;  i
     shl.w #1,A0
     mova -260[FB],A1 ;  data
     add.l A1,A0
     add.w [A0],-4[FB] ;  sum
     add.b #01H,-1[FB] ;  i
     jmp  L1
   L5:
   ;## # C_SRC : }
     popm A0,A1
     exitd
```

图 5.30 缩减所生成的代码 (1)

您可以缩减使用长型声明的数据所生成的代码数量，并且在以下范围之内和符合以下标准：

从 0 至 65535：更改成无符号的整数类型。

从 -32768 至 32767：更改成有符号的整数类型。

通过缩减变量类型的大小，您可以缩减比较、加法，以及其他指令的大小，从而提高 ROM 效率。

之前

```
unsigned int data[65535];

void main(void)
{

    long cnt = 65535;
    long i;

    int sum;

    for( i = 0 ; i < cnt ; i++ ){
        sum += data[i];
    }
}
```

```
   ;## # C_SRC : long cnt = 65535;
     mov.l #0000ffffH,-10[FB] ;  cnt
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     mov.l #00000000H,-6[FB] ;  i
   L1:
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     cmp.l -10[FB],-6[FB] ;  cnt  i
     jge L5
   ;## # C_SRC : sum += data[i];
     mov.l -6[FB],A0 ;  i
     shl.l #1,A0
     add.w _data:16[A0],-2[FB] ;  sum
     add.l #00000001H,-6[FB] ;  i
     jmp  L1
   L5:
   ;## # C_SRC : }
     popm A0
   exitd
```

之后

```
unsigned int data[65535];

void main(void)
{

    unsigned int cnt = 65535;
    unsigned int i;

    int sum;

    for( i = 0 ; i < cnt ; i++ ){
        sum += data[i];
    }
}
```

```
   ;## # C_SRC : unsigned int cnt = 65535;
     mov.w #0ffffH,-6[FB] ;  cnt
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     mov.w #0000H,-4[FB] ;  i
   L1:
   ;## # C_SRC : for( i = 0 ; i < cnt ; i++ )
     cmp.w -6[FB],-4[FB] ;  cnt  i
     jgeu L5
   ;## # C_SRC : sum += data[i];
     mov.w -4[FB],A0 ;  i
     shl.l #1,A0
     add.w _data:16[A0],-2[FB] ; sum
     add.w #0001H,-4[FB] ;  i
     jmp  L1
   L5:
   ;## # C_SRC : }
     popm A0
   exitd
```

图 5.31 删除所生成的代码 (2)

M16C 族 C 编译器应用笔记

使用 HEW 调试程序

第 6 节 使用 HEW 调试程序

本章介绍如何有效使用 HEW 调试程序 （模拟器调试程序），内容如下：

编号	类别	项目	节
1	使用虚拟中断函数	通过点击按钮插入虚拟中断	6.1.1
2		以规则的时间间隔插入虚拟中断	6.1.2
3		在指定的周期插入虚拟中断	6.1.3
4		在指定地址的指令运行时插入虚拟中断	6.1.4
5	使用虚拟端口输入/输出函数	通过点击按钮输入数据	6.2.1
6		在指定地址被读取时，通过虚拟端口输入数据	6.2.2
7		在指定的周期通过虚拟端口输入数据	6.2.3
8		当虚拟中断发生时，通过虚拟端口输入数据	6.2.4
9		检查虚拟输出端口的输出数据	6.2.5
10	使用虚拟 发光二极管 (LED) 或标签 (Label) 来检查存储器内容		6.3
11	使用 printf 进行调试		6.4
12	使用 I/O 脚本		6.5

6.1 使用虚拟中断函数

6.1.1 通过点击按钮插入虚拟中断

■ 说明：

中断可以通过点击虚拟中断按钮来手动产生，就好像点击按钮引发中断。

为按钮设置中断优先权和中断条件

■ 怎样创立一个手动产生虚拟中断的按钮：

1. 从菜单选择［查看->图形->GUI I/O (View -> Graphics -> GUI I/O)］来显示图形用户界面（GUI）窗口。
2. 您可以点击创建按钮图标，或右键点击从菜单选择［创建按钮 （Create button）］来创建一个用来产生虚拟中断的按钮。

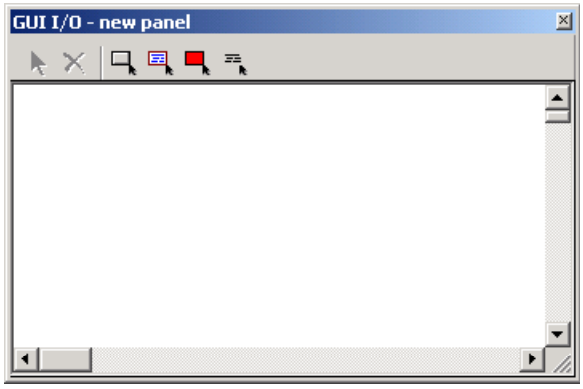


图 6.1 GUI 窗口

3. 点击已创建的按钮来显示该按钮的设置窗口。

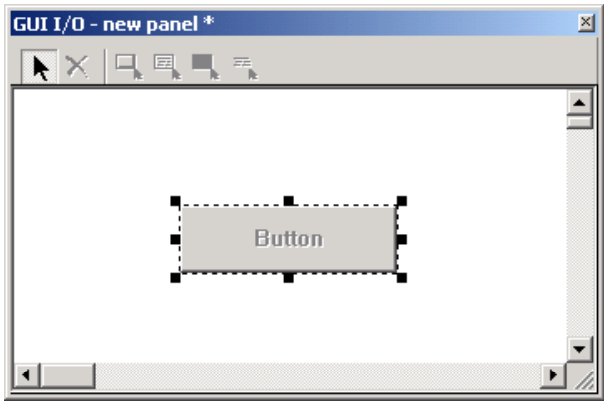


图 6.2 GUI 窗口 （放置按钮以后）

4. 选择按钮类别［中断（Interrupt）］，然后设置中断向量和 IPL（中断优先等级）

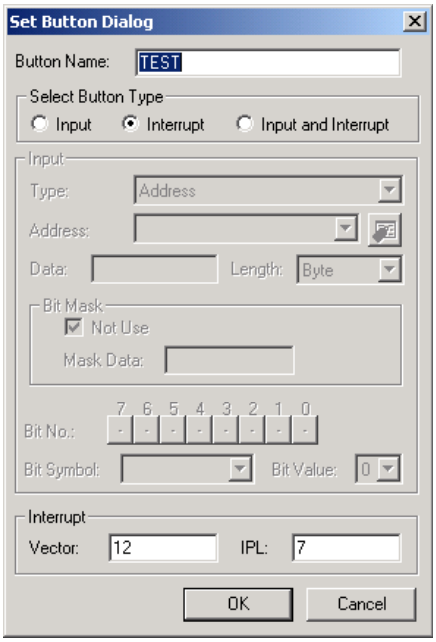


图 6.3 按钮设置窗口

5. 在设置向量表的文件中，设置中断向量。在此情况下，在［vector 12］行，［.lword dummy_int］变为［.lword _test］，当按钮按下时引用源程序中的 test 函数。

```
;-----------------------------------------------------------------
; variable vector section
;-----------------------------------------------------------------
 .section vector,ROMDATA   ; varible vector table
 .org VECTOR_ADR

 .lword dummy_int          ; vector 0
 .lword dummy_int          ; vector 1

  :

 .glb    _test
 .lword  _test             ; vector 12

  :
```

6. 在调试程序中，点击步骤 2 创立的按钮来产生虚拟中断。

6.1.2 以规则的时间间隔插入虚拟中断

■ 说明：

可以通过［I/O 时序设置（I/O Timing Setting)］窗口设置与指定时间间隔同步的虚拟中断

■ 如何设置：

1. 从菜单，选择［查看->CPU-> I/O 时序设置（View -> CPU -> I/O Timing Setting)］来显示［I/O 时序设置（I/O Timing Setting)］窗口。
2. 您可以点击时间间隔同步中断的图标，或右键点击从菜单选择［计时器（Timer）］来显示设定与指定间隔同步的中断的窗口。

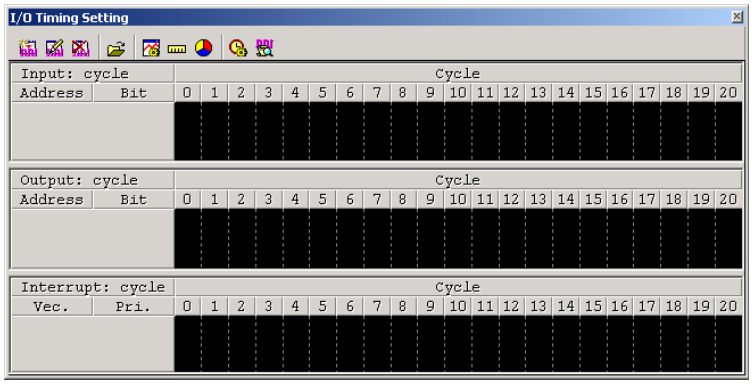


图 6.4 I/O 时序设置窗口

3. 单击［载入（Load）］键来引入设置文件。
4. 除非您正在引入设置文件，输入时间间隔，向量和优先级，然后点击［添加（Add)］按钮。

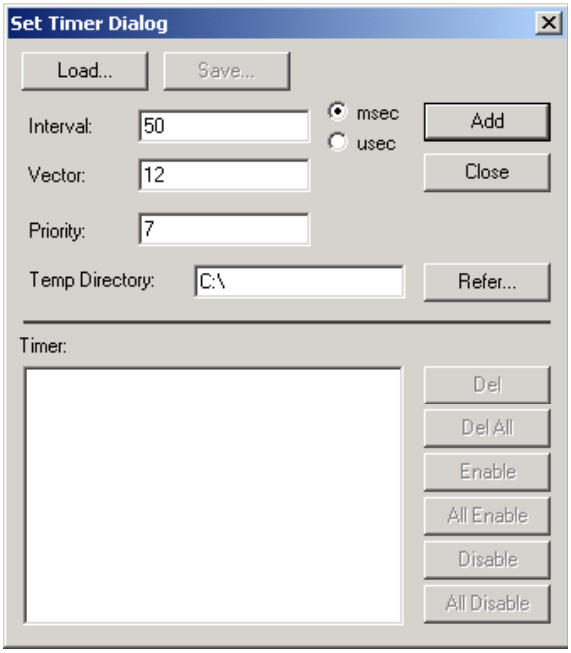


图 6.5 设置计时器对话框

5. 显示输入设置。
6. 点击［保存（Save）］按钮来保存当前设置。

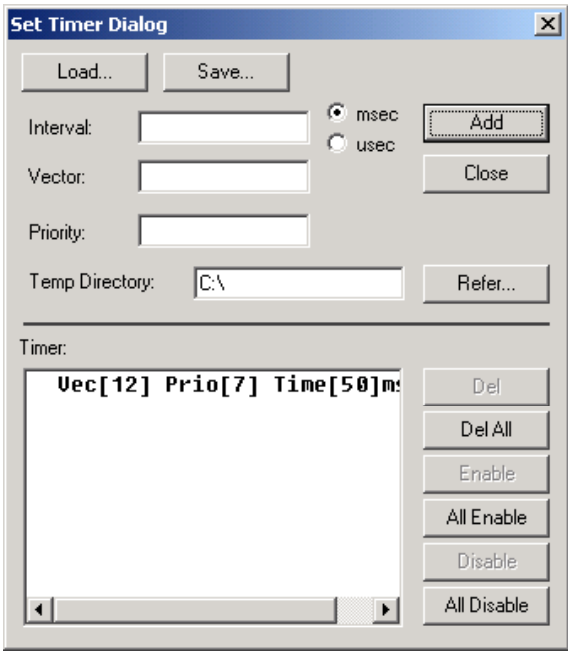


图 6.6 设置计时器对话框（在输入设置后）

7. 您可根据需要添加多种设置。
8. 您还可以切换每一个设置，根据需要允许或禁用。

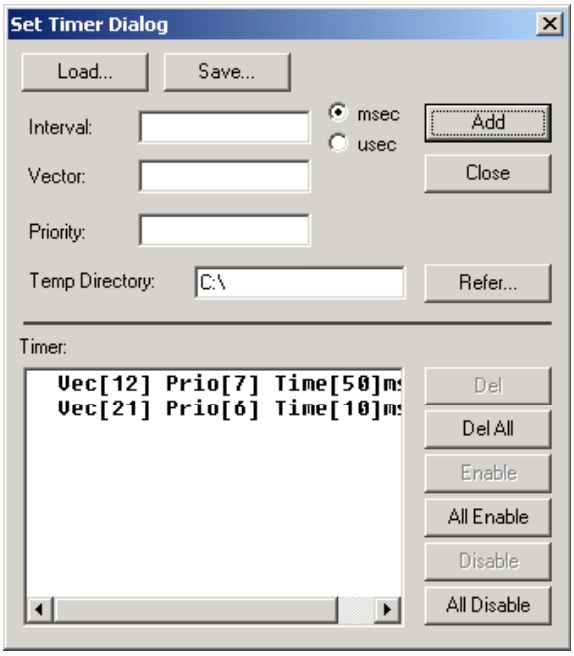


图 6.7 设置计时器对话框（多样设定）

6.1.3 在指定的周期插入虚拟中断

■ 说明：

可以通过［I/O 时序设置（I/O Timing Setting)］窗口设置在指定周期发生的虚拟中断

■ 如何设置:

1. 从菜单，选择［查看->CPU-> I/O 时序设置（View -> CPU -> I/O Timing Setting)］来显示［I/O 时序设置（I/O Timing Setting)］窗口。
2. 您可以点击数据设置（Data setting）图标，或右键点击从菜单选择［设置（Setup）］。

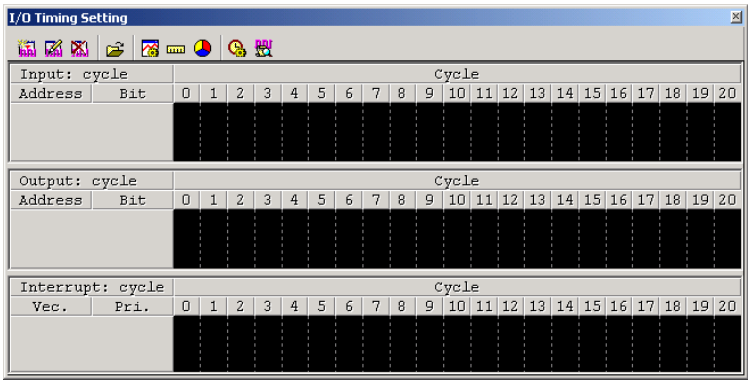


图 6.8 I/O 时序设置窗口

3. 选择［设置虚拟中断（Set virtual interrupt）］，然后点击［下一步>（Next>）］按钮。

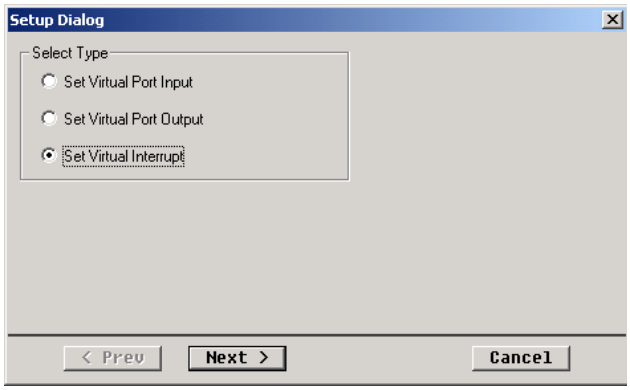


图 6.9 设置虚拟中断

4. 在中断将要发生的时间上，选择［时钟周期（Cycle）］。
5. 设置起始周期，结束周期，向量和优先级，然后点击［下一步>（Next>）］按钮。

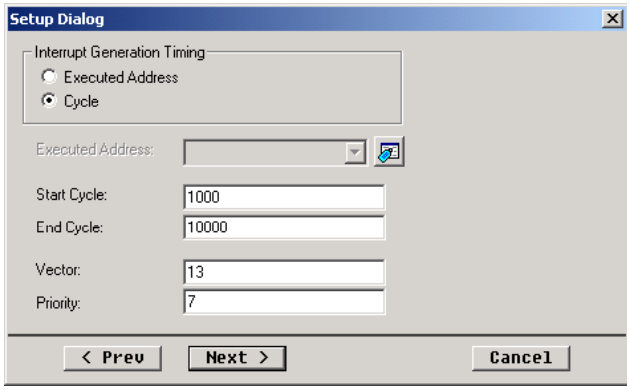


图 6.10 设置周期和向量

6. 把鼠标指向您想设置虚拟中断周期的格子，左键点击来设置将要发生的中断。
7. 当所有中断设置好以后，点击［下一步>（Next>）］按钮。

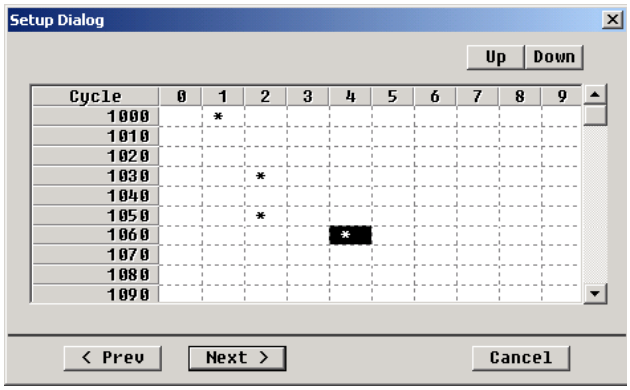


图 6.11 设置将要发生的中断

8. 保存 I/O 脚本文件，它已被自动修改。

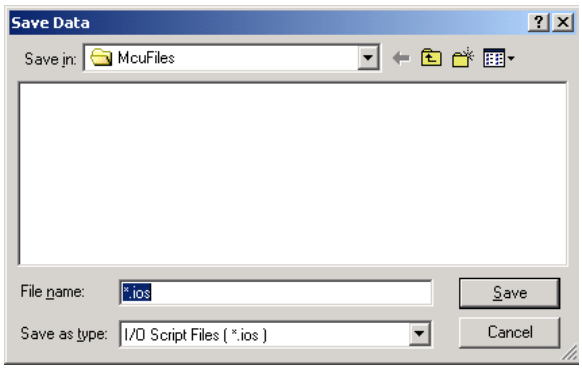


图 6.12 保存（Save） 对话框

9. I/O 脚本文件如下所示：

```
; IOSCRIPT FILE FOR I/O WINDOW (INT WAITC)
{
cycle 1001
int 13 , 7
waitc 31
int 13 , 7
waitc 20
int 13 , 7
waitc 12
int 13 , 7
}
```

6.1.4 在指定地址的指令运行时插入虚拟中断

■ 说明：

在指定地址的指令运行发生的虚拟中断可以在［I/O 时序设置 (I/O Timing Setting)］窗口中设置。

■ 如何设置:

1. 执行“6.1.3 在指定的周期插入虚拟中断”中的步骤 1 到 3，来显示设置虚拟中断的对话框。
2. 在中断将要发生的时间上，选择［执行地址（Executed address）］。
3. 设置执行地址，向量和优先级，然后点击［下一步>（Next>）］按钮。

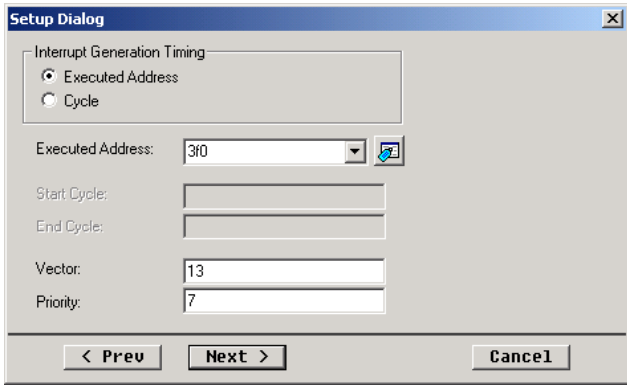


图 6.13 设置执行地址和向量

4. 设置在执行地址来产生虚拟中断所需指令运行次数。鼠标指向所需运行次数的格子，左键点击设置将要发生的中断。
5. 当所有中断设置好以后，点击［下一步>（Next>）］按钮。

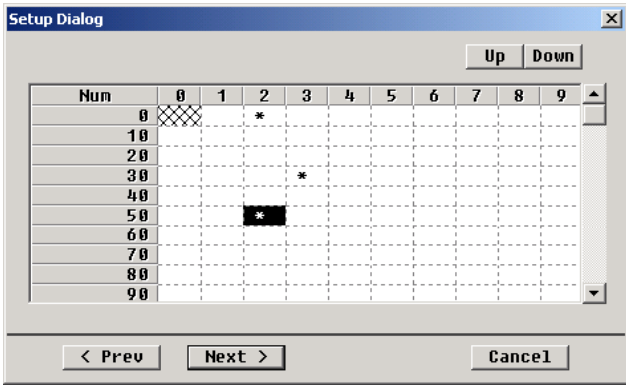


图 6.14 设置将要发生的中断

6. 保存 I/O 脚本文件，它已被自动修改。

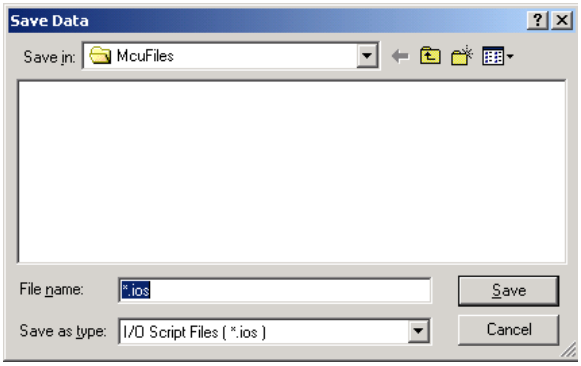


图 6.15 保存（Save） 对话框

7. I/O 脚本文件如下所示：

```
; IOSCRIPT FILE FOR I/O WINDOW (INT ISFETCH)
{
pass #isfetch:0x3f0 , 2
int 13 , 7
pass #isfetch:0x3f0 , 31
int 13 , 7
pass #isfetch:0x3f0 , 19
int 13 , 7
}
```

6.2 使用虚拟端口输入/输出函数

6.2.1 点击按钮输入数据

■ 说明：

虚拟端口输入可以通过点击按钮手动产生

指明按钮的地址和位符号（bit symbol）。

■ 怎样创立一个手动产生虚拟端口输入的按钮：

1. 在 GUI 窗口放置按钮。细节请看“6.1.1 通过点击按钮插入虚拟中断”，步骤 1 到 3。
2. 按钮类型选择［输入（Input）］。
3. 在类型（type）选项中选择［地址（Address）］，［地址和比特编号（Address & Bit No.）］，或［位符号（Bit Symbol）］。
4. 如果选择［地址（Address）］，指明地址，数据和数据大小。

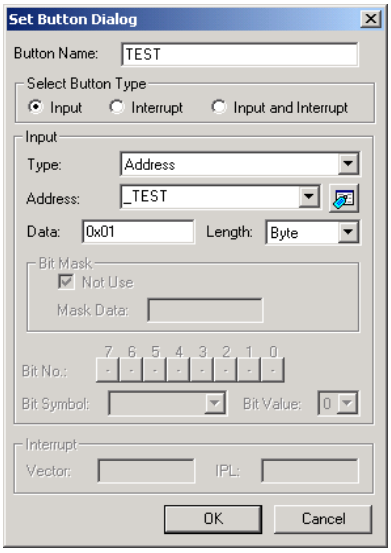


图 6.16 按钮设置窗口

5. 如果选择了［地址和比特编号（Address & Bit No.）］，在不使用位屏蔽的情况下设置地址和比特编号 如果使用位屏蔽（Bit mask）的话，指明地址，数据，数据大小和屏蔽值

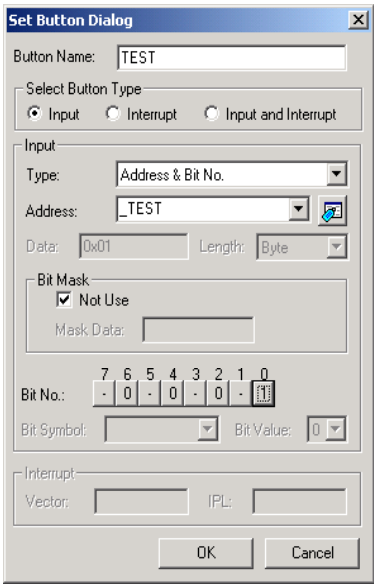


图 6.17 按钮设置

6. 如果选择［位符号（Bit Symbol）］，设置位符号（Bit symbol）和位值（Bit value）。

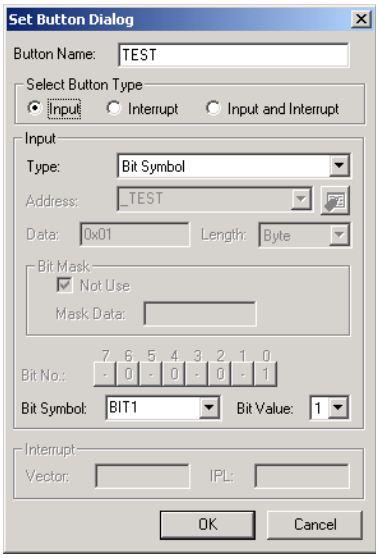


图 6.18 按钮设置

7. 点击按钮后，指定地址或位符号的值就被输入了。

6.2.2 在指定地址被读取时，通过虚拟端口输入数据

■ 说明：

在指定地址被读取时，虚拟端口的输入可以通过［I/O 时序设置（I/O Timing Setting)］窗口来设置。

■ 如何设置:

1. 从菜单，选择［查看->CPU-> I/O 时序设置（View -> CPU -> I/O Timing Setting)］来显示［I/O 时序设置（I/O Timing Setting)］窗口。
2. 您可以点击数据设置（Data setting）图标，或右键点击从菜单选择［设置（Setup）］。

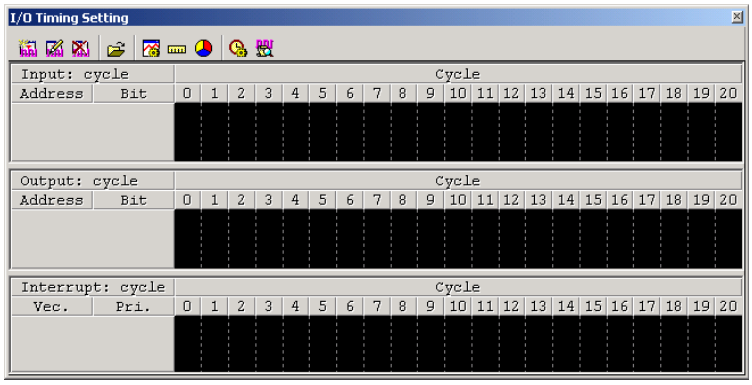


图 6.19 I/O 时序设置窗口

3. 选择［设置虚拟端口输入（Set virtual port input）］，然后点击［下一步>（Next>）］按钮。

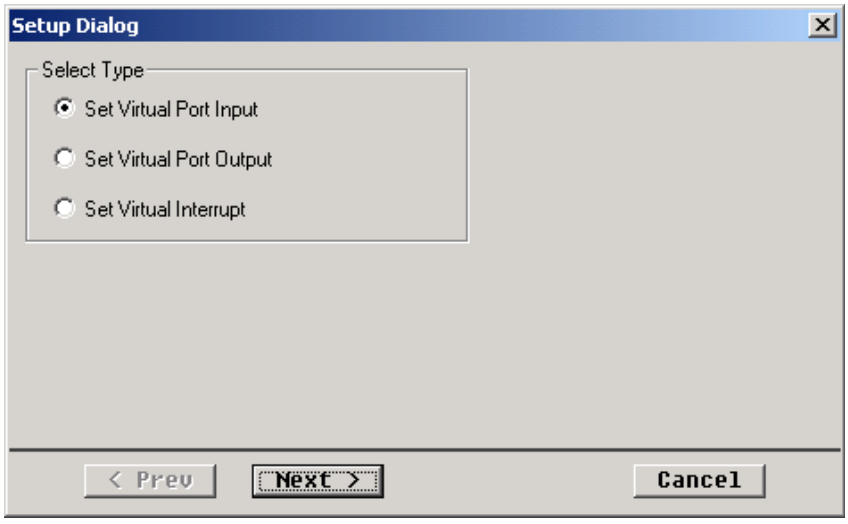


图 6.20 设置进程类型

4. 把数据输入时序（Data input timing）设置成［读访问（Read access）］
5. 在［输入地址（Input address）］区，使用 16 进制数输入虚拟端口输入发生的地址。
6. 在［读访问（Read address）］区，使用 16 进制数输入存储器地址（在此地址出现读访问时虚拟端口就有输入发生）。
7. 当使用相同地址作为输入地址和读取地址时，此地址被访问时下一段数据就被引用。
8. 点击［下一步>（Next>）］按钮。

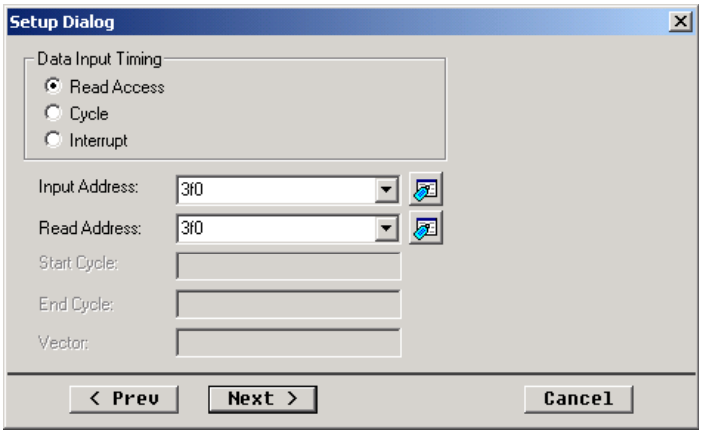


图 6.21 读访问设定

9. 设定从虚拟输入端口输入的数据。鼠标指向设置数据需要读访问次数的格子，双击左键来显示输入控制台，然后使用 16 进制数输入数据。
10. 数据输入后，点击［下一步>（Next>）］按钮。

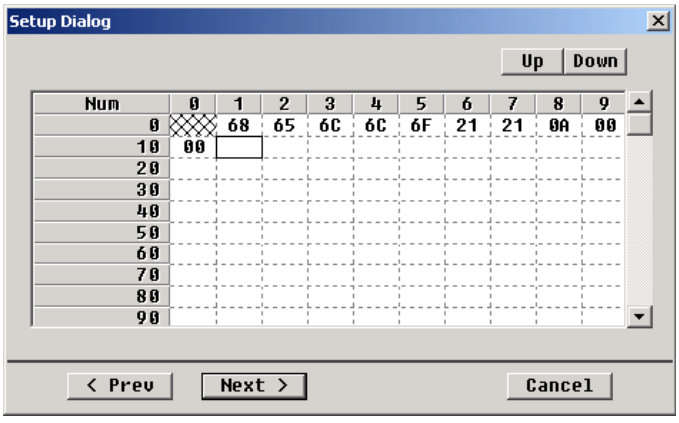


图 6.22 输入数据设置

11. 保存 I/O 脚本文件，它已被自动修改。

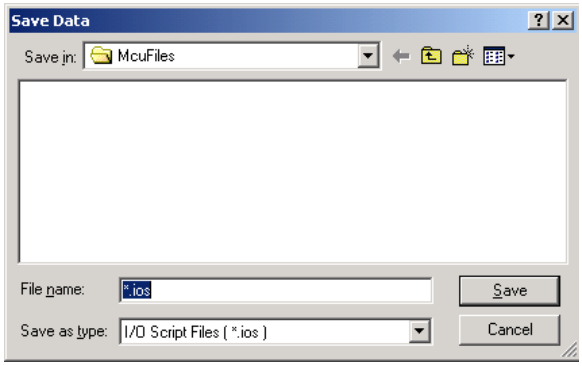


图 6.23 保存（Save） 对话框

12. I/O 脚本文件如下所示：

```
; IOSCRIPT FILE FOR I/O WINDOW (SET ISREAD)
{
pass #isread:0x3f0 , 1
set [0x3f0] = 0x68
pass #isread:0x3f0 , 1
set [0x3f0] = 0x65
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6f
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0xa
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
}
```

13. 下面程序是虚拟端口输入的实例。

```
#include <stdio.h>
char *PORT_IN;
int i;
char buf[10];

void main(void)
{
   PORT_IN = (char *)0x3f0;      → [Input address] 或 [Read address]中指明的地址
      for(i=0; i<10; i++){          → for 语句
            buf[i] = *PORT_IN;         → 读访问时，下一个数据被引用
            if(buf[i] == '\0')         → 0x00 用作结束码
         break;
   }

   printf("%s", buf);             → 将输入数据标准输出
}
```

6.2.3 在指定的周期通过虚拟端口输入数据

■ 说明：

在指定的周期从虚拟端口输入的数据可以在［I/O 时序设置 (I/O Timing Setting)］窗口设定。

■ 如何设置:

1. 数据设定对话框从［I/O 时序设置 (I/O Timing Setting)］窗口显示。执行“6.2.2 在指定地址被读取时，通过虚拟端口输入数据”的步骤 1 到 3。
2. 将输入数据时序设定成［时钟周期（Cycle）］。
3. 在［输入地址（Input address）］区，使用 16 进制数输入虚拟端口输入发生的地址。
4. 设定起始周期和结束周期，然后按［下一步>（Next>）］按钮。

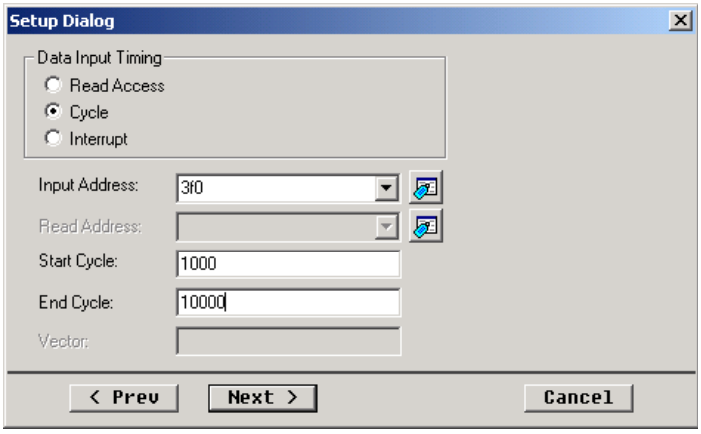


图 6.24 周期设置

5. 设定从虚拟输入端口输入的数据。鼠标指向设置数据需要读访问次数的格子，双击左键来显示输入控制台，然后使用 16 进制数输入数据。
6. 数据输入后，点击［下一步>（Next>）］按钮。

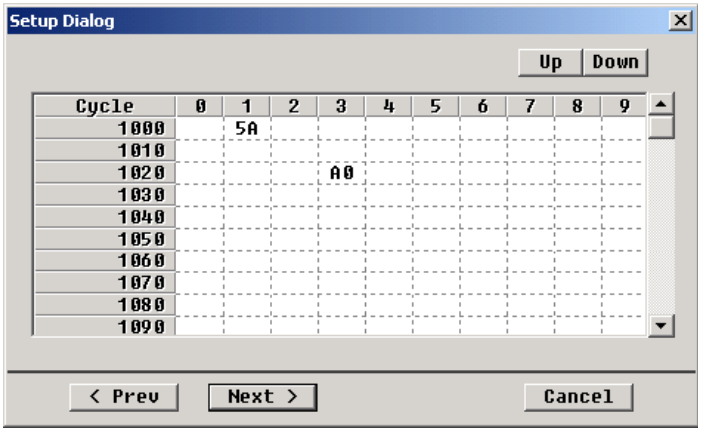


图 6.25 输入数据设置

7. 保存 I/O 脚本文件，它已被自动修改。

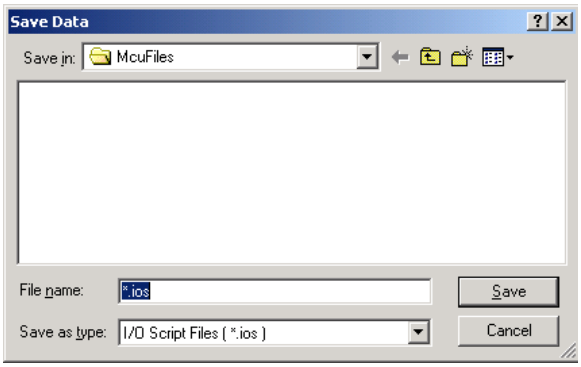


图 6.26 保存（Save） 对话框

8. I/O 脚本文件如下所示：

```
; IOSCRIPT FILE FOR I/O WINDOW (SET WAITC)
{
cycle 1001
set [0x3f0] = 0x5a
waitc 22
set [0x3f0] = 0xa0
}
```

6.2.4 当虚拟中断发生时，通过虚拟端口输入数据

■ 说明：

和虚拟中断一起发生的端口输入可以在［I/O 时序设置 (I/O Timing Setting)］窗口设定。

■ 如何设置:

1. 数据设定对话框从［I/O 时序设置 (I/O Timing Setting)］窗口显示。执行“6.2.2 在指定地址被读取时，通过虚拟端口输入数据”的步骤 1 到 3。
2. 将输入数据定时设定到［中断（Interrupt）］。
3. 在［输入地址（Input address）］区，使用 16 进制数输入虚拟端口输入发生的地址。
4. 指明将被监控的中断向量，然后按［下一步>（Next>）］按钮。

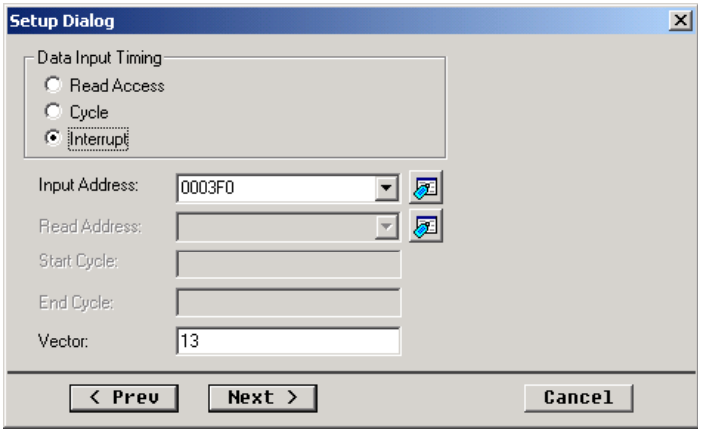


图 6.27 中断设置

5. 设定从虚拟输入端口输入的数据。鼠标指向设置数据需要读访问次数的格子，双击左键来显示输入控制台，然后使用 16 进制数输入数据。
6. 数据输入后，点击［下一步>（Next>）］按钮。

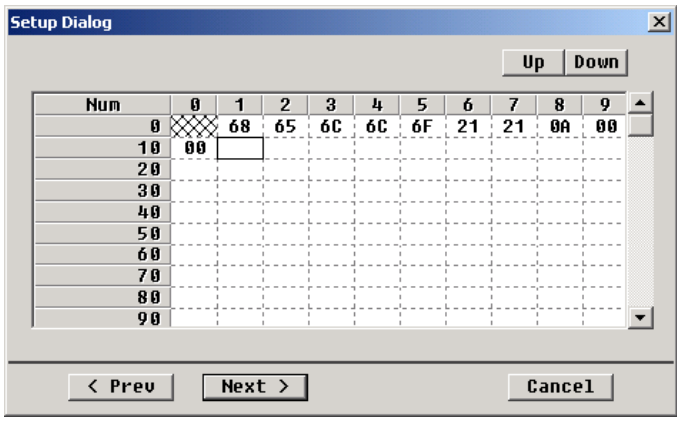


图 6.28 输入数据设置

7. 保存 I/O 脚本文件，它已被自动修改。

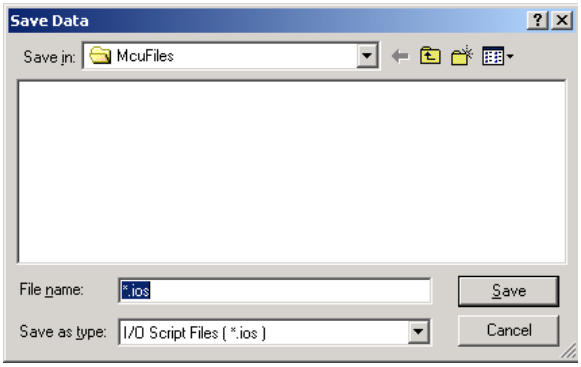


图 6.29 保存（Save） 对话框

8. I/O 脚本文件如下所示：

```
; IOSCRIPT FILE FOR I/O WINDOW (SET ISINT)
{
pass #isint:13 , 1
set [0x3f0] = 0x68
pass #isint:13 , 1
set [0x3f0] = 0x65
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6f
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0xa
pass #isint:13 , 1
set [0x3f0] = 0x0
pass #isint:13 , 1
set [0x3f0] = 0x0
}
```

6.2.5 检查虚拟输出端口的输出数据

■ 说明:

虚拟端口的输出数据科技通过［I/O 时序设置 (I/O TimingSetting)］窗口和输出端口窗口来检查。

(1) ［I/O 时序设置 (I/O Timing Setting)］窗口

1. 从菜单，选择［查看->CPU-> I/O 时序设置 (View -> CPU -> I/O Timing Setting)］来显示［I/O 时序设置 (I/O Timing Setting)］窗口。
2. 您可以点击数据设置（Data setting）图标，或右键点击从菜单选择［设置（Setup）］。

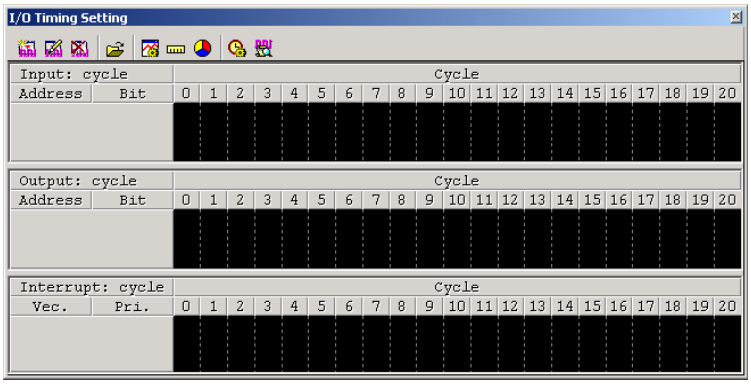


图 6.30 I/O 时序设置窗口

3. 选择［设置虚拟输出端口（Set virtual output port）］，然后点击［下一步>（Next>）］按钮。

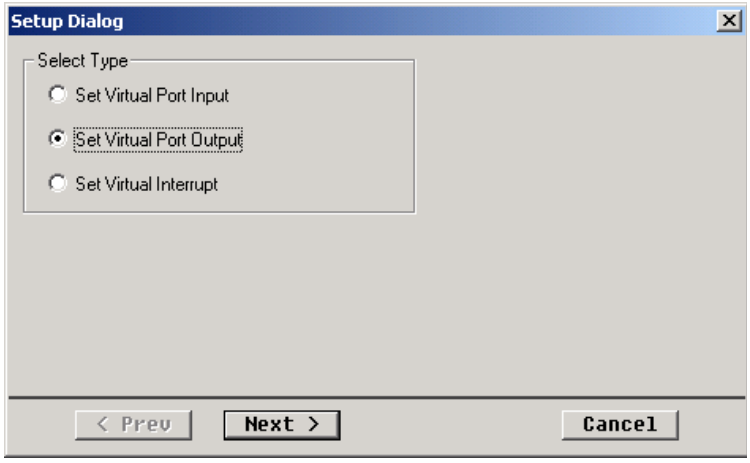


图 6.31 选择处理类型

4. 设定输出地址，然后点击［下一步>（Next>）］按钮。

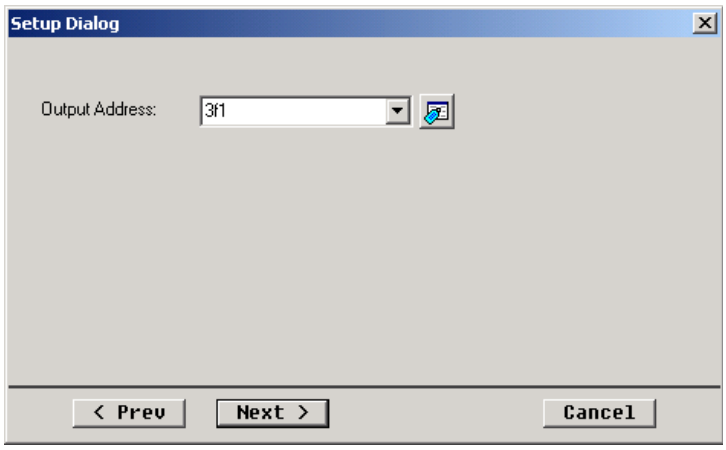


图 6.32 设置输出地址

5. 显示对话框，用以指明虚拟端口输出结果将要保存在的 I/O 脚本文件。指明文件名，保存文件。

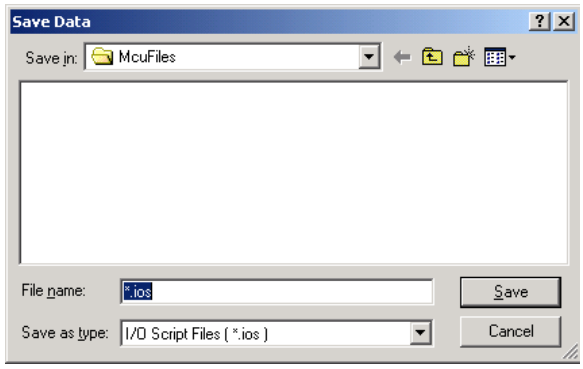


图 6.33 保存（Save） 对话框

6. 下面程序是虚拟端口输出的实例。

```
#include <stdio.h>
char *PORT_OUT;
static int i;
char buf[10];

void main(void)
{
   PORT_OUT = (char *)0x3f1;            → 输出地址

   sprintf(buf, "hello!!");             → 数据输出

   for(i = 0; i < 10; i++) {            → for 语句
            *PORT_OUT = buf[i];                → 端口输出
            if(buf[i] == '\0')          → 0x00 用作结束码
         break;
   }

}
```

7. ［I/O 时序设置 (I/O Timing Setting)］窗口的输出结果如下显示。

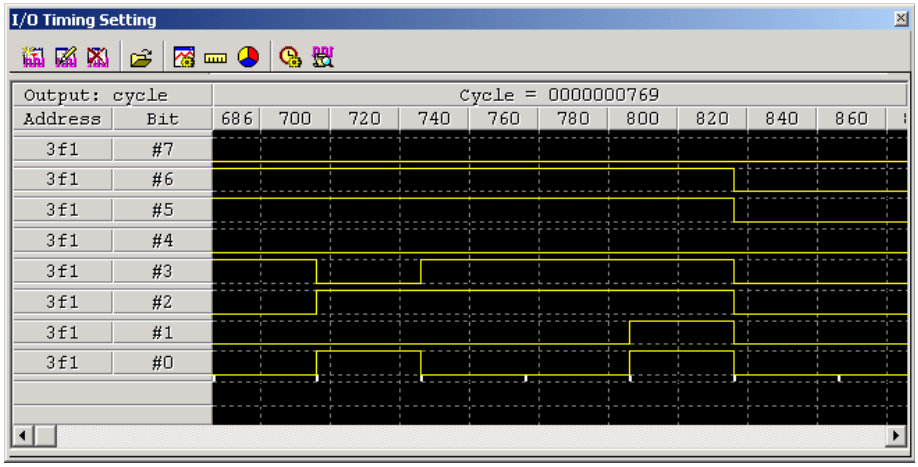


图 6.34 I/O 时序设置窗口

8. 当步骤 6 的程序执行后，步骤 5 的 I/O 脚本如下。

```
; IOSCRIPT FILE FOR I/O WINDOW (SET WAITC)
{
waitc 1145
set [0x3f1] = 0x68
waitc 30
set [0x3f1] = 0x65
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6f
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x0

}
```

(2)　输出端口窗口

1. 从菜单，选择［查看->CPU->输出端口 (View -> CPU -> Output Port)］来显示［输出端口（Output Port）］窗口。
2. 您可以点击端口设置（port setting）图标，或右键点击从显示菜单选择［设定（Set）］。

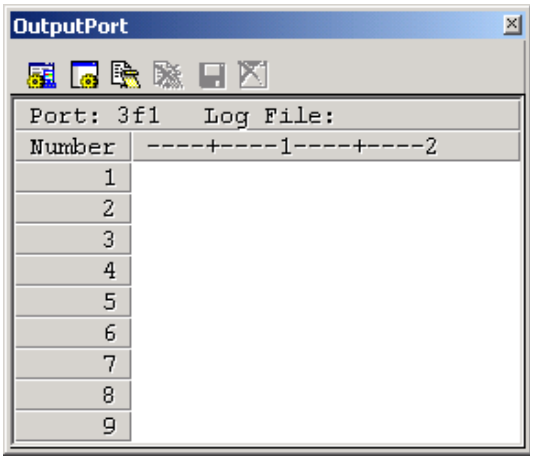


图 6.35 输出端口窗口

3. 显示端口设置对话框。
4. 选择［地址（Address）］，然后输入标签或地址。
5. 单击［确定（OK）］按钮来完成设定。

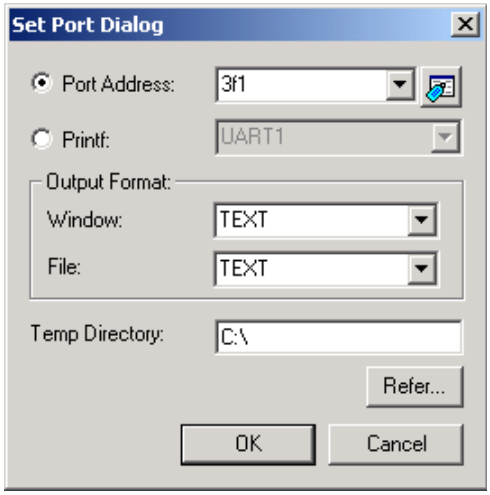


图 6.36 端口设置

6. 在程序运行时，如果有输出到指定地址，输出内容将显示在［输出端口（Output Port）］窗口。

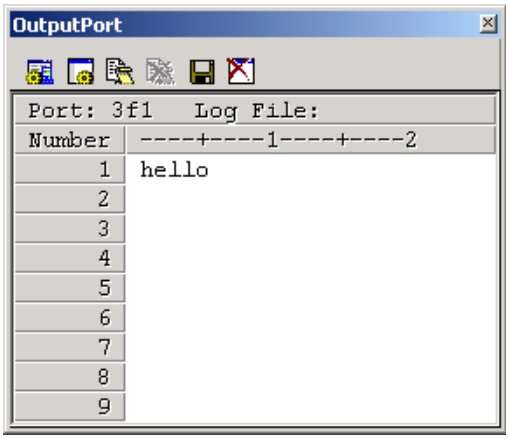


图 6.37 输出端口窗口

7. 点击开始日志（log start）图标，把端口输出内容放在日志文件中。

6.3 使用虚拟发光二极管（LED）或标签（Label）来检查存储器内容

■ 说明:

虚拟 LED 的颜色和标签的显示文本可以按照检测的存储内容实时更改。

■ 怎样使用虚拟 LED 和标签检测存储内容:

1. 从菜单选择［查看->图形->GUI I/O (View -> Graphics -> GUI I/O)］来显示图形用户界面（GUI）窗口。
2. 您可以点击 LED 创建（LED creation）图标或标签创建（label creation）图标，或右键点击从显示菜单选择［创建 LED（Create LED）］或［创建标签（Create Label）］。然后，放置虚拟 LED 或标签。

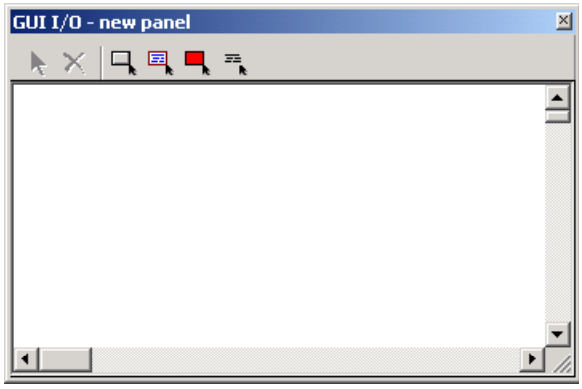


图 6.38 GUI 窗口

3. 点击放置好的目标来显示设置窗口。

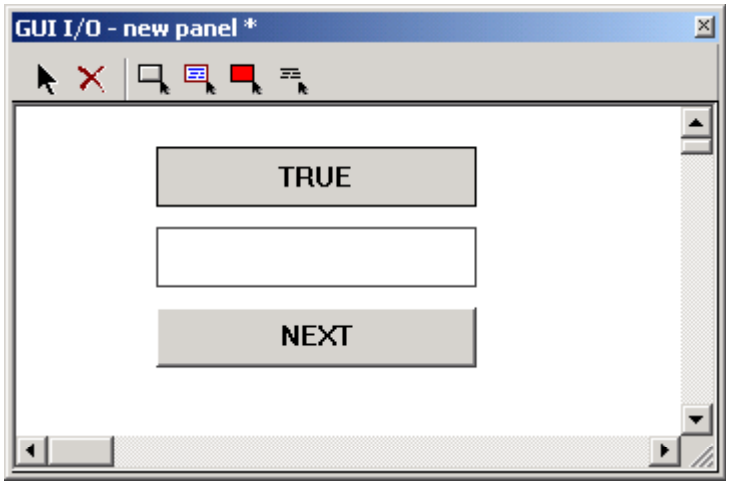


图 6.39 GUI 窗口（放好后）

4. 设定地址，选择［位（Bit）］或［数据（Data）］作为数据类型。
5. 对于虚拟 LED，设置显示颜色。对于标签，设置显示字串。
6. 对于［位（Bit）］数据类型，选择［正（Positive）］（显示颜色 1 或字串 1 当条件为正时）或［负（Negative）］（显示颜色 1 或字串 1 当条件为负时）。

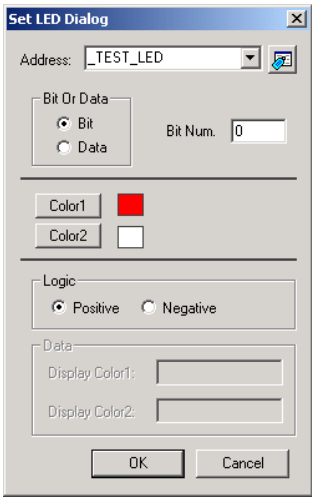


图 6.40 设置LED对话框

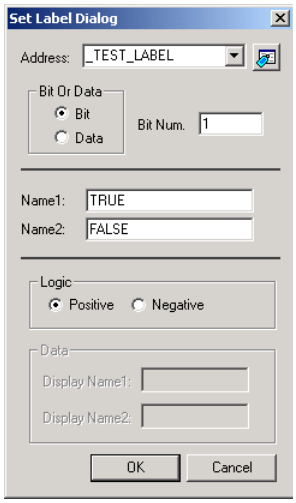


图 6.41 设置标签对话框

7. 以下为编程示例。在此例中，虚拟 LED 和标签根据从 0 到 200 的一个整数的位 0 和位 1 改变。

```
void main(void)
{
   for ( i = 0; i < 200; i++){
         TEST_LABEL = i;
         TEST_LED = i;
         while ( NEXT != 1);
         NEXT = 0;
   }
}
```

8. 对于［数据（Data）］数据类型，输入显示 1 和显示 2 的数据。

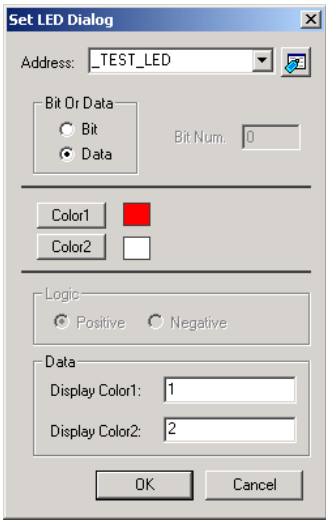


图 6.42 设置LED对话框

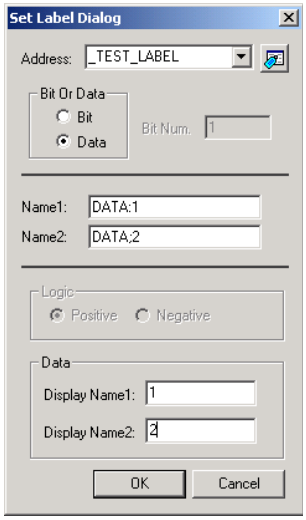


图 6.43 设置标签对话框

9. 当以下程序按步骤 8 中设定运行时，操作与步骤 7 中一样。

```
void main(void)
{
   for ( i = 0; i < 200; i++){
         TEST_LABEL = ((i >> 1) & 1) ?1 : 2;
         TEST_LED = ( i & 1) ?1 : 2;
         while ( NEXT != 1);
         NEXT = 0;
   }
}
```

6.4 使用 printf 进行调试

■ 说明：

源程序中通过 printf 输出的内容同样可以输出到［输出端口（Output Port）］窗口和日志文件。

■ 怎样输出：

1. 从菜单，选择［查看->CPU->输出端口（View -> CPU -> Output Port)］来显示［输出端口（Output Port）］窗口。
2. 您可以点击端口设置（port setting）图标，或右键点击从显示菜单选择［设定（Set）］。

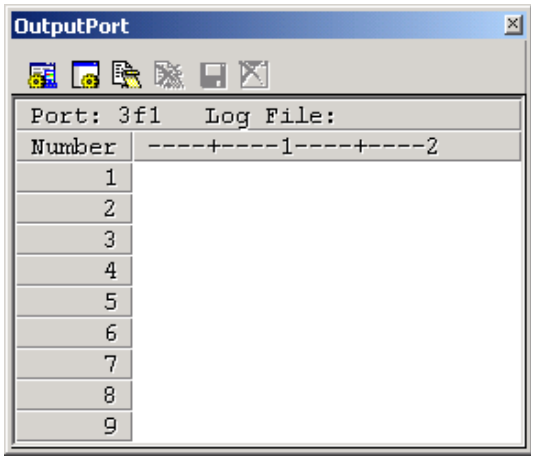


图 6.44 输出端口窗口

3. 显示端口设置对话框。
4. 选择［printf］然后选择［UART1］。
5. 单击［确定（OK）］按钮来完成设定。

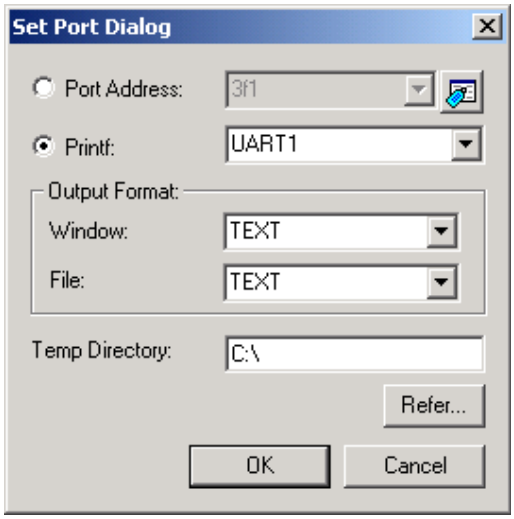


图 6.45 端口设置

6. 在程序运行中，如果用 printf 来输出，此端口的输出内容将会在［输出端口（Output Port）］窗口显示。程序运行后的输出显示如下。

```
for(i = 0; i < 10; i++) {
      :
#ifdef DEBUG
      printf ( "i=%d\n",i);
#endif
      :
}
```

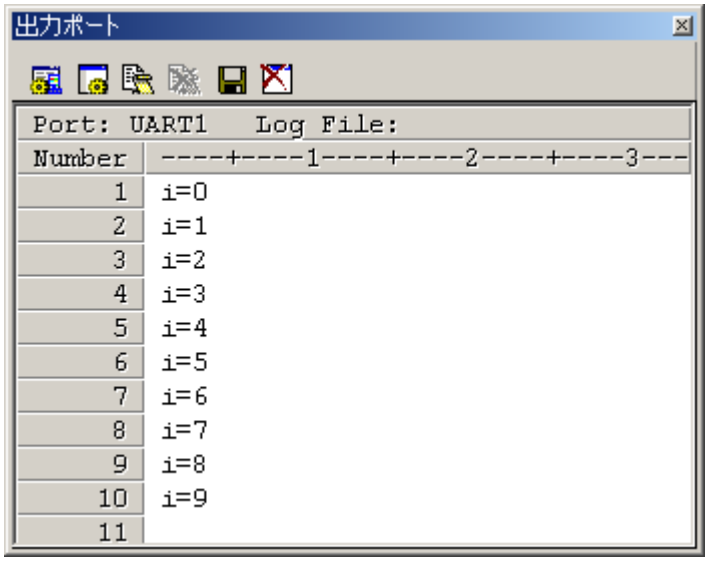


图 6.46 输出端口窗口

7. 点击开始日志（log start）图标，把端口输出内容放在日志文件中。

6.5 使用 I/O 脚本

■ 说明：

可以在文件中通过使用脚本格式指明虚拟端口输入和虚拟中断的设置。这种脚本被称作 I/O 脚本，它们所在的文件被称作 I/O 脚本文件。虽然 I/O 脚本文件可以通过［I/O 时序设置（I/O Timing Setting)］窗口自动生成，它们也可以被直接编辑以取得更适合的设置。例如，以下几种设置可以在［I/O 时序设置（I/O Timing Setting)］窗口外设置。

如果您想要生成周期性虚拟中断（好像计时器中断），您可以使用 while 语句来指明生成虚拟中断的循环。

您可指明中断控制寄存器的中断优先级选择位所设的优先级被用来解析生成的虚拟中断的优先级。

作为进入虚拟端口输入或生成虚拟中断的条件，您可指明程序取出、存储器读写访问或存储器比较的条件组合。

■ 使用 I/O 脚本文件的方法.

1. 预先使用文本编辑器建立一个 I/O 脚本文件。把文件扩展名设成 ios。
2. 在［I/O 时序设置（I/O Timing Setting)］窗口，点击引入（import）图标，或右键点击从显示的菜单选择［载入（Load）］来显示用来引入 I/O 脚本文件的对话框。
3. 选择要引入的 I/O 脚本文件。

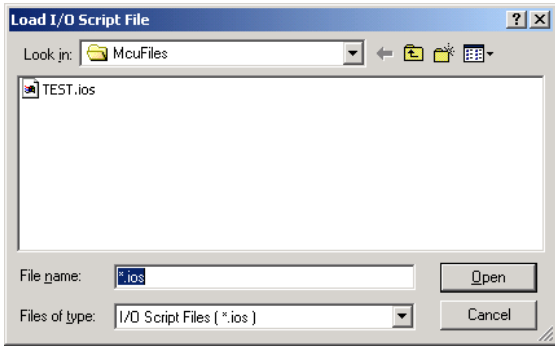


图 6.47 载入I/O脚本文件的对话框

■ 计时器 A0 的计时器模式的定义示例

以下示例为计时器 A0 的计时器模式的定义程序示例。

在此示例中，计时器 A0 中断发生在每一个为计时器 A0 指定的 divide-by-ratio （周期数）。中断控制寄存器中指明的值被用来确定这个计时器的中断优先级。

```
; 虚拟中断示例
{                                  → 程序开始
while(1){                          → while 语句
      if([0x380].b & 0x01) {       → 检查计时器A0计时起始标志
         waitc[0x386].w+1          → 在I/O脚本运行前，等待计时器A0中所设频率的周期数
         int   21,[0x55] & 0x7     → 产生计时器A0中断
                                      （中断控制寄存器被用来确定优先级）
      } else {
   waiti 100                       → 在I/O脚本运行前，等待100条指令
      }
}
}                                  → 程序结束
```

■ 与周期同步的虚拟端口输入的示例

以下示例为与周期同步的虚拟端口输入的程序示例

在此示例中，程序中执行 5000 个时钟周期时发生虚拟端口输入。[I/O 时序设置（I/O Timing Setting_] 窗口只能用来设定以字节为单位的虚拟端口输入，但 I/O 脚本文件却可以用来设定以字或长字为单位的虚拟端口输入。

```
; 虚拟端口输入示例
{                                  → 程序开始
   waitc 5000                      → 在I/O脚本运行前，等待5000个周期
   set[0x3e0] = 0x34               → 输入 0x34 到地址 0x3e0
   waitc 5000
   set[0x3e0].w = 0x4126           → 从地址 0x3e0 输入2字节数据0x4126
}                                  → 程序结束
```

M16C 族 C 编译器应用笔记

MISRA C

第 7 节 MISRA C

7.1 MISRA C

7.1.1 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。

7.1.2 规则实例

本小节将介绍一些实际的 MISRA C 规则。图 7.1 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 7.1 规则 62

图 7.2 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：

```
i = 1;
x = ++i + i;        x = 2 + 2?   x = 2 + 1?

j = 1;
func(j, ++j);       func(1, 2)? func(2, 2)?
```

图 7.2 规则 46

图 7.3 显示规则 38：移位运算符的右操作数应该大于零并且小于左操作数的位宽度减一。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 7.3 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。

实例：

```
unsigned short us;

us << 16;     ← 未定义的操作
us >> -1      ← 未定义的操作
```

图 7.3 规则 38

图 7.4 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。

实例：

```
if( 1UL - 2UL )          ← 预期什么：-1 或 0xFFFFFFFF?

*(char*)(0xfffffffeUL + 2);   ← 结果是 0 地址。
```

图7.4 规则 51

7.1.3 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 7.1 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 7.1 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
...	...	...	...	...

7.1.4 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。

7.1.5 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。

7.2 SQMlint

7.2.1 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 7.5 所示。它不会影响编译程序所生成的代码。

表 7.2 列出 SQMlint 支持的规则。

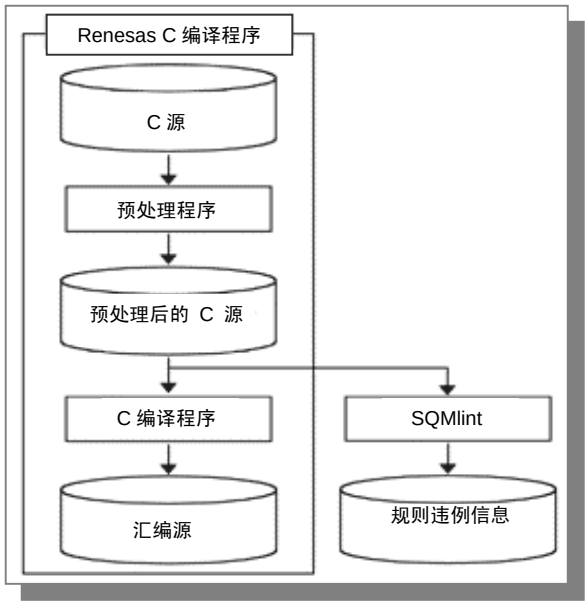


图 7.5 SQMlint 的定位

表 7.2 SQMlint 支持的规则

规则	测试	规则	测试	规则	测试	规则	测试	规则	测试	规则	测试
1	○	26	×	51	○*	76	○	101	○	126	○
2	×	27	×	52	×	77	○	102	○	127	○
3	×	28	○	53	○	78	○	103	○		
4	×	29	○	54	○*	79	○	104	○		
5	○	30	×	55	○	80	○	105	○		
6	×	31	○	56	○	81	×	106	○*		
7	×	32	○	57	○	82	○	107	×		
8	○	33	○	58	○	83	○	108	○		
9	×	34	○	59	○	84	○	109	×		
10	×	35	○	60	○	85	○	110	○		
11	×	36	○	61	○	86	×	111	○		
12	○	37	○	62	○	87	×	112	○		
13	○	38	○	63	○	88	×	113	○		
14	○	39	○	64	○	89	×	114	×		
15	×	40	○	65	○	90	×	115	○		
16	×	41	×	66	×	91	×	116	×		
17	○*	42	○	67	×	92	×	117	×		
18	○	43	○	68	○	93	×	118	○		
19	○	44	○	69	○	94	×	119	○		
20	○	45	○	70	○*	95	×	120	×		
21	○*	46	○*	71	○	96	×	121	○		
22	○*	47	×	72	○*	97	×	122	○		
23	×	48	○	73	○	98	×	123	○		
24	○	49	○	74	○	99	○	124	○		
25	×	50	○	75	○	100	×	125	○*		

○:可测试　×:不可测试　*:具有限制的可测试

表 7.3 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

7.2.2 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 7.6 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。

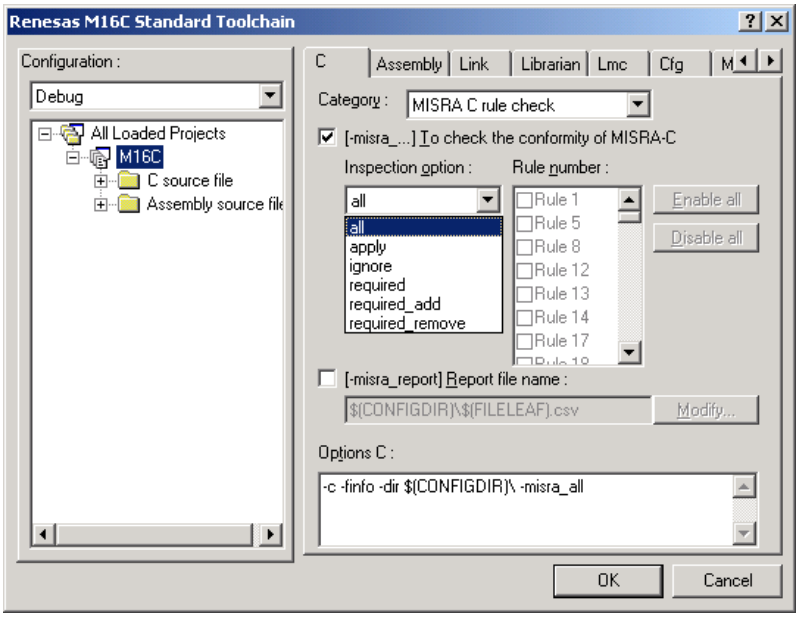


图 7.6 HEW 选项窗口

如此，SQMlint 将会在编译时间启动。以下提供每个选项的含义：

- [all]： 对所有规则执行测试。
- [apply]： 仅对指定的规则执行测试。
- [ignore]： 除了特定编号的规则，对所有规则执行测试。
- [require]： 根据 MISRA C 规则，仅对需要的规则执行测试。
- [require_add]： 根据 MISRA C 规则，仅对所有以及后续编号的规则执行测试。
- [require_remove]： 根据 MISRA C 规则，仅对需要的规则执行测试，除了特定编号的规则。

7.2.3 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。可以执行标签跳转，使用改正编译错误相同的操作来简易地修正源代码。

(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

同时显示源文件和测试结果，如图 7.7 所示。

```
1 : void func(void);
2 : void func(void)
3 : {
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 7.7 SQMmerger

7.2.4 开发程序

图 7.8 显示如何使用 SQMlint 执行开发。

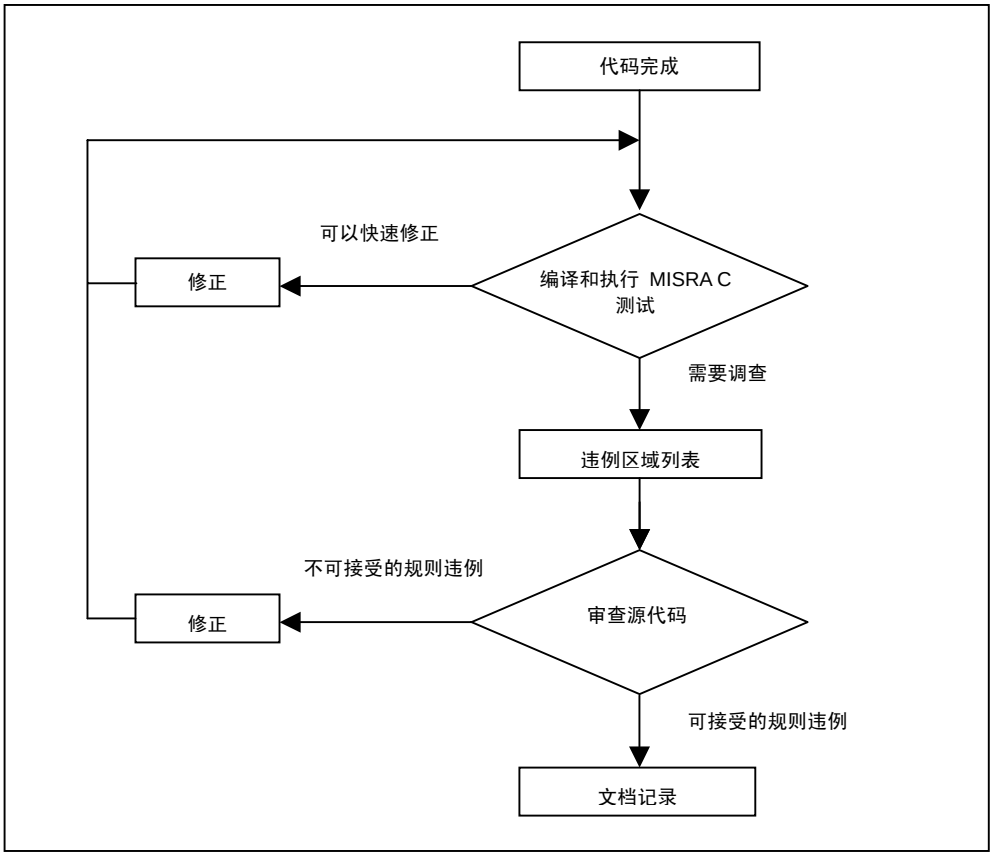


图 7.8 使用 SQMlint 的开发程序

- 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。
- 建立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。

7.2.5 支持的编译程序

SQMlint 支持下列编译程序：

- M3T-NC30WA，版本 5.20 版本 1 和更新版本
- M3T-NC308WA，版本 5.10 版本 1 和更新版本
- M3T-CC32R，版本 4.10 版本 1 和更新版本

M16C 族 C 编译器应用笔记

常见问题集

第 8 节 常见问题集

本章包含用户常见问题的解答。

8.1 C 编译程序 (M3T-NC308WA)

8.1.1 位字段

■ 问题：

我如何声明结合 10 位宽度、11 位宽度和 3 位宽度的位字段，而且字段之间没有空的位（填充）？

■ 解答：

通过将整个序列的字段声明为相同类型，而且该类型包含的位多于字段中的位总和，您可以建立不含空位的包型位字段。

实例程序：

```
typedef struct {
      unsigned long   BIT0_9   :10;
      unsigned long   BIT10_20 :11;
      unsigned long   BIT21_23 : 3;
} BIT0_23;
```

8.1.2 存储器管理函数

■ 问题：

我如何防止应用程序不需要的函数，如 MALLOC() 和 MEMCPY() 在 Nc*xx*WA 中连接？

■ 解答：

MALLOC() 和 MEMCPY() 函数用于保护使用 “ncrt0.a30” 和 “sectxx.inc”启动程序设定的 “堆” (HEAP) 区域。不使用如 MALLOC() 和 MEMCPY() 的存储器管理函数时，请在汇编“ncrt0.a30”和 “sectxx.inc”时指定 “-D__HEAP__=1” 汇编选项。

8.1.3 -ONBSD 选项

■ 问题：

编译程序优化选项 “－ONBSD” 会制止哪类优化？ 有可能对 “－ONBSD” 制止的优化执行更详细的设定吗？

■ 解答：

下列类型的优化将会被制止：

(1) 将公用表达式析出（以便使相同的表达式不会多次执行）
(2) 将公用指令块分组（例如，用于转移到相同系列的指令）
(3) 按字执行的每字节存储到连续区域
(4) 按字执行的 1 字节常数的连续推动运算
(5) 将连续移位分组（例如，将 b=a<<2; b<<=2; 转变为 b=a<<4;）
(6) 将位运算分组为 OR 和 AND 计算
(7) for 语句中的比较运算符移到循环的结束以缩减所执行的求值次数
(8) 以 return 语句直接替换无条件转移 return 语句（仅限于 NC308）
(9) 将使用指数 2 的指数计算更改为移位运算
(10) 将自动变量自动分配到寄存器
(11) 将常数合并

- 请注意，根据相应位置前面或后面的代码而定，这些优化并非始终被应用。

没有选项可以对优化制止执行更详细的设定。

8.1.4 优化选项的优先级

■ 问题：

指定多个优化选项时，这些选项将以什么优先级处理？
同时，哪个具有优先权：执行优化的选项，或制止它的选项？

■ 解答：

指定多个优化选项时，生效的选项如下所示：

(1) 指定 “-O1” 到 “-O5” 多个选项时，最后一个指定的将会生效。

(2) 指定 “-Onumber” 选项和 “-OR” 选项时，两者均会生效。

(3) 指定 “-Onumber” 选项和 “-OS” 选项时，两者均会生效。

- 请注意，“-OR” 和 “-OS” 选项不能一起指定。

例如，如果同时指定 “-OR” 和 “-O1”，-OR 优化将会执行，但 -O1 制止的优化则不会执行。

同时指定执行优化的选项和制止优化的选项时，后者将拥有优先权，而优化的执行将会以所指定的优化制止选项为根据。

8.1.5 将函数添加到程序库

■ 问题：

我该如何将函数添加到编译程序程序库？

■ 解答：

您可以按照下列以 NC308WA 为根据之实例中显示的方式将函数添加到编译程序程序库。

编译 C 源文件以生成可再定位文件。

要生成 “new.r30”：

```
>nc308 -c new.c
```

使用库管理程序 lb308 将可再定位文件添加到程序库。

要将 “new.r30” 文件添加到 “nc308” 程序库文件：

```
>lb308 -a nc308lib.lib new.r30
```

注意：
有关如何使用 lb308 的详情，请参阅 AS308 的文档，您可以在产品 CD-ROM 上的 nc308wa/manual directory（手册目录）中找到该文档。如果使用 Windows，此文档也会在产品安装期间安装，因此您也可以从 Windows [开始 (Start)] 菜单找到它。

8.1.6 将常数声明放置在 ROM 段中

■ 问题：

我以为使用常数声明的所有项目都会放置在 ROM 段中，但使用下列代码时，它却放置在数据段中。我应该怎么做以将它放置在 ROM 段中？

```
const S_TBL *sp_tbl[]={
      p00,  /* 指向结构的指针 */
      p01,  /* 指向结构的指针 */
      p02,  /* 指向结构的指针 */
  };
```

■ 解答：

可将如上所示的代码更改成如下所示的代码以将 sp_tbl 放置在 ROM 中：

```
S_TBL *const sp_tbl[]={
      p00,  /* 指向结构的指针 */
      p01,  /* 指向结构的指针 */
      p02,  /* 指向结构的指针 */
  };
```

此外，您可以使用以下代码，除了 sp_tbl，您还可将 p00、p01 和 p02 放置在 ROM 中：

```
const S_TBL *const sp_tbl[]={
      p00,  /* 指向结构的指针 */
      p01,  /* 指向结构的指针 */
      p02,  /* 指向结构的指针 */
  };
```

注意：
对指针使用常数声明时，放置在 ROM 中的项目将根据常数位置而有所不同。

在以下实例中，a、b 和 c 将放置在 ROM 中：

```
const int *i={a,b,c};
```

在以下实例中，i 将放置在 ROM 中：

```
int * const i={a,b,c};
```

8.1.7 通过寄存器传递参数

■ 问题：

函数参数会在什么时候通过寄存器传递？

此外，与堆栈传递比较，当函数参数通过寄存器传递时的大小和速度有什么差异？

■ 解答：

函数参数会在下列情况通过寄存器传递：

(1) 函数存在原型声明，以及参数类型是在函数调用时规定。

(2) 原型声明未使用… 变量参数。

(3) 函数参数的类型符合下表所显示的：

对于 NC30：

参数	参数类型	使用的寄存器
第一个参数	字符类型	R1L 寄存器
第一个参数	整数类型或 near 指针类型	R1 寄存器
第二个参数	整数类型或 near 指针类型	R2 寄存器

对于 NC308：

参数	参数类型	使用的寄存器
第一个参数	字符类型	R0L 寄存器
第一个参数	整数类型或 near 指针类型	R0 寄存器

此外，在大小和速度方面，通过寄存器传递参数都会更好。

8.1.8 函数参数如何传递

■ 问题：

函数参数的传递方式会影响程序的易传性吗？

■ 解答：

如果存在函数的原型声明，编译程序将决定函数参数的传递方式。因此，程序的易传性将不会受到影响。

8.1.9 原型声明

■ 问题：

原型声明会决定函数参数的传递方式吗？

■ 解答：

原型声明不单会决定函数参数的传递方式，它还会指定诸如参数大小等的属性。因此，当调用的函数不存在原型声明时，存在于个别文件中的函数将可能出现参数的一致性问题。为了防止这类问题，我们建议使用原型声明。

8.1.10 结构位字段中的成员放置

■ 问题：

我在 C 程序中定义了一个结构位字段，但是当我编译该程序时，位字段成员的顺序发生变化。我该如何防止这个问题？

■ 解答：

结构位字段中的成员放置根据下列条件决定：

(1) “unpack” 和 “arrange” 未在 #pragma STRUCT[#1] 中使用。

(2) 如果存在相同类型的连续位字段，成员将会连续放置。

(3) 如果存在不同类型的位字段，但前一个位字段是相同类型，成员将会放置在此前一个位字段的连续后面。

(4) 如果不存在相同类型的前一个位字段，成员将会从下一个地址放置。

(5) 成员将不会在非位字段成员中连续放置。

要将符合上述条件的成员以他们的指定顺序放置，请使用下列对策之一：

对策 1： 用括号将不同类型的每个位字段在其本身的结构中括起来 { }。

对策 2： 声明您要连续放置的字段时使用相同类型。

#1 不可使用选项或 #pragma STRUCT 更改。

以下是成员顺序发生变化的实例，以及上述各对策的实例。

成员顺序发生变化的实例：

```
struct tagData {
    unsigned char   c0a :  5;  /* (1) */
    unsigned char   c0b :  3;  /* (2) */
    unsigned char   c1a :  6;  /* (3) */
    unsigned char   c1b :  2;  /* (4) */
    unsigned int    i2a : 10;  /* (5) */
    unsigned int    i2b :  6;  /* (6) */
    unsigned char   c4a :  3;  /* (7) */
    unsigned char   c4b :  5;  /* (8) */
    unsigned char   c5;        /* (9) */
    unsigned char   c6a :  5;  /* (10) */
    unsigned char   c6b :  3;  /* (11) */
} s;
```

对于上述代码，成员将按照下列方式放置：

(1)、(2)、(3) 和 (4) 将会连续放置。

由于类型与 (5) 和 (6) 不同，相同类型的 (7) 和 (8) 成员将会先放置。

由于 (9) 是非位字段成员，(10) 和 (11) 不能从 (7) 和 (8) 继续。

(5) 和 (6) 将会从 (7) 和 (8) 之后的地址放置，因为它们的类型不同。

(9) 将会从 (5) 和 (6) 之后的地址放置，因为它是非位字段成员。

(10) 和 (11) 将会从 (9) 之后的地址放置。

对策 1： 用括号将不同类型的每个位字段在其本身的结构中括起来 { }。

```
struct tagData {
    struct {
        unsigned char   c0a :  5;  /* (1) */
        unsigned char   c0b :  3;  /* (2) */
        unsigned char   c1a :  6;  /* (3) */
        unsigned char   c1b :  2;  /* (4) */
    } s1;
    struct {
        unsigned int    i2a : 10;  /* (5) */
        unsigned int    i2b :  6;  /* (6) */
    } s2;
    struct {
        unsigned char   c4a :  3;  /* (7) */
        unsigned char   c4b :  5;  /* (8) */
    } s3;
    unsigned char   c5;            /* (9) */
    struct {
        unsigned char   c6a :  5;  /* (10) */
        unsigned char   c6b :  3;  /* (11) */
    } s4;
} s;
```

- 请注意，参考此结构的区域需要更改。

  之前： s.c0a = 1;

  之后： s.s1.c0a = 1;

对策 2： 声明您要连续放置的字段时使用相同类型。

```
struct tagData {
    unsigned int    c0a :  5;  /* (1) */
    unsigned int    c0b :  3;  /* (2) */
    unsigned int    c1a :  6;  /* (3) */
    unsigned int    c1b :  2;  /* (4) */
    unsigned int    i2a : 10;  /* (5) */
    unsigned int    i2b :  6;  /* (6) */
    unsigned int    c4a :  3;  /* (7) */
    unsigned int    c4b :  5;  /* (8) */
    unsigned char   c5;        /* (9) */
    unsigned char   c6a :  5;  /* (10) */
    unsigned char   c6b :  3;  /* (11) */
} s;
```

成员 (1) 到 (8) 和 (10) 到 (11) 将会连续放置。

使用此方法可能会稍微增加代码数量。

8.1.11 增量和减量运算符

■ 问题：

我编程了 (1) 所示的 C 代码。

检查生成的代码时，它在增量前的变量 x 是与 5 比较。如果要使变量 x 在它增量后与 5 比较，我需要将这个代码更改为类似 (2) 所显示的吗？

```
(1)   if (x++ == 5){
          aaasub();
      }
(2)   x++;
      if (x ==5){
          aaasub();
      }
```

■ 解答：

有两种方法可以使用 ++ 增量运算符和 -- 减量运算符： 变量前面，以及变量后面。

变量后面： 增量/减量在使用变量之后执行。

变量前面： 增量/减量在使用变量之前执行。

由于您程序中使用的增量运算符是放置在变量后面，因此变量 x 会在执行增量前与 5 比较。要实现预期的结果，请如下所示将增量运算符放置在变量前面：

```
if (++x == 5){
   aaasub();
}
```

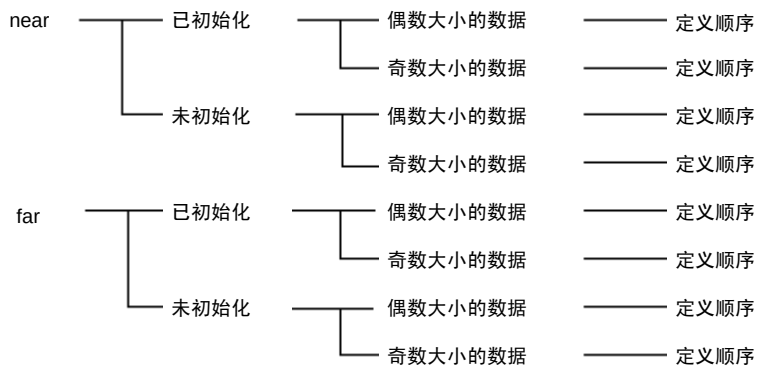
8.1.12 放置外部变量

■ 问题：

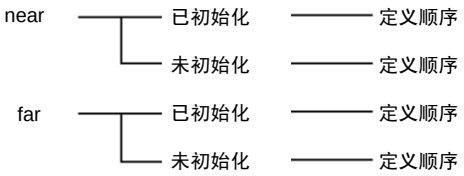
编写 C 程序时，我该如何按照外部变量的定义顺序放置它们？

■ 解答：

使用标准 Nc*xx*WA，外部变量将会分组放置在下列各属性中：



您不能将已初始化和未初始化的数据组合在一起，但是您可以在编译时指定“-fno_even”命令选项组合偶数大小的数据和奇数大小的数据，按照下列方式放置外部变量：



8.1.13 将数组放置到 far 区域中

■ 问题：

我应该在 NC30WA 中使用什么声明来将数组放置到 “far” 区域中，但参考它的指针是在 “near” 区域中？

■ 解答：

作为一个实例，使用类似下列的声明来声明短类型的 64 字节数组：

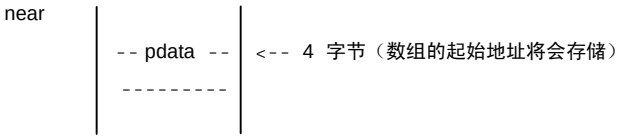
```
/* (1) 声明数组本身 */

    short far data[64];

 /* (2) 声明指向数组的指针 */

    short far *pdata;
```

(2) 所示的指针将会存储在 “near” 区域中，如下所示。此指针指向 “far” 区域。



您也可以更改放置指向数组的指针的区域，以及指针指向的区域，如下所示。

`short * far pdata`
在上述表达式中，pdata 将放置在 “far” 区域中，以及它所指向的将放置在 “near” 区域中。如果省略“far”，这将处理为“near”。

`short far * far pdata`
在上述表达式中，pdata 将放置在“far” 区域中，以及它所指向的将放置在 “far” 区域中。

`short far * pdata`
在上述表达式中，pdata 将放置在 “near” 区域中，以及它所指向的将放置在 “far” 区域中。

`short * pdata`
在上述表达式中，pdata 将放置在 “near” 区域中，以及它所指向的将放置在 “near” 区域中。

8.1.14 在固定地址放置函数

■ 问题：

我该如何在 C 程序中的绝对（固定）地址放置函数？

■ 解答：

NC30WA 会在程序段中放置函数。因此，要在绝对地址中放置函数，请指定程序段的位置。

由于程序段中的地址在 “sect30.inc” 文件中的 .section 程序行后面，因此可以使用 .org pseudo-instruction 来指定它们。

要在 10000H 地址放置函数，请在 “sect30.inc” 文件中指定以下：

```
.section  program
.org   10000H
```

要在个别地址放置每个函数，请对每个函数使用 #pragma SECTION 建立具备程序段名称以外的名称的段，然后更改该代码以便指定该段的地址。

要将 func() 函数放置在称为 program1 的段，然后在 20000H 放置该段：

(1) 您可以按照下列方式指定要放置函数的段的名称：

```
#pragma SECTION program program1
func()
{
}
```

(2) 在 “sect30.inc” 文件中，按照下列方式指定段地址：

```
.section  program1
.org   20000H
```

8.1.15 使用 #pragma ADDRESS 指定绝对地址

■ 问题：

我该如何使用 #pragma ADDRESS 指定和以下相同的绝对地址？

```
#define AAA (*(volatile unsigned char *)0x000406)
```

■ 解答：

要使用 #pragma ADDRESS 指定如上所示的 0x0406，请使用下列方式：

```
#pragma ADDRESS AAA 000406h
unsigned char AAA;
```

请注意，由 #pragma ADDRESS 声明的变量具有不具备其指定的 volatile 属性。

8.1.16 使用 #define 定义字符串

■ 问题：

当我使用预处理命令 #define 定义字符串中的数值时，为何使用定义的字符串的表达式返回未预期的结果？

例如，在以下表达式中，cul 值并非我所期待的 0x04AAAA。

```
#define      V1       0x040000
#define      V2       0x0AAAA + V1

cul = (WORD *)V2
```

■ 解答：

对于上述程序，#define 扩展的表达式结果如下（请注意，#define 扩展只包含字符串替换）：

```
cul = ( WORD *)0x0AAAA+ 0x040000;
```

在此表达式中，(WORD *) 只转型为已添加 0x040000 的 0x0AAAA。由于这将导致添加到指针到整数，它将变成 0x80000。

要获得预期的值，用括号将 #define 定义的表达式括起来，如下所示：

```
#define      V2       (0x0AAAA + V1)
```

8.1.17 位字段成员的类型

■ 问题：

当我执行以下程序时，我以为长类型变量应该被 －1 取代，但它却是替换为 0xF (15)。我该如何将它设定为 -1？

```
void main( void )
{
    struct
    {
        short r : 10 ;
        short i : 4 ;
        short s : 2 ;
    } buf ;
    long l ;


    buf.i = -1 ;
    l = (long)buf.i ;
}
```

■ 解答：

位字段成员使用无符号作为其类型处理。因此，具有大类型的变量在存储或转移时将是零扩展。

如上所示，要将此类变量作为有符号扩展处理，请明确地声明该位字段成员的类型为“有符号”。

```
struct
{
    signed short r : 10 ;
    signed short i : 4 ;
    signed short s : 2 ;
} buf ;
```

8.1.18 重复变量定义

■ 问题：

在 C 程序中，如果在同个文件中超过一次声明具有相同名称和类型的变量，会出现错误吗？

■ 解答：

如果在同个文件中超过一次声明具有相同名称的变量将不会出现错误或警告。

这是因为采用下列 ANSI 指定：

注意：
“只要变量的类型是分配到已知不会更改的同个文件中，该名称将可存在任何数量的外部声明。”

下列将不会出现错误：

```
int i;
int i=1;

main()
{
}
```

下列将出现重复定义错误：

```
int i;
char i;

main()
{
}
```

8.1.19 函数的原型声明

■ 问题：

为何连接期间会输出 “*function-name*: value is undefined”（*函数名称*：值未定义）错误信息，即使存在该函数？

■ 解答：

当其中一边的函数名称调用函数时，或被调用的一边是 *_function-name* 而另一边是 $*function-name* 时就会发生此错误。这通常是因为该函数不存在原型声明，或函数的参数类型不符合。请检查原型定义。

8.1.20 不具备 extern 声明之函数的外部参考

■ 问题：

当函数在不执行 extern 声明的情形下执行外部参考时，会出现编译程序错误吗？

■ 解答：

如果存在 extern 声明（原型声明），ANSI 指定将会规定函数的返回值解释为整数值。因此，当您对外部参考使用整数类型时，将不会出现错误。

要检测不具备 extern 声明（原型声明）的函数，请在编译时指定“-Wno_prototype (-WNP) ”或“-Wall”。这将对不具备原型声明的任何函数输出警告。

8.1.21 优化期间的代码删除

■ 问题：

使用优化选项时为何没有输出表达式的代码（例如，正在读取端口）？

■ 解答：

这是因为代码在优化期间被当着是无意义而删除。对于具有目的（例如，读取）的运算，请使用 volatile 声明来声明任何变量。

8.1.22 合并位存取

■ 问题：

由于连续位存取运算合并为一个运算，当位存取期间出现中断时，程序有时就会出现故障。我该怎么做？

■ 解答：

有时发生的故障是因为优化所导致的连续位计算合并为一个计算。要避免此故障，您可以通过在编译时指定 “-ONB (-Ono_bit) ” 选项来制止此类优化。

8.1.23 在 ROM 地址放置程序库函数

■ 问题：

我该如何指定要在连接期间放置程序库函数的 ROM 地址？ 此外，是否具备程序库函数的段以便让我可以在连接期间指定段的放置？

■ 解答：

抱歉，没有程序库函数的段。

要更改任何程序库函数的段，请在程序库函数的源程序中，指定 #pragma SECTION 扩展函数中的段名称。NC30WA 和 NC308WA 都具备各程序库的源。

在程序库函数的起始部分，添加下列 #pragma SECTION 行。

```
#pragma SECTION program library
```

将放置程序库函数的段的名称是 “library”。您可以按照需要指定段名称。

接着，按照正规函数的方法编译和汇编程序库函数之后，将 “library” 段放置在所需的地址，然后执行连接。

注意：

#pragma SECTION 不能用于在汇编程序中更改程序库的段名称。直接更改 .section 程序。

程序库函数的源存储在 NC30WA(NC308WA) 安装目录的 src30\lib(src308\lib) 中，如果在安装期间作如此选择。

8.1.24 负整数计算的处理

■ 问题：

关于以下计算如何执行：

(1) 使用 / 运算符舍入的负整数除法之结果的方向是什么？

(2) 使用 % 运算符的取模运算之结果的符号是什么？

(3) 当负整数使用 >> 运算符进行位移位时，将执行哪一项：算术移位（MSB 变成符号位）或逻辑移位（MSB 变成 0）？

■ 解答：

(1) 使用 / 运算符的负整数除法之结果将舍入为 0。

表 ﾋ式	结果
(-10) / 3	-3
(-10) % 3	-1
10 / (-3)	-3
10 % (-3)	1
(-10) / (-3)	3
(-10) % (-3)	-1

(2) 使用 % 运算符的取模运算之结果的符号是被除数的符号。

表 ﾋ式	结果
(-10) % 3	-1
10 % (-3)	1
(-10) % (-3)	-1

注意：
如果指定 “-fround_under_div(-fRUD) ” 选项，将使用除数的符号。

(3) 当负整数使用 >> 运算符进行位移位时，将执行算术移位，MSB 将变成符号位。

8.1.25 整数类型大小

■ 问题：

在以下程序中，为何 aaa = (signed long)(BBB << CCC); 表达式的右边产生 0x0000f000？ 在 VISUAL C++ 中，它产生 0x0ffff000。

实例程序	生成的汇编代码

`#define BBB (unsigned short)0xffff`
`#define CCC (unsigned char)12`
 	
`signed long aaa;`
 	
`test( void )`
`{`
`aaa = (signed long)( BBB << CCC );`
`}`
	`_test:`
	`mov.w #0f000H,_aaa`
	`mov.w #00000H,_aaa+2`
	`rts`

■ 解答：

编译程序按照下列顺序处理上述程序的表达式：

(1) 将 BBB 转换成整数类型。

（如果之前的值可表达为整数类型，将使用有符号的整数。）

（如果之前的值不能表达为整数类型，将使用无符号的整数。）

(2) 对 CCC 执行移位。

(3) 结果将转换为有符号的长型，以及存储在 aaa 中。

由于 NC30 和 NC308 处理 16 位整数类型的转换，表达式 aaa = (signed long)(BBB << CCC); 将如下处理：

(1) 0xffff -> 0xffff（这将不能表达为整数类型，并且会转换为无符号的整数类型。）

(2) 0xffff<<12 -> 0xf000

(3) 由于 0xf000 没有符号，它将保持 0x0000f000，即使是在转型到 signed long. -> aaa 时。

结果将是 0x0000f000。

在 NC30 和 NC308 中，要使表达式 aaa = (signed long)(BBB << CCC); 的结果为 0x0ffff000，请指定用于转型 BBB 变量的代码，而不是将移位的结果转型为有符号的长型。

```
#define BBB   (unsigned short)0xffff
#define CCC   (unsigned char)12

signed long aaa;

test( void )
{
    aaa = (signed long)BBB<<CCC;
}
```

以下汇编代码会在编译上述程序时生成。

```
;## # C_SRC :     aaa =  (signed long)BBB << CCC;
        mov.w    #0f000H,_aaa
        mov.w    #00fffH,_aaa+2
```

注意：

VISUAL C++ 将整数类型处理为 32 位。

因此，在 VISUAL C++ 中，表达式 aaa = (signed long)(BBB << CCC); 将如下处理：

(1) 0xffff -> 0x0000ffff

(2) 0x0000ffff << 12 -> 0x0fff000

(3) 0x0fff000 -> aaa

此计算的结果将是 0x0fff000。

8.1.26 控制 enter 指令

■ 问题：

使用 NC308WA 版本 3.00 Release 1，enter 指令在函数的开始输出。但由于在使用版本 3.10 Release 2 时没有输出，堆栈参数不能在内联汇编代码中参考。有方法可以控制 enter 指令吗？

■ 解答：

使用 NC308WA 版本 3.10，已加强优化为不输出不必要的 enter 指令。然而，由于未将内联汇编代码的处理考虑在内，当参数和自动变量不在从 #pragma ASM 至 #pragma ENDASM 或其他 asm() 范围以外参考时，enter 指令将被当作不必要，因此不会输出。

NC308WA 支持使用 asm("... $$ or $@ ...", variable-name); 来使用内联汇编代码以参考 C 变量。

由于此方法可以让您从编译程序正确检查是否使用该值，因此您可以在必要时生成 enter 指令。如果您使用其他方法参考 C 变量，运算可能变成不可预知，或可能遗失版本之间的兼容性。在使用此方法参考变量和参数时，不要设定函数返回值。

我们建议您使用如下列实例中的代码。

实例 1：

```
void memset(char *p, char c, unsigned short n)
{
  asm(" PUSHM  A1,R0,R3   ");     // 使用的寄存器必须保存
  asm(" MOV.L  $$[FB],A1  ", d); // 参数 d 转移到 A1
  asm(" MOV.B  $$[FB],R0L ", c); // 参数 c 转移到 R0L
  asm(" MOV.W  $$[FB],R3  ", n); // 参数 n 转移到 R3
  asm(" SSTR.B            ");
  asm(" POPM   A1,R0,R3   ");     // 使用的寄存器已恢复
}
```

实例 2：

```
void memset(char *p, char c, unsigned short n)
{
  asm(" PUSHM  A1,R0,R3   ");     // 使用的寄存器必须保存
  asm(" MOV.L  $@,A1    ", d); // 参数 d 转移到 A1
  asm(" MOV.B  $@,R0L   ", c); // 参数 c 转移到 R0L
  asm(" MOV.W  $@,R3    ", n); // 参数 n 转移到 R3
  asm(" SSTR.B          ");
  asm(" POPM   A1,R0,R3   ");     // 使用的寄存器已恢复
}
```

对于自动变量，$$ 将替换为 FB 寄存器值的偏移。对于外部变量、符号，或寄存器变量，$$ 将替换为寄存器名称。

$@ 将替换为标示自动变量、外部变量，或寄存器变量的操作数。与 $@ 相应的变量类型将由编译程序自动决定。

优化可能会将函数参数和自动变量更改为寄存器变量，但不会是使用 asm("$$ or $@",variable-name) 指定的变量。因此，不会出现这样的情况，例如，想要写入自动变量 FB 之偏移的 $$ 变成寄存器变量，以及意外扩展至寄存器名称。

8.1.27 浮点数库(Floating-point Library)的性能

■ 问题:

浮点数库 (Floating-point Library)的运算性能如何?

■ 解答:

浮点数库 (Floating-point Library)的运算性能如下:

测试条件

(1) 编译程序: M3T-NC30WA V.5.30 Release 02

(2) 仿真器: M16C/Tiny 系列使用的仿真器 (M30290T2-CPE)

时钟 20 MHz 高速(High-speed) 模式

测试结果

- 算术运算

	float	double
加法 (7.1+2.9)	33 μsec	37 μsec
减法 (7.9-2.9)	35 μsec	40 μsec
除法 (3.5/0.5)	79 μsec	164 μsec
乘法 (7.5*10.2)	22 μsec	37 μsec

- 数学函数

acos(0.5)	9.718 msec
asin(0.5)	10.682 msec
atan(0.5)	7.984 msec
atan2(1.5,0.5)	13.119 msec
ceil(10.8)	0.104 msec
cos(0.5)	2.775 msec
cosh(0.5)	1.934 msec
exp(10.4)	1.533 msec
fabs(x)	0.010 msec
fmod(10.5,1.5)	0.218 msec
ldexp(10.4,3)	0.006 msec
log(2.4)	4.587 msec
log10(100.3)	5.295 msec
pow(10.3,4.5)	7.061 msec
sin(0.4)	3.047 msec
sinh(0.6)	1.938 msec
sqrt(100.1)	2.112 msec
tan(0.5)	2.717 msec
tanh(0.9)	1.966 msec

8.2 连接程序

8.2.1 ln308 和 ln30 的 “-LOC” 选项

■ 问题：

在什么情况下我该使用 ln308 或 ln30 的 “-LOC” 选项？

■ 解答：

此选项可在 RAM 中运行之程序的应用程序中使用。在此情形下，在 RAM 中运行之程序的 RAM 地址将使用 “-ORDER” 选项定义。然后，寄存转移至 RAM 之程序的 ROM 地址将使用 “-LOC” 选项定义。

请注意，“-LOC” 选项功能仅在指定的地址寄存定义的程序，并且不具备在执行期间将该程序转移至地址区域的任何功能。

8.2.2 连接期间的警告

■ 问题：

连接期间输出的 “16-bits unsigned value is out of range 0 -- 65535. address='*xxxx*'”（16 位无符号值超出范围 0 -- 65535. address='*xxxx*'）警告，在减少 RAM 大小时并没有出现。发生什么事？

■ 解答：

上述警告会在位运算输出到位运算无法到达的区域（0H 至 1FFFH 以外）时输出。减少 RAM 大小可防止该警告输出的原因是，从 0H 至 1FFFH 以外区域的变量适合 0H 至 1FFFH 区域，并且是位运算可以到达的。放置变量的地址不能在编译时决定，因为它在连接期间决定。

请执行以下操作以删除输出位运算的设定：

(1) 如果在编译期间指定 -fbit（或 -fB），请清除此指定。

(2) 如果相应变量在 #pragma BIT 中指定，请清除此指定。

8.2.3 更改起始地址

■ 问题：

我该如何将起始地址编码为 0xFC0000 的程序转换为具备 0x000000 起始地址的 Motorola S 格式？

此外，当我运行 lmc308 时为何起始地址没有如下更改？

```
>lmc308 -E 00 test.x30
```

■ 解答：

lmc308 不能更改起始地址。“-E” 命令选项用于寄存执行起始地址，它以 S8 的数字 5 至 8 记录在 Motorola S 格式的最后一行。然而，请紧记此选项不能更改起始地址。

8.3 Stk 阅读器

8.3.1 Stk 阅读器堆栈大小

■ 问题：

Stk 阅读器中显示的堆栈大小有受到保护，虽然如此还是出现溢出。是否 Stk 阅读器中显示的堆栈大小并不足够？

■ 解答：

Stk 阅读器中显示的堆栈大小仅供参考。

由 Stk 阅读器调用的 Stk，根据编译程序所输出的绝对模块文件中的堆栈信息计算堆栈大小，但由于此大小只是理论上的，它不是从跟踪程序计算的实际大小。

同时，请注意，Stk 并没有将中断函数考虑在内。

要计算将中断函数考虑在内的堆栈大小，请将如 Stk 阅读器中显示的出现中断的函数或汇编例程的堆栈大小，添加到中断函数的堆栈大小。

8.4 SQMLint

8.4.1 选择测试规则

■ 问题：

是否有方法从那些可以使用 SQMlint 测试的规则中只选择必要的规则？

■ 解答：

您可以从那些可以使用 SQMlint 测试的规则中选择要测试的特定规则。您可以如下选择这些规则：

(1) 测试所有可以使用 SQMlint 测试的规则。

(2) 在可以使用 SQMlint 测试的规则中，仅测试 MISRA C 认为是“必需”的。

(3) 在可以使用 SQMlint 测试的规则中，仅测试所有认为是“必需”的，以及那些认为是已指定成员之“咨询”的。

(4) 在可以使用 SQMlint 测试的规则中，仅测试那些已指定成员的。

有关可使用 SQMlint 及其成员测试之规则的详细信息，请在第 7 章中参考*表 7.2 SQMlint 支持的规则*。

8.4.2 输出报告文件

■ 问题：

我想要在编译多个 C 源文件的同时，将使用 SQMlint 检查的结果输出至报告文件。我该如何输出每个源文件的个别报告文件？

■ 解答：

使用 HEW 或 TM 嵌入式工作区。

- 使用 HEW：

对于 M3T-NC30WA 或 M3T-NC308 WA：

(1) 选择 [选项（Options）] 菜单，然后选取 [瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）]。

(2) 在显示的对话框的 [C] 标签中，在 [类别 (Category)] 选取 [MISRA C 规则检查（MISRA C Rule Check）]。

(3) 如下设定 [[-r] 报告文件名：（[-r] Report file name:）] ：

```
$(CONFIGDIR)\$(PROJECTNAME).csv
```

对于 M3T-CC32R

(1) 选择 [选项（Options）] 菜单，然后选取 [瑞萨 M32R 标准工具链（Renesas M32R Standard Toolchain）]。

(2) 在显示的对话框的 [C] 标签中，在 [类别 (Category)] 选取 [MISRA C 规则检查（MISRA C Rule Check）]。

(3) 如下设定 [-misra_report 报告文件名：（-misra_report Report file name:）] ：

```
$(CONFIGDIR)\$(PROJECTNAME).csv
```

- 使用 TM：

对于 M3T-NC30 WA 或 M3T-NC308 WA：

(1) 选择 [工程（Project）] 菜单，然后选取 [选项浏览器（Option Browser）]。

(2) 选择 [CFLAGS]，然后选取 [编辑...（Edit...）]。

(3) 在选项更改类别中，选取 [MISRA-C 检查选项（MISRA-C check option）]。

(4) 选取 [- sqmlint] 选项，然后如下设定参数字符串：

```
-misra all -r $*.csv
```

对于 M3T-CC32R

(1) 选择 [工程（Project）] 菜单，然后选取 [选项浏览器（Option Browser）]。

(2) 选择 [CFLAGS]，然后选取 [编辑...（Edit...）]。

(3) 在选项更改类别中，选取 [MISRA-C 检查选项（MISRA-C check option）]。

(4) 选取 [-misra_report] 选项，然后如下设定参数字符串：

```
$*.csv
```

注意：
从命令行输入命令时，请如下指定每个所编译文件的报告文件的名称：

```
>nc308 -c test.c -sq -sqmlint "-misra all -r test.csv"
```

也就是说，为每个命令编译一个 C 源文件。

例如，当以下在命令行执行时，只有 test2.c 的报告结果会保存在 “report.csv” 文件中。

```
>nc308 test1.c test2.c -sq -sqmlint "-misra all -r report.csv"
```

8.4.3 报告信息 (1)

■ 问题：

当我指定原型声明之参数的枚举类型，以及将计数指定为调用函数时的实际参数时，下列信息会在即使它们是相同类型时输出。为何会输出此信息？

实例 C 源：

```
enum E { A, B, C };
void func1(enum E);

void func2()
{
   func1(A);
}
```

信息输出：

```
Rule 77 (Complaining) "parameter type shall be compatible with prototype, the 1st
parameter"
```

■ 解答：

枚举类型是整数。

由于 SQMlint 对虚设参数及其相应实际参数执行严格比较，因此枚举 E 类型和整数类型被当作是不同的，所以会输出报告信息。请忽略所有的此类信息。

8.4.4 报告信息 (2)

■ 问题：

当我在 the : 三元运算符 的两边指定一个计数，以及用枚举变量取代结果的返回值时，下列信息会在即使它们是相同类型时输出。为何会输出此信息？

实例 C 源：

```
enum E { A, B, C };

void func(int i)
{
    enum E e;
    e = (i == 0) ? A : B;
}
```

信息输出：

```
Rule 43 (Complaining) "information loss conversion (from 'signed int' to 'enum
E') in assignment operation"

Rule 29 (Complaining) "enum type object to which has not been assigned own
enumerator"
```

■ 解答：

当您在 the : 三元运算符 的两边指定计数时，运算符会返回整数类型的值。因此，这和用枚举变量取代整数类型常数的情况一样，所以会输出报告信息。请忽略所有的此类信息。

8.5 HEW

8.5.1 文件的连接顺序

■ 问题：

我想要在 HEW 中建立一个工程，但文件的连接变成按文件名的字母顺序进行。我该如何让启动文件 (ncrt0.r30) 先连接？

■ 解答：

请执行以下操作：

- 对于 M3T-NC30WA 版本 5.20 R1：

(1) 选择 [选项（Options）] 菜单，然后单击 [瑞萨 M16C 标准工具链...（Renesas M16C Standard Toolchain...）]。

（[瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）] 对话框将会显示。）

(2) 单击 [连接（Link）] 标签。

(3) 在 [类别：（Category:）] 中，选取 [输入（Input）]。

(4) 在 [显示有关项目：（Show entries for:）] 中，选取 [可再定位文件（Relocatable files）]。

(5) 单击 [添加（Add）] 按钮。

（[添加可再定位文件（Add Relocatable Files）] 对话框将会显示。）

(6) 在 [相对于：（Relative to:）] 中，选取 [配置目录（Configuration directory）]。

(7) 在 [文件路径：（File path:）] 中，输入“ncrt0.r30”。

(8) 在 [添加可再定位文件（Add Relocatable Files）] 对话框中，单击 [确定（OK）] 按钮。

(9) 在 [瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）] 对话框中，单击 [确定（OK）] 按钮。

这将可以让您先连接“ncrt0.r30”。

- 对于 M3T-NC30WA 版本 5.30 R1、M3T-NC308WA 版本 5.20 R1，以及 M3T-NC8C 版本 5.30 R1：

建立新的工程工作空间时，只有启动程序的名称是“ncrt0.a30”时才会先自动连接文件。

否则，可按照下列方式完成：

(1) 选择 [选项（Options）] 菜单，然后单击 [瑞萨 M16C 标准工具链...（Renesas M16C Standard Toolchain...）][#1]。

（[瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）] 对话框将会显示。）

#1
对于 M3T-NC308WA，将是 [瑞萨 M32C 标准工具链...（Renesas M32C Standard Toolchain...）]。

对于 M3T-NC8C，将是 [瑞萨 R8C 标准工具链...（Renesas R8C Standard Toolchain...）]。

(2) 单击 [连接（Link）] 标签。

(3) 从 [类别：（Category:）]，选取 [输入（Input）]。

选取 [启动程序连接到最前面。（Start-up program is linked to a head.）] 选项使启动文件先连接。

对于文件名不是“ncrt0.a30”的启动程序，请使用上述的第一个方法。

[可再定位文件（Relocatable files）] 列表的概述：

当寄存在工作空间的文件添加到 [可再定位文件（Relocatable files）] 列表时，它们会在文件寄存到工作空间之前先连接。

当没有寄存在工作空间的文件添加到 [可再定位文件（Relocatable files）] 列表时，它们会在文件寄存到工作空间之后连接。

添加到 [可再定位文件（Relocatable files）] 列表的文件将按列表顺序连接。

8.5.2 可再定位文件的连接顺序

■ 问题：

我该如何更改可再定位文件之连接顺序？

■ 解答：

可再定位文件以下列顺序连接：

(1) 同时寄存到工作空间和 [可再定位文件（Relocatable files）] 列表的文件将按 [可再定位文件（Relocatable files）] 列表的顺序连接。

(2) 仅寄存到工作空间的文件将按字母顺序连接。

(3) 仅寄存到[可再定位文件（Relocatable files）] 列表的文件将按列表顺序连接。

按照需要将文件添加到 [可再定位文件（Relocatable files）] 列表。

8.5.3 生成 Motorola S 格式文件

■ 问题：

我该如何生成 Motorola S 格式文件？

■ 解答：

请执行以下操作：

(1) 选择 [选项（Options）] 菜单，然后单击 [创建阶段（Build Phase）]。

[创建阶段（Build Phase）] 对话框将会显示。

(2) 单击 [创建顺序（Build Order）] 标签。

创建阶段的顺序列表将会显示。

对于 M3T-NC30WA 版本 5.20 Release 1，请选取 [M16C S 类型转换器（M16C Stype Converter）]。

对于 M3T-NC30WA 版本 5.30 Release 1，请选取 [M16C 装入模块转换器（M16C Load Module Converter）]。

对于 M3T-NC308WA 版本 5.20 Release 1，请选取 [M32C 装入模块转换器（M32C Load Module Converter）]。

对于 M3T-NC8C 版本 5.30 Release 1，请选取 [R8C 装入模块转换器（R8C Load Module Converter）]。

对于 M3T-CC32R，请选取 [M32R S 类型转换器（M32R Stype Converter）]。

(3) 单击 [确定（OK）] 按钮以关闭对话框。

(4) 选择 [选项（Options）] 菜单，然后单击 [瑞萨 M16C 标准工具链...（Renesas M16C Standard Toolchain...）][#1]。

[瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）] 对话框将会显示。

#1
对于 M3T-NC308WA，将是 [瑞萨 M32C 标准工具链...（Renesas M32C Standard Toolchain...）]。

对于 M3T-NC8C，将是 [瑞萨 R8C 标准工具链...（Renesas R8C Standard Toolchain...）]。

对于 M3T-CC32R，将是 [瑞萨 M32R 标准工具链...（Renesas M32R Standard Toolchain...）]。

(5) 在 [瑞萨 M16C 标准工具链（Renesas M16C Standard Toolchain）] 对话框中，单击 [Lmc] 标签。

[装入模块转换器的选项设定（Option Settings for Loading Module Converters）] 窗口将会显示。

(6) 设定必要选项，然后单击 [确定（OK）] 按钮以关闭对话框。

如此即完成了这些设定。

从现在开始，装入模块转换器将会在创建时执行。

8.5.4 安装 HEW (1)

■ 问题：

我在已安装 HEW 环境的机器上安装了 M3T-NC30WA 随附的 HEW 版本。但即使是在启动 HEW 时，M16C 都没有显示为工具链。

■ 解答：

安装 M3T-NC30WA 随附的 HEW 版本时，请确定将它安装到已安装 HEW 的相同目录中，以覆盖先前的安装。此外，由于 HEW 环境需要在 M3T-NC30WA 之前安装，请确定使用 M3T-NC30WA 编译程序的安装程序。

对于使用 SHC 版本 7 和 H8C 版本 5 的用户：

请确定从下列 URL 下载和安装更新编译程序包的 HEW 修订版，然后安装 M3T-NC30WA 编译程序：

```
http://download.renesas.com/eng/mpumcu/upgrades/IDEs_and_project_managers/hew/index.html
```

8.5.5 安装 HEW (2)

■ 问题：

当我启动 HEW 和尝试建立新的工程工作空间时不能成功，因为工程类型字段中没有显示任何项目。为什么？

■ 解答：

对于 M3T-NC30WA 随附的 HEW 环境版本，请确定使用 M3T-NC30WA 安装程序。此外，在已安装 HEW 环境的机器上安装 HEW 的 M3T-NC30WA 时，请确定通过覆盖现有的 HEW 环境执行安装 。

如果此操作无法解决问题，请执行以下操作进行工具链注册：

(1) 启动 HEW。

(2) 当 [欢迎！（Welcome!）] 对话框显示时，单击 [取消（Cancel）] 按钮。

(3) 在 HEW 菜单中，选择 [工具（Tools）]，然后单击 [管理（Administration）]。[工具管理（Tool Administration）] 对话框将会显示。

(4) 在 [工具管理（Tool Administration）] 对话框中，单击 [注册（Registration）] 按钮。[选择 HEW 注册文件（Choose HEW Registration File）] 对话框将会显示。

(5) 在 [选择 HEW 注册文件（Choose HEW Registration File）] 对话框的文件位置中，定位至 M3T-NC30WA 安装目录（默认为“\MTOOL”）。

(6) 选取“nc30wa.hrf”，然后单击 [选取（Select）] 按钮。

(7) 当您返回 [工具管理（Tool Administration）] 对话框时，单击 [确定（OK）] 按钮。

8.5.6 取消创建

■ 问题：

我该如何在执行创建出现编译程序错误时取消创建？

■ 解答：

使用默认 HEW 设定，即使是在创建期间出现错误，该创建将不会取消，而且会通过连接执行处理。您可以执行下列设定以在一出现错误时取消创建。

(1) 从 [工具（Tools）] 菜单，单击 [选项（Options）]。[选项（Options）] 对话框将会显示。

(2) 在 [选项（Options）] 对话框中，单击 [创建（Build）] 标签。创建处理的设定将会显示。

(3) 选取 [如果错误数量超过此数量时停止创建：（Stop build if the number of errors is exceeded:）]。（您可以在相邻的文本框中指定此错误数量。）

(4) 单击 [确定（OK）] 按钮以关闭对话框。

要在出现警告时取消创建，请选取 [如果警告数量超过此数量时停止创建处理：（Stop build processing if the number of warnings is exceeded:）]。和错误一样，您可以在相邻的文本框中指定此警告数量。

8.5.7 选取创建目标

■ 问题：

当我使用 HEW3 执行创建时，创建的文件并非指定的文件（在工作空间窗口的 [工程（Project）] 标签树中选定的文件），而是编辑程序中打开的文件。为什么？

■ 解答：

当编辑程序中打开的文件具有焦点（您可以看到光标）时，它将会变成创建目标。这是因为源文件通常在编辑之后创建。如果编辑程序没有焦点，在工作空间窗口的 [工程（Project）] 标签树中选定的文件，将是创建目标。

8.5.8 创建配置

■ 问题：

为何即使在选取创建配置的 [发布（Release）] 时出现错误，仍然可以在选取 [调试（Debug）] 时正确执行编译？

■ 解答：

这是因为 [调试（Debug）] 和 [发布（Release）] 所设定的选项是不同的。

关于创建配置的实用程序：

在 HEW 中建立工程时，将会建立两个创建配置：[发布（Release）] 和 [调试（Debug）]。您可以对各配置设定不同的选项样式，以便使这些选项样式可以通过切换创建配置轻松切换。

- 请注意，当 HEW 初次建立这些创建配置时，各配置的选项样式是一样的。

基于上述原因，当 HEW 的其中一个配置的选项样式更改时，这些更改将不会自动应用到另一个配置。

8.5.9 输出调试信息

■ 问题：

我目前使用的是 NC8C 版本 5.30 Release 1。调试信息会在默认情形下于建立工程时输出吗？

■ 解答：

调试信息并没有设定为在默认情形下于建立工作空间时输出。

要输出调试信息，请如下设定相应选项：

(1) 从 [选项（Options）] 菜单，单击 [瑞萨 R8C 标准工具链（Renesas R8C Standard Toolchain）]。

(2) [瑞萨 R8C 标准工具链（Renesas R8C Standard Toolchain）] 对话框将会显示。

(3) 单击 [C] 标签。

(4) 从 [类别（Category）]，选取 [目标（Object）]。

(5) 从 [调试选项（Debug options）] 列表，选取 [-g]。

(6) 单击 [确定（OK）] 按钮。

8.6 SBDATA 声明实用程序

8.6.1 SBDATA 声明实用程序

■ 问题：

为何一些变量会在我使用 SBDATA 声明实用程序 (utlxx) 来生成“sbdata.h”时注出？

■ 解答：

在程序中已声明为 SBDATA 的变量将会先映像至 SBDATA 区域。SBDATA 声明实用程序会将其余部分映像至变量。因此，未映像至 SBDATA 区域的变量将会作为注解输出。共有两种类型的注解输出：

(1) 已在 #pragma SBDATA 中声明之变量的注解。

@ 字符将会附加到注解的末端。

```
//#pragma  SBDATA  ***********          /* size=(   1) / ref=[    22] @ */
```

(2) 应该已经但尚未映像至 SBDATA 的变量的注解。

```
//#pragma  SBDATA  ***********          /* size=(   1) / ref=[    22] */
```

M16C 族 C 编译器应用笔记

附录

附录

附录 A 附加功能

A.1 版本 1.00 Release 1 和版本 2.00 Release 1 之间的附加功能

(1) SB308： SBDATA 声明实用程序

此实用程序分析各种用途的信息，并且以存取频率顺序输出 SBDATA 声明 (#pragma SBDATA)。

您可以通过包含输出标题文件来缩减代码大小，然后重新编译。

(2) SP308： Special 页声明实用程序

此实用程序分析函数调用的信息，并且以函数调用次数的顺序输出 special 页声明 (#pragma SPECIAL)。

您可以通过包含输出标题文件来缩减代码大小，然后重新编译。

(3) 优化选项的级别

- 优化选项划分成五个级别：“-O1”到“-O5”来帮助指定。

-O1	仅执行不会影响调试信息（行信息）的优化。
-O2	使用版本 2.00 Release 2，此级别与 -O1 相同。
-O3	执行优化，包括会影响调试信息的优化。指定 -O 将具有与指定 -O3 时的相同效果。
-O4	除了执行 -O3 级别的优化外，也执行用常数直接替换对外部常数的参考的优化。
-O5	除了执行 -O4 级别的优化外，也执行忽略变量别名（例如，间接参考）的公用子表达式优化。 请注意，在使用同个变量时，如果运算包含直接和指针间接参考，或间接使用多个指针，所建立的代码可能会有未预期的行为。因此，只有在将所生成的代码考虑在内时使用此选项。

- -Ono_logical_or_combine (-ONLOC)

  防止结合连续逻辑 OR 的优化。此选项在指定“-O3”或更大级别时生效。

- -Ocompare_byte_to_word (-OCBTW)

  按字比较顺序存储器中的顺序字节。

  此选项可不考虑其他指定的选项而生效。

(4) 警告选项

- -Wlarge_to_small (-WLTS)

  对从大的大小转换为小的大小之隐含类型发出警告。

  除非明确指定，否则此选项将不会生效，即使 “-Wall” 选项已指定。

- -Wuninitialize_variable (-WUV)

  使用未初始化的自动变量时发出警告。

  指定 “-Wall” 选项时，此选项将会在即使没有指定的情形下自动生效。

  请注意，在用户应用程序中，当初始化按通过 “if” 或 “for” 条件的转移执行时，编译程序会将此作为未执行初始化求值，并且会输出一则警告。

  实例：

```
main()
{
   int i;
   int val;
   for(i=0;i<2;i++){
      f();
      val = 1; //初始化永远在此执行
   }
   ff( val );
}
```

(5) 更改输出代码的选项

- -finfo

  输出 SB308 和 SP308 的信息文件。

- -fuse_CLIP (-fUC)

  生成代码，使用 CLIP 指令。

- -fuse_MAX (-fUM)

  生成代码，使用 MIN 和 MAX 指令。

(6) asm 函数的变量名称扩展处理

已添加变量名称扩展的支持，使用 $@。指定 $@ 时，编译程序将在求值相应变量是自动变量、寄存器变量，或外部变量之后执行输出。

实例：`asm("    mov.w   #10H,$@",I);`

(7) AS308 功能

包含字符串的运算现在已可在汇编程序用法和助记码操作数中指定。

(8) XRF308 功能

xrf308 可同时打开的最大文件数已更改为 600。

A.2 版本 2.00 Release 1 和版本 2.00 Release 2 之间的附加功能

无。

A.3 版本 2.00 Release 2 和版本 3.00 Release 1 之间的附加功能

(1) aopt308： 汇编程序优化器

已添加汇编程序优化器 aopt308，允许使用 adjnz 命令的条件转移进行优化。请注意，aopt308 会在使用编译驱动器 nc308 执行编译并指定其中一个 “-O”、“O3”、“-O4”、“-O5”、“-OR” 或 “-OS” 选项时自动运行。

(2) utl308： SBDATA 声明和 special 页声明实用程序

SBDATA 声明实用程序和 special 页函数声明实用程序已合并到 utl308 中，而现有的 sb308 和 sp308 实用程序已停止使用。

您可以通过包含此实用程序的输出标题文件来缩减代码大小。

(3) STK 阅读器（仅限 W95J 和 Solaris 版本）

已为堆栈使用计算实用程序建立一个 GUI，以提高可用性和查看功能。

(4) 映像阅读器（仅限 W95J 版本）

已添加用于查看映像信息的实用程序，允许更简易地查看映像信息。

(5) 支持集成开发环境 TM 版本 3.00（仅限 W95J 版本）

已添加集成开发环境 TM 版本 3.00 的支持。

请注意，TM 版本 2.01 和更早的版本已不再支持。

(6) 堆栈信息和实用程序信息的合并

直至目前为止，堆栈信息 (.stk) 以及 SBDATA 声明和 special 页声明实用程序 (.utl) 的信息个别为每个源文件输出。这些信息已经和检查器信息合并，从而不再需要建立个别数据文件。

请注意，检查器信息由集成开发环境 TM 版本 3.00 使用，并且存储在可再定位文件 (.r30) 和绝对模块文件 (.x30) 中。

(7) NC308 功能

(a) 优化选项

- “-Oloop_unroll[=*循环最大次数*](-OLU)” 命令选项

  通过在编译期间扩展循环次数已清除的 “for” 循环来提高执行速度。

  如果已指定循环的最大次数，在指定次数内的循环之循环将会扩展。

  如果已省略循环的最大次数，循环五次或更少的 “for” 循环将会扩展。

(b) 加强警告功能和制止选项

- 标头文件中没有添加标准程序库函数的警告

  指定 “-Wnon_prototype” 或 “-Wall” 选项时，将使用标准程序库函数。如果未添加这些函数需要的标题文件，将会输出警告信息。

- “-Wno_warning_stdlib(-WNWS)” 命令选项

  制止在未添加上述标题文件时输出警告信息。

  此选项在指定 “-Wnon_prototype (-WNP)” 或 “-Wall” 选项时生效。

(c) 在默认情形下，函数会开始对齐，并对应制止选项。

函数指令位置的偶数对齐的开始现在已可在默认情形下执行。

由此，“-fno_align”选项已添加来制止函数指令位置的偶数对齐的开始，而用以进行偶数对齐的“-falign”选项则已被停止。

(d) 实用程序信息输出选项的更改

- “-finfo” 命令选项功能的更改

  由于堆栈信息和实用程序已经合并，检查器信息（包含堆栈信息和实用程序信息）现在可使用“-finfo” 选项输出。

  以前的 “-fshow_stack_usage(-fSSU)” 选项已停止使用。

(e) 默认情形下使用的 CLIP、MAX 和 MIN 指令。

使用 CLIP、MAX 和 MIN 指令时需要的 “-fuse_CLIP(-fUC)” 和 “-fuse_MAX(-fUM)” 选项已经停止使用，因此这些指令可以在不指定任何选项的情形下使用。

(8) AS308 功能

(a) 建立检查器信息的功能

- “-finfo” 命令选项

  已添加 “-finfo” 命令选项，用于建立检查器信息。

- 指令

  下列指令已添加。

- .INSF： 显示检查器信息的函数（子例程）起始信息。
- .EINSF： 显示检查器信息的函数（子例程）终止信息。
- .CALL： 显示检查器信息的函数（子例程）调用目标信息。
- .STK： 显示检查器信息的堆栈信息。

(b) 执行转移优化

转移优化可以使用 adjnz 和 sbjnz 指令执行。请注意，这相等于优化选择规则的条件转移指令。

(9) LMC308 功能

- “-A” 命令选项

  已添加 “-A” 命令选项，用于指定机器语言数据输出至所建立文件的地址范围。

- “-F” 命令选项

  已添加 “-F” 命令选项，用于输出未在指定绝对模块文件中寄存的地址的可选数据。

- 输出文件名指定选项的扩展名功能

  输出文件名指定选项 “-O” 现在已可用于指定输出文件名的扩展名。

- 原始 Mitsubishi HEX 格式文件生成条件的功能更改

  以前，在指定“-H”命令选项时，寄存在指定绝对模块文件中所有段的文件将以原始 Mitsubishi HEX 格式建立，并以设定数据的最大地址值作为根据。在此版本中，这将只在具备超过 1MB 的 CODE 或 ROMDATA 属性之段的地址时发生。请注意，建立原始 Mitsubishi HEX 格式文件时，将会输出“已生成 mitsubishi 微型计算机的原始 HEX 格式”(Original HEX format for mitsubishi microcomputers is generated) 警告信息。

- 不存在 ROM 数据时的功能更改

  以前，当指定绝对模块文件中具备 CODE 或 ROMDATA 属性的段不包含任何数据时，将会建立不包含数据的机器语言文件。在此版本中，将会发生错误。

- 执行路径显示

  lmc308 的执行状态现在将显示为一个路径，与汇编程序一样。

A.4 版本 3.00 Release 1 和版本 3.10 Release 1 之间的附加功能

(1) NC308 功能

(a) 加强优化

编译程序体系结构中的条件转移、位运算，以及寄存器变量的优化已经获得加强。

汇编程序的优化也已经改进，包括结合如同顺序区域和寄存器的逻辑运算，以及机器固定的指令，如 btsts。

(b) 加强警告功能和附加选项

已添加 “-Wno_used_argument(-WNUA)” 命令选项。

指定此选项时，将发出函数定义期间未使用参数的警告。

此函数不会生效，即使是在已指定 “-Wall” 选项时，并且需要个别指定。

(c) 汇编列表文件的选项

已添加 “-dsource_in_list(-dSL)” 命令选项。

指定此选项时，一旦建立可再定位文件，汇编列表文件也会建立，并且作为注出的 C 源行输出。

请注意，此选项在指定 “-P” 、“-E” 或 “-S” 选项时不会发生作用，因为汇编并没有执行。

(d) “-dsource(-dS)” 选项功能的更改

指定此选项和注出的 C 源行输出至生成的汇编源时，不必要的可再定位文件 (.r30) 将不再保持未删除。

(2) AS308 功能

对位运算指令的 “BTST” 操作数指定绝对寻址 “base:19” 时，代码现在将以 S 格式生成。

(3) utl308 的加强功能： SBDATA 声明和 special 页函数声明实用程序

- “-sb308” 和 “-sp308” 命令选项现在已可同时指定。

  SBDATA 处理和 special 页处理现在已可同时执行。

- 已添加 “-fsection” 命令选项。

  指定此选项时，在 #pragma SECTION 中用于更改段位置的变量和函数也会处理。

- “-o” 命令选项的处理已经更改，而 “-fover_write(-fOW)” 已添加。

  使用 “-O” 选项指定输出文件时，如果该文件已经存在，结果将会输出至标准输出，而该文件将不会被覆盖。

  同时指定 “-fover_write(-fOW)” 选项时，如果现有文件是为 “-o” 指定，该文件将会被强制覆盖。

A.5 版本 3.10 Release 1 和版本 3.10 Release 2 之间的附加功能

(1) NC308 功能

(a) 提高 #include 的最大嵌套

可使用 #include 指令装入的嵌套文件最大数量已从 8 个提高到 40 个。

(b) 注解处理方式的更改

注解处理已更改为配合用于公用 C++ 处理的方式。

在以前的版本中，// 和换行代码之间的部分是在 /* 和 */ 之间括起来的部分处理之后处理。在此版本中，注解处理将会先从注解描写字符开始。

因此，请紧记注解处理已更改为以下实例中显示的方式。

```
i = 4     //* 注解 */
    +2;
```

在以前的版本中，/* 和 */ 之间的所有内容都被当成注解，使 i = 4 / +2;。

在此版本中，// 和行结束之间的所有内容都被当成注解，使 i = 4 +2;。

(c) 小写字母 #pragma 扩展函数名称的支持

#pragma 指令（如 ADDRESS、INTERRUPT、SBDATA 和 ASM）后面的指定扩展函数字现在可使用小写字母指定。不论使用大写还是小写字母，该功能都不会更改。

(d) #pragma ASM、#pragma ENDASM 的更改

在以前的版本中，如果在 #pragma ASM 和 #pragma ENDASM 之间指定非成对引号字符（' 或 "），在记号分析期间可能会出现错误。在此版本中，这是可以允许的。

```
#pragma     ASM

    nop     ; Insert "NOP"    // 双引号字符是成对的，不存在问题。

    nop     ; Don't care      // 只存在一个引号字符。
                              // 这在以前的版本将导致错误，
                              // 但现在已可允许。

#pragma     ENDASM
```

(e) 加强警告功能

在以前的版本中，指定 “-Wno_used_argument(-WNUA)” 命令选项时，将会发出未使用堆栈传递参数的警告。在此版本中，也会发出未使用寄存器传递参数的警告。

A.6 版本 3.10 Release 2 和版本 3.10 Release 3 之间的附加功能

无。

A.7 版本 3.10 Release 3 和版本 5.00 Release 1 之间的附加功能

(1) Linux 版本

已添加 Linux（支持日文 Turbolinux 7 工作站）版本。

(2) NC308 功能

(a) “加长”类型、“_Bool”类型，以及“restrict”限定语符的支持

已为 ISO/IEC 9899:1999 (ANSI C99) 添加新建立之类型（“加长”类型和“_Bool”类型）以及“restrict”限定语符的支持。

(b) 扩展函数 #pragma BITADDRESS

已添加扩展函数 #pragma BITADDRESS，允许“_Bool”类型外部变量分配到指定绝对地址的 1 位。

(c) 扩展函数 #pragma SB16DATA

已添加扩展函数 #pragma SB16DATA，允许使用 dsp16[SB] 寻址存取外部变量。

(d) 扩展函数 #pragma DMAC

已添加扩展函数 #pragma DMAC，允许将外部变量分配到 DMAC 寄存器，以便可以使用 C 来存取 DMAC 寄存器。此功能支持 DMAC 通道 0 和 1。

(e) 加强优化功能

包含下列功能的优化（尤其是改进执行速度）已加强：

- 内联函数功能
- 常数传输优化
- 用于转移的分析优化，如“if”语句。
- 诸如用于算术计算的优化

(f) 扩展函数 #pragma SECTION 处理方式的更改

#pragma SECTION 的处理已经更改，以允许特定源文件中的“data”和“rom”段名称可以多次更改。

(g) M32C/80 系列之预定义的宏的更改

使用 M32C/80 系列的“-M82”代码生成选项时，会将“M32C80”定义为预定义的宏，而不是“M16C80”。

(h) 更快的中断处理函数

中断处理函数的寄存器保存与恢复处理现在已变得更快。

(i) asm 函数扩展

现在可以在 asm 函数中使用多达两个 $$ 和 $@。

(j) 整数常数的二进制支持

现在可为整数常数指定二进制编号。以“0b”前缀开头的整数常数将当作二进制编号。例如，您能够以二进制指定“0b00010010”来代表十六进制编号“0x12”。

(3) 标准程序库函数的内联扩展宏

已将 strcpy、strcmp、memcpy 和 memset 标准程序库函数的内联扩展宏添加至标准标题“string.h”。

(4) 加强 AS308 功能

(a) 用于地址寄存器相对寻址的优化

当地址寄存器相对值 0 用于标准指令和位指令时，将选定地址寄存器间接寻址。

```
mov.w   #1234h,0[A0]    →  mov.w   #1234h,[A0]
bclr            1.0[A0]       →        bclr          1,[A0]
```

(b) SBSYM16 指令

参考使用此指令定义的符号时，将选定 dsp16[SB] 寻址。

```
.glb            sym
.sbsym16        sym
abs.b sym               →       dsp:16[SB] 将选定
```

(5) 加强 LN308 功能

(a) 命令行之指定规则的更改

一行可指定的字符数已从 255 个增加至 2,048 个。

(6) 加强映像阅读器功能

下列功能已添加至映像阅读器。

- 映像信息打印
- 在左窗口中滚动
- 显示存储器大小图像的扩展/缩减

A.8 版本 5.00 Release 1 和版本 5.10 Release 1 之间的附加功能

(1) NC308 功能

(a) “-fdouble_32[-fD32]” 命令选项

指定双精度类型和浮点类型一样以 32 位数据长度处理。

注意：
使用此选项时，请确定指明函数原型。如果没有原型声明，代码可能不会正确生成。

(b) “-Wno_used_static_function[-WNUSF]” 命令选项

指定 “-Ostatic_to_inline[-OSTI]” 选项时，下列警告信息将会在静态函数是扩展内联，或未从任何地点参考时显示：

[Warning(ccom):xxx.c,line xx] 静态函数（*函数名称*）的代码生成可通过使用 -ferase_static_function(-fESF) 选项制止。

(c) “-ferase_static_function=*静态函数的名称*[-fESF=*静态函数的名称*]” 命令选项

防止为指定的静态函数生成代码。

注意：
将静态函数的此选项指定为由 “-Wno_used_static_function[-WNUSF]” 命令选项检测。

(d) 加强 “-Oconst” 命令选项

将对以常数初始化的数组的参考替换成对常数的参考。

(e) 总和计算的加强处理

rmpa 指令现在已为 “for” 语句中的总和计算处理生成。

程序实例：

```
signed  char ch1[10];
signed  char ch2[10];
int    ih1[10] ;
int    ih2[10] ;

#define  LOOP_MAX  9

signed  char *pc1,*pc2;
int    *pi1,*pi2;

int i;
long   l;

void   func_c1(void)
{
    intj;

    i = 0;
    for( j=0; j<LOOP_MAX; j++ ){
       i = i + (int)*pc1 * (int)*pc2;
       pc1++;
       pc2++;
    }
}

void   func_c2(void)
{
    intj;

    i = 0;
```

```
    for( j=0; j<LOOP_MAX; j++ ){
        i = i + (int)ch1[j] * (int)ch2[j];
    }
}
```

生成的代码实例：

```
_func_c1:
    pushm     R1,R2,R3,A0,A1
    mov.w     #0000H,_i:16
    mov.w     _i:16,R0
    mov.l     _pc1:16,A0
    mov.l     _pc2:16,A1
    mov.w     #0009H,R3
    rmpa.b
    mov.w     R0,_i:16
    add.l     #00000009H,_pc1:16
    add.l     #00000009H,_pc2:16
    popm      R1,R2,R3,A0,A1
    rts

_func_c2:
    pushm     R1,R2,R3,A0,A1
    mov.w     #0000H,_i:16
    mov.w     _i:16,R0
    mov.w       #(_ch1&0FFFFH),A0
    mov.w       #(_ch2&0FFFFH),A1
    mov.w       #0009H,R3
    rmpa.b
    mov.w     R0,_i:16
    popm      R1,R2,R3,A0,A1
    rts
```

注意：

- **需要指定其中一个“-O[3-to-5]”、“-OS”或“-OR”优化选项。**
- **rmpa 指令可能不会生成，取决于“for”语句中之总和计算的处理。**

(2) AS308 功能

(a) AS308 选项“-PATCH_TA”和“-PATCH_Tan”

生成代码以保存定时器功能的警惕，用于控制三相电机。

(b) LN308 选项“-U”

输出未使用函数的警告信息输出。

(c) LMC308 选项“-F”

现在可指定在可用区域中嵌入任意数据的处理范围。

A.9 版本 5.10 Release 1 和版本 5.20 Release 1 之间的附加功能

(1) NC308 功能

(a) HEW (High-performance Embedded Workshop)
已添加 HEW，它是一个结合与批次管理各类工具的集成开发环境，包括编辑程序和调试程序。

(b) “-fno_switch_table [-fNST]” 命令选项
生成永远执行 “switch” 语句之比较的转移代码。

注意：
指定此选项以防止建立使用 “switch” 语句之表跳转的代码。

(c) “-fswitch_other_section [-fSOS]” 命令选项
指定程序段以外的段中的 “switch” 语句的表代码。

注意：
此选项在指定 “-fno_switch_table [-fNST]” 选项时没有作用。

(d) “-fmake_vector_table [-fMVT]” 命令选项
自动生成中断向量表。

(e) “-fmake_special_table [-fMST]” 命令选项
自动生成 special 页表。

(f) “-Oforward_function_to_inline [-OFFTI]” 命令选项
对所有内联函数执行内联扩展。

(g) “-Ofloat_to_inline [-OFTI]” 命令选项
对浮点运行库执行内联扩展，以提高浮点计算处理的速度。

注意：
此选项在同时指定 “-M82” 编译选项时生效。

(h) “-Oglb_jmp [-OGJ]” 命令选项
根据连接期间的转移距离，选择优化转移指令。

(i) “-Wno_used_function [-WNUF]” 命令选项
检测到未使用的公式时输出警告。

(j) “-Wstop_at_link [-WSAL]” 命令选项
防止连接期间出现警告时建立绝对模块文件。

(k) “-Wundefined_macro [-WUM]” 命令选项
在 “#if” 语句中使用未定义的宏时输出警告。

(l) 扩展函数 #pragma INTHANDLER(#pragma HANDLER) 的选项
现在已允许在输入中断处理器后立即执行多个中断。

注意：
此功能在 #pragma INTHANDLER (#pragma HANDLER) 中指定 “/E” 选项时生效。

(2) AS308 功能

(a) “－fMVT” 命令选项

自动生成变量向量表。

(b) “－fMST” 命令选项

自动生成 special 页表。

(c) “－JOPT” 命令选项

优化参考全局标签的转移指令。

(d) .ID 指令

设定 ID 代码检查函数的 ID 代码

(e) .PROTECT 指令

设定用于 ROM 代码保护的控制地址的值

(f) .RVECTOR 指令

设定软件中断编号和软件中断名称。

(g) .SVECTOR 指令

设定 special 页编号和 special 页名称。

(3) LN308 功能

(a) “－fMVT” 命令选项

自动生成变量向量表。

(b) “－fMST” 命令选项

自动生成 special 页表。

(c) “－VECT” 命令选项

设定执行变量向量表的自动生成时可用区域的地址。

(d) “－JOPT” 命令选项

优化参考全局标签的转移指令。

(e) “－W” 命令选项

防止出现警告时建立绝对模块文件。

(f) 删除 “－L” 命令选项的警惕

以前版本的下列警惕已经删除：

指定多个程序库文件时，连接程序将会按它们的指定顺序参考它们。因此，在满足下列条件时，将会出现未定义符号的错误：

- 可再定位文件 (sample.r30) 参考寄存在程序库文件 (A.LIB) 中的全局符号。
- 连接在 (1) 中的程序库文件 (A.LIB) 中的可再定位的模块参考寄存在另一个程序库文件 (B.LIB) 中的全局符号。
- 在 “-L” 选项中，(2) 的程序库文件 (B.LIB) 在 (1) 的程序库文件 (A.LIB) 前面指定，如以下实例所示：

```
>ln308 sample.r30 -L B.LIB A.LIB
```

(4) LMC308 功能

(a) “－protectx”命令选项

设定用于 ROM 代码保护的控制地址的值。